Image Processing Toolbox™

User's Guide

MATLAB®

R2018b

MathWorks®

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 1993–2018 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)
March 2015	Online only	Revised for Version 9.2 (Release 2015a)
September 2015	Online only	Revised for Version 9.3 (Release 2015b)
March 2016	Online only	Revised for Version 9.4 (Release 2016a)
September 2016	Online only	Revised for Version 9.5 (Release 2016b)
March 2017	Online only	Revised for Version 10.0 (Release 2017a)
September 2017	Online only	Revised for Version 10.1 (Release 2017b)
March 2018	Online only	Revised for Version 10.2 (Release 2018a)
September 2018	Online only	Revised for Version 10.3 (Release 2018b)

Contents

1 Getting Started

Image Processing Toolbox Product Description 1-2
Key Features 1-2

Configuration Notes 1-3

Compilability 1-4

Basic Image Import, Processing, and Export 1-5

Correct Nonuniform Illumination and Analyze Foreground Objects 1-11

Getting Help 1-21
Product Documentation 1-21
Image Processing Examples 1-21
MATLAB Newsgroup 1-22

Acknowledgments 1-23

2 Introduction

Images in MATLAB 2-2

Image Coordinate Systems 2-3
Pixel Indices 2-3
Spatial Coordinates 2-4

Define World Coordinates Using XData and YData Properties 2-7
Display an Image using Nondefault Spatial Coordinates 2-7

Define World Coordinates Using Spatial Referencing 2-10

Image Types in the Toolbox 2-13

Binary Images 2-15

Indexed Images 2-16

Grayscale Images 2-18

Truecolor Images 2-20

Display Separated Color Channels of an RGB Image 2-23

Convert Between Image Types 2-26

Convert Image Data Between Classes 2-28
Overview of Image Class Conversions 2-28
Losing Information in Conversions 2-28
Converting Indexed Images 2-28

Process Multiframe Image Arrays 2-30

Perform an Operation on a Sequence of Images 2-31

Detecting Cars in a Video of Traffic 2-34

Batch Processing Using the Image Batch Processor App . . . 2-42
Open Image Batch Processor App 2-42
Load Images into the Image Batch Processor App 2-43
Specify the Batch Processing Function 2-45
Perform the Operation on the Images 2-48
Obtain the Results of the Batch Processing Operation 2-52

Process Large Set of Images Using MapReduce Framework and Hadoop 2-57
Download Sample Data 2-57
View Image Files and Test Algorithm 2-58

Testing Your MapReduce Framework Locally: Data Preparation 2-60
Testing Your MapReduce Framework Locally: Running your algorithm 2-63
Running Your MapReduce Framework on a Hadoop Cluster . 2-67

What Is an Image Sequence? 2-72

Toolbox Functions That Work with Image Sequences 2-73

Image Arithmetic Functions 2-76

Image Arithmetic Saturation Rules 2-77

Nest Calls to Image Arithmetic Functions 2-78

Find Vegetation in a Multispectral Image 2-79

3 Reading and Writing Image Data

Get Information About Graphics Files 3-2

Read Image Data into the Workspace 3-3

Read Multiple Images from a Single Graphics File 3-5

Read and Write 1-Bit Binary Images 3-6

Write Image Data to File in Graphics Format 3-8

Determine Storage Class of Output Files 3-10

DICOM Support in the Image Processing Toolbox 3-11

Read Metadata from DICOM Files 3-12
Private DICOM Metadata 3-12
Create Your Own Copy of DICOM Dictionary 3-13

Read Image Data from DICOM Files **3-14**
View DICOM Images **3-14**

Write Image Data to DICOM Files **3-15**
Include Metadata with Image Data **3-15**

Explicit Versus Implicit VR Attributes **3-17**

Remove Confidential Information from a DICOM File **3-18**

Create New DICOM Series **3-19**

Mayo Analyze 7.5 Files **3-22**

Interfile Files **3-23**

Work with High Dynamic Range Images **3-24**
Read High Dynamic Range Image **3-24**
Create High Dynamic Range Image **3-24**
Write High Dynamic Range Image to File **3-25**

Display High Dynamic Range Image **3-26**

4 Displaying and Exploring Images

Image Display and Exploration Overview **4-2**

Display an Image in a Figure Window **4-4**
Overview **4-4**
Specifying the Initial Image Magnification **4-6**
Controlling the Appearance of the Figure **4-7**

Display Multiple Images **4-10**
Display Multiple Images in Separate Figure Windows **4-10**
Display Multiple Images in a Montage **4-10**
Display Images Individually in the Same Figure **4-12**
Compare a Pair of Images **4-13**

View Thumbnails of Images in Folder or Datastore **4-15**
View Thumbnails of Images in Folder **4-15**
View Thumbnails of Images in Image Datastore **4-20**
Remove Image from Image Browser and Create Datastore . . **4-25**

Interact with Images Using Image Viewer App **4-32**
Open the Image Viewer App **4-33**
Initial Image Magnification in the Image Viewer App **4-34**
Choose the Colormap Used by the Image Viewer App **4-35**
Import Image Data from the Workspace into the Image Viewer App **4-36**
Export Image Data from the Image Viewer App to the Workspace **4-37**
Save Image Data Displayed in Image Viewer **4-38**
Close the Image Viewer App **4-39**
Print Images Displayed in Image Viewer App **4-40**

Create and Open Reduced Resolution Files **4-41**
Open a Reduced Resolution File **4-41**

Explore Images with Image Viewer App **4-42**
Explore Images Using the Overview Tool **4-42**
Pan Images Displayed in Image Viewer App **4-45**
Zoom Images in the Image Viewer App **4-45**
Specify Image Magnification in Image Viewer **4-46**

Get Pixel Information in Image Viewer App **4-49**
Determine Individual Pixel Values in Image Viewer **4-49**
Determine Pixel Values in an Image Region **4-51**
Determine Image Display Range in Image Viewer **4-54**

Measure Distance Between Pixels in Image Viewer App **4-57**
Determine Distance Between Pixels Using Distance Tool **4-57**
Export Endpoint and Distance Data **4-59**
Customize the Appearance of the Distance Tool **4-59**

Get Image Information in Image Viewer App **4-61**

Adjust Image Contrast in Image Viewer App **4-63**
Open the Adjust Contrast Tool **4-63**
Adjust Image Contrast Using the Histogram Window **4-65**
Adjust Image Contrast Using Window/Level Tool **4-66**
Make Contrast Adjustments Permanent **4-69**

Interactive Contrast Adjustment **4-70**

Crop Image Using Image Viewer App **4-72**

Explore 3-D Volumetric Data with Volume Viewer App **4-77**
Load Volume Data into the Volume Viewer **4-77**
View the Volume Data in the Volume Viewer **4-79**
View the Volume Data in the Volume Viewer **4-82**
Refine the View with the Rendering Editor **4-86**

View Image Sequences in Video Viewer App **4-89**
View MRI Sequence Using Video Viewer App **4-89**
Configure Video Viewer App **4-92**
Specifying the Frame Rate **4-95**
Specify Color Map **4-95**
Get Information about an Image Sequence **4-96**

Convert Multiframe Image to Movie **4-97**

Display Different Image Types **4-98**
Display Indexed Images **4-98**
Display Grayscale Images **4-99**
Display Binary Images **4-100**
Display Truecolor Images **4-102**

Add Colorbar to Displayed Image **4-104**

Print Images **4-106**
Graphics Object Properties That Impact Printing **4-106**

Manage Display Preferences **4-107**
Retrieve Values of Toolbox Preferences **4-107**
Set Values of Toolbox Preferences **4-107**

5 Building GUIs with Modular Tools

Build Custom Image Processing Apps Using Modular Interactive Tools **5-2**

Interactive Modular Tool Workflow **5-9**
Display the Target Image in a Figure Window **5-11**
Associate Modular Tools with the Target Image **5-11**
Associate Modular Tools with a Particular Target Image **5-12**
Get Handle to Target Image **5-14**
Specify the Parent of a Modular Tool **5-17**
Position Modular Tools in a GUI **5-21**
Adding Navigation Aids to a GUI **5-23**

Build App To Display Pixel Information **5-27**

Build App for Navigating Large Images **5-30**

Customize Modular Tool Interactivity **5-32**

Build Image Comparison Tool **5-33**

Create Your Own Modular Tools **5-37**

Create Angle Measurement Tool Using ROI Objects **5-39**

6 Geometric Transformations

Resize an Image with imresize Function **6-2**

Rotate an Image **6-6**

Crop an Image **6-9**

Translate an Image using imtranslate Function **6-11**

2-D and 3-D Geometric Transformation Process Overview . . **6-14**
Create Geometric Transformation Object **6-14**
Perform the Geometric Transformation **6-18**

Matrix Representation of Geometric Transformations **6-19**
2-D Affine Transformations **6-19**
2-D Projective Transformations **6-21**
Create Composite 2-D Affine Transformations **6-22**

3-D Affine Transformations 6-24

Specify Fill Values in Geometric Transformation Output 6-27

Perform Simple 2-D Translation Transformation 6-30

N-Dimensional Spatial Transformations 6-35

Register Two Images Using Spatial Referencing to Enhance Display 6-37

Create a Gallery of Transformed Images 6-44

Exploring a Conformal Mapping 6-65

Exploring Slices from a 3-Dimensional MRI Data Set 6-79

Padding and Shearing an Image Simultaneously 6-88

7 Image Registration

Approaches to Registering Images 7-3
Registration Estimator App 7-4
Intensity-Based Automatic Image Registration 7-5
Control Point Registration 7-6
Automated Feature Detection and Matching 7-7

Register Images Using the Registration Estimator App 7-9
Load Images into Registration Estimator App 7-9
Obtain Initial Registration Estimate 7-11
Refine the Registration Settings 7-12
Export Registration Results 7-16

Load Images into Registration Estimator App 7-18
Load Images from File or from the Workspace 7-18
Provide Spatial Referencing Information 7-20
Provide an Initial Geometric Transformation 7-21

Tune Registration Settings in Registration Estimator App . . 7-22
Geometric Transformations Supported by Registration Estimator App 7-22
Feature-Based Registration Settings 7-23
Intensity-Based Registration Settings 7-24
Nonrigid and Post-Processing Settings 7-25

Export the Results from Registration Estimator App 7-26
Export Results to the Workspace 7-26
Generate a Function 7-26

Techniques Supported by Registration Estimator App 7-28
Feature-Based Registration 7-28
Intensity-Based Registration 7-29
Nonrigid Registration 7-29

Intensity-Based Automatic Image Registration 7-31

Create an Optimizer and Metric for Intensity-Based Image Registration 7-34

Use Phase Correlation as Preprocessing Step in Registration 7-36

Register Multimodal MRI Images 7-42

Register Multimodal 3-D Medical Images 7-56

Registering an Image Using Normalized Cross-Correlation . 7-65

Control Point Registration 7-72

Geometric Transformation Types for Control Point Registration 7-75

Control Point Selection Procedure 7-77

Start the Control Point Selection Tool 7-80

Find Visual Elements Common to Both Images 7-83
Use Scroll Bars to View Other Parts of an Image 7-83
Use the Detail Rectangle to Change the View 7-83
Pan the Image Displayed in the Detail Window 7-84

Zoom In and Out on an Image 7-84
Specify the Magnification of the Images 7-85
Lock the Relative Magnification of the Moving and Fixed Images 7-86

Select Matching Control Point Pairs 7-88
Pick Control Point Pairs Manually 7-88
Use Control Point Prediction 7-90
Move Control Points 7-93
Delete Control Points 7-94

Export Control Points to the Workspace 7-95

Find Image Rotation and Scale 7-98

Find Image Rotation and Scale Using Automated Feature Matching 7-103

Use Cross-Correlation to Improve Control Point Placement 7-110

Register an Aerial Photograph to a Digital Orthophoto 7-111

Registering an Aerial Photo to an Orthophoto 7-116

8 Designing and Implementing Linear Filters for Image Data

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

Integral Image 8-5

Filter Grayscale and Truecolor (RGB) Images using imfilter Function 8-7

imfilter Boundary Padding Options 8-12

Filter Images Using Predefined Filters **8-16**

Generate HDL Code for Image Sharpening **8-21**

What is Guided Image Filtering? **8-29**

Perform Flash/No-flash Denoising with Guided Filter **8-31**

Segment Thermographic Image after Edge-Preserving Filtering **8-36**

Apply Multiple Filters to Integral Image **8-41**

Reducing Noise in Image Gradients **8-47**

Design Linear Filters in the Frequency Domain **8-56**
Two-Dimensional Finite Impulse Response (FIR) Filters **8-56**
Transform 1-D FIR Filter to 2-D FIR Filter **8-57**
Frequency Sampling Method **8-60**
Windowing Method **8-61**
Creating the Desired Frequency Response Matrix **8-62**
Computing the Frequency Response of a Filter **8-63**

9 Transforms

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**
Image Compression with the Discrete Cosine Transform **9-17**

Hough Transform **9-21**
Detect Lines in Images Using Hough **9-21**

Radon Transform 9-27
Plot the Radon Transform of an Image 9-29
Viewing the Radon Transform as an Image 9-32

Detect Lines Using the Radon Transform 9-34

The Inverse Radon Transformation 9-39
Inverse Radon Transform Definition 9-39
Reconstructing an Image from Parallel Projection Data 9-41

Fan-Beam Projection 9-45
Image Reconstruction from Fan-Beam Projection Data 9-48
Reconstruct Image using Inverse Fanbeam Projection 9-49

Reconstructing an Image from Projection Data 9-55

10 Morphological Operations

Morphological Dilation and Erosion 10-2
Processing Pixels at Image Borders (Padding Behavior) 10-3

Structuring Elements 10-5
Determine the Origin of a Structuring Element 10-7
Structuring Element Decomposition 10-8

Dilate an Image to Enlarge a Shape 10-10

Erode an Image To Remove Thin Lines 10-15

Operations That Combine Dilation and Erosion 10-17
Use Morphological Opening to Extract Large Image Features 10-17

Skeletonization 10-23
Skeletonize Binary Image 10-23

Find Object Perimeters 10-25
Find Perimeter Pixels in Binary Image 10-25

Pixel Connectivity **10-27**
Defining Connectivity in an Image **10-27**
Choosing a Connectivity **10-29**
Specifying Custom Connectivities **10-29**

Understanding Morphological Reconstruction **10-31**
Understanding the Marker and Mask **10-33**
Finding Peaks and Valleys **10-34**

Flood-Fill Operations **10-43**
Specifying Connectivity **10-43**
Specifying the Starting Point **10-44**
Filling Holes **10-44**

Detecting a Cell Using Image Segmentation **10-46**

Granulometry of Snowflakes **10-53**

Distance Transform of a Binary Image **10-58**

Label and Measure Objects in a Binary Image **10-60**
Understanding Connected-Component Labeling **10-60**
Selecting Objects in a Binary Image **10-62**
Finding the Area of the Foreground of a Binary Image **10-63**
Finding the Euler Number of a Binary Image **10-63**

Lookup Table Operations **10-65**
Creating a Lookup Table **10-65**
Using a Lookup Table **10-65**

11 Analyzing and Enhancing Images

Pixel Values **11-3**
Determine Values of Individual Pixels in Images **11-3**

Intensity Profile of Images **11-5**
Create an Intensity Profile of an Image **11-5**
Create Intensity Profile of an RGB Image **11-6**

Contour Plot of Image Data **11-9**
Create Contour Plot of Image Data **11-9**

Measuring Regions in Grayscale Images **11-11**

Finding the Length of a Pendulum in Motion **11-17**

Create Image Histogram **11-25**

Image Mean, Standard Deviation, and Correlation Coefficient **11-27**

Edge Detection **11-28**
Detect Edges in Images **11-28**

Boundary Tracing in Images **11-31**
Trace Boundaries of Objects in Images **11-31**
Select First Step and Direction for Tracing **11-35**

Quadtree Decomposition **11-37**
Perform Quadtree Decomposition on an Image **11-37**

Detect and Measure Circular Objects in an Image **11-41**

Identifying Round Objects **11-58**

Measuring Angle of Intersection **11-65**

Measuring the Radius of a Roll of Tape **11-73**

Texture Analysis **11-77**

Detect Regions of Texture in Images **11-79**

Texture Analysis Using the Gray-Level Co-Occurrence Matrix (GLCM) **11-81**

Create a Gray-Level Co-Occurrence Matrix **11-82**

Specify Offset Used in GLCM Calculation **11-84**

Derive Statistics from GLCM and Plot Correlation **11-85**

Adjust Image Intensity Values to Specified Range **11-88**

Gamma Correction **11-90**
Specify Gamma when Adjusting Contrast **11-90**

Contrast Enhancement Techniques **11-92**

Specify Contrast Adjustment Limits **11-97**
Specify Contast Adjustment Limits as Range **11-97**
Set Image Intensity Adjustment Limits Automatically **11-98**

Histogram Equalization **11-99**
Adjust Intensity Values Using Histogram Equalization **11-99**
Plot Transformation Curve for Histogram Equalization . . . **11-101**

Adaptive Histogram Equalization **11-103**
Adjust Contrast using Adaptive Histogram Equalization . . **11-103**

Enhance Color Separation Using Decorrelation Stretching **11-106**
Simple Decorrelation Stretching **11-106**
Linear Contrast Stretching **11-112**
Decorrelation Stretch with Linear Contrast Stretch **11-113**

Enhancing Multispectral Color Composite Images **11-115**

Low-Light Image Enhancement **11-128**

Apply Gaussian Smoothing Filters to Images **11-141**

Noise Removal **11-151**
Remove Noise by Linear Filtering **11-151**
Remove Noise Using an Averaging Filter and a Median Filter **11-151**
Remove Noise By Adaptive Filtering **11-155**

Compute 3-D Superpixels of Input Volumetric Intensity Image **11-159**

Image Quality Metrics **11-162**
Full-Reference Quality Metrics **11-162**
No-Reference Quality Metrics **11-163**

Train and Use a No-Reference Quality Assessment Model **11-164**
NIQE Workflow **11-164**
BRISQUE Workflow **11-167**

Obtain Local Structural Similarity Index **11-169**

Compare Image Quality at Various Compression Levels . . **11-172**

Anatomy of an eSFR Chart **11-174**
Slanted Edge Features **11-174**
Gray Patch Features **11-176**
Color Patch Features **11-177**
Registration Markers **11-178**

Evaluate Quality Metrics on eSFR Test Chart **11-179**

Correct Colors Using Color Correction Matrix **11-194**

Install Sample Data Using the Add-Ons Explorer **11-204**

12 ROI-Based Processing

Create a Binary Mask **12-2**
Create a Binary Mask from a Grayscale Image **12-2**
Create Binary Mask Using an ROI Object **12-2**
Create Binary Mask Based on Color Values **12-3**
Create Binary Mask Without an Associated Image **12-3**

ROI Creation Overview **12-5**
ROI Creation Classes **12-7**
ROI Creation Functions **12-7**

Overview of ROI Filtering **12-9**

Sharpen Region of Interest in an Image **12-10**

Apply Custom Filter to Region of Interest in Image **12-13**

Fill Region of Interest in an Image **12-16**

Image Region Properties 12-20

Calculate Region Properties Using Image Region Analyzer 12-21

Filter Images on Region Properties Using Image Region Analyzer App 12-29

Create Image Comparison Tool Using ROIs 12-36

Use Freehand ROIs to Refine Segmentation Masks 12-44

Rotate Rectangle ROI to interact with imrotate 12-50

Subsample or Simplify a Freehand ROI 12-56

Measure Distances in an Image 12-67

Use Polyline to Create An Angle Measurement Tool 12-75

Create Freehand ROI Editing Tool 12-78

13 Image Segmentation

Texture Segmentation Using Gabor Filters 13-2

Texture Segmentation Using Texture Filters 13-9

Color-Based Segmentation Using the L*a*b* Color Space . . 13-27

Color-Based Segmentation Using K-Means Clustering 13-34

Marker-Controlled Watershed Segmentation 13-41

Image Segmentation Using the Color Thresholder App 13-57
Open Image in Color Thresholder 13-57
Segment Image Using the Color Selector in the Color Thresholder 13-62

Segment Image Using Color Component Controls in the Color Thresholder 13-65
Create an Image Mask Using the Color Thresholder 13-69

Acquire Live Images in the Color Thresholder App **13-76**

Image Segmentation Using Point Clouds in the Color Thresholder App **13-83**
Load Image into the Color Thresholder App 13-83
Choose a Color Space 13-87
Segment the Image Using the Color Cloud 13-89

Image Segmentation Using the Image Segmenter App **13-91**
Segmentation Tools Available in the Image Segmenter App 13-92
Tools to Refine the Segmentation in the Image Segmenter . 13-94

Open Image Segmenter App and Load Image **13-97**

Save Mask Image Created Using Image Segmenter **13-101**

The Image Segmenter Workflow **13-103**
Segment Using Threshold Technique in Image Segmenter 13-103
Segment By Drawing Regions in Image Segmenter 13-105
Use Active Contours to Refine Segmentation in Image Segmenter 13-107
Use Morphology to Refine Mask in Image Segmenter 13-111

Load Existing Binary Mask into Image Segmenter **13-114**

Segmentation Using Graph Cut in Image Segmenter **13-115**

Segmentation Using Local Graph Cut (Grabcut) in Image Segmenter **13-122**

Segmentation Using Find Circles in Image Segmenter . . . **13-129**

Segmentation Using Auto Cluster in Image Segmenter . . **13-136**

Plot Land Classification with Color Features and Superpixels **13-141**

Segment Lungs from 3-D Chest Scan and Calculate Lung Volume **13-146**
Prepare the Data **13-146**
Step 1: Segment the Lungs **13-147**
Step 2: Compute the Volume of the Segmented Lungs . . . **13-155**

14 Image Deblurring

Image Deblurring **14-2**
Deblurring Functions **14-4**

Deblur with the Wiener Filter **14-6**
Refining the Result **14-6**

Deblurring Images Using a Wiener Filter **14-7**

Deblur with a Regularized Filter **14-15**
Refining the Result **14-16**

Deblurring Images Using a Regularized Filter **14-18**

Deblur with the Lucy-Richardson Algorithm **14-30**
Overview **14-30**
Reducing the Effect of Noise Amplification **14-30**
Accounting for Nonuniform Image Quality **14-31**
Handling Camera Read-Out Noise **14-31**
Handling Undersampled Images **14-31**
Example: Using the deconvlucy Function to Deblur an Image **14-32**
Refining the Result **14-34**

Deblurring Images Using the Lucy-Richardson Algorithm . **14-35**

Deblur with the Blind Deconvolution Algorithm **14-50**
Deblur images using blind deconvolution **14-50**
Refining the Result **14-58**

Deblurring Images Using the Blind Deconvolution Algorithm **14-60**

Create Your Own Deblurring Functions **14-72**

Avoid Ringing in Deblurred Images **14-73**

15 Color

Display Colors **15-2**

Reduce the Number of Colors in an Image **15-4**
Reduce Colors of Truecolor Image Using Color Approximation **15-4**
Reduce Colors of Indexed Image Using imapprox **15-9**
Reduce Colors Using Dithering **15-10**

Profile-Based Color Space Conversions **15-13**
Read ICC Profiles **15-13**
Write ICC Profile Information to a File **15-14**
Convert RGB to CMYK Using ICC Profiles **15-14**
What is Rendering Intent in Profile-Based Conversions? . . . **15-16**

Device-Independent Color Spaces **15-17**
Convert Between Device-Independent Color Spaces **15-17**
Color Space Data Encodings **15-18**

Understanding Color Spaces and Color Space Conversion . **15-20**

Convert from HSV to RGB Color Space **15-21**
Convert RGB image to HSV and Explore HSV Color Space . **15-22**

Convert from YIQ to RGB Color Space **15-26**

Convert from YCbCr to RGB Color Space **15-27**

Determine If L*a*b* Value Is in RGB Gamut **15-28**

Comparison of Auto White Balance Algorithms **15-29**

16 Neighborhood and Block Operations

Neighborhood or Block Processing: An Overview **16-2**

Sliding Neighborhood Operations **16-3**
Determine the Center Pixel **16-4**
General Algorithm of Sliding Neighborhood Operations **16-4**
Border Padding Behavior in Sliding Neighborhood Operations **16-4**
Implementing Linear and Nonlinear Filtering as Sliding Neighborhood Operations **16-5**

Distinct Block Processing **16-7**
Implement Block Processing Using the blockproc Function **16-7**
Apply Padding **16-9**

Block Size and Performance **16-11**
TIFF Image Characteristics **16-11**
Choosing Block Size **16-11**

Parallel Block Processing on Large Image Files **16-15**
What is Parallel Block Processing? **16-15**
When to Use Parallel Block Processing **16-15**
How to Use Parallel Block Processing **16-15**

Perform Block Processing on Image Files in Unsupported Formats **16-17**
Learning More About the LAN File Format **16-17**
Parsing the Header **16-18**
Reading the File **16-19**
Examining the LanAdapter Class **16-19**
Using the LanAdapter Class with blockproc **16-24**

Use Column-wise Processing to Speed Up Sliding Neighborhood or Distinct Block Operations **16-25**
Using Column Processing with Sliding Neighborhood Operations **16-25**
Using Column Processing with Distinct Block Operations . . **16-26**

Block Processing Large Images **16-29**

Computing Statistics for Large Images **16-36**

17 Deep Learning

Train and Apply Denoising Neural Networks **17-2**
Denoise Images Using Pretrained Network **17-2**
Train a Denoising Network Using Predefined Layers **17-3**
Train Fully Customized Denoising Neural Network **17-4**

Remove Noise from Color Image Using Pretrained Neural Network **17-6**

Single Image Super-Resolution Using Deep Learning **17-13**

JPEG Image Deblocking Using Deep Learning **17-32**

Semantic Segmentation of Multispectral Images Using Deep Learning **17-50**

Image Processing Operator Approximation Using Deep Learning **17-72**

18 Code Generation for Image Processing Toolbox Functions

Code Generation for Image Processing **18-2**

List of Supported Functions with Usage Notes **18-3**

Generate Code from Application Containing Image Processing Functions **18-8**
Setup Your Compiler **18-8**
Generate Code **18-9**

Understand Code Generation with Image Processing Toolbox 18-24

19 GPU Computing with Image Processing Toolbox Functions

Image Processing on a GPU 19-2

List of Supported Functions with Limitations and Other Notes 19-4

Perform Thresholding and Morphological Operations on a GPU 19-7

Perform Element-Wise Operations on a GPU 19-11

1

Getting Started

This topic presents 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 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
- “Compilability” on page 1-4
- “Basic Image Import, Processing, and Export” on page 1-5
- “Correct Nonuniform Illumination and Analyze Foreground Objects” on page 1-11
- “Getting Help” on page 1-21
- “Acknowledgments” on page 1-23

Image Processing Toolbox Product Description

Perform image processing, analysis, and algorithm development

Image Processing Toolbox provides a comprehensive set of reference-standard algorithms and workflow apps for image processing, analysis, visualization, and algorithm development. You can perform image segmentation, image enhancement, noise reduction, geometric transformations, image registration, and 3D image processing.

Image Processing Toolbox apps let you automate common image processing workflows. You can interactively segment image data, compare image registration techniques, and batch-process large datasets. Visualization functions and apps let you explore images, 3D volumes, and videos; adjust contrast; create histograms; and manipulate regions of interest (ROIs).

You can accelerate your algorithms by running them on multicore processors and GPUs. Many toolbox functions support C/C++ code generation for desktop prototyping and embedded vision system deployment.

Key Features

- Image analysis, including segmentation, morphology, statistics, and measurement
- Apps for image region analysis, image batch processing, and image registration
- 3D image processing workflows, including visualization and segmentation
- Image enhancement, filtering, geometric transformations, and deblurring algorithms
- Intensity-based and non-rigid image registration methods
- Support for CUDA enabled NVIDIA GPUs (with Parallel Computing Toolbox™)
- C-code generation support for desktop prototyping and embedded vision system deployment

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`).

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

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 I
```

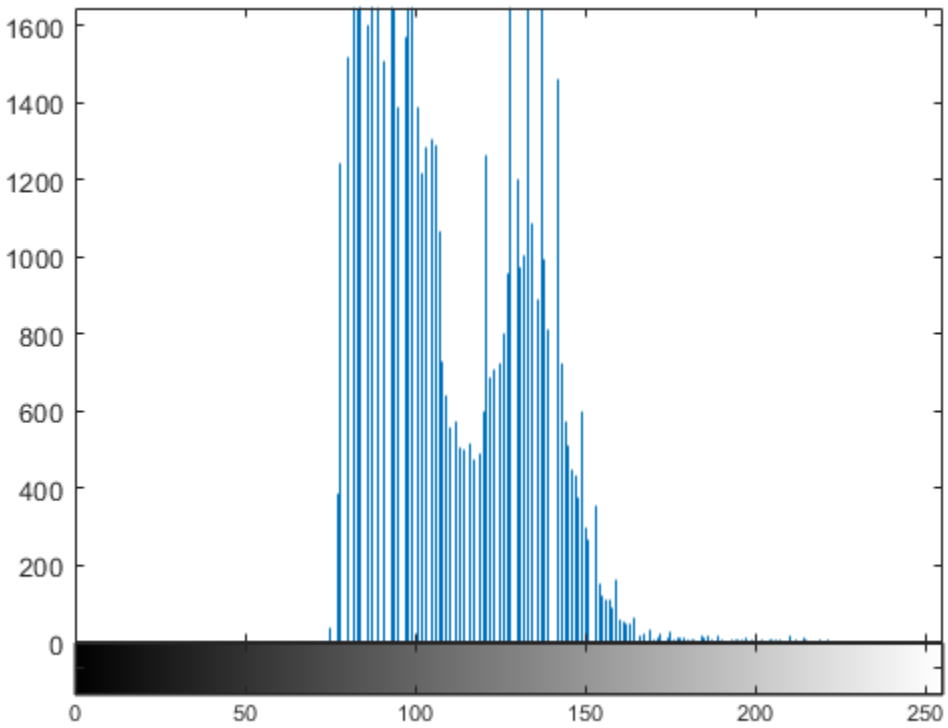
```
  Name        Size              Bytes  Class    Attributes

  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)
```



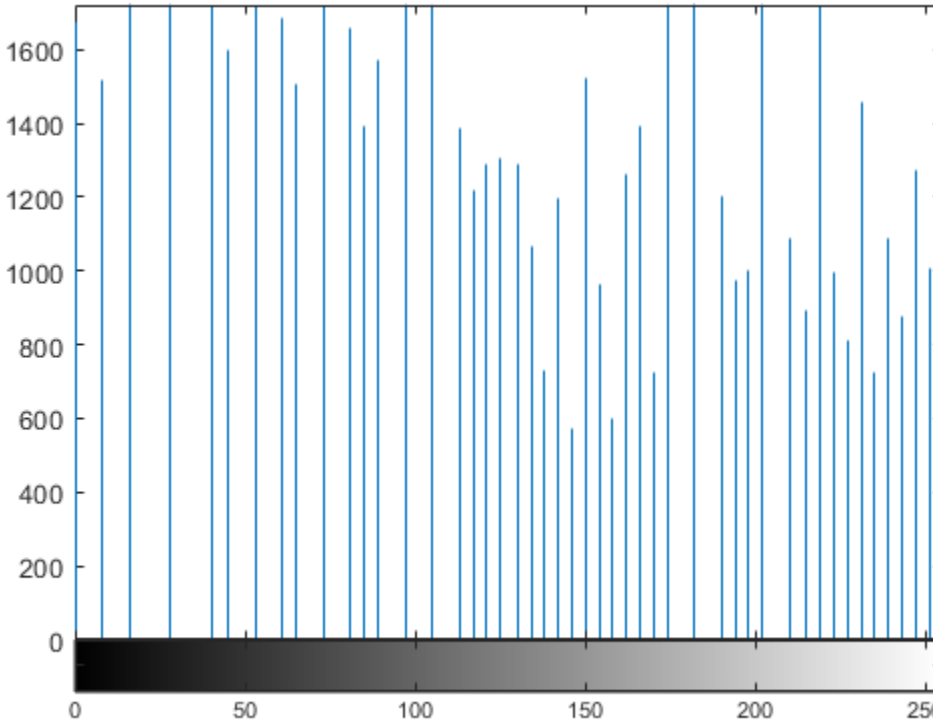
Improve the contrast in an image, using the `histeq` function. Histogram equalization spreads the intensity values over the full range of the image. Display the image. (The toolbox includes several other functions that perform contrast adjustment, including

`imadjust` and `adapthisteq`, 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 = struct with fields:
                  Filename: 'C:\TEMP\Bdoc18b_943130_7372\ib632619\30\tpe2e744f9\images
                FileModDate: '27-Aug-2018 16:11:47'
                   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: '27 Aug 2018 20:11:47 +0000'
                      Title: []
                     Author: []
                  Description: []
                  Copyright: []
               CreationTime: []
                   Software: []
                 Disclaimer: []
                    Warning: []
                     Source: []
                    Comment: []
                  OtherText: []
```

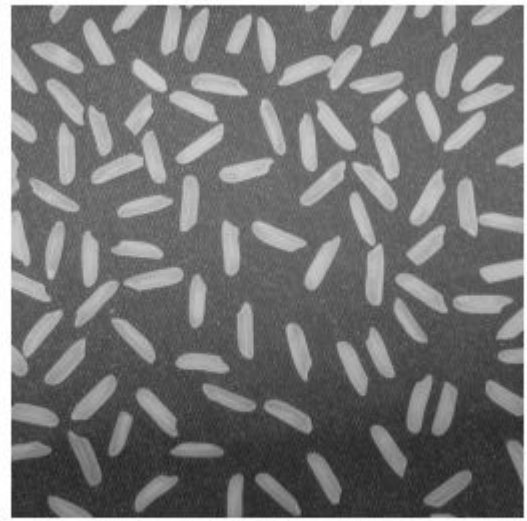
Correct Nonuniform Illumination and Analyze Foreground Objects

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 to make it easy to identify foreground objects (individual grains of rice). You can then analyze the objects, such as finding the area of each grain of rice, and you can compute statistics for all objects in the image.

Preprocess the Image

Read an image into the workspace.

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



The background illumination is brighter in the center of the image than at the bottom. Preprocess the image to make the background illumination more uniform.

As a first step, remove all the foreground (rice grains) using morphological opening. The opening operation removes small objects that cannot completely contain the structuring element. Define a disk-shaped structuring element with a radius of 15, which does fit entirely inside a single grain of rice.

```
se = strel('disk',15)
```

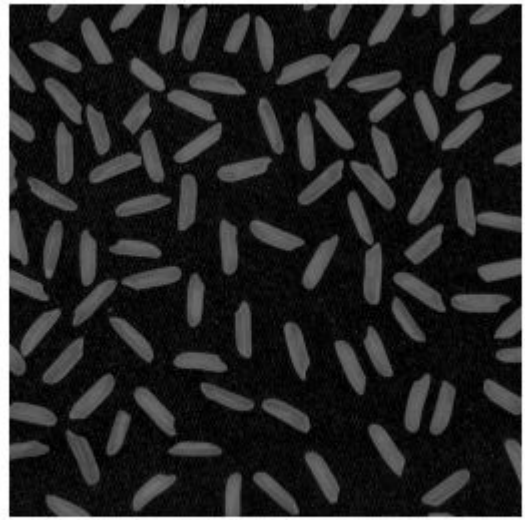
```
se =
strel is a disk shaped structuring element with properties:

      Neighborhood: [29x29 logical]
    Dimensionality: 2
```

To perform the morphological opening, use `imopen` with the structuring element.

```
background = imopen(I,se);
imshow(background)
```

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)
```

Note that the prior two steps could be replaced by a single step using `imtophat` which first calculates the morphological opening and then subtracts it from the original image.

```
I2 = imtophat(I,strel('disk',15));
```

Create a binary version of the processed image so you can use toolbox functions for analysis. Use the `imbinarize` function to convert the grayscale image into a binary image. Remove background noise from the image with the `bwareaopen` function.

```
bw = imbinarize(I3);
bw = bwareaopen(bw,50);
imshow(bw)
```

Identify 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)
```

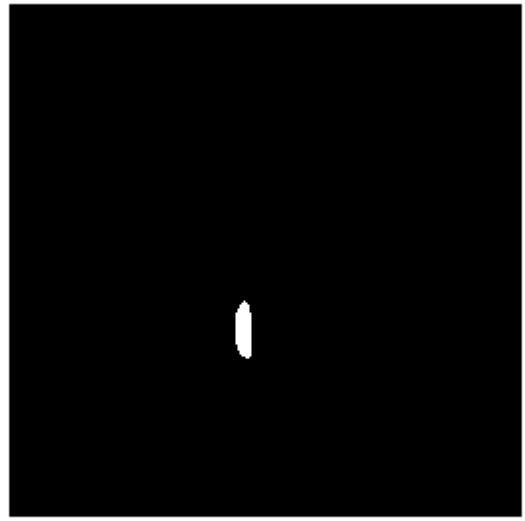
```
cc = struct with fields:
    Connectivity: 4
       ImageSize: [256 256]
      NumObjects: 95
    PixelIdxList: {1x95 cell}
```

```
cc.NumObjects
```

```
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 by creating a label matrix and then displaying it as a pseudocolor indexed image.

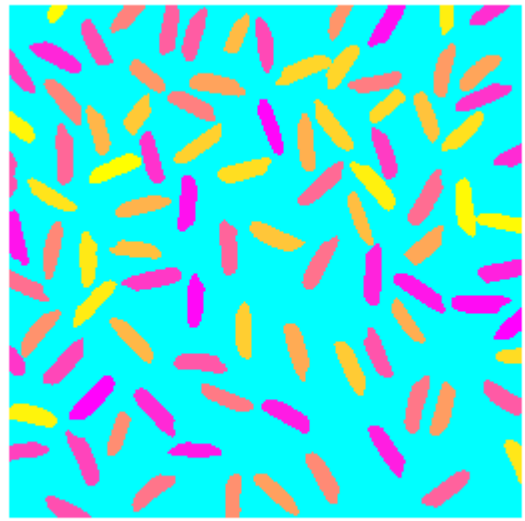
Use `labelmatrix` to create a label matrix from the output of `bwconncomp`. Note that `labelmatrix` stores the label matrix in the smallest numeric class necessary for the number of objects.

```
labeled = labelmatrix(cc);
whos labeled
```

```
  Name           Size              Bytes  Class    Attributes

  labeled      256x256             65536  uint8
```

Use `label2rgb` to choose the colormap, the background color, and how objects in the label matrix map to colors in the colormap. In the pseudocolor image, the label identifying each object in the label matrix maps to a different color in an associated colormap matrix.

```
RGB_label = label2rgb(labeled,'spring','c','shuffle');
imshow(RGB_label)
```



Compute Area-Based Statistics

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
```

Create a new vector `grain_areas`, which holds the area measurement for each grain.

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

Find the area of the 50th component.

```
grain_areas(50)
```

```
ans = 194
```

Find and display the grain with the smallest area.

```
[min_area, idx] = min(grain_areas)
```

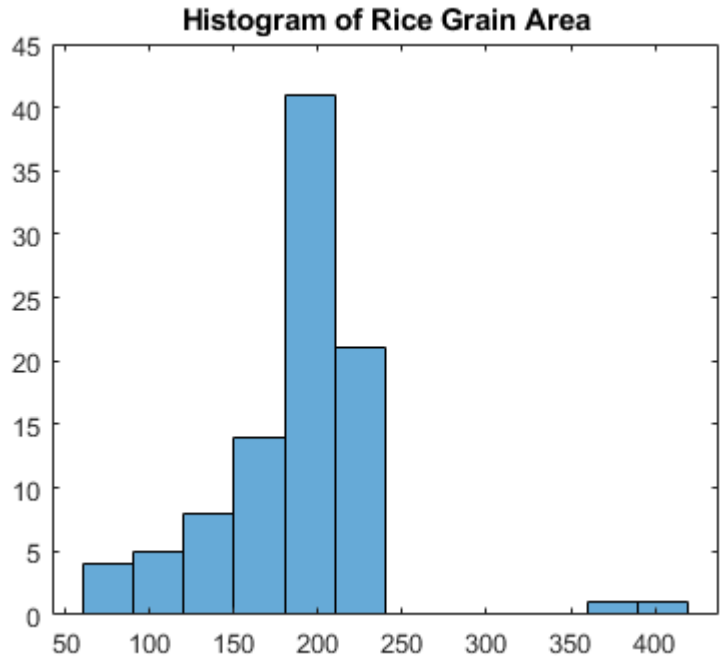
```
min_area = 61
```

```
idx = 16
```

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

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

```
histogram(grain_areas)
title('Histogram of Rice Grain Area')
```



See Also

bwareaopen | bwconncomp | imadjust | imbinarize | imopen | imread | imshow | label2rgb | labelmatrix | regionprops

Getting Help

In this section...

"Product Documentation" on page 1-21
"Image Processing Examples" on page 1-21
"MATLAB Newsgroup" on page 1-22

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., PhD., Johns Hopkins University School of Medicine.
`trees`	*Trees with a View,* 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
- “Image Coordinate Systems” on page 2-3
- “Define World Coordinates Using XData and YData Properties” on page 2-7
- “Define World Coordinates Using Spatial Referencing” on page 2-10
- “Image Types in the Toolbox” on page 2-13
- “Binary Images” on page 2-15
- “Indexed Images” on page 2-16
- “Grayscale Images” on page 2-18
- “Truecolor Images” on page 2-20
- “Display Separated Color Channels of an RGB Image” on page 2-23
- “Convert Between Image Types” on page 2-26
- “Convert Image Data Between Classes” on page 2-28
- “Process Multiframe Image Arrays” on page 2-30
- “Perform an Operation on a Sequence of Images” on page 2-31
- “Detecting Cars in a Video of Traffic” on page 2-34
- “Batch Processing Using the Image Batch Processor App” on page 2-42
- “Process Large Set of Images Using MapReduce Framework and Hadoop” on page 2-57
- “What Is an Image Sequence?” on page 2-72
- “Toolbox Functions That Work with Image Sequences” on page 2-73
- “Image Arithmetic Functions” on page 2-76
- “Image Arithmetic Saturation Rules” on page 2-77
- “Nest Calls to Image Arithmetic Functions” on page 2-78
- “Find Vegetation in a Multispectral Image” on page 2-79

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 matrices, in which each element of the matrix corresponds to a single discrete *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, represent images using a three-dimensional array. In truecolor images, 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 numeric data, and makes the full power of MATLAB available for image processing applications.

For more information on how Image Processing Toolbox assigns pixel indices and how to relate pixel indices to continuous spatial coordinates, see “Image Coordinate Systems” on page 2-3.

See Also

`imread` | `imshow`

Related Examples

- “Basic Image Import, Processing, and Export” on page 1-5

More About

- “Image Types in the Toolbox” on page 2-13

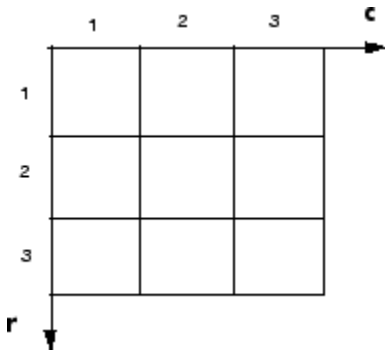
Image Coordinate Systems

As described in “Images in MATLAB” on page 2-2, 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. To access locations in images, the Image Processing Toolbox uses several different image coordinate systems as conventions for representing images as arrays.

- “Pixel Indices” on page 2-3—Because images are arrays, you can use standard MATLAB indexing.
- “Spatial Coordinates” on page 2-4—You can consider locations in images as positions on a plane using Cartesian coordinates.

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

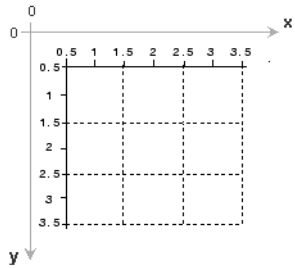
Another method for expressing locations in an image is to use a system of continuously varying coordinates rather than discrete indices. This lets you consider an image as covering a square patch, for example. In a *spatial coordinate system* like this, locations in an image are positions on a plane, 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).

The Image Processing Toolbox defines two types of spatial coordinate systems

- “Intrinsic Coordinates” on page 2-4—A spatial coordinate system that corresponds to pixel indices
- “World Coordinates” on page 2-5—A spatial coordinate system that relates the image to some other coordinate space

Intrinsic Coordinates

By default, the toolbox uses a spatial coordinate system for an image that corresponds to the image’s pixel indices. It is 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 (`numCols` + 0.5, `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 (`numRows`, `numCols`). The center of the upper left pixel is (1.0, 1.0) and the center of the lower right pixel is (`numCols`, `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.

There are several ways to define a world coordinate system:

- “Define World Coordinates Using XData and YData Properties” on page 2-7
- “Define World Coordinates Using Spatial Referencing” on page 2-10

Define World Coordinates Using XData and YData Properties

To define a world coordinate system for an image, 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 axis 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. (Another way to define a world coordinate system is to use spatial referencing—see "Define World Coordinates Using Spatial Referencing" on page 2-10.)

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 is also possible to set `XData` or `YData` such that the *x*-axis or *y*-axis is reversed. You would 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.

Several toolbox functions accept this `XData` and `YData` as arguments and return coordinates in the world coordinate system: `bwselect`, `imcrop`, `impixel`, `roipoly`, and `imtransform`.

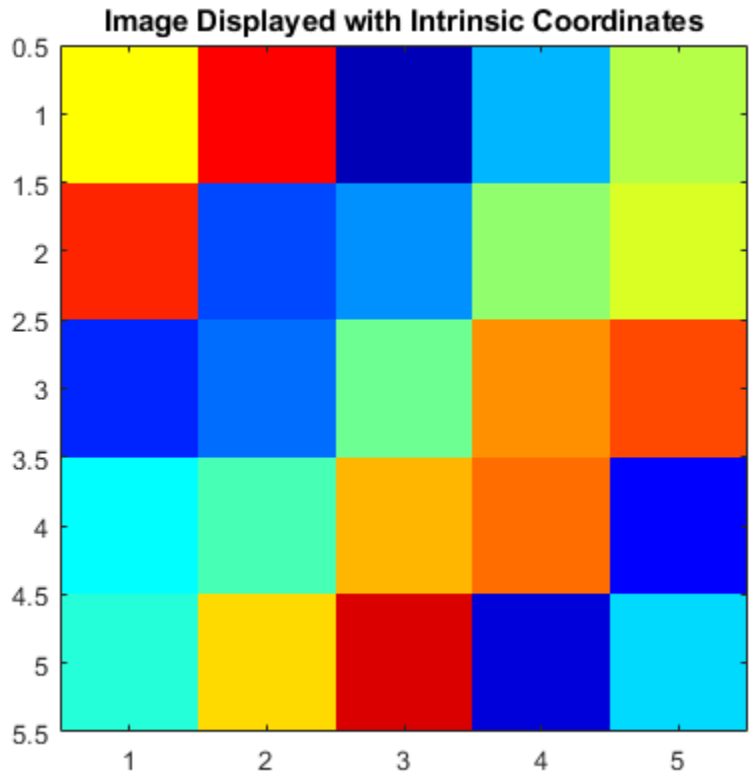
Display an Image using Nondefault Spatial Coordinates

Create an image of a 5-by-5 magic square.

```
A = magic(5);
```

Display this image with intrinsic coordinates. Here, the x- and y-coordinate ranges are both [1 5].

```
image(A)
axis image
colormap(gca,jet(25))
title('Image Displayed with Intrinsic Coordinates');
```



Specify nondefault x- and y-coordinate ranges.

```
x = [19.5 23.5];
y = [8.0 12.0];
```

Display the image with the nondefault spatial coordinates using the `XData` and `YData` image properties.

```
image(A,'XData',x,'YData',y)
axis image
colormap(gca,jet(25))
title('Image Displayed with Nondefault Coordinates');
```

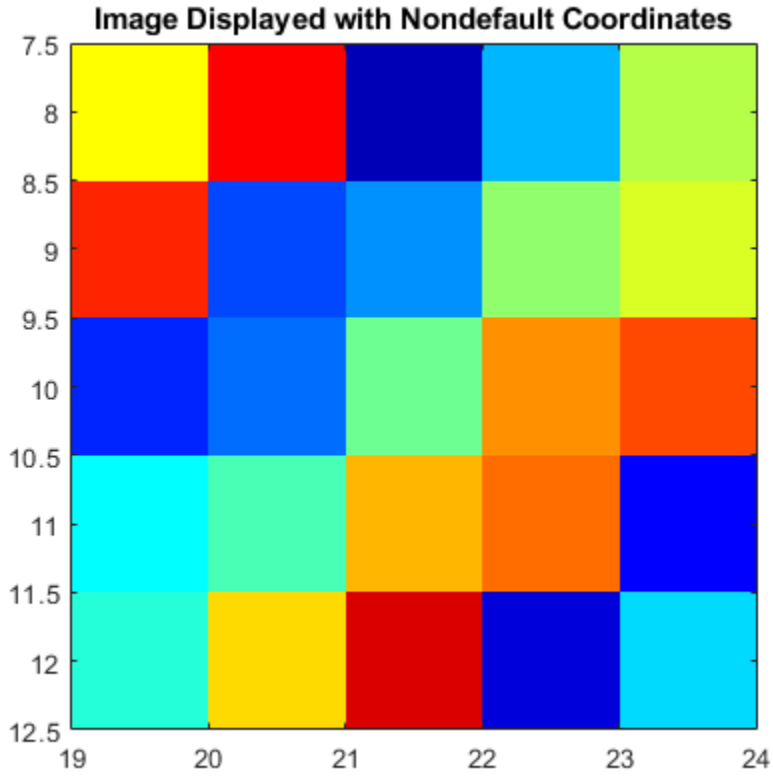


Image Displayed with Nondefault Coordinates
7.5
8
8.5
9
9.5
10
10.5
11
11.5
12
12.5
19
20
21
22
23
24

Define World Coordinates Using Spatial Referencing

To specify a world (nondefault spatial) coordinate system for an image, you can use spatial referencing. The Image Processing Toolbox uses includes two spatial referencing objects, `imref2d` and `imref3d`, that let you define the location of the image in a world coordinate system. You can also use these objects to specify the image resolution, including nonsquare pixel shapes. (Another way to define a world coordinate system is to use image object `XData` and `YData` properties—see "Define World Coordinates Using XData and YData Properties" on page 2-7.)

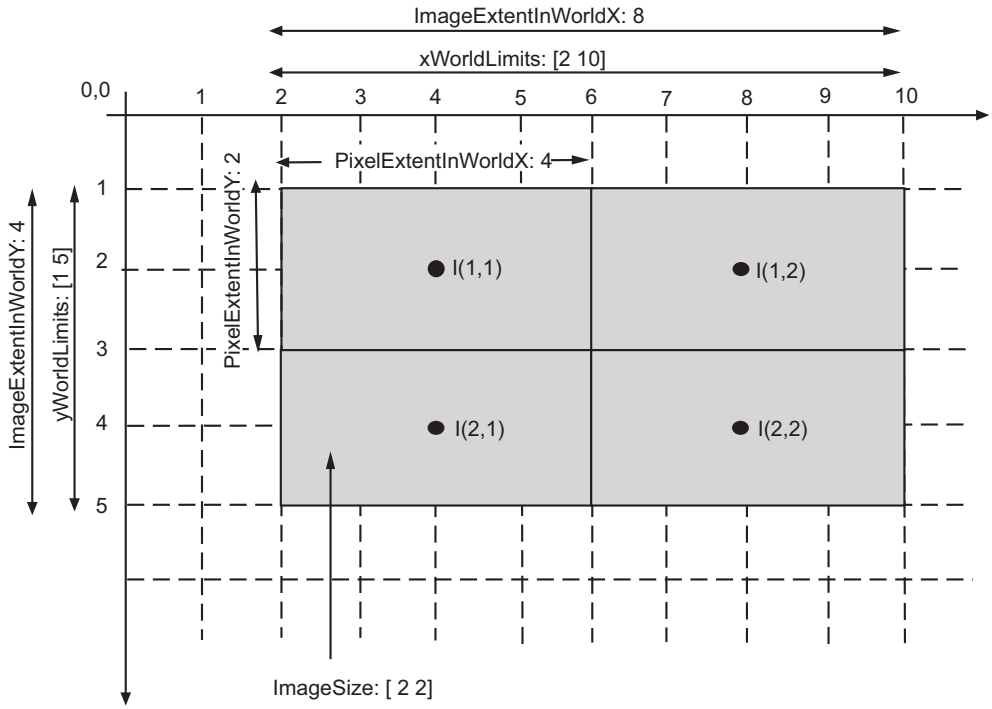
When you create a spatial referencing object, you associate it with a particular image. The object contains information about the image, some of it provided by you and some of it derived by the object. The following table provides descriptions of spatial referencing object fields.

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

To illustrate, this example creates a spatial referencing object associated with a 2-by-2 image. In this image, the world extent is 4 units/pixel in the **x** direction and 2 units/pixel in the y direction. To create a spatial referencing object, specify the image dimensions and individual pixel dimensions as arguments. The object generates the world limits of this image 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 following figure illustrates how these properties map to elements of an image.



Spatial referencing objects 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`.

Image Types in the Toolbox

The Image Processing Toolbox software defines four basic types of images, summarized in the following table. These image types determine the way MATLAB interprets array elements as pixel intensity values.

All images in Image Processing Toolboxare assumed to have real, nonsparse, numeric or logical values unless otherwise specified.

Image Type	Interpretation
“Binary Images” on page 2-15	Image data are stored as an *m*-by-*n* logical array. Array values of 0 and 1 are interpreted as black and white, respectively.
“Indexed Images” on page 2-16	Image data are stored as an *m*-by-*n* numeric matrix whose elements are direct indices into a color map. Each row of the color map specifies the red, green, and blue components of a single color. • For `single` or `double` arrays, integer values range from [1, *p*]. • For `logical`, `uint8`, or `uint16` arrays, values range from [0, *p*-1]. The colormap is a *c*-by-3 array of class `double`.
“Grayscale Images” on page 2-18 (Also known as an *intensity* image)	Image data are stored as an *m*-by-*n* numeric array whose elements specify intensity values. • For `single` or `double` arrays, values range from [0, 1]. • For `uint8` arrays, values range from [0,255]. • For `uint16`, values range from [0, 65535]. • For `int16`, values range from [-32768, 32767].

Image Type	Interpretation
“Truecolor Images” on page 2-20 (Also known as an *RGB* image)	Image data are stored as an *m*-by-*n*-by-3 numeric array whose elements specify the intensity values of one of the three color channels. For RGB images, the three channels represent the red, green, and blue signals of the image. • For `single` or `double` arrays, RGB values range from [0, 1]. • For `uint8` arrays, RGB values range from [0,255]. • For `uint16`, RGB values range from [0, 65535]. There are other models, called color spaces, that describe colors using three color channels. For these color spaces, the range of each data type may differ from the range allowed by images in the RGB color space. For example, pixel values in the L*a*b* color space of data type `double` can be negative or greater than 1. For more information, see “Understanding Color Spaces and Color Space Conversion” on page 15-20.

See Also

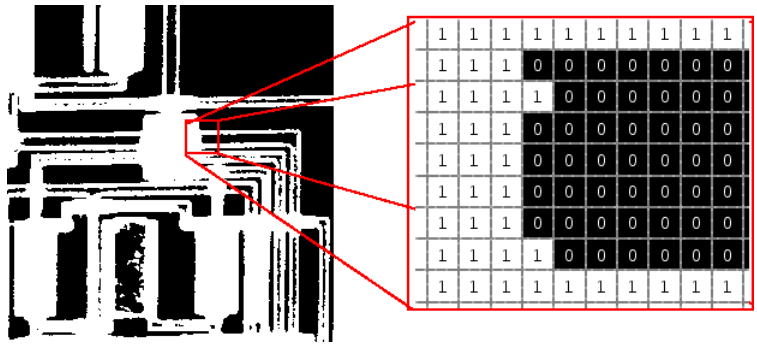
More About

- “Convert Between Image Types” on page 2-26
- “Understanding Color Spaces and Color Space Conversion” on page 15-20
- “Work with High Dynamic Range Images” on page 3-24
- “What Is an Image Sequence?” on page 2-72

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

See Also

More About

- “Image Types in the Toolbox” on page 2-13

Indexed Images

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

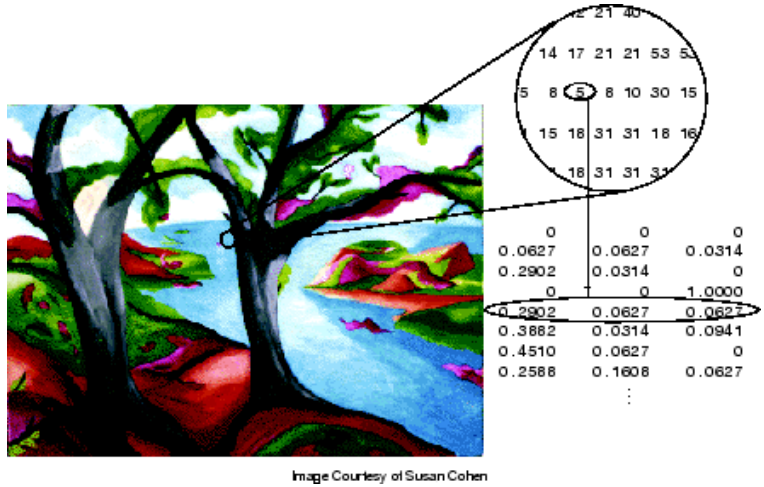
The colormap matrix is an *m*-by-3 array of class `double` containing floating-point values in the range [0,1]. Each row of `map` specifies the red, green, and blue components of a single color.

The pixel values in the array are direct indices into a colormap. The color of each image pixel is determined by using the corresponding value of `X` as an index into `map`. 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`, the colormap normally contains integer values in the range [1, *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 colormap normally contains integer values in the range [0, *p–1*]. The value 0 points to the first row in the colormap, the value 1 points to the second row, and so on.

A colormap is often stored with an indexed image and is automatically loaded with the image when you use the `imread` function. After you read the image and the colormap into the 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 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.



Pixel Values Index to Colormap Entries in an Indexed Image

See Also

More About

- “Image Types in the Toolbox” on page 2-13

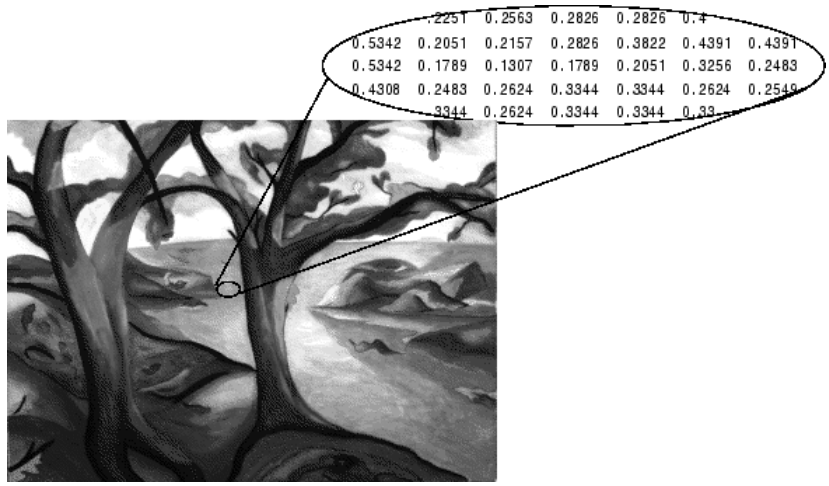
Grayscale Images

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

See Also

More About

- “Image Types in the Toolbox” on page 2-13

Truecolor Images

A truecolor image is an image in which each pixel is specified by three values — one each for the red, blue, and green components of the pixel's color. MATLAB store truecolor images as an *m*-by-*n*-by-3 data array that defines red, green, and blue color components for each individual pixel. Truecolor images do not use a colormap. The color of each pixel is determined by the combination of the red, green, and blue intensities stored in each color plane at the pixel's location.

Graphics file formats store truecolor images as 24-bit images, where the red, green, and blue components are 8 bits each. This yields a potential of 16 million colors. The precision with which a real-life image can be replicated has led to the commonly used term truecolor image.

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

The following figure depicts a truecolor image of class `double`.

The Color Planes of a Truecolor Image

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

```
0.5176 0.1608 0.0627
```

See Also

More About

- “Image Types in the Toolbox” on page 2-13
- “Display Separated Color Channels of an RGB Image” on page 2-23

Display Separated Color Channels of an RGB Image

This example creates a simple truecolor image and then separates the color channels (planes). The example displays each color channel and the original image.

Create an RGB image with uninterrupted areas of red, green, and blue. The dimensions of this image are 200-by-200 pixels. Display the image.

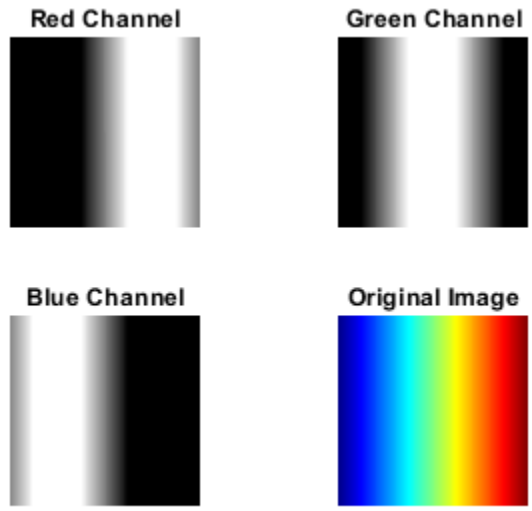
```
RGB=reshape(ones(200,1)*reshape(jet(200),1,600),[200,200,3]);
imshow(RGB)
title('Original RGB Image')
```

Separate the three color channels.

```
[R,G,B] = imsplit(RGB);
```

Display each color channel separately, along with the original 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 Channel 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`.

```
subplot(2,2,1)
imshow(R)
title('Red Channel')

subplot(2,2,2)
imshow(G)
title('Green Channel')

subplot(2,2,3)
imshow(B)
title('Blue Channel')

subplot(2,2,4)
imshow(RGB)
title('Original Image')
```



Create an all-black channel.

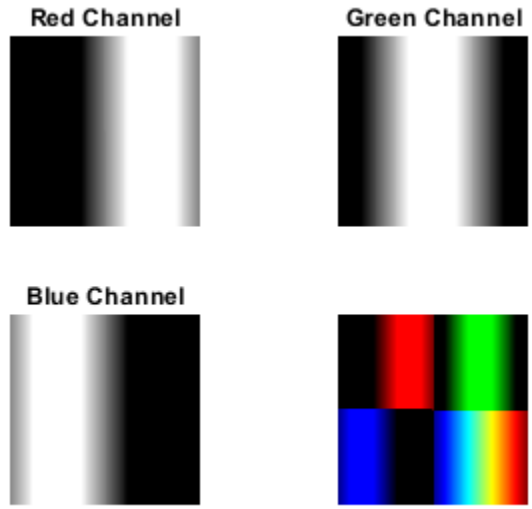
```
allBlack = zeros(size(RGB, 1), size(RGB, 2), class(RGB));
```

Create color versions of the individual color channels.

```
justRed = cat(3, R, allBlack, allBlack);
justGreen = cat(3, allBlack, G, allBlack);
justBlue = cat(3, allBlack, allBlack, B);
```

Display all the channels and the original image in a montage.

```
montage({justRed, justGreen, justBlue, RGB}, 'ThumbnailSize', []);
```



See Also

`imsplit`

More About

- “Truecolor Images” on page 2-20
- “Image Types in the Toolbox” on page 2-13

Convert 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
`demosaic`	Convert Bayer pattern encoded image to truecolor (RGB) image.
`dither`	Use dithering to convert a grayscale image to a binary image or to convert a truecolor image to an indexed image.
`gray2ind`	Convert a grayscale image to an indexed image.
`grayslice`	Convert a grayscale image to an indexed image by using multilevel thresholding.
`ind2gray`	Convert an indexed image to a grayscale image.
`ind2rgb`	Convert an indexed image to a truecolor image.

Function	Description
`mat2gray`	Convert a data matrix to a grayscale image, by scaling the data.
`rgb2gray`	Convert a truecolor image to a grayscale image. 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 “Understanding Color Spaces and Color Space Conversion” on page 15-20.
`rgb2ind`	Convert a truecolor image to an indexed image.

Convert Image Data Between Classes

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 “Reduce Colors of Indexed Image Using imapprox” on page 15-9 for more information.

Process Multiframe Image Arrays

The toolbox includes two functions, `immovie` and `montage`, that work with a specific type of multidimensional array called a multiframe 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 multiframe 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 multiframe 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 multiframe array. For example, the following stores a group of images 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_1m4_*.tif'));
fileNames = {dirOutput.name}'
numFrames = numel(fileNames)
```

```
fileNames =

  10x1 cell array

    {'AT3_1m4_01.tif'}
    {'AT3_1m4_02.tif'}
    {'AT3_1m4_03.tif'}
    {'AT3_1m4_04.tif'}
    {'AT3_1m4_05.tif'}
    {'AT3_1m4_06.tif'}
    {'AT3_1m4_07.tif'}
    {'AT3_1m4_08.tif'}
    {'AT3_1m4_09.tif'}
    {'AT3_1m4_10.tif'}


numFrames =

    10
```

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

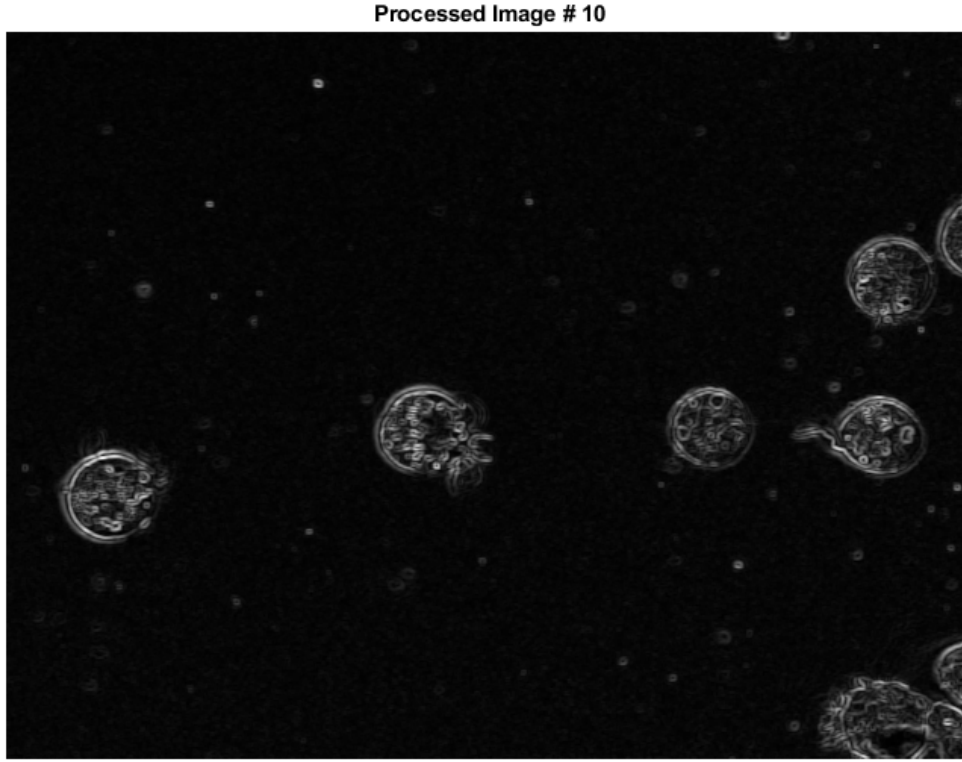
```
I = imread(fileNames{1});
sequence = zeros([size(I) numFrames],class(I));
sequence(:,:,1) = I;

for p = 2:numFrames
    sequence(:,:,p) = imread(fileNames{p});
end
```

Process each image in the sequence, performing standard deviation filtering. 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));
```

View each input image followed by its processed image.

```
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
```


Processed Image # 10

Detecting Cars in a Video of Traffic

This example shows how to use Image Processing Toolbox™ to visualize and analyze videos or image sequences. This example uses `VideoReader` (MATLAB®), `implay`, and other Image Processing Toolbox functions to detect light-colored cars in a video of traffic. Note that `VideoReader` has platform-specific capabilities and may not be able to read the supplied Motion JPEG2000 video on some platforms.

Step 1: Access Video with VideoReader

The `VideoReader` function constructs a multimedia reader object that can read video data from a multimedia file. See the documentation for `VideoReader` for information on which formats are supported on your platform.

Use `VideoReader` to access the video and get basic information about it.

```
trafficVid = VideoReader('traffic.mj2')
```

```
trafficVid = 

  VideoReader with properties:

   General Properties:
            Name: 'traffic.mj2'
            Path: 'B:\matlab\toolbox\images\imdata'
        Duration: 8
     CurrentTime: 0
             Tag: ''
        UserData: []

   Video Properties:
           Width: 160
          Height: 120
       FrameRate: 15
    BitsPerPixel: 24
     VideoFormat: 'RGB24'
```

The get method provides more information on the video such as its duration in seconds.

```
get(trafficVid)
```

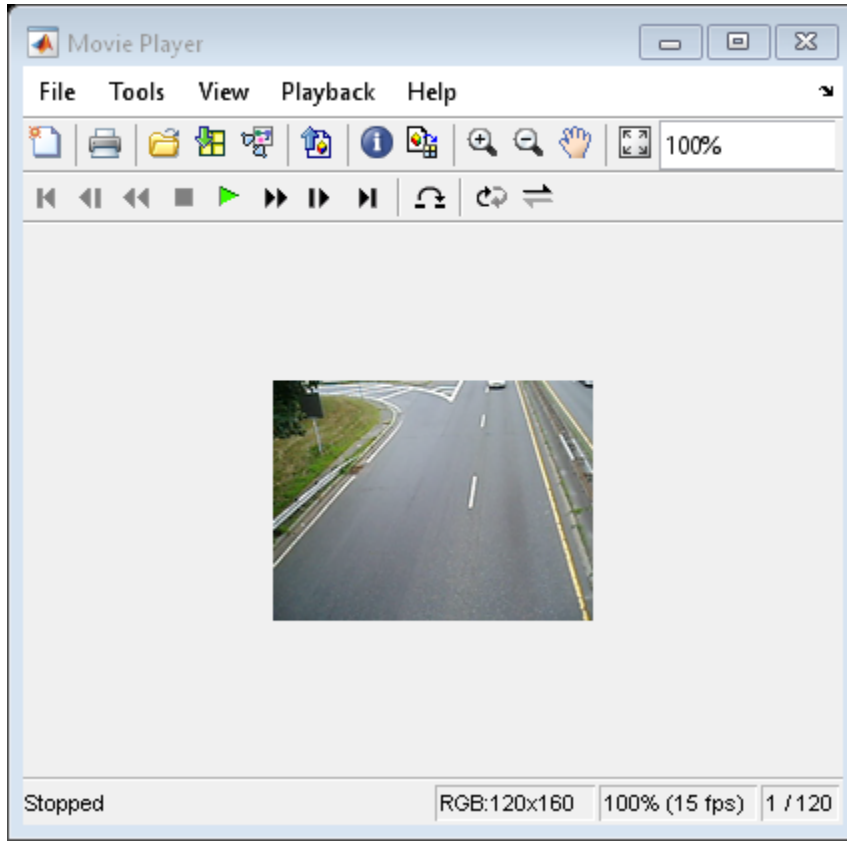
```
obj = 
```

```
  VideoReader with properties:

   General Properties:
            Name: 'traffic.mj2'
            Path: 'B:\matlab\toolbox\images\imdata'
        Duration: 8
     CurrentTime: 0
             Tag: ''
        UserData: []

   Video Properties:
           Width: 160
          Height: 120
       FrameRate: 15
    BitsPerPixel: 24
     VideoFormat: 'RGB24'
```

Step 2: Explore Video with IMPLAY

Explore the video in `implay`.

```
implay('traffic.mj2');
```



Step 3: Develop Your Algorithm

When working with video data, it can be helpful to select a representative frame from the video and develop your algorithm on that frame. Then, this algorithm can be applied to the processing of all the frames in the video.

For this car-tagging application, examine a frame that includes both light-colored and dark-colored cars. When an image has many structures, like the traffic video frames, it is useful to simplify the image as much as possible before trying to detect an object of interest. One way to do this for the car tagging application is to suppress all objects in the image that are not light-colored cars (dark-colored cars, lanes, grass, etc.). Typically, it takes a combination of techniques to remove these extraneous objects.

One way to remove the dark-colored cars from the video frames is to use the `imextendedmax` function. This function returns a binary image that identifies regions with intensity values above a specified threshold, called regional maxima. All other objects in the image with pixel values below this threshold become the background. To eliminate the dark-colored cars, determine the average pixel value for these objects in the image. (Use `rgb2gray` to convert the original video from RGB to grayscale.) You can use the pixel region tool in `implay` to view pixel values. Specify the average pixel value (or a value slightly higher) as the threshold when you call `imextendedmax`. For this example, set the value to 50.

```
darkCarValue = 50;
darkCar = rgb2gray(read(trafficVid,71));
noDarkCar = imextendedmax(darkCar, darkCarValue);
imshow(darkCar)
figure, imshow(noDarkCar)
```

In the processed image, note how most of the dark-colored car objects are removed but many other extraneous objects remain, particularly the lane-markings. The regional maxima processing will not remove the lane markings because their pixel values are above the threshold. To remove these objects, you can use the morphological function `imopen`. This function uses morphological processing to remove small objects from a binary image while preserving large objects. When using morphological processing, you must decide on the size and shape of the structuring element used in the operation. Because the lane-markings are long and thin objects, use a disk-shaped structuring element with radius corresponding to the width of the lane markings. You can use the pixel region tool in `implay` to estimate the width of these objects. For this example, set the value to 2.

```
sedisk = strel('disk',2);
noSmallStructures = imopen(noDarkCar, sedisk);
imshow(noSmallStructures)
```

To complete the algorithm, use `regionprops` to find the centroid of the objects in `noSmallStructures` (should just be the light-colored cars). Use this information to position the tag on the light-colored cars in the original video.

Step 4: Apply the Algorithm to the Video

The car-tagging application processes the video one frame at a time in a loop. (Because a typical video contains a large number of frames, it would take a lot of memory to read and process all the frames at once.)

A small video (like the one in this example) could be processed at once, and there are many functions that provide this capability. For more information, see the Documentation.

For faster processing, preallocate the memory used to store the processed video.

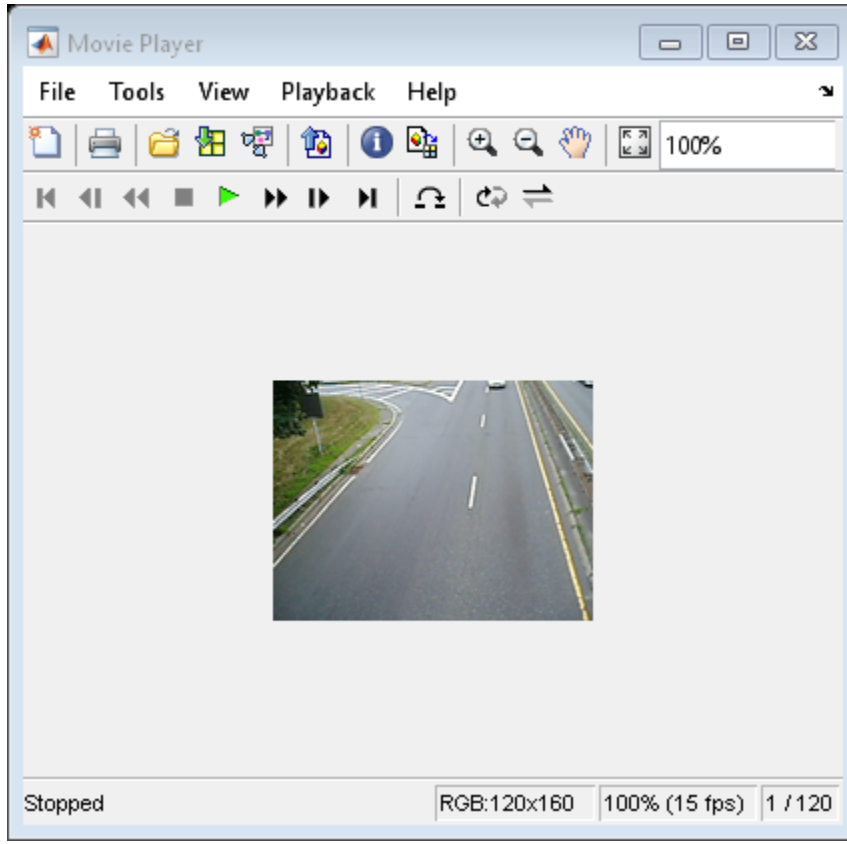
```
nframes = trafficVid.NumberOfFrames;
I = read(trafficVid, 1);
taggedCars = zeros([size(I,1) size(I,2) 3 nframes], class(I));

for k = 1 : nframes
    singleFrame = read(trafficVid, k);

    % Convert to grayscale to do morphological processing.
    I = rgb2gray(singleFrame);

    % Remove dark cars.
    noDarkCars = imextendedmax(I, darkCarValue);
```

```
    % Remove lane markings and other non-disk shaped structures.
    noSmallStructures = imopen(noDarkCars, sedisk);

    % Remove small structures.
    noSmallStructures = bwareaopen(noSmallStructures, 150);

    % Get the area and centroid of each remaining object in the frame. The
    % object with the largest area is the light-colored car.  Create a copy
    % of the original frame and tag the car by changing the centroid pixel
    % value to red.
    taggedCars(:,:,:,k) = singleFrame;

    stats = regionprops(noSmallStructures, {'Centroid','Area'});
    if ~isempty([stats.Area])
        areaArray = [stats.Area];
        [junk,idx] = max(areaArray);
        c = stats(idx).Centroid;
        c = floor(fliplr(c));
        width = 2;
        row = c(1)-width:c(1)+width;
        col = c(2)-width:c(2)+width;
        taggedCars(row,col,1,k) = 255;
        taggedCars(row,col,2,k) = 0;
        taggedCars(row,col,3,k) = 0;
    end
end
```

Step 5: Visualize Results

Get the frame rate of the original video and use it to see `taggedCars` in `implay`.

```
frameRate = trafficVid.FrameRate;
implay(taggedCars,frameRate);
```



Copyright 2007-2018 The MathWorks, Inc.

See Also

`VideoReader` | `bwareaopen` | `imextendedmax` | `imopen` | `implay` | `regionprops` | `rgb2gray`

More About

- “What Is an Image Sequence?” on page 2-72
- “Perform an Operation on a Sequence of Images” on page 2-31

Batch Processing Using the Image Batch Processor App

This example shows how to use the Image Batch Processor app to process a group of images in the same folder. The batch processing operation typically follows these steps.

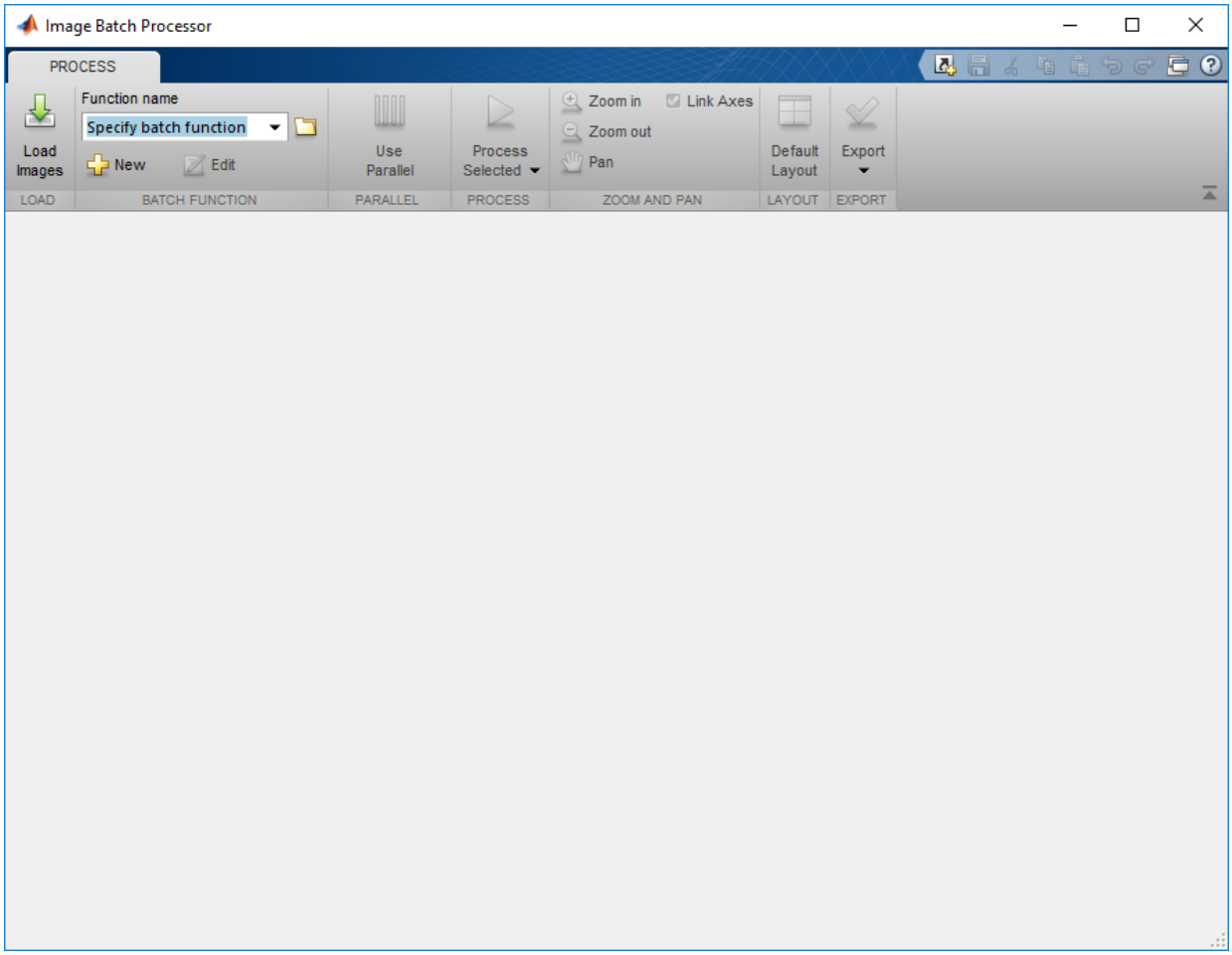
- “Open Image Batch Processor App” on page 2-42 — Select the app icon from the apps gallery, or open it from the command line.
- “Load Images into the Image Batch Processor App” on page 2-43 — Specify the name of a folder containing image files. The app can retrieve images from subfolders as well, retaining the hierarchical folder structure.
- “Specify the Batch Processing Function” on page 2-45 — Provide the app with the function you want applied to all the images in the folder. You can specify the name of an existing function or create a new function using a batch processing function template.
- “Perform the Operation on the Images” on page 2-48 — Process the group of images. You can process all files in the folder or select files to process. The app displays the results in the Results tab.
- “Obtain the Results of the Batch Processing Operation” on page 2-52 — You can save the results to a variable in the workspace or, for image results, files. You can also get the MATLAB code that the app used to generate the results.

Open Image Batch Processor App

This part of the example shows how to open the Image Batch Processor app.

From the MATLAB Toolstrip, on the Apps tab, in the Image Processing and Computer Vision group, click **Image Batch Processor** . You can also open the app at the command line using the `imageBatchProcessor` command. (If you have Parallel Computing Toolbox, the app includes the **Use Parallel** button.)

```
imageBatchProcessor
```



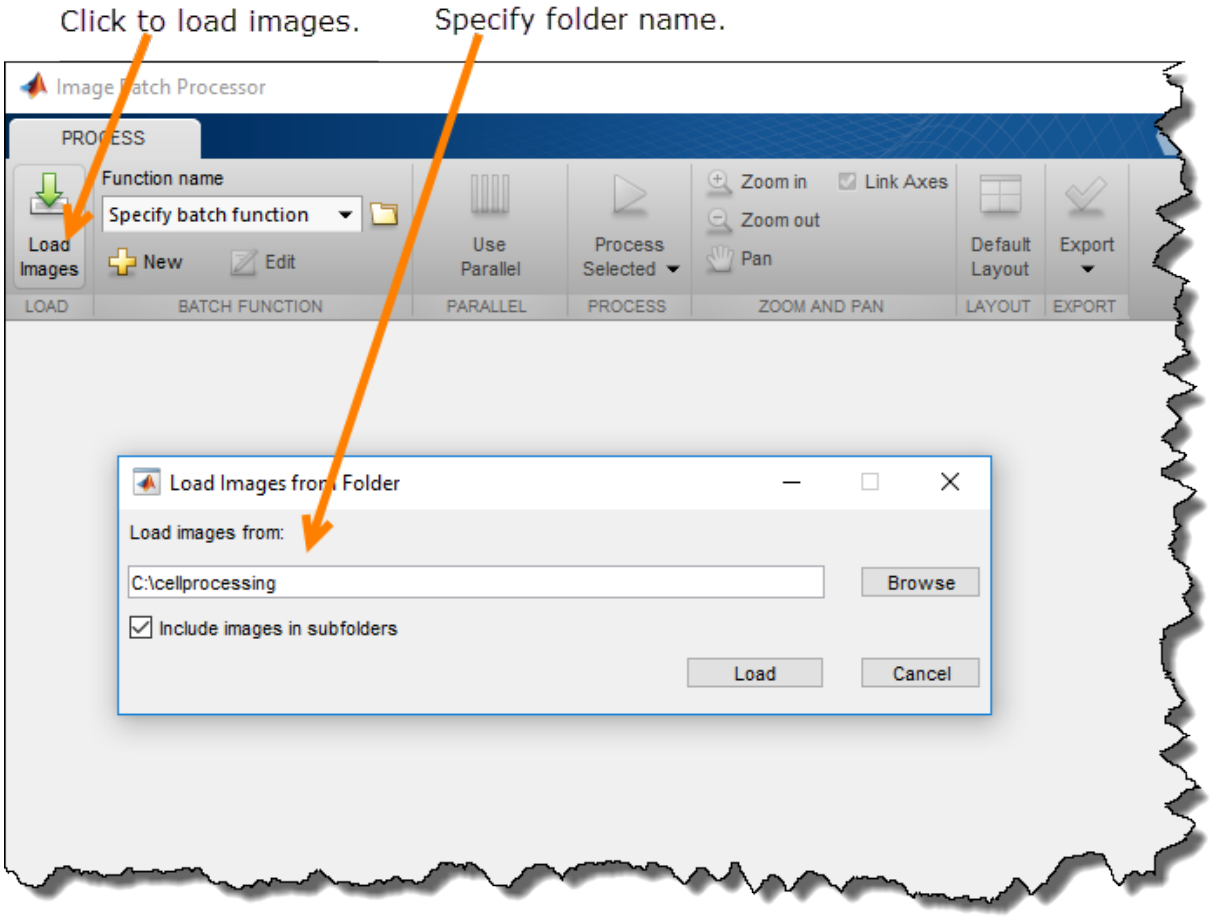
Load Images into the Image Batch Processor App

This part of the example shows how to load images into the Image Batch Processor app.

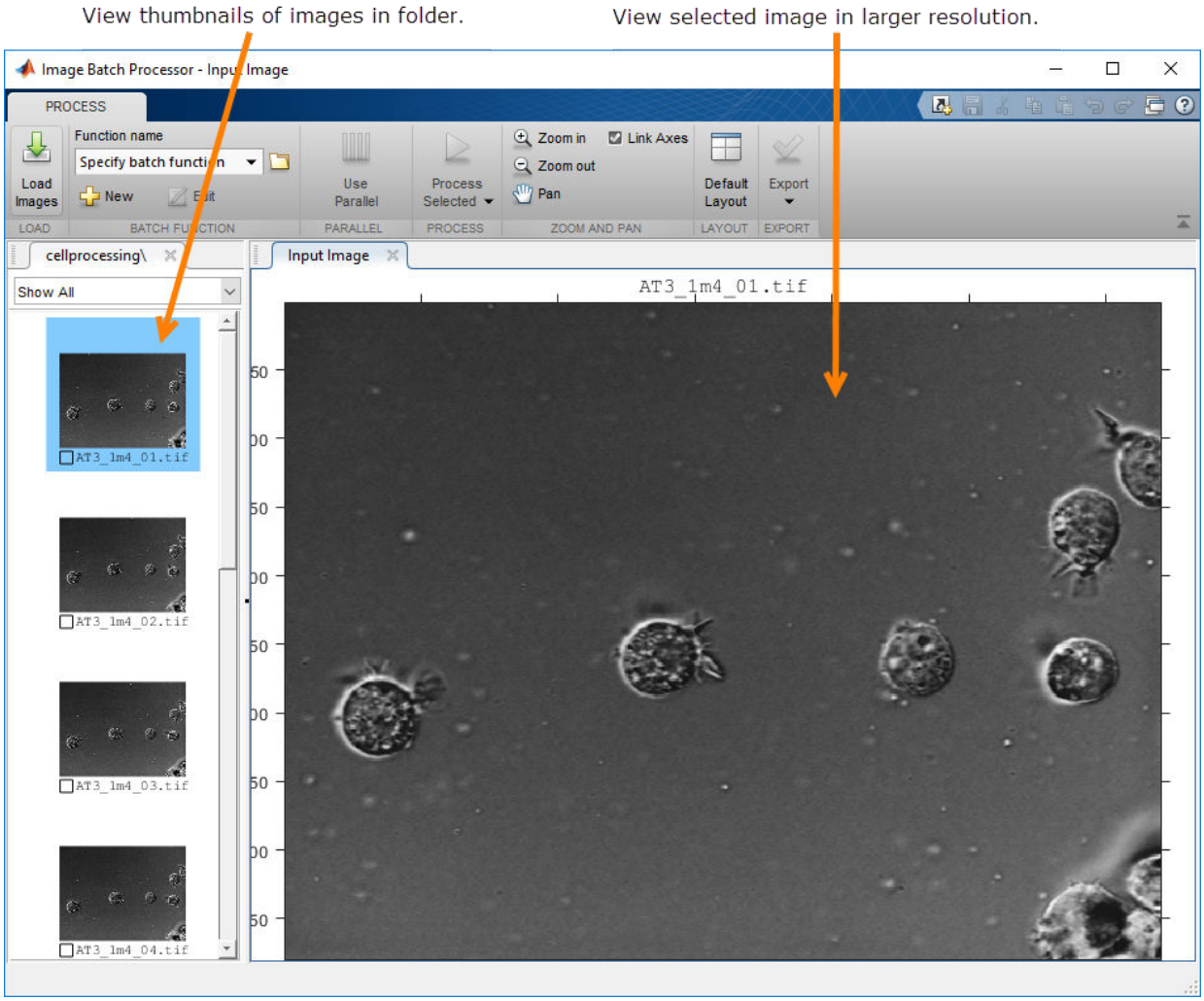
For this example, create a new folder in an area where you have write permission, and load a set of 10 images from the Image Processing Toolbox `imdata` folder.

```
mkdir('cellprocessing');
copyfile(fullfile(matlabroot,'toolbox','images','imdata','AT3*.tif'),'cellprocessing');
```

Click **Load Images** and specify the folder containing the images you want to process in the **Load Images from Folder** dialog box. For this example, specify the folder that you created in the first step, `cellprocessing`, and then click **Load**. By default, the app includes images in subfolders. To change this behavior, clear the **Include images in subfolders** check box.



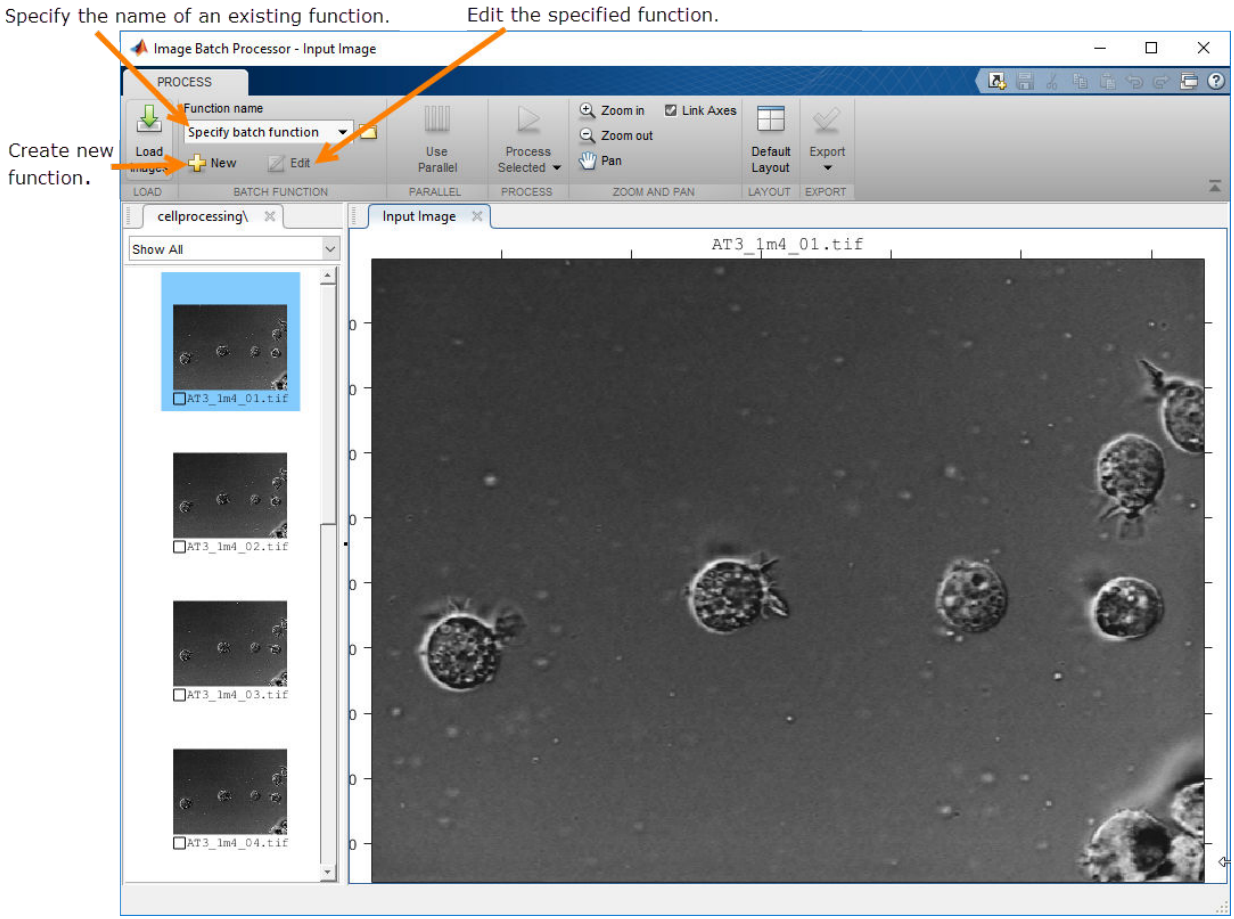
The Image Batch Processor app creates thumbnails of the images in the folder and displays them in a scrollable pane. The app displays the first selected image in larger resolution in the Input Image pane.



Specify the Batch Processing Function

This part of the example shows how to specify the function you want to execute on all the images.

Specify the name of the function you want to use to process the images in the folder. To specify an existing function, type its name in the **Function name** field, or click the folder icon to browse and select the function. To create a new batch processing function, click **New**. The app opens the batch processing function template in the MATLAB Editor. This example creates a new function-click **New**.

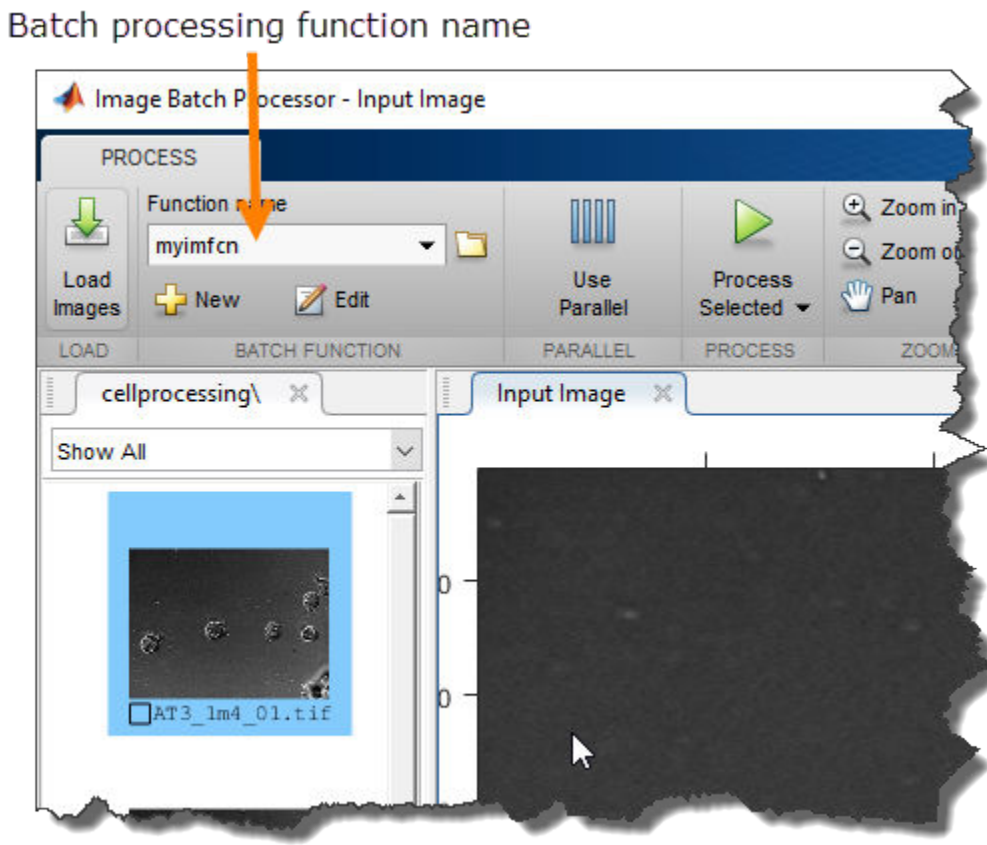


Edit the batch processing function template opened in the MATLAB Editor. Paste your code into the space reserved in the template file and click **Save**. The example uses the default function name, `myimfcn`, but you can give the function another name. The

example code creates a mask image, calculates the total number of cells in the image, and creates a thresholded version of the original image.

```
function results = myimfcn(im)
%Image Processing Function
%
% IM      - Input image.
% RESULTS - A scalar structure with the processing results.
%

%--------------------------------------------------------------------------
% Auto-generated by imageBatchProcessor App.
%
% When used by the App, this function will be called for every input image
% file automatically. IM contains the input image as a matrix. RESULTS is a
% scalar structure containing the results of this processing function.
%
%--------------------------------------------------------------------------

imstd = stdfilt(im,ones(27));
bw    = imstd>30;

thresholdMask = imfuse(im, bw);
[~, n] = bwlabel(bw);

results.bw = bw;
results.thresholdMask = thresholdMask;
results.numCells = n;
```

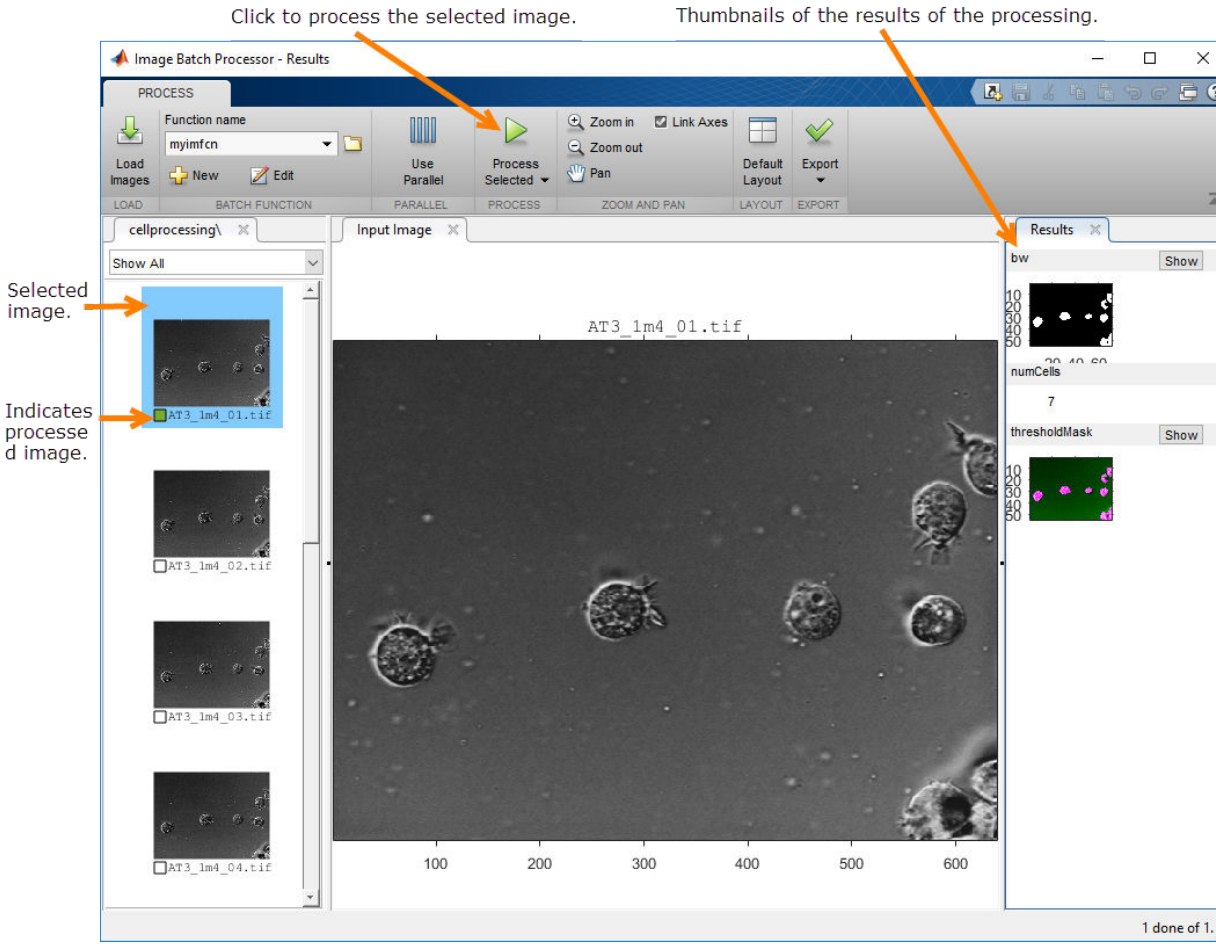
When you save the file, the app displays the name of your new function in the **Function name** field.



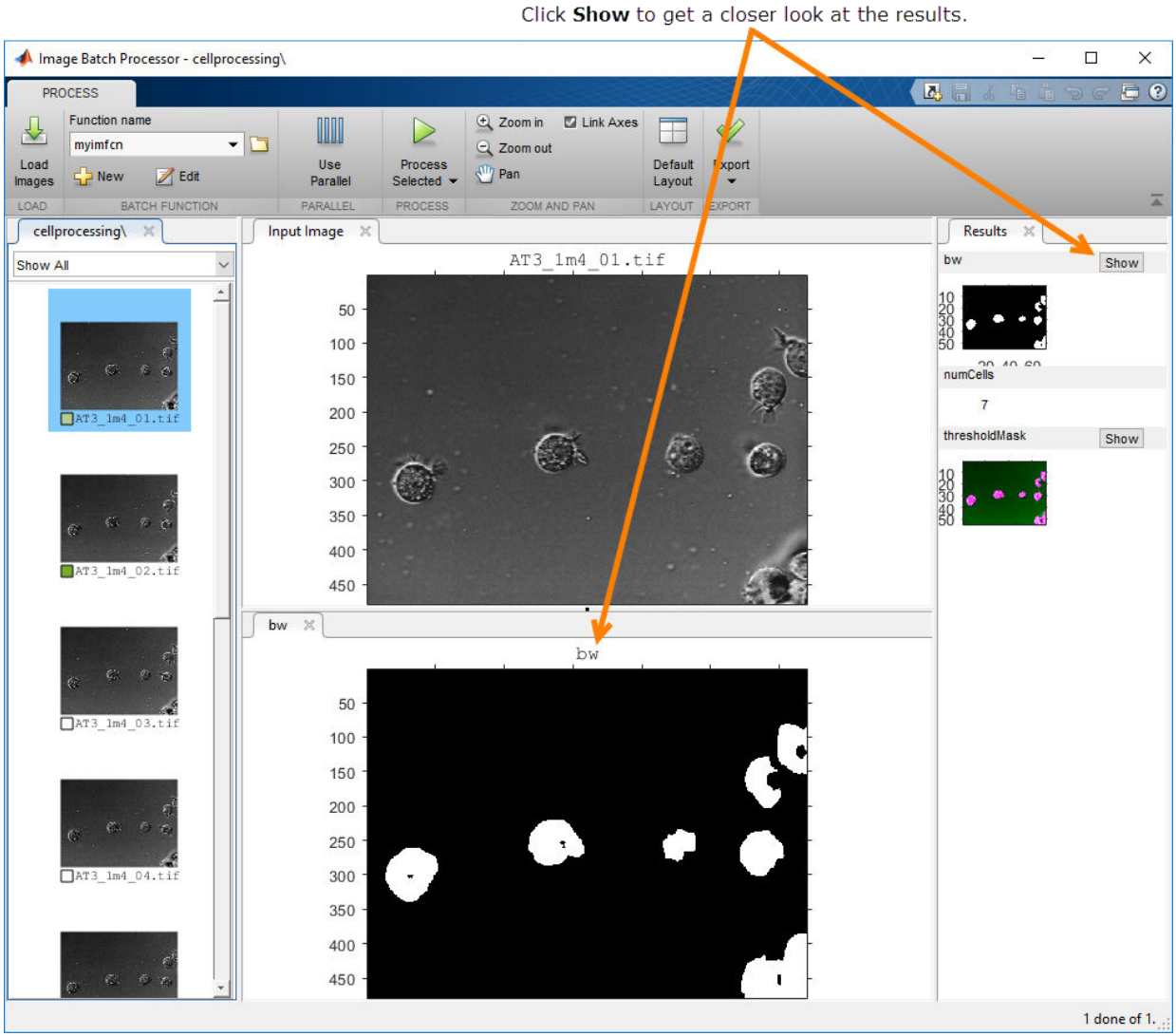
Perform the Operation on the Images

This part of the example shows how to process all the images in a folder (or a subset of the images) in a single operation .

Run the batch processor on one of your images as a test of your function. It can be useful to try out your batch processing function on one of the images and check the results before running it on all the images. When you are satisfied that the function is working properly, run the function on all the images in the folder. With one image selected, click **Process Selected** to process the selected image.



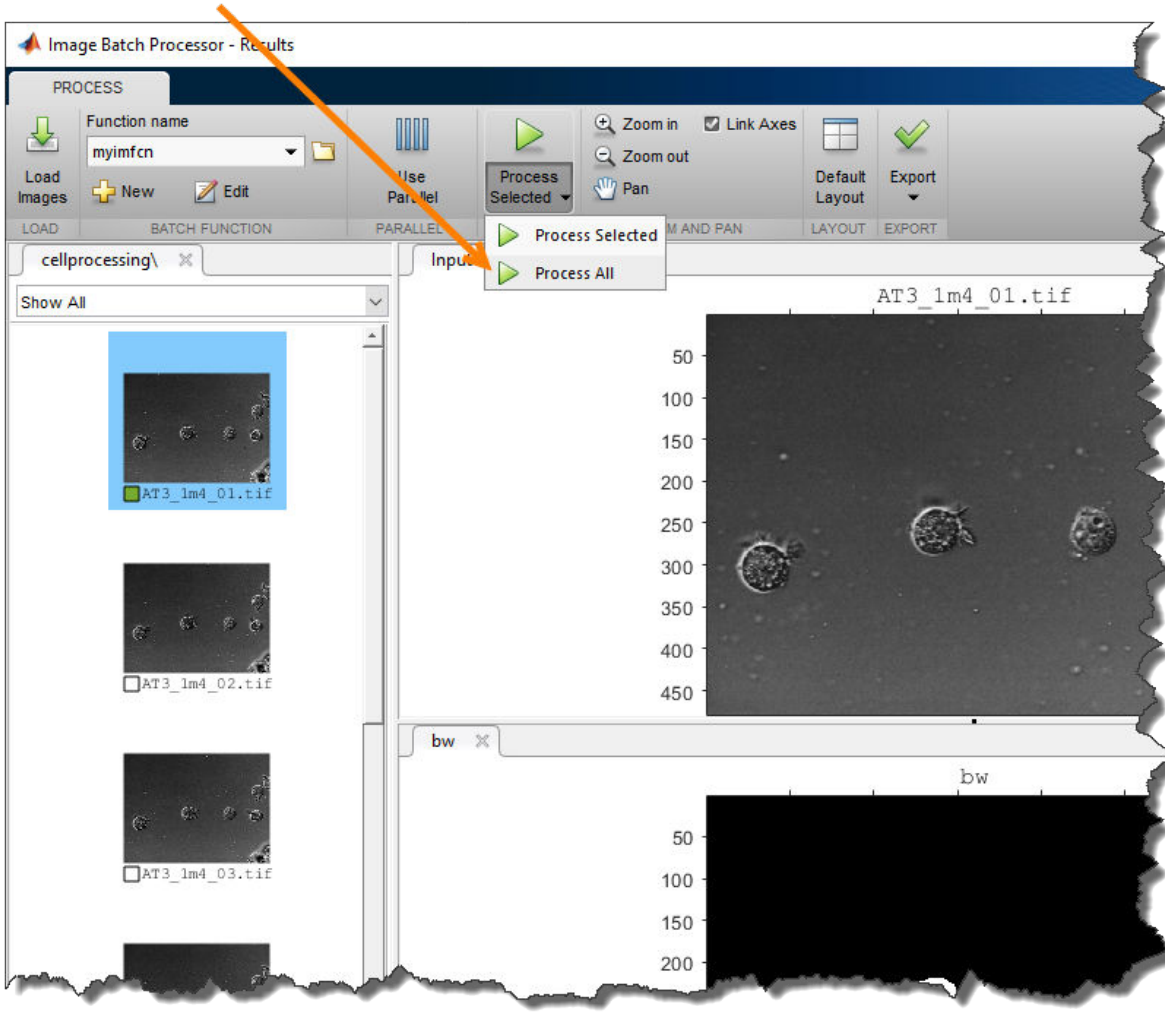
Examine the results of the test run. The app displays the results of the processing in a new tab called **Results**. For this example, the app displays the binary mask, a count of the number of objects (cells) in the image, and a thresholded version of the image. To get a closer view of the image results, click **Show**. The app open a larger resolution version of the image in another tab. You can use the zoom and pan options to examine the image. When zooming and panning, the app links the result image to the original image—moving or zooming on one causes the other image to move.



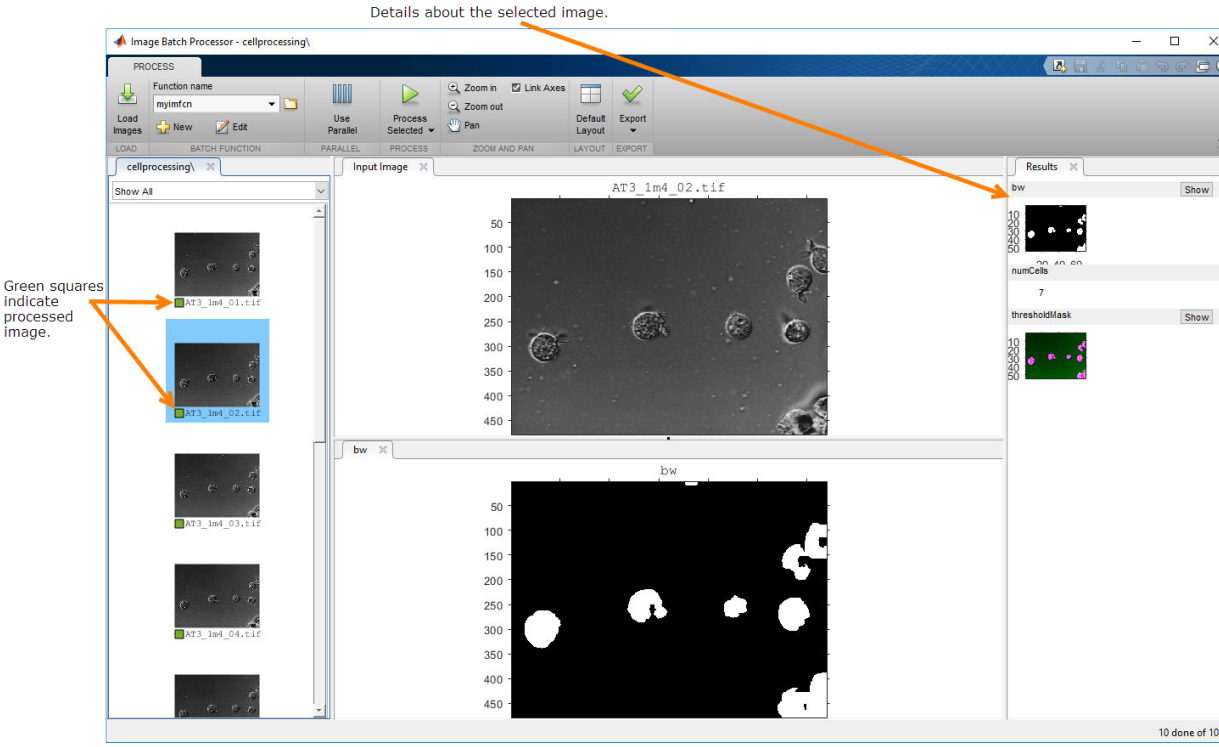
If the results of the test run are successful, execute the function on all the images in your input folder. Click the additional options menu on the **Process Selected** button, and select **Process All**. You can also select the multiple images to process using either **Ctrl-**

Click or **Shift-Click**. If you have Parallel Computing Toolbox, you can click **Use Parallel** to process the images on a local parallel pool.

Select the **Process All** option.



The app processes all the images in the specified folder, filling in the square next to the thumbnail names with green. The **Results** tab contains the results of the selected image.

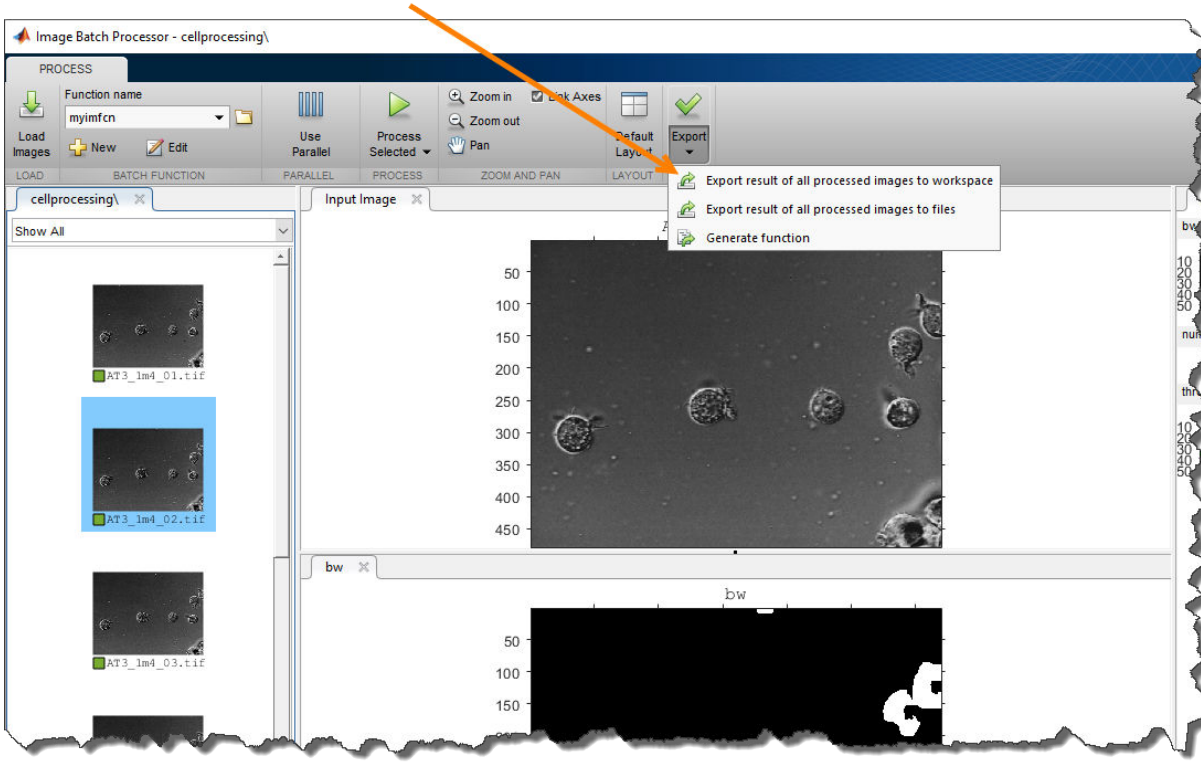


Obtain the Results of the Batch Processing Operation

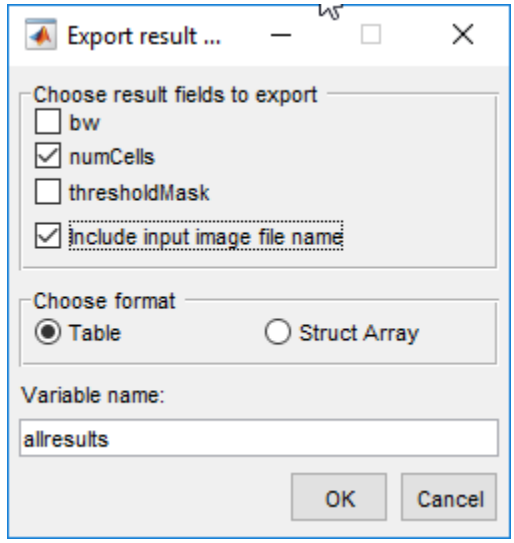
This part of the example shows how to obtain the results of the batch processing operation

Click **Export** to view the options available. You can export results to the workspace or to a file, or you can get the MATLAB code the app used to generate the results.

Click **Export** and choose export option.



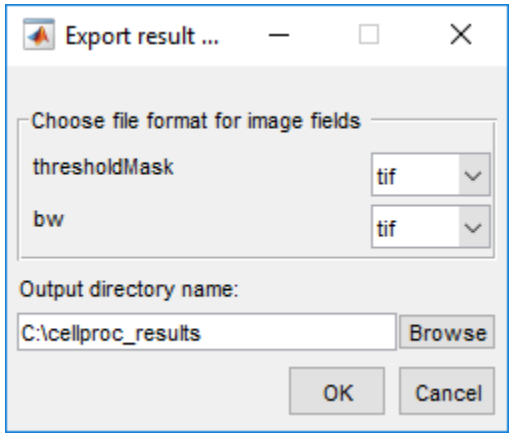
Choose the**Export result of all processed images to workspace** option to save the results in a workspace variable. Select the results you want to save, and click **OK**. A common approach is to save the nonimage results to the workspace and save the images that result from the processing in files. The example saves the cell count along with the name of the input file to the workspace variable `numCells`.



By default, the app returns the results you choose in a table named `allresults`. You can specify another name and you can also choose to store the results in a structure instead. If you select **Include input image file name**, the app includes the name of the image associated with the results.

```
numCells                        fileName
 ________     ____________________________________________

 7            'C:\cellprocessing\cellimages\AT3_1m4_10.tif'
 7            'C:\cellprocessing\cellimages\AT3_1m4_01.tif'
 7            'C:\cellprocessing\cellimages\AT3_1m4_02.tif'
 7            'C:\cellprocessing\cellimages\AT3_1m4_03.tif'
 7            'C:\cellprocessing\cellimages\AT3_1m4_04.tif'
 7            'C:\cellprocessing\cellimages\AT3_1m4_05.tif'
 6            'C:\cellprocessing\cellimages\AT3_1m4_06.tif'
 7            'C:\cellprocessing\cellimages\AT3_1m4_07.tif'
 9            'C:\cellprocessing\cellimages\AT3_1m4_08.tif'
 7            'C:\cellprocessing\cellimages\AT3_1m4_09.tif'
```

Choose the **Export result of all processed images to files** option to save the images produced by the batch processing. Select the file format for each returned file, and click **OK**. By default, the app stores the files in the same folder that you specified when you loaded your images, but you can specify another folder.

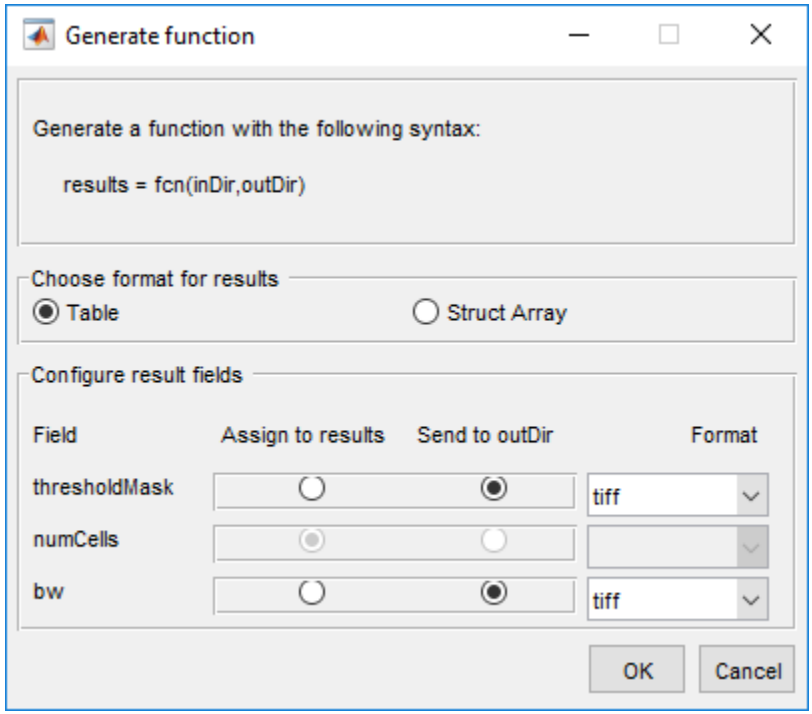


Here is what the folder looks like after saving results in files.

```
06/19/2005  10:06 PM           310,202 AT3_1m4_01.tif
12/12/2015  11:22 AM             3,822 AT3_1m4_01_bw.tif
12/12/2015  11:22 AM           929,544 AT3_1m4_01_thresholdMask.tif
06/19/2005  10:06 PM           310,434 AT3_1m4_02.tif
12/12/2015  11:22 AM             3,840 AT3_1m4_02_bw.tif
12/12/2015  11:22 AM           929,550 AT3_1m4_02_thresholdMask.tif
06/19/2005  10:06 PM           310,314 AT3_1m4_03.tif
12/12/2015  11:22 AM             3,756 AT3_1m4_03_bw.tif
12/12/2015  11:22 AM           929,508 AT3_1m4_03_thresholdMask.tif
06/19/2005  10:06 PM           310,336 AT3_1m4_04.tif
12/12/2015  11:22 AM             3,806 AT3_1m4_04_bw.tif
12/12/2015  11:22 AM           929,512 AT3_1m4_04_thresholdMask.tif
06/19/2005  10:06 PM           310,378 AT3_1m4_05.tif
12/12/2015  11:22 AM             3,744 AT3_1m4_05_bw.tif
12/12/2015  11:22 AM           929,518 AT3_1m4_05_thresholdMask.tif
06/19/2005  10:06 PM           310,342 AT3_1m4_06.tif
12/12/2015  11:22 AM             3,730 AT3_1m4_06_bw.tif
12/12/2015  11:22 AM           929,538 AT3_1m4_06_thresholdMask.tif
06/19/2005  10:06 PM           310,352 AT3_1m4_07.tif
12/12/2015  11:22 AM             3,756 AT3_1m4_07_bw.tif
12/12/2015  11:22 AM           929,532 AT3_1m4_07_thresholdMask.tif
06/19/2005  10:06 PM           310,364 AT3_1m4_08.tif
12/12/2015  11:22 AM             3,746 AT3_1m4_08_bw.tif
12/12/2015  11:22 AM           929,482 AT3_1m4_08_thresholdMask.tif
06/19/2005  10:06 PM           310,266 AT3_1m4_09.tif
12/12/2015  11:22 AM             3,732 AT3_1m4_09_bw.tif
```

```
12/12/2015  11:22 AM           929,500 AT3_1m4_09_thresholdMask.tif
06/19/2005  10:06 PM           310,288 AT3_1m4_10.tif
12/12/2015  11:22 AM             3,718 AT3_1m4_10_bw.tif
12/12/2015  11:22 AM           929,504 AT3_1m4_10_thresholdMask.tif
```

Choose the **Generate function** option to get the MATLAB code the app used to process your files. The app generates a function that accepts the input folder name and the output folder name as input arguments. By default, the function returns a table with the results, but you can choose a structure instead. For image results, you can specify the file format and whether you want the function to write the image to the specified output folder. To get the code, click **OK**.



Process Large Set of Images Using MapReduce Framework and Hadoop

This example shows how to execute a cell counting algorithm on a large number of images using Image Processing Toolbox with MATLAB MapReduce and Datastore. MapReduce is a programming technique for analyzing data sets that do not fit in memory. The example also uses MATLAB Distributed Computing Server™ to run parallel MapReduce programs on Hadoop clusters. The example shows how to create a subset of the images so that you can test your algorithm on a local system before moving it to the Hadoop cluster.

In this section...

"Download Sample Data" on page 2-57
"View Image Files and Test Algorithm" on page 2-58
"Testing Your MapReduce Framework Locally: Data Preparation" on page 2-60
"Testing Your MapReduce Framework Locally: Running your algorithm" on page 2-63
"Running Your MapReduce Framework on a Hadoop Cluster" on page 2-67

Download Sample Data

This part of the example shows how to download the BBBC005v1 data set from the Broad Bioimage Benchmark Collection. This data set is an annotated biological image set designed for testing and validation. The image set provides examples of in- and out-of-focus synthetic images, which can be used for validation of focus metrics. The data set contains almost 20,000 files. For more information, see this introduction to the data set.

At the system prompt on a Linux system, use the `wget` command to download the zip file containing the BBBC data set. Before running this command, make sure that your target location has enough space to hold the zip file (1.8 GB) and the extracted images (2.6 GB).

```
wget http://www.broadinstitute.org/bbbc/BBBC005/BBBC005_v1_images.zip
```

At the system prompt on a Linux system, extract the files from the zip file.

```
unzip BBBC005_v1_images.zip
```

Examine the image file names in this data set. They are constructed in a specific format to contain useful information about each image. For example, the file name

`BBBC005_v1_images/SIMCEPImages_A05_C18_F1_s16_w1.TIF` indicates that the image contains 18 cells (`C18`) and was filtered with a Gaussian low-pass filter with diameter 1 and a sigma of 0.25x diameter to simulate focus blur (`F1`). The `w1` identifies the stain used.

View Image Files and Test Algorithm

This example shows how to view the files in the BBBC data set and test an algorithm on a small subset of the files using the Image Batch Processor app. The example tests a simple algorithm that segments the cells in the images. (The example uses a modified version of this cell segmentation algorithm to create the cell counting algorithm used in the MapReduce implementation.)

Open the Image Batch Processor app. From the MATLAB Toolstrip, on the Apps tab, in the Image Processing and Computer Vision group, click **Image Batch Processor**. You can also open the app from the command line using the `imageBatchProcessor` command.

Load the cell image data set into the app. In the Image Batch Processor app, click **Load Images** and navigate to the folder in which you stored the downloaded data set

Specify the name of the function that implements your cell segmentation algorithm. To specify an existing function, type its name in the **Function name** field or click the folder icon to browse and select the function. To create a new batch processing function, click **New** and enter your code in the template file opened in the MATLAB editor. For this example, create a new function containing the following image segmentation code.

```
function imout = cellSegmenter(im)
% A simple cell segmenter

% Otsu thresholding
t  = graythresh(im);
bw = im2bw(im,t);

% Show thresholding result in app
imout = imfuse(im,bw);

% Find area of blobs
stats = regionprops('table',bw,{'Area'});

% Average cell diameter is about 33 pixels (based on random inspection)
cellArea = pi*(33/2)^2;
```

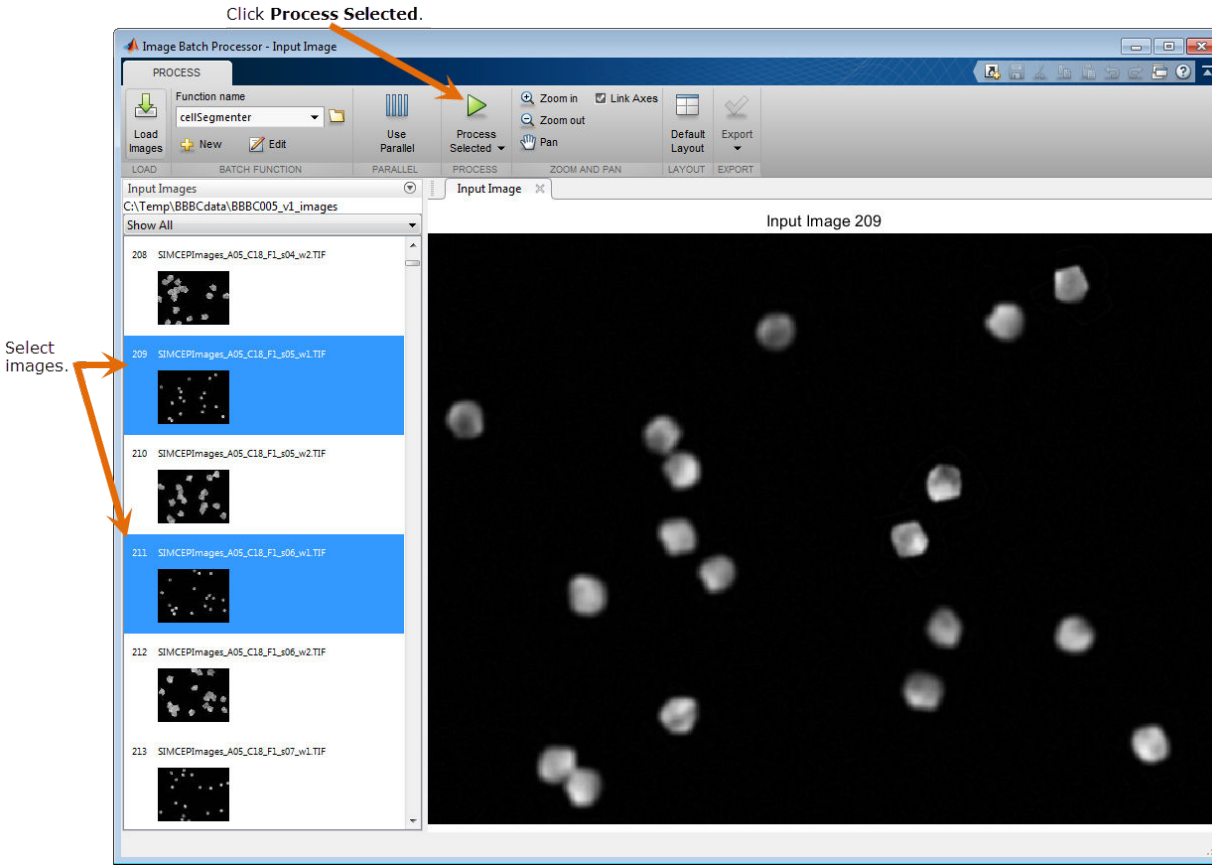
```
% Estimate cell count based on area of blobs
cellsPerBlob = stats.Area/cellArea;
cellCount = sum(round(cellsPerBlob));
disp(cellCount);
```

```
18
```

```
18
```

Select a few images displayed in the app, using the mouse, and click **Run** to execute a test run of your algorithm.

Note For this example, choose only images with the "`w1`" stain. The segmentation algorithm works best with these images.



Examine the results of running your algorithm to verify that your segmentation algorithm found the correct number of cells in the image. The names of the images contain the cell count in the `C` number. For example, the image named `SIMCEPImages_A05_C18_F1_s05_w1.TIF` contains 18 cells. Compare this number to the results returned at the command line for both images.

Testing Your MapReduce Framework Locally: Data Preparation

This example shows how to set up a small test version on your local system of the large scale processing you want to perform. You should test your processing framework before running it on thousands of files. To do this, you must first create an image datastore to

contain the images. MapReduce uses a datastore to process data in small chunks that individually fit into memory. For local testing, select a subset of the images in the datastore to facilitate a quicker, iterative development process. Once you have created the datastore, convert the sample subset of images into Hadoop sequence files, a format used by the Hadoop cluster.

Create an `ImageDatastore`. Because the cell segmentation algorithm implemented in `cellSegmenter.m` works best with the cell body stain, select only the files with the indicator `w1` in their file names.

```
localDataFolder = '/your_data/broad_data/BBBC005_v1_images/';
w1FilesOnly     = fullfile(localDataFolder,'*w1*');
localimds       = imageDatastore(w1FilesOnly);
disp(localimds);
```

```
 ImageDatastore with properties:

      Files: {
             ' .../broad_data/BBBC005_v1_images/SIMCEPImages_A01_C1_F1_s01_w1.TIF';
             ' .../broad_data/BBBC005_v1_images/SIMCEPImages_A01_C1_F1_s02_w1.TIF';
             ' .../broad_data/BBBC005_v1_images/SIMCEPImages_A01_C1_F1_s03_w1.TIF'
              ... and 9597 more
             }
    ReadFcn: @readDatastoreImage
```

Note that there still are over 9000 files in the subset.

Subset this sample further, selecting every 100th file from the thousands of files in the data set. This brings the number of files down to a more manageable number.

```
localimds.Files = localimds.Files(1:100:end);
disp(localimds);
```

```
ImageDatastore with properties:

      Files: {
             ' .../broad_data/BBBC005_v1_images/SIMCEPImages_A01_C1_F1_s01_w1.TIF';
             ' .../broad_data/BBBC005_v1_images/SIMCEPImages_A05_C18_F1_s01_w1.TIF';
             ' .../broad_data/BBBC005_v1_images/SIMCEPImages_A09_C35_F1_s01_w1.TIF'
              ... and 93 more
             }
    ReadFcn: @readDatastoreImage
```

Repackage the subset of images into Hadoop sequence files. Note that this step simply changes the data from one storage format to another without changing the data value.

For more information about sequence files, see “Getting Started with MapReduce” (MATLAB).

You can use the MATLAB `mapreduce` function to do this conversion. You must create a “map” function and a “reduce” function which you pass to the `mapreduce` function. To convert the image files to Hadoop sequence files, the map function should be a no-op function. For this example, the map function simply saves the image data as-is, using its file name as a key.

```
function identityMap(data, info, intermKVStore)
    add(intermKVStore, info.Filename, data);
end
```

Create a reduce function that converts the image files into a key-value datastore backed by sequence files.

```
function identityReduce(key, intermValueIter, outKVStore)
    while hasnext(intermValueIter)
        add(outKVStore, key, getnext(intermValueIter));
    end
end
```

Call `mapreduce`, passing your map and reduce functions. The example first calls the `mapreducer` function to specify where the processing takes place. To test your set up and perform the processing on your local system, specify `0`. (When run locally, `mapreduce` creates a key-value datastore back by MAT-files.) In the following code, the `mapreducer` function specifies that the operation should take place on your local system.

```
mapreducer(0);

matFolder = 'BBBC005_v1_subset_images_w1_mat';
localmatds = mapreduce(localimds, ...
    @identityMap, @identityReduce,...
    'OutputFolder', matFolder);
disp(localmatds);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  10% Reduce   0%
Map  20% Reduce   0%
Map  30% Reduce   0%
Map  40% Reduce   0%
```

```
Map  50% Reduce   0%
Map  60% Reduce   0%
Map  70% Reduce   0%
Map  80% Reduce   0%
Map  90% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce  10%
Map 100% Reduce  20%
Map 100% Reduce  30%
Map 100% Reduce  40%
Map 100% Reduce  50%
Map 100% Reduce  60%
Map 100% Reduce  70%
Map 100% Reduce  80%
Map 100% Reduce  90%
Map 100% Reduce 100%
  KeyValueDatastore with properties:

       Files: {
              ' .../results_1_12-Jun-2015_10-41-25_187.mat';
              ' .../results_2_12-Jun-2015_10-41-25_187.mat'
              }
    ReadSize: 1 key-value pairs
    FileType: 'mat'
```

Testing Your MapReduce Framework Locally: Running your algorithm

This example shows how to test your MapReduce framework on your local system. After creating the subset of image files for testing, and converting them to a key-value datastore, you are ready to test the algorithm. Modify your original cell segmentation algorithm to return the cell count. (The Image Batch Processor app, where this example first tested the algorithm, can only return processed images, not values such as the cell count.)

Modify the cell segmentation function to return a cell count.

```
function cellCount = cellCounter(im)
% A simple cell counter

% Otsu thresholding
t  = graythresh(im);
bw = im2bw(im,t);
```

```
% Show thresholding result in app
% imout = imfuse(im,bw);

stats = regionprops('table',bw,{'Area'});

% Average cell diameter is about 33 pixels (based on random inspection)
cellArea = pi*(33/2)^2;

% Estimate cell count based on area of blobs
cellsPerBlob = stats.Area/cellArea;
cellCount = sum(round(cellsPerBlob));
```

Create map and reduce functions that perform the desired processing. For this example, create a map function that calculates the error count for a specific image. This function gets the actual cell count for an image from the file name coding (the `C` number) and compares it to the cell count returned by the segmentation algorithm.

```
function mapImageToMisCountError(data, ~, intermKVStore)

% Extract the image
im = data.Value{1};

% Call the cell counting algorithm
actCount = cellCounter(im);

% The original file name is available as the key
fileName = data.Key{1};
[~, name] = fileparts(fileName);
% Extract expected cell count and focus blur from the file name
strs = strsplit(name, '_');
expCount  = str2double(strs{3}(2:end));
focusBlur = str2double(strs{4}(2:end));

diffCount = abs(actCount-expCount);

% Note: focus blur is the key
add(intermKVStore, focusBlur, diffCount);
end
```

Create a reduce function that computes the average error in cell count for each focus value.

```
function reduceErrorCount(key, intermValueIter, outKVStore)
```

```
focusBlur = key;

% Compute the sum of all differences in cell count for this value of focus
% blur
count = 0;
totalDiff = 0;
while hasnext(intermValueIter)
    diffCount = getnext(intermValueIter);
    count = count +1;
    totalDiff = totalDiff+diffCount;
end

% Average
meanDiff = totalDiff/count;

add(outKVStore, focusBlur, meanDiff);
end
```

Run the `mapreduce` job on your local system.

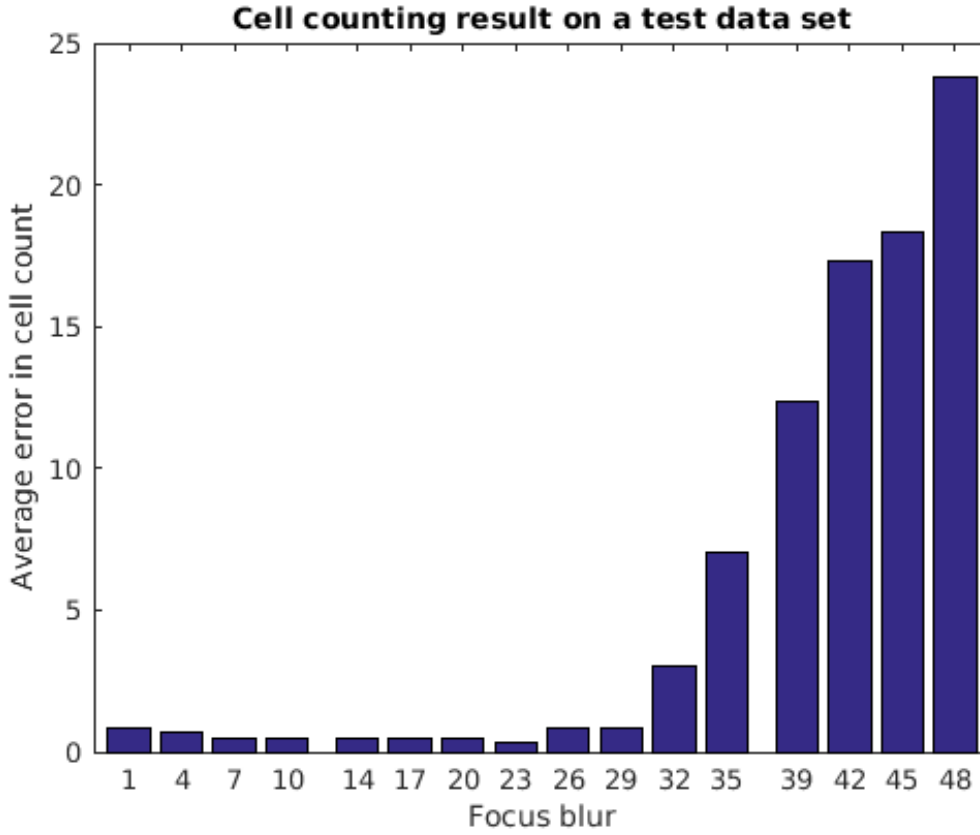
```
focusErrords = mapreduce(localmatds, @mapImageToMisCountError, @reduceErrorCount);

% Gather the result
focusErrorTbl = readall(focusErrords);
disp(focusErrorTbl);
averageErrors = cell2mat(focusErrorTbl.Value);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  10% Reduce   0%
Map  20% Reduce   0%
Map  30% Reduce   0%
Map  40% Reduce   0%
Map  50% Reduce   0%
Map  75% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce  13%
Map 100% Reduce  25%
Map 100% Reduce  38%
Map 100% Reduce  50%
Map 100% Reduce  63%
Map 100% Reduce  75%
Map 100% Reduce  88%
```

```
Map 100% Reduce 100%
    Key       Value
    ___     _________

     1      [ 0.8333]
     4      [ 0.6667]
     7      [ 0.5000]
    10      [ 0.5000]
    14      [ 0.5000]
    17      [ 0.5000]
    20      [ 0.5000]
    23      [ 0.3333]
    26      [ 0.8333]
    29      [ 0.8333]
    32      [      3]
    35      [      7]
    39      [12.3333]
    42      [17.3333]
    45      [18.3333]
    48      [23.8333]
```

Inspect the results on the subset. The simple cell counting algorithm used here relies on the average area of a cell or a group of cells. Increasing focus blur diffuses cell boundaries, and thus the area. The expected result is for the error to go up with increasing focus blur. Plot the results.

```
bar(focusErrorTbl.Key, averageErrors);
ha = gca;
ha.XTick = sort(focusErrorTbl.Key);
ha.XLim  = [min(focusErrorTbl.Key)-2 max(focusErrorTbl.Key)+2];
title('Cell counting result on a test data set');
xlabel('Focus blur');
ylabel('Average error in cell count');
```



Running Your MapReduce Framework on a Hadoop Cluster

This example shows how to load all the image data into the Hadoop file system and run your MapReduce framework on a Hadoop cluster.

Load the image data into the Hadoop file system using the following shell commands.

```
hadoop fs -mkdir /user/broad_data/
hadoop fs -copyFromLocal /your_data/broad_data/BBBC005_v1_images /user/broad_data/BBBC0
```

Set up access to the MATLAB Distributed Computing Server cluster.

```
setenv('HADOOP_HOME', '/mathworks/AH/hub/apps_PCT/LS_Hadoop_hadoop01glnxa64/hadoop-2.5
cluster = parallel.cluster.Hadoop;
cluster.HadoopProperties('mapred.job.tracker') = 'hadoop01glnxa64:54311';
cluster.HadoopProperties('fs.default.name') = 'hdfs://hadoop01glnxa64:54310';
disp(cluster);
% Change mapreduce execution environment to point to the remote cluster
mapreducer(cluster);
```

```
Hadoop with properties:

                 HadoopInstallFolder: '/your_data/apps_PCT/LS_Hadoo...'
             HadoopConfigurationFile: ''
                    HadoopProperties: [2x1 containers.Map]
                  ClusterMatlabRoot: '/your_cluster...'
    RequiresMathWorksHostedLicensing: 0
                       LicenseNumber: ''
                     AutoAttachFiles: 1
                       AttachedFiles: {}
                     AdditionalPaths: {}
```

Convert all the image data into Hadoop sequence files. This is similar to what you did on your local system when you converted a subset of the images for prototyping. You can reuse the map and reduce functions you used previously.

```
% Use the internal Hadoop cluster ingested with Broad Institute files
broadFolder = 'hdfs://hadoop01glnxa64:54310/user/broad_data/BBBC005_v1_images';

% Pick only the 'cell body stain' (w1) files for processing
w1Files = fullfile(broadFolder, '*w1*.TIF');

% Create an ImageDatastore representing all these files
imageDS = imageDatastore(w1Files);

% Specify the output folder.
seqFolder = 'hdfs://hadoop01glnxa64:54310/user/datasets/images/broad_data/broad_sequen

% Convert the images to a key-value datastore.
seqds = mapreduce(imageDS, @identityMap, @identityReduce,'OutputFolder', seqFolder);
```

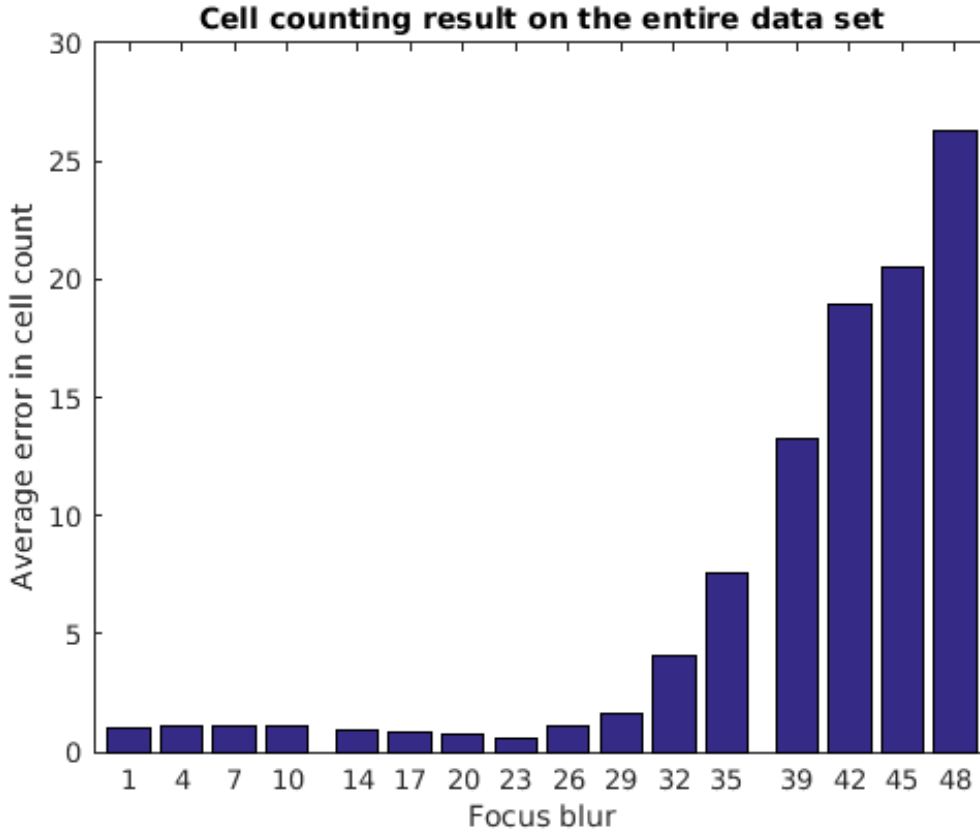
Run the cell counting algorithm on the entire data set stored in the Hadoop file system using the MapReduce framework. The only change from the local version now is that the input and output locations are on the Hadoop file system.

```
% Output location for error count.
output = 'hdfs://hadoop01glnxa64:54310/user/broad_data/BBBC005_focus_vs_errorCount';
```

```
tic;
focusErrords = mapreduce(seqds, @mapImageToMisCountError, @reduceErrorCount,...
    'OutputFolder',output);
toc
% Gather result
focusErrorTbl = readall(focusErrords);
disp(focusErrorTbl);
averageErrors = cell2mat(focusErrorTbl.Value);

% Plot
bar(focusErrorTbl.Key, averageErrors);
ha = gca;
ha.XTick = sort(focusErrorTbl.Key);
ha.XLim  = [min(focusErrorTbl.Key)-2 max(focusErrorTbl.Key)+2];
title('Cell counting result on the entire data set');
xlabel('Focus blur');
ylabel('Average error in cell count');
```

```
Parallel mapreduce execution on the Hadoop cluster:
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map   7% Reduce   0%
Map  10% Reduce   0%
Map  12% Reduce   0%
Map  20% Reduce   0%
Map  23% Reduce   0%
Map  25% Reduce   0%
Map  28% Reduce   0%
Map  30% Reduce   0%
Map  32% Reduce   0%
Map  33% Reduce   0%
Map  38% Reduce   0%
Map  41% Reduce   0%
Map  43% Reduce   0%
Map  48% Reduce   0%
Map  50% Reduce   0%
Map  51% Reduce   5%
Map  53% Reduce   7%
Map  55% Reduce  10%
Map  56% Reduce  11%
Map  58% Reduce  11%
Map  62% Reduce  12%
Map  64% Reduce  12%
```

```
Map  65% Reduce  12%
Map  67% Reduce  16%
Map  69% Reduce  16%
Map  71% Reduce  16%
Map  74% Reduce  17%
Map  75% Reduce  17%
Map  76% Reduce  17%
Map  79% Reduce  20%
Map  83% Reduce  23%
Map  85% Reduce  24%
Map  88% Reduce  24%
Map  92% Reduce  24%
Map  94% Reduce  25%
Map  96% Reduce  28%
Map  97% Reduce  29%
Map 100% Reduce  66%
Map 100% Reduce  69%
Map 100% Reduce  78%
Map 100% Reduce  87%
Map 100% Reduce  96%
Map 100% Reduce 100%
Elapsed time is 227.508109 seconds.
    Key      Value
    ___    _________

     4     [ 1.1117]
     7     [ 1.0983]
    10     [ 1.0500]
    14     [ 0.9317]
    17     [ 0.8650]
    20     [ 0.7583]
    23     [ 0.6050]
    26     [ 1.0600]
    29     [ 1.5750]
    32     [ 4.0633]
    42     [18.9267]
    48     [26.2417]
     1     [ 1.0083]
    35     [ 7.5650]
    39     [13.2383]
    45     [20.5500]
```



See Also

`ImageDatastore` | `mapreduce`

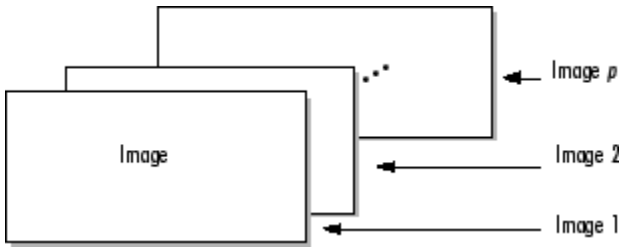
More About

- “Getting Started with MapReduce” (MATLAB)

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 2-72.) 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” on page 4-89.

Function	Image Sequence Dimensions	Guideline When Used with an Image Sequence
`bwlabeln`	*m*-by-*n*-by-*p* only	Must use the `bwlabeln(BW,conn)` syntax with a 2-D connectivity.
`deconvblind`	*m*-by-*n*-by-*p* or *m*-by-*n*-by-3-by-*p*	`PSF` argument can be either 1-D or 2-D.
`deconvlucy`	*m*-by-*n*-by-*p* or *m*-by-*n*-by-3-by-*p*	`PSF` argument can be either 1-D or 2-D.
`edgetaper`	*m*-by-*n*-by-*p* or *m*-by-*n*-by-3-by-*p*	`PSF` argument can be either 1-D or 2-D.
`entropyfilt`	*m*-by-*n*-by-*p* only	`nhood` argument must be 2-D.
`imabsdiff`	*m*-by-*n*-by-*p* or *m*-by-*n*-by-3-by-*p*	Image sequences must be the same size.
`imadd`	*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.
`imbothat`	*m*-by-*n*-by-*p* only	`SE` argument must be 2-D.
`imclose`	*m*-by-*n*-by-*p* only	`SE` argument must be 2-D.
`imdilate`	*m*-by-*n*-by-*p* only	`SE` argument must be 2-D.
`imdivide`	*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
`imerode`	*m*-by-*n*-by-*p* only	`SE` argument must be 2-D.
`imextendedmax`	*m*-by-*n*-by-*p* only	Must use the `imextendedmax(I,h,conn)` syntax with a 2-D connectivity.
`imextendedmin`	*m*-by-*n*-by-*p* only	Must use the `imextendedmin(I,h,conn)` syntax with a 2-D connectivity.
`imfilter`	*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.
`imhmax`	*m*-by-*n*-by-*p* only	Must use the `imhmax(I,h,conn)` syntax with a 2-D connectivity.
`imhmin`	*m*-by-*n*-by-*p* only	Must use the `imhmin(I,h,conn)` syntax with a 2-D connectivity.
`imlincomb`	*m*-by-*n*-by-*p* or *m*-by-*n*-by-3-by-*p*	Image sequences must be the same size.
`immultiply`	*m*-by-*n*-by-*p* or *m*-by-*n*-by-3-by-*p*	Image sequences must be the same size.
`imopen`	*m*-by-*n*-by-*p* only	`SE` argument must be 2-D.
`imregionalmax`	*m*-by-*n*-by-*p* only	Must use the `imextendedmax(I,conn)` syntax with a 2-D connectivity.
`imregionalmin`	*m*-by-*n*-by-*p* only	Must use the `imextendedmin(I,conn)` syntax with a 2-D connectivity.
`imsubtract`	*m*-by-*n*-by-*p* or *m*-by-*n*-by-3-by-*p*	Image sequences must be the same size.
`imtophat`	*m*-by-*n*-by-*p* only	`SE` argument must be 2-D.
`imwarp`	*m*-by-*n*-by-*p* or *m*-by-*n*-by-3-by-*p*	`TFORM` argument must be 2-D.
`padarray`	*m*-by-*n*-by-*p* or *m*-by-*n*-by-3-by-*p*	`PADSIZE` argument must be a two-element vector.
`rangefilt`	*m*-by-*n*-by-*p* only	`NHOOD` argument must be 2-D.

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

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	`uint8`	255
-45	`uint8`	0
10.5	`uint8`	11

Nest 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
```

Find Vegetation in a Multispectral Image

This example shows how to use MATLAB® array arithmetic to process images and plot image data. In particular, this example works with a three-dimensional image array where the three planes represent the image signal from different parts of the electromagnetic spectrum, including the visible red and near-infrared (NIR) channels.

Image data differences can be used to distinguish different surface features of an image, which have varying reflectivity across different spectral channels. By finding differences between the visible red and NIR channels, the example identifies areas containing significant vegetation.

Step 1: Import Color-Infrared Channels from a Multispectral Image File

This example finds vegetation in a LANDSAT Thematic Mapper image covering part of Paris, France, made available courtesy of Space Imaging, LLC. Seven spectral channels (bands) are stored in one file in the Erdas LAN format. The LAN file, `paris.lan`, contains a 7-channel 512-by-512 Landsat image. A 128-byte header is followed by the pixel values, which are band interleaved by line (BIL) in order of increasing band number. Pixel values are stored as unsigned 8-bit integers, in little-endian byte order.

The first step is to read bands 4, 3, and 2 from the LAN file using the MATLAB® function `multibandread`.

Channels 4, 3, and 2 cover the near infrared (NIR), the visible red, and the visible green parts of the electromagnetic spectrum. When they are mapped to the red, green, and blue planes, respectively, of an RGB image, the result is a standard color-infrared (CIR) composite. The final input argument to `multibandread` specifies which bands to read, and in which order, so that you can construct a composite in a single step.

```
CIR = multibandread('paris.lan',[512, 512, 7],'uint8=>uint8',...
                    128,'bil','ieee-le',{'Band','Direct',[4 3 2]});
```

Variable `CIR` is a 512-by-512-by-3 array of class `uint8`. It is an RGB image, but with false colors. When the image is displayed, red pixel values signify the NIR channel, green values signify the visible red channel, and blue values signify the visible green channel.

In the CIR image, water features are very dark (the Seine River) and green vegetation appears red (parks and shade trees). Much of the image appearance is due to the fact that healthy, chlorophyll-rich vegetation has a high reflectance in the near infrared. Because the NIR channel is mapped to the red channel in the composite image, any area

with a high vegetation density appears red in the display. A noticeable example is the area of bright red on the left edge, a large park (the Bois de Boulogne) located west of central Paris within a bend of the Seine River.

```
imshow(CIR)
title('CIR Composite')
text(size(CIR,2),size(CIR,1) + 15,...
  'Image courtesy of Space Imaging, LLC',...
  'FontSize',7,'HorizontalAlignment','right')
```

CIR Composite

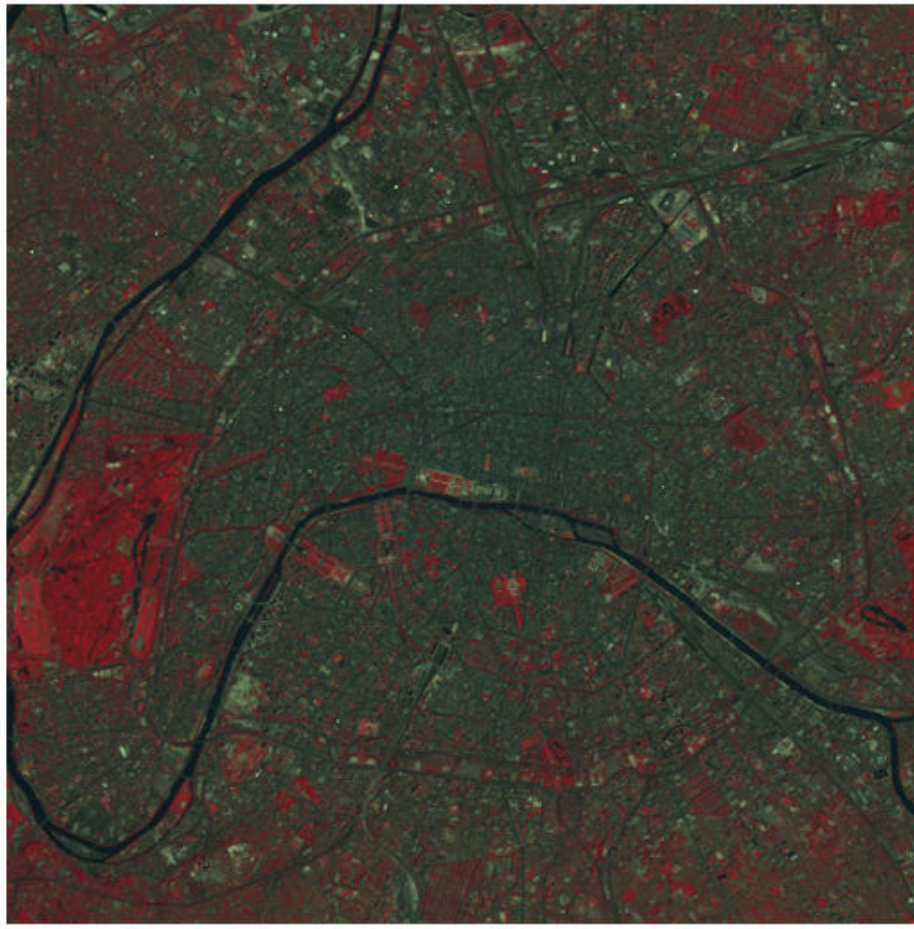


Image courtesy of Space Imaging, LLC

By analyzing differences between the NIR and red channels, you can quantify this contrast in spectral content between vegetated areas and other surfaces such as pavement, bare soil, buildings, or water.

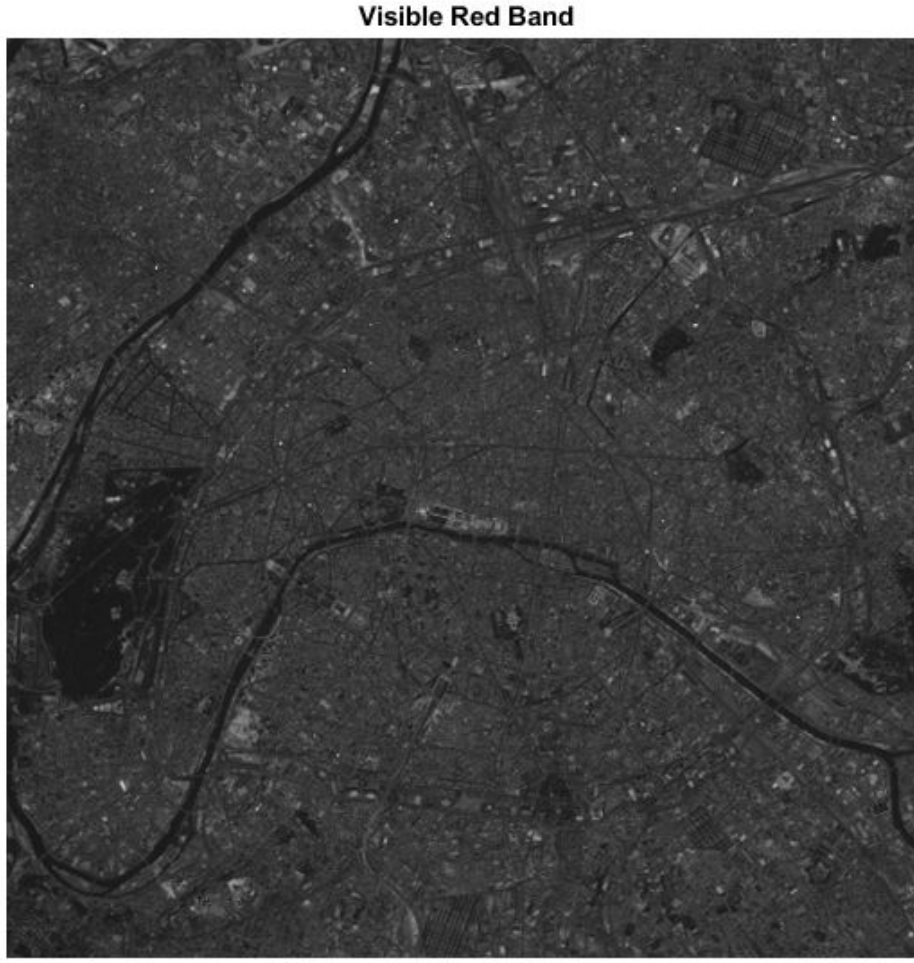
Step 2: Construct an NIR-Red Spectral Scatter Plot

A scatter plot is a natural place to start when comparing the NIR channel (displayed as red pixel values) with the visible red channel (displayed as green pixel values). It's convenient to extract these channels from the original CIR composite into individual variables. It's also helpful to convert from class `uint8` to class `single` so that the same variables can be used in the NDVI computation below, as well as in the scatter plot.

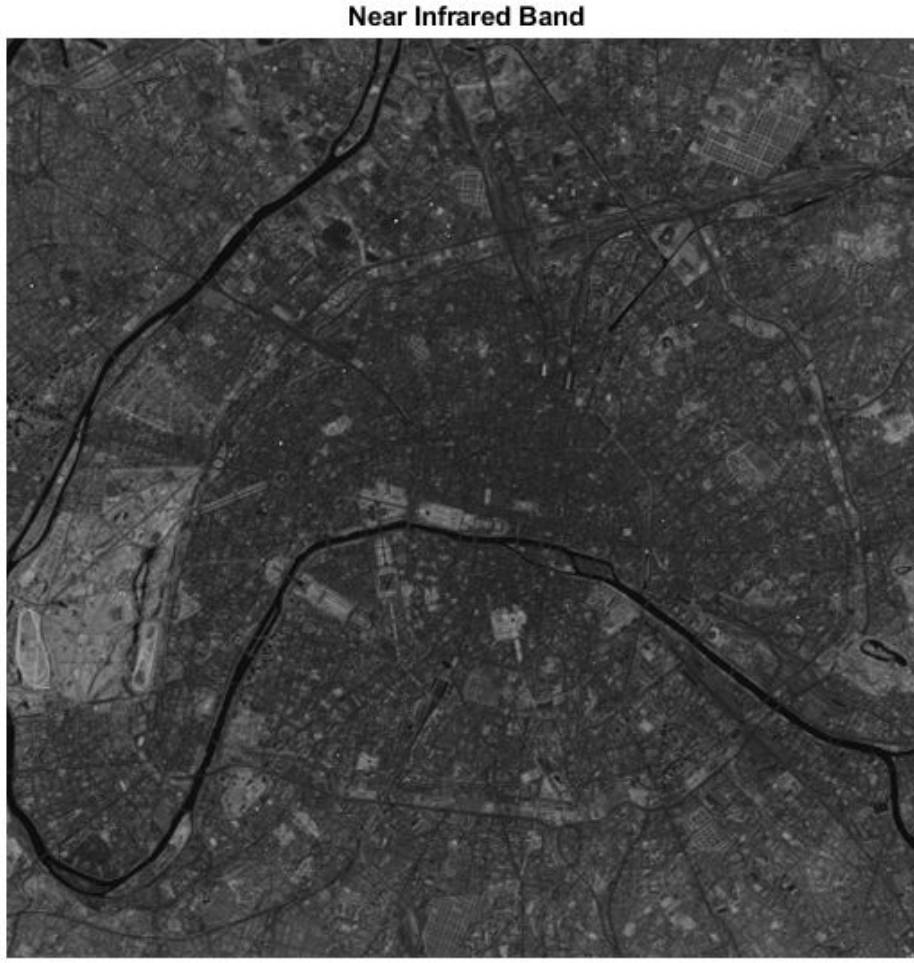
```
NIR = im2single(CIR(:,:,1));
R = im2single(CIR(:,:,2));
```

Viewing the two channels together as grayscale images, you can see how different they look.

```
imshow(R)
title('Visible Red Band')
```

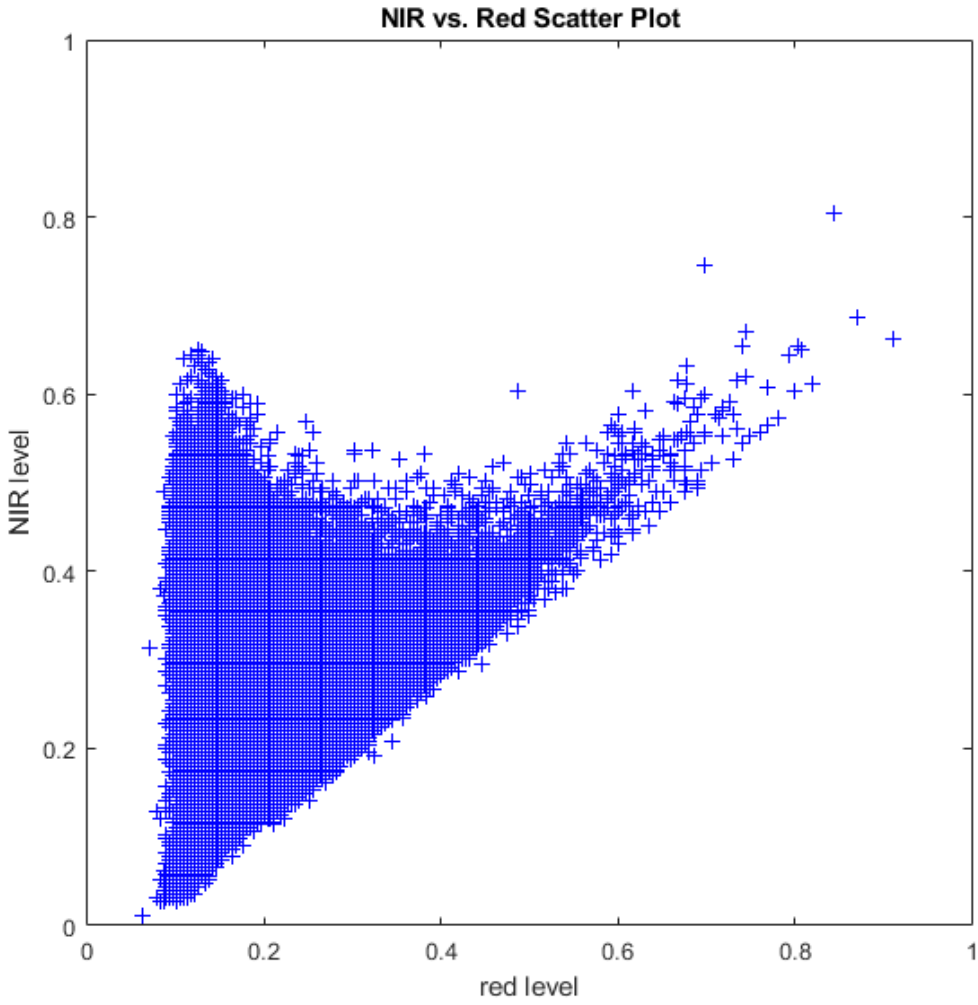



```
imshow(NIR)
title('Near Infrared Band')
```



With one simple call to the `plot` command in MATLAB, you can create a scatter plot displaying one point per pixel (as a blue cross, in this case), with its x-coordinate determined by its value in the red channel and its y-coordinate by the value its value in the NIR channel.

```
plot(R,NIR,'+b')
ax = gca;
ax.XLim  = [0 1];
ax.XTick = 0:0.2:1;
ax.YLim  =  [0 1];
ax.YTick = 0:0.2:1;
axis square
xlabel('red level')
ylabel('NIR level')
title('NIR vs. Red Scatter Plot')
```



The appearance of the scatter plot of the Paris scene is characteristic of a temperate urban area with trees in summer foliage. There's a set of pixels near the diagonal for which the NIR and red values are nearly equal. This "gray edge" includes features such as road surfaces and many rooftops. Above and to the left is another set of pixels for which

the NIR value is often well above the red value. This zone encompasses essentially all of the green vegetation.

Step 3: Compute Vegetation Index via MATLAB® Array Arithmetic

Observe from the scatter plot that taking the ratio of the NIR level to red level would be one way to locate pixels containing dense vegetation. However, the result would be noisy for dark pixels with small values in both channels. Also notice that the difference between the NIR and red channels should be larger for greater chlorophyll density. The Normalized Difference Vegetation Index (NDVI) is motivated by this second observation. It takes the (NIR - red) difference and normalizes it to help balance out the effects of uneven illumination such as the shadows of clouds or hills. In other words, on a pixel-by-pixel basis subtract the value of the red channel from the value of the NIR channel and divide by their sum.

```
ndvi = (NIR - R) ./ (NIR + R);
```

Notice how the array-arithmetic operators in MATLAB make it possible to compute an entire NDVI image in one simple command. Recall that variables `R` and `NIR` have class `single`. This choice uses less storage than class `double` but unlike an integer class also allows the resulting ratio to assume a smooth gradation of values.

Variable `ndvi` is a 2-D array of class `single` with a theoretical maximum range of [-1 1]. You can specify these theoretical limits when displaying `ndvi` as a grayscale image.

```
figure
imshow(ndvi,'DisplayRange',[-1 1])
title('Normalized Difference Vegetation Index')
```

The Seine River appears very dark in the NDVI image. The large light area near the left edge of the image is the park (Bois de Boulogne) noted earlier.

Step 4: Locate Vegetation -- Threshold the NDVI Image

In order to identify pixels most likely to contain significant vegetation, apply a simple threshold to the NDVI image.

```
threshold = 0.4;
q = (ndvi > threshold);
```

The percentage of pixels selected is thus

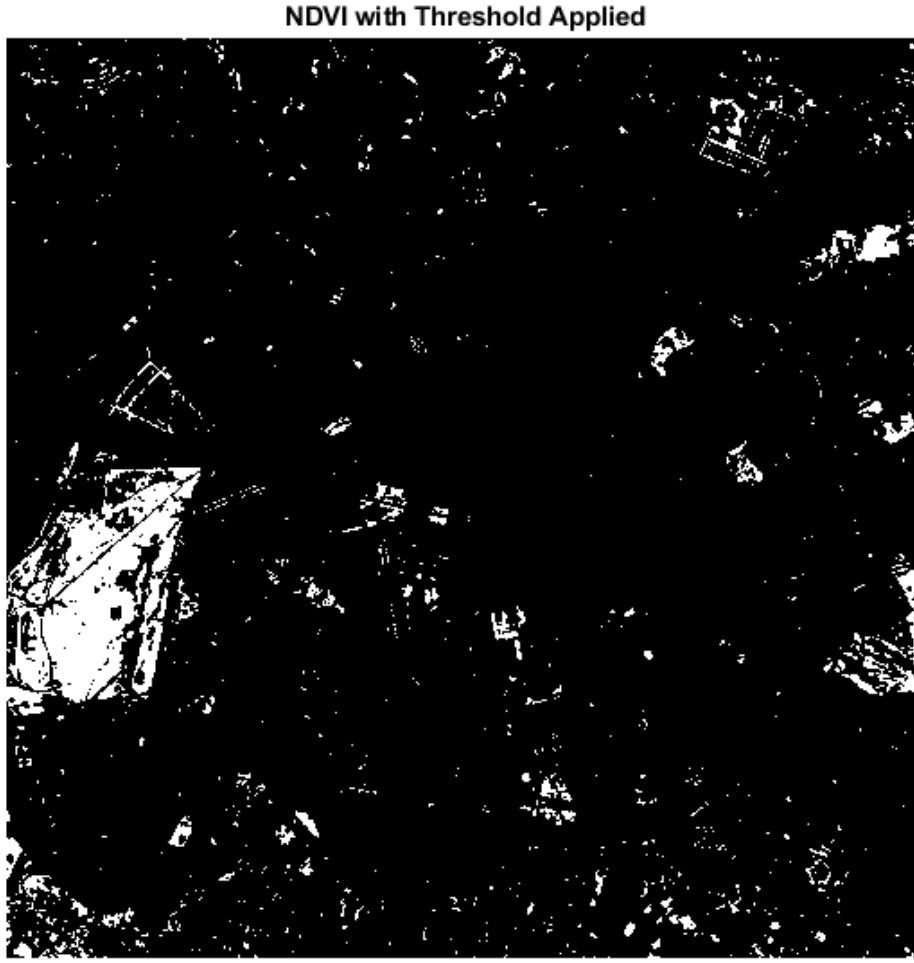
```
100 * numel(NIR(q(:))) / numel(NIR)
```

```
ans = 5.2204
```

or about 5 percent.

The park and other smaller areas of vegetation appear white by default when displaying the logical (binary) image `q`.

```
imshow(q)
title('NDVI with Threshold Applied')
```



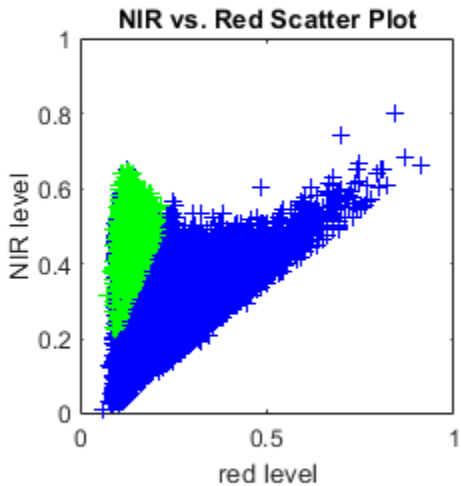
Step 5: Link Spectral and Spatial Content

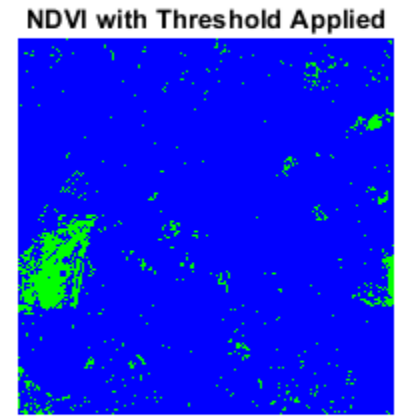
To link the spectral and spatial content, you can locate above-threshold pixels on the NIR-red scatter plot, re-drawing the scatter plot with the above-threshold pixels in a contrasting color (green) and then re-displaying the threshold NDVI image using the

same blue-green color scheme. As expected, the pixels having an NDVI value above the threshold appear to the upper left of the rest and correspond to the redder pixels in the CIR composite displays.

Create the scatter plot, then display the thresholded NDVI.

```
figure
subplot(1,2,1)
plot(R,NIR,'+b')
hold on
plot(R(q(:)),NIR(q(:)),'g+')
axis square
xlabel('red level')
ylabel('NIR level')
title('NIR vs. Red Scatter Plot')

subplot(1,2,2)
imshow(q)
colormap([0 0 1; 0 1 0]);
title('NDVI with Threshold Applied')
```





See Also

`im2single` | `imshow` | `multibandread`

Related Examples

- “Enhancing Multispectral Color Composite Images” on page 11-115

More About

- “Images in MATLAB” on page 2-2

3

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.

- “Get Information About Graphics Files” on page 3-2
- “Read Image Data into the Workspace” on page 3-3
- “Read Multiple Images from a Single Graphics File” on page 3-5
- “Read and Write 1-Bit Binary Images” on page 3-6
- “Write Image Data to File in Graphics Format” on page 3-8
- “Determine Storage Class of Output Files” on page 3-10
- “DICOM Support in the Image Processing Toolbox” on page 3-11
- “Read Metadata from DICOM Files” on page 3-12
- “Read Image Data from DICOM Files” on page 3-14
- “Write Image Data to DICOM Files” on page 3-15
- “Explicit Versus Implicit VR Attributes” on page 3-17
- “Remove Confidential Information from a DICOM File” on page 3-18
- “Create New DICOM Series” on page 3-19
- “Mayo Analyze 7.5 Files” on page 3-22
- “Interfile Files” on page 3-23
- “Work with High Dynamic Range Images” on page 3-24
- “Display High Dynamic Range Image” on page 3-26

Get 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 4-61.

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

Read Image Data into the Workspace

This example shows to read image data from a graphics file into the MATLAB workspace using the `imread` function.

Read a truecolor image into the workspace. The example reads the image data from a graphics file that uses JPEG format.

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

If the image file format uses 8-bit pixels, `imread` returns the image data as an m-by-n-by-3 array of `uint8` values. For graphics file formats that support 16-bit data, such as PNG and TIFF, `imread` returns an array of `uint16` values.

```
whos
```

```
  Name        Size                 Bytes  Class    Attributes

  RGB       256x320x3            245760  uint8
```

Read a grayscale image into the workspace. The example reads the image data from a graphics file that uses the TIFF format. `imread` returns the grayscale image as an m-by-n array of `uint8` values.

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

```
  Name        Size                 Bytes  Class    Attributes

  I         256x256                65536  uint8
  RGB       256x320x3            245760  uint8
```

Read an indexed image into the workspace. `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');
whos
```

```
  Name        Size                 Bytes  Class     Attributes

  I         256x256                65536  uint8
  RGB       256x320x3            245760  uint8
```

```
  X         258x350                90300  uint8
  map       256x3                   6144  double
```

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 the 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 image formats supported, see `imread` and `imformats` reference pages.

```
pep = imread('peppers.png','png');
whos
```

```
  Name        Size                Bytes  Class     Attributes

  I         256x256               65536  uint8
  RGB       256x320x3            245760  uint8
  X         258x350               90300  uint8
  map       256x3                  6144  double
  pep       384x512x3            589824  uint8
```

See Also

`imread`

More About

- “Image Types in the Toolbox” on page 2-13
- “Read Multiple Images from a Single Graphics File” on page 3-5
- “Write Image Data to File in Graphics Format” on page 3-8

Read Multiple Images from a Single Graphics File

This example shows how to read multiple images from a single graphics file. Some graphics file formats allow you to store multiple images. You can read these images using format-specific parameters with `imread`. By default, `imread` imports only the first image in the file.

Preallocate a 4-D array to hold the images to be read from a file.

```
mri = zeros([128 128 1 27],'uint8');
```

Read the images from the file, using a loop to read each image sequentially.

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

```
  Name         Size                    Bytes  Class     Attributes

  frame        1x1                         8  double
  map        256x3                      6144  double
  mri        128x128x1x27             442368  uint8
```

See Also

`imread`

More About

- “Read Image Data into the Workspace” on page 3-3

Read and Write 1-Bit Binary Images

This example shows how to read and write 1-bit binary images.

Check the bit depth of the graphics file containing a binary image, `text.png`. Note that the file stores the binary image in 1-bit format.

```
info = imfinfo('text.png');
info.BitDepth
```

```
ans = 1
```

Read the binary image from the file into the workspace. When you read a binary image stored in 1-bit format, `imread` represents the data in the workspace as a logical array.

```
BW = imread('text.png');
whos
```

```
  Name        Size             Bytes  Class      Attributes

  BW        256x256            65536  logical
  ans         1x1                  8  double
  info        1x1               4454  struct
```

Write the binary image to a file in 1-bit format. If the file format supports it, `imwrite` exports a binary image as a 1-bit image, by default. To verify this, use `imfinfo` to get information about the newly created file and check the `BitDepth` field. When writing binary files, `imwrite` sets the `ColorType` field to `grayscale`.

```
imwrite(BW,'test.tif');
info = imfinfo('test.tif');
info.BitDepth
```

```
ans = 1
```

See Also

`imfinfo` | `imread` | `imwrite`

More About

- “Image Types in the Toolbox” on page 2-13

- “Read Image Data into the Workspace” on page 3-3

Write Image Data to File in Graphics Format

This example shows how to write image data from the MATLAB workspace to a file in one of the supported graphics file formats using the `imwrite` function.

Load image data into the workspace. This example loads the indexed image `X` from a MAT-file, `clown.mat`, along with the associated colormap `map`.

```
load clown
whos
```

```
  Name           Size              Bytes  Class     Attributes

  X            200x320            512000  double
  caption        2x1                   4  char
  map           81x3                1944  double
```

Export the image data as a bitmap file using `imwrite`, specifying the name of the variable and the name of the output file you want to create. If you include an extension in the filename, `imwrite` attempts to infer the desired file format from it. For example, the file extension `.bmp` specifies the Microsoft Windows Bitmap format. You can also specify the format explicitly as an argument to `imwrite`.

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

Use format-specific parameters with `imwrite` to control aspects of the export process. For example, with PNG files, you can specify the bit depth. To illustrate, read an image into the workspace in TIFF format and note its bit depth.

```
I = imread('cameraman.tif');
s = imfinfo('cameraman.tif');
s.BitDepth
```

```
ans = 8
```

Write the image to a graphics file in PNG format, specifying a bit depth of 4.

```
imwrite(I,'cameraman.png','Bitdepth',4)
```

Check the bit depth of the newly created file.

```
newfile = imfinfo('cameraman.png');
newfile.BitDepth
```

```
ans = 4
```

See Also

`imwrite`

More About

- “Image Types in the Toolbox” on page 2-13
- “Read Image Data into the Workspace” on page 3-3

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, `imwrite` creates a 1-bit image file. If the output image file format specified does not support 1-bit images, `imwrite` exports the image data as a `uint8` grayscale image.
`uint8`	If the output image file format supports unsigned 8-bit images, `imwrite` creates an unsigned 8-bit image file.
`uint16`	If the output image file format supports unsigned 16-bit images (PNG or TIFF), `imwrite` creates an unsigned 16-bit image file. If the output image file format does not support 16-bit images, `imwrite` scales the image data to class `uint8` and creates an 8-bit image file.
`int16`	Partially supported; depends on file format.
`single`	Partially supported; depends on file format.
`double`	MATLAB scales the image data to `uint8` and creates an 8-bit image file, because most image file formats use 8 bits.

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?” on page 2-72.

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” on page 16-2.

DICOM Support in the Image Processing Toolbox

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 files that comply 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 DICOM Files

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 3-14. If you just want to view the metadata in a DICOM file, for debugging purposes, you can use the `dicomdisp` function.

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

```
info = dicominfo('CT-MONO2-16-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 contains characters, `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 3-13.

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 file name 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 `imtool`. 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 Image Data to DICOM Files

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-MONO2-16-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');
```

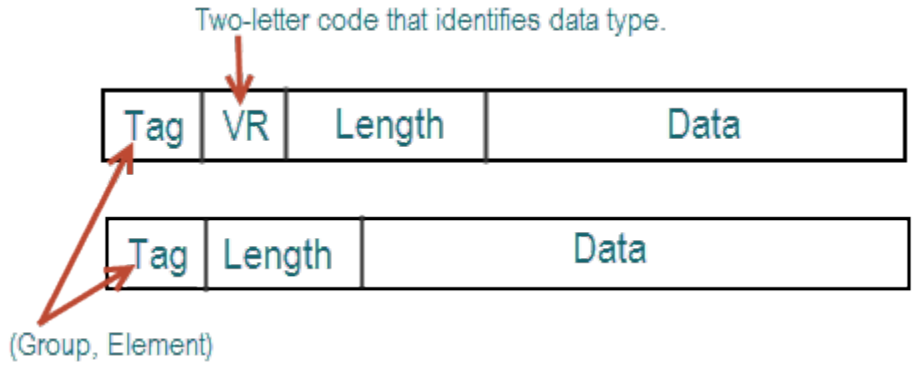
```
info2.SOPInstanceUID
```

```
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 New DICOM Series” on page 3-19 with a call to the `dicomanon` function.

Create 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-MONO2-16-ankle.dcm');
   ```

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

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

2. Read the metadata from the same DICOM file.

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

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

   ```
   info.SeriesInstanceUID
   ```

```
ans =

1.2.840.113619.2.1.2411.1031152382.365.736169244
```

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

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

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

```
ans =

    4080
```

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

```
ans =

    32
```

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

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

View the processed image.

```
imshow(Imodified,[])
```

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

```
uid = dicomuid
```

```
uid =

1.3.6.1.4.1.9590.100.1.1.56461980611264497732341403390561061497
```

`dicomuid` is guaranteed to generate a unique UID.

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

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

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

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

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

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

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 workspace.

Note The Analyze 7.5 format uses the same dual-file data set organization and the same file name 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 3-23.

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 workspace.

Note The Interfile format uses the same dual-file data set organization and the same file name 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 3-22.

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');
```

Work with High Dynamic Range Images

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 3-26.

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);
```

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');
```

See Also

`hdrread` | `hdrwrite` | `makehdr`

Related Examples

- “Display High Dynamic Range Image” on page 3-26

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)
```

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
- “Display an Image in a Figure Window” on page 4-4
- “Display Multiple Images” on page 4-10
- “View Thumbnails of Images in Folder or Datastore” on page 4-15
- “Interact with Images Using Image Viewer App” on page 4-32
- “Create and Open Reduced Resolution Files” on page 4-41
- “Explore Images with Image Viewer App” on page 4-42
- “Get Pixel Information in Image Viewer App” on page 4-49
- “Measure Distance Between Pixels in Image Viewer App” on page 4-57
- “Get Image Information in Image Viewer App” on page 4-61
- “Adjust Image Contrast in Image Viewer App” on page 4-63
- “Interactive Contrast Adjustment” on page 4-70
- “Crop Image Using Image Viewer App” on page 4-72
- “Explore 3-D Volumetric Data with Volume Viewer App” on page 4-77
- “View Image Sequences in Video Viewer App” on page 4-89
- “Convert Multiframe Image to Movie” on page 4-97
- “Display Different Image Types” on page 4-98
- “Add Colorbar to Displayed Image” on page 4-104
- “Print Images” on page 4-106
- “Manage Display Preferences” on page 4-107

Image Display and Exploration Overview

The Image Processing Toolbox software includes two display functions, `imshow` and `imtool`. Both functions work within the 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 “Display an Image in a Figure Window” 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`, opens 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 “Interact with Images Using Image Viewer App” on page 4-32.

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 graphics object 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.

Property	Indexed Images	Grayscale Images	Binary Images	Truecolor Images
`CData` (Image)	Set to the data in `X`	Set to the data in `I`	Set to data in `BW`	Set to data in `RGB`

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

Display an Image in a Figure Window

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.

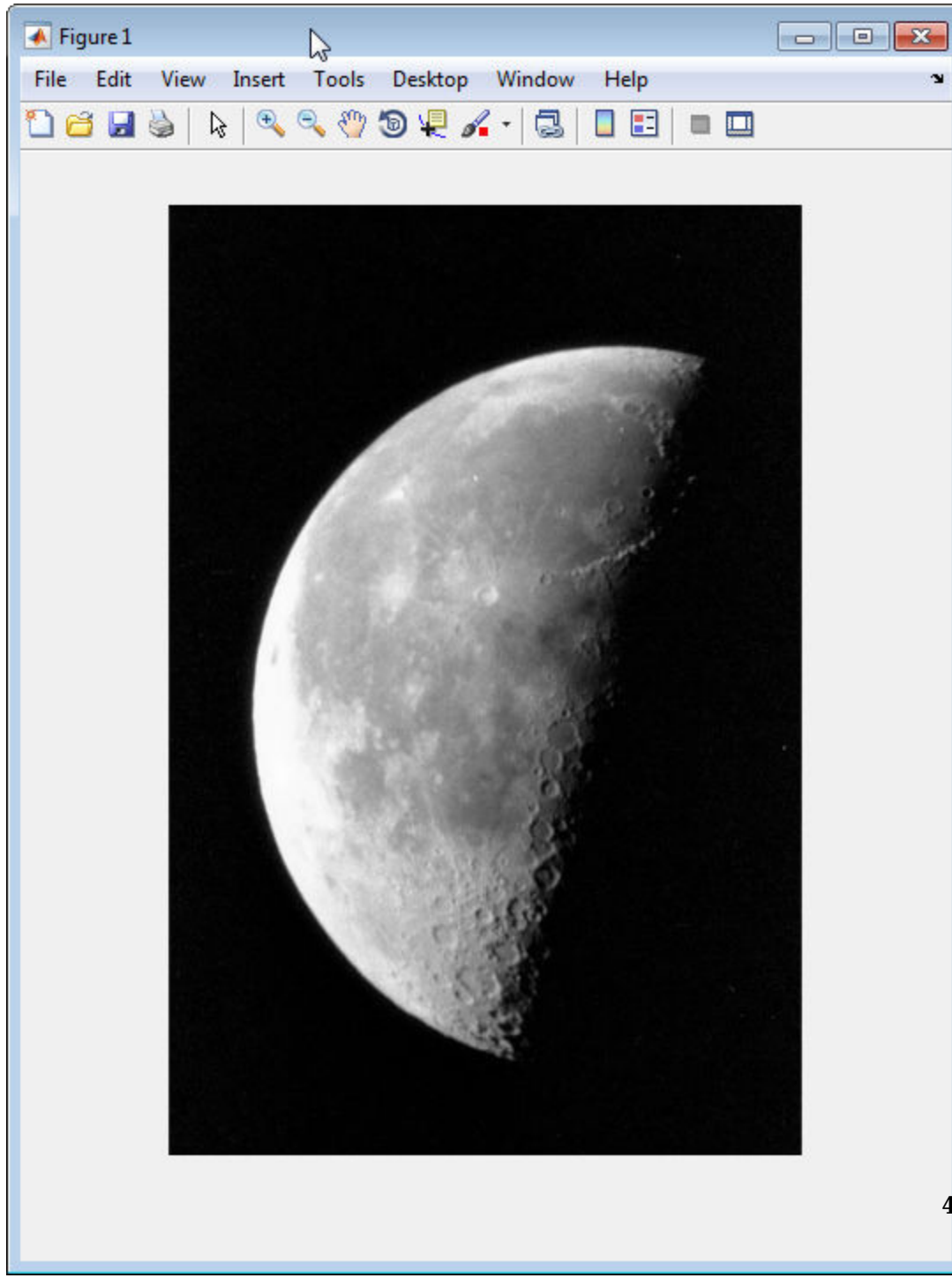


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help

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 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 4-98.

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 `'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 box 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

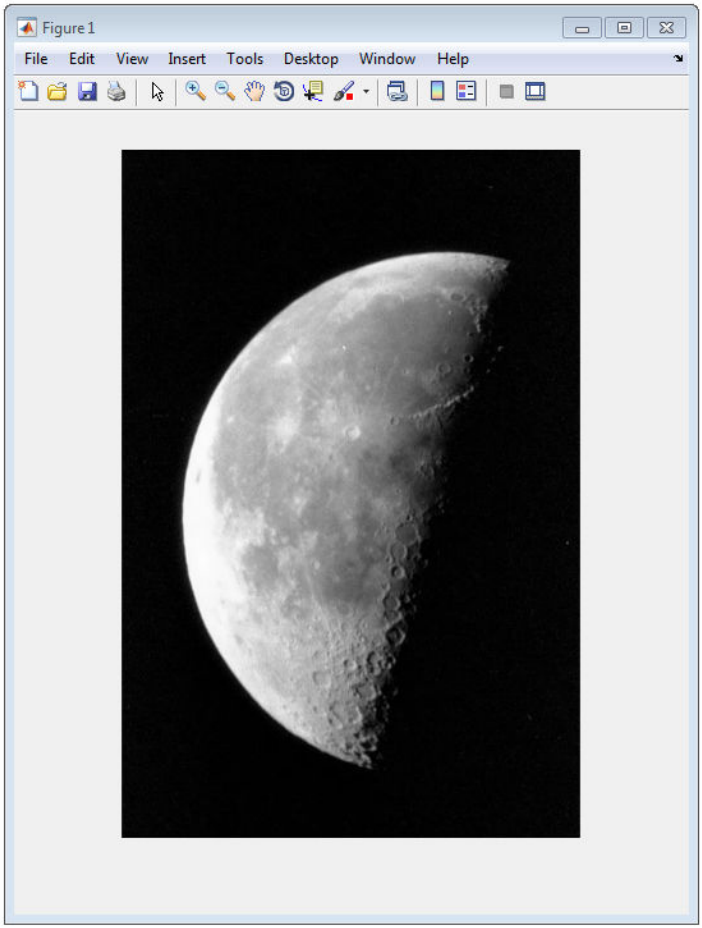
information about specifying interpolation methods, see “Resize an Image with imresize Function” on page 6-2.

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`.

See Also

More About

- “Display Multiple Images” on page 4-10

Display Multiple Images

This section describes various ways you can view multiple images at the same time.

Display Multiple Images in Separate Figure Windows

The simplest way to display multiple images at the same time 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. The following example views the first three frames in an array of grayscale images `I`.

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

Display Multiple Images in a Montage

You can view multiple images as a single image object in a figure window using the function. By default, the montage function scales the images, depending on the number of images and the size of your screen, and arranges them to form a square, preserving the aspect ratio of the original images. You can specify the size of the thumbnails using the parameter.

The images in the montage can be of different types and sizes. `montage` converts indexed images to RGB using the colormap present in the file.

By default, the `montage` function does not include any blank space between the images in the montage. You can specify the amount of blank space between the image using the `BorderSize` parameter. You can also specify the color of the space between images using the `BackgroundColor` parameter.

The following example shows how to view a sequence of images as a montage.

View Image Sequence as Montage

This example shows how to view multiple frames in a multiframe array at one time, using 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]);
```

Display Images Individually in the Same Figure

You can use the `imshow` function with the MATLAB `subplot` function to display multiple images in a single figure window. For additional options, see "What Is an Image Sequence?" on page 2-72.

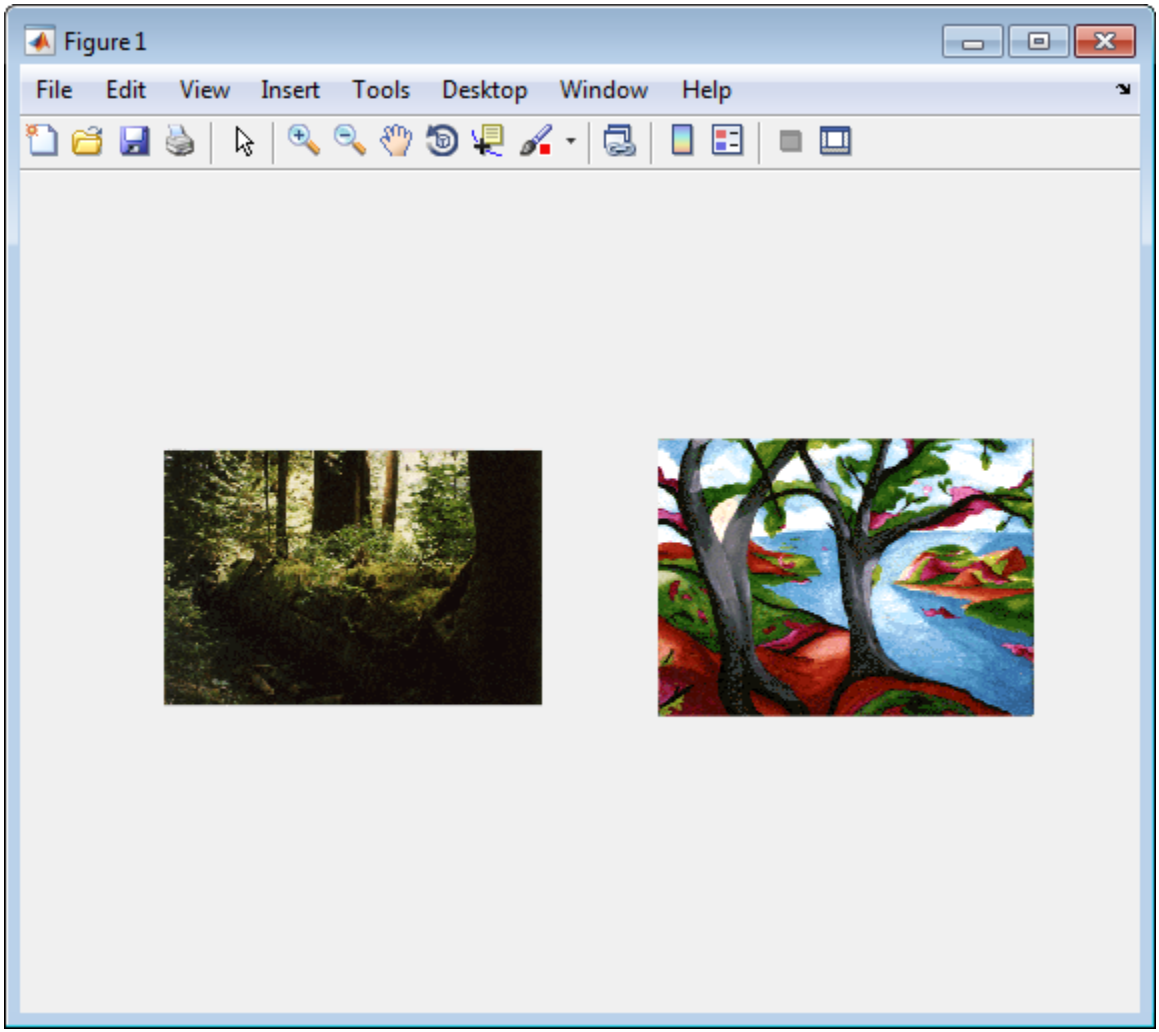
Note The Image Viewer app (`imtool`) does not support this capability.

Divide a Figure Window into Multiple Display Regions

`subplot` divides a figure into multiple display regions. Using the syntax `subplot(m,n,p)`, you define an `m`-by-`n` matrix of display regions and specify which region, `p`, is active.

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)
```



Compare a Pair of Images

The `imshowpair` function combines a pair of images in the same figure window to assist in the comparison of the images. `imshowpair` supports many visualization methods, including:

- falsecolor, in which the two images are overlaid in different color bands. Gray regions indicate where the images have the same intensity, and colored regions indicate where the image intensity values differ. RGB images are converted to grayscale before display in falsecolor.
- alpha blending, in which the intensity of the display is the mean of the two input images. Alpha blending supports grayscale and truecolor images.
- checkerboard, in which the output image consists of alternating rectangular regions from the two input images.
- the difference of the two images. RGB images are converted to grayscale.
- montage, in which the two images are displayed alongside each other. This visualization mode is similar to the display using the `montage` function.

`imshowpair` uses optional spatial referencing information to display the pair of images.

See Also

`bwskel`

More About

- “Operations That Combine Dilation and Erosion” on page 10-17

View Thumbnails of Images in Folder or Datastore

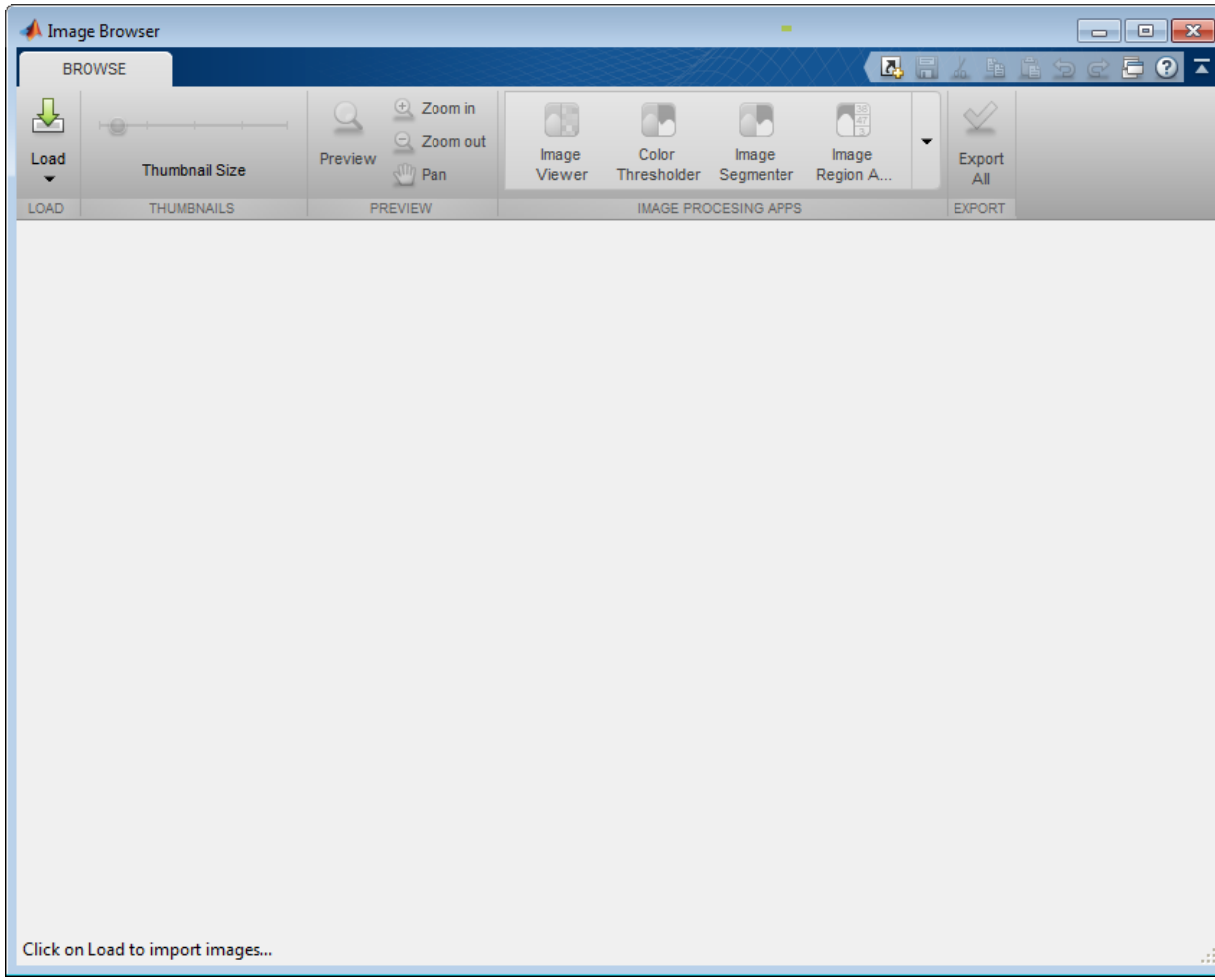
You can use the Image Browser app to view reduced-size versions, called thumbnails, of all the images in a folder or an image datastore.

- “View Thumbnails of Images in Folder” on page 4-15
- “View Thumbnails of Images in Image Datastore” on page 4-20
- “Remove Image from Image Browser and Create Datastore” on page 4-25

View Thumbnails of Images in Folder

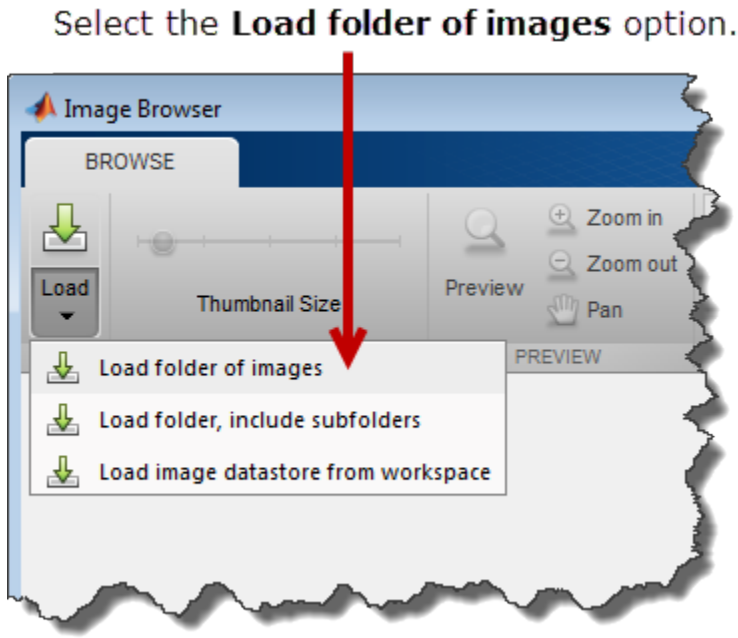
This example shows how to use the Image Browser to view thumbnails of all the images in a folder. The example looks in the Image Processing Toolbox sample image folder, `imdata`, for a binary image that would be suitable to use with the Image Region Analyzer app. The example shows how to and open the image in the Image Region Analyzer app from within the Image Browser app.

Open the Image Browser app. From the MATLAB Toolstrip, on the Apps tab, in the Image Processing and Computer Vision group, click **Image Browser** . You can also open the app at the command line using the `imageBrowser` command.

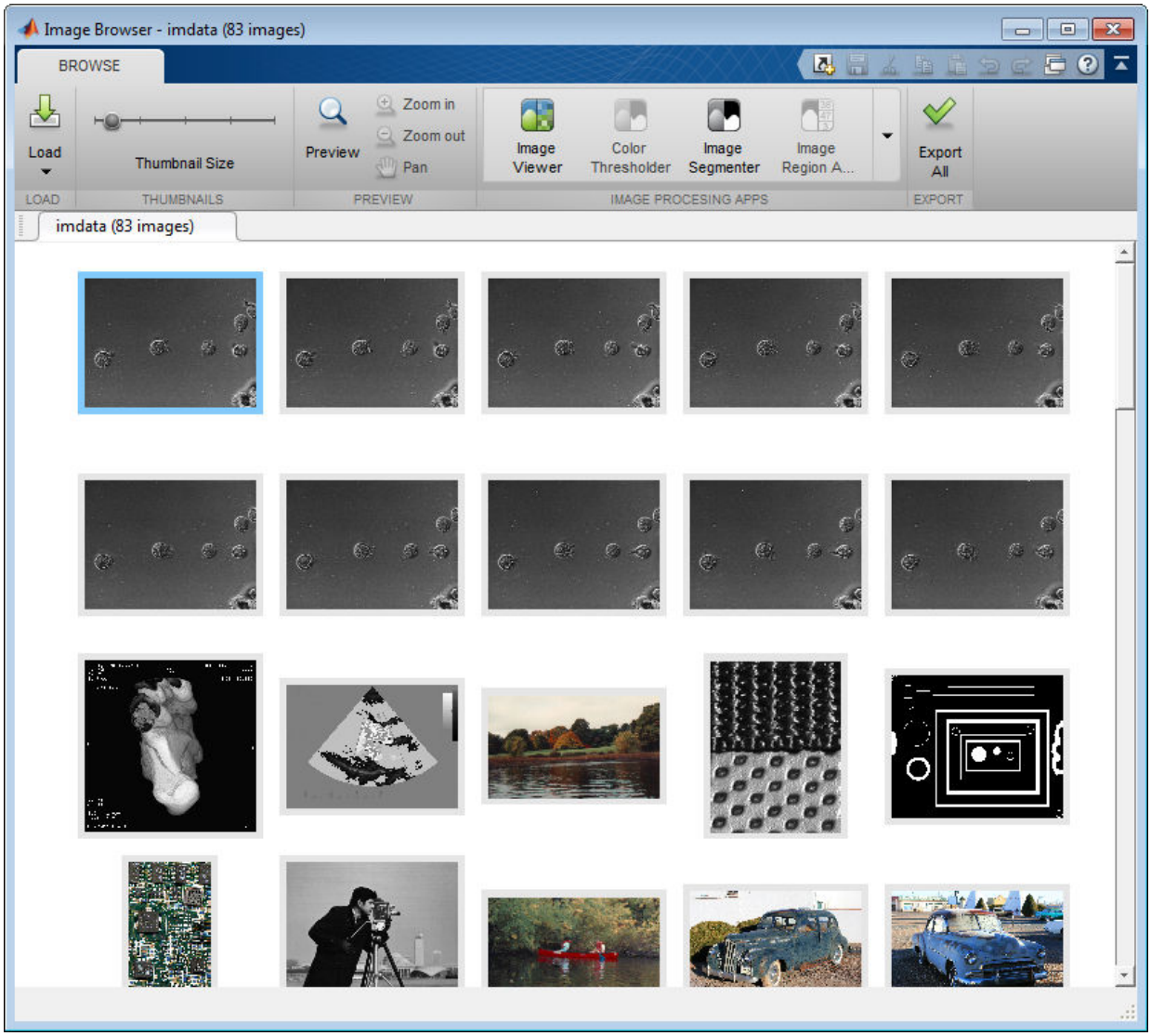


Load images into the app. Click **Load** and select **Load folder of images**. For this example, load all the images in the Image Processing Toolbox sample image folder. In the file selection dialog, navigate to the `imdata` folder. You can also specify a folder name when you open the Image Browser app at the command line.

```
imageBrowser(fullfile(matlabroot,'toolbox/images/imdata/'));
```

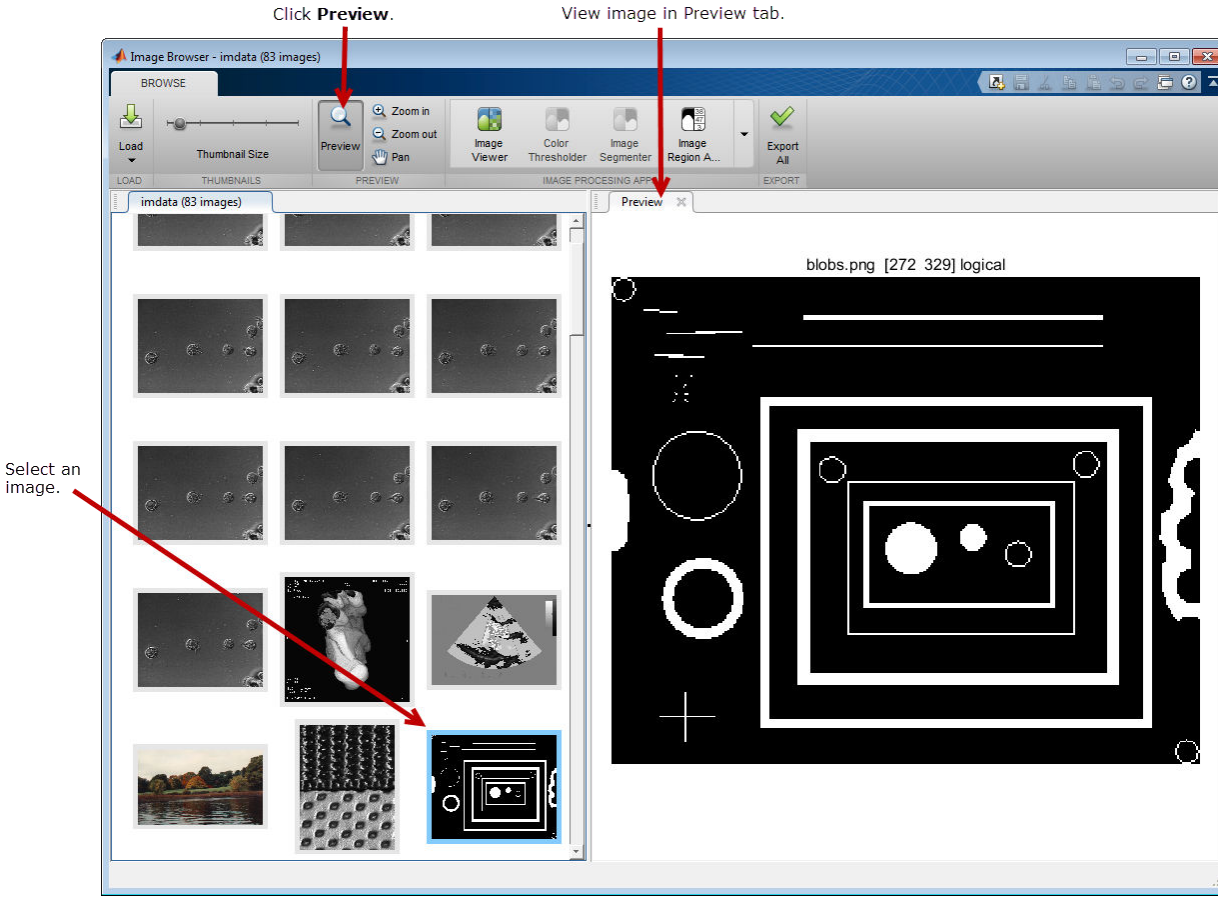


The Image Browser app displays thumbnails of all the images in the folder.

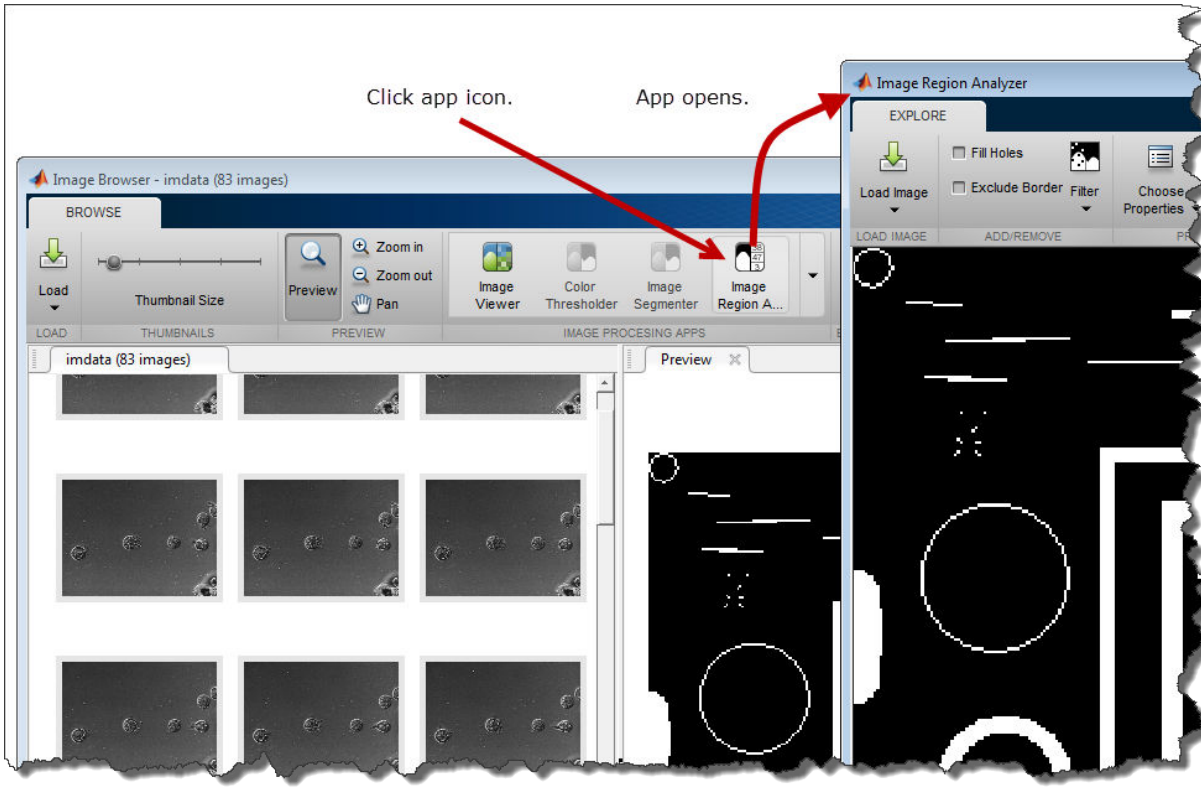


View thumbnails of the sample images. To see more detail, you can use the **Thumbnail Size** slider to make the thumbnails bigger. You can also view an individual image in a **Preview** tab. For this example, select the binary image that contains many different shapes and click **Preview**. The Image Browser displays the image at in a Preview tab. If

you want to explore the image further, you can use the zoom and pan options with the image in the Preview tab.



Open the selected image in the Image Region Analyzer app. Because you selected a binary image, the Image Browser app enables the Image Region Analyzer app icon in the Image Processing Apps section of the toolstrip. Click **Image Region Analyzer** to open the app with the image you selected.



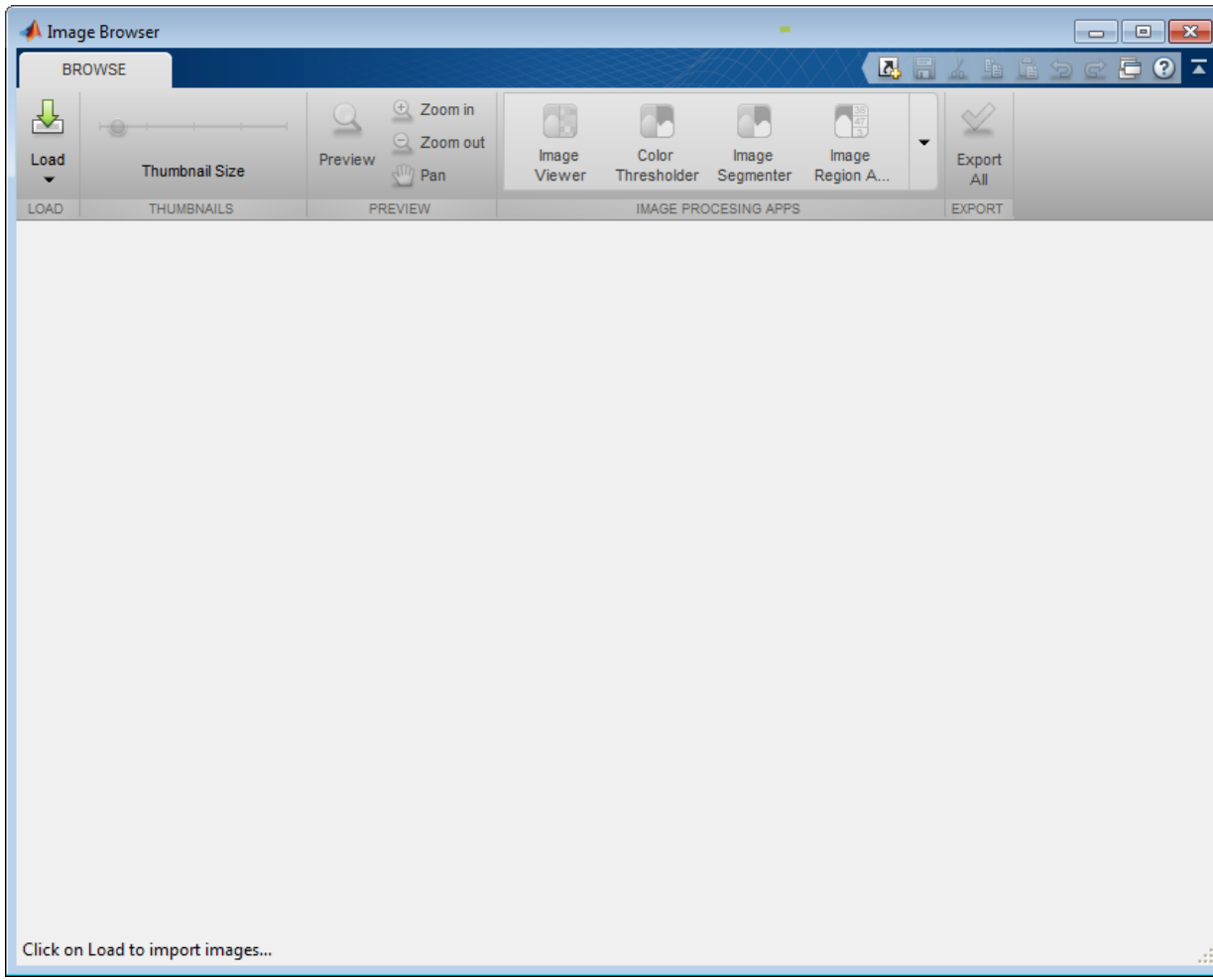
View Thumbnails of Images in Image Datastore

This example shows how to use the Image Browser app to view thumbnails of images in an image datastore and export one of images to the workspace. For more information about datastores, see “Getting Started with Datastore” (MATLAB).

First, create an image datastore. The example creates an image datastore that contains all the sample images included with the Image Processing Toolbox.

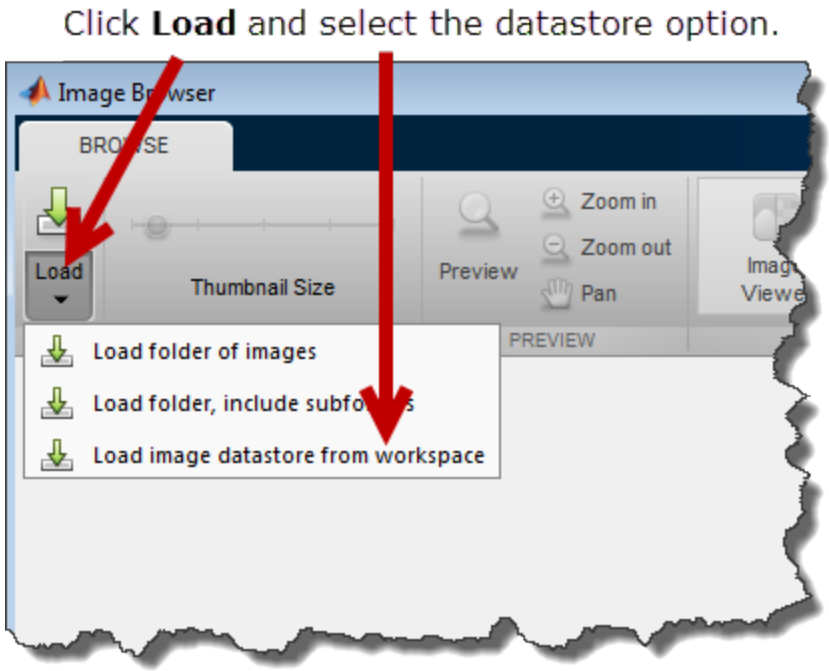
```
imds = imageDatastore(fullfile(matlabroot,'toolbox/images/imdata/'));
```

Open the Image Browser app. From the MATLAB Toolstrip, on the Apps tab, in the Image Processing and Computer Vision group, click **Image Browser** . You can also open the app at the command line using the `imageBrowser` command.

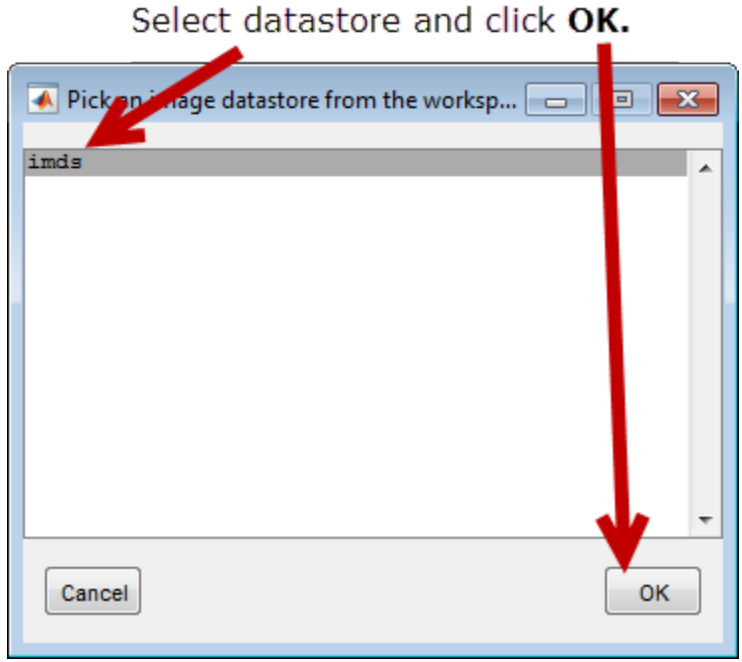


Load the images from the image datastore. Click **Load** and select the **Load image datastore from workspace** option. You can also specify the name of an image datastore in the workspace when you open the Image Browser app at the command line.

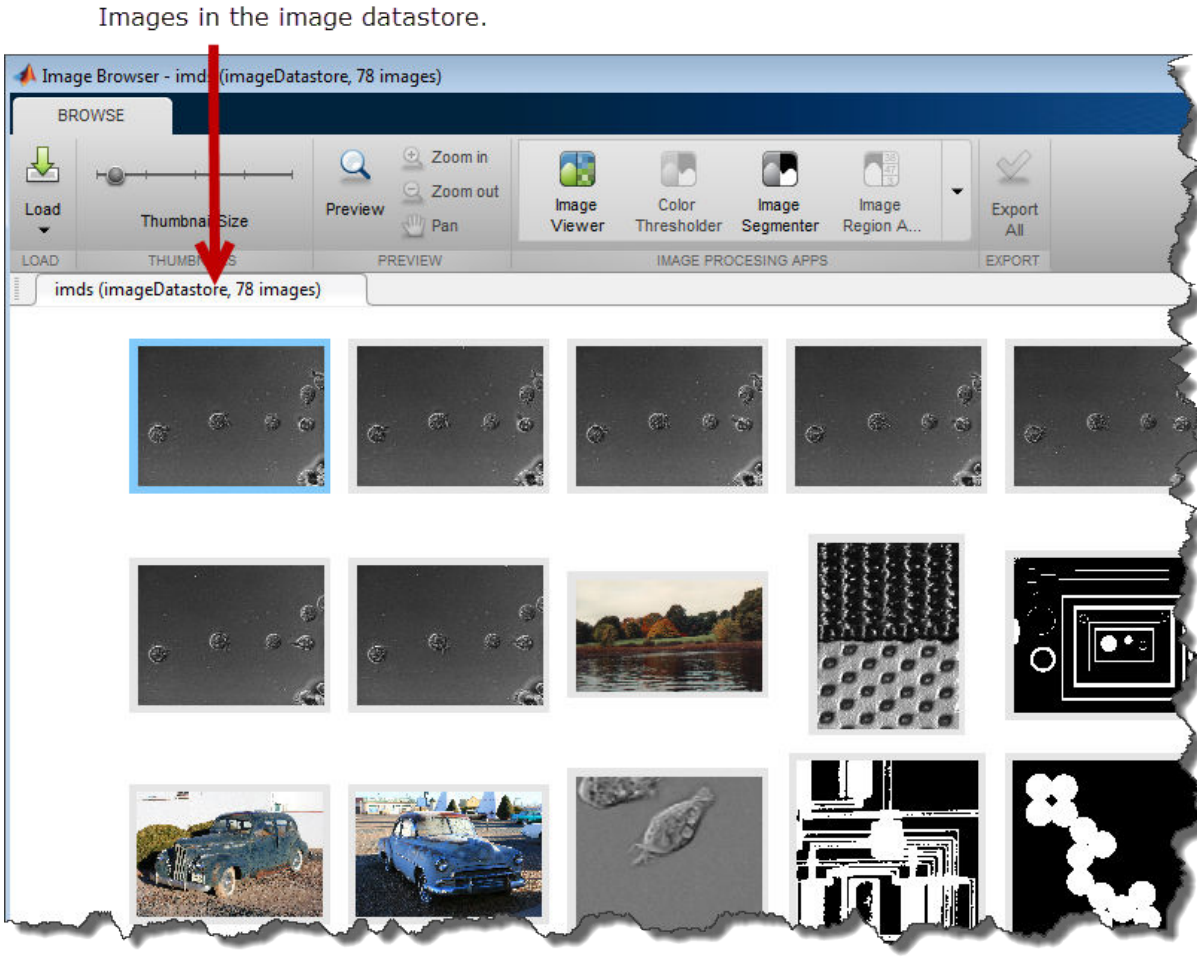
```
imageBrowser(imds);
```



In the **Pick an image datastore from the workspace** dialog box, select the image datastore and click **OK**.

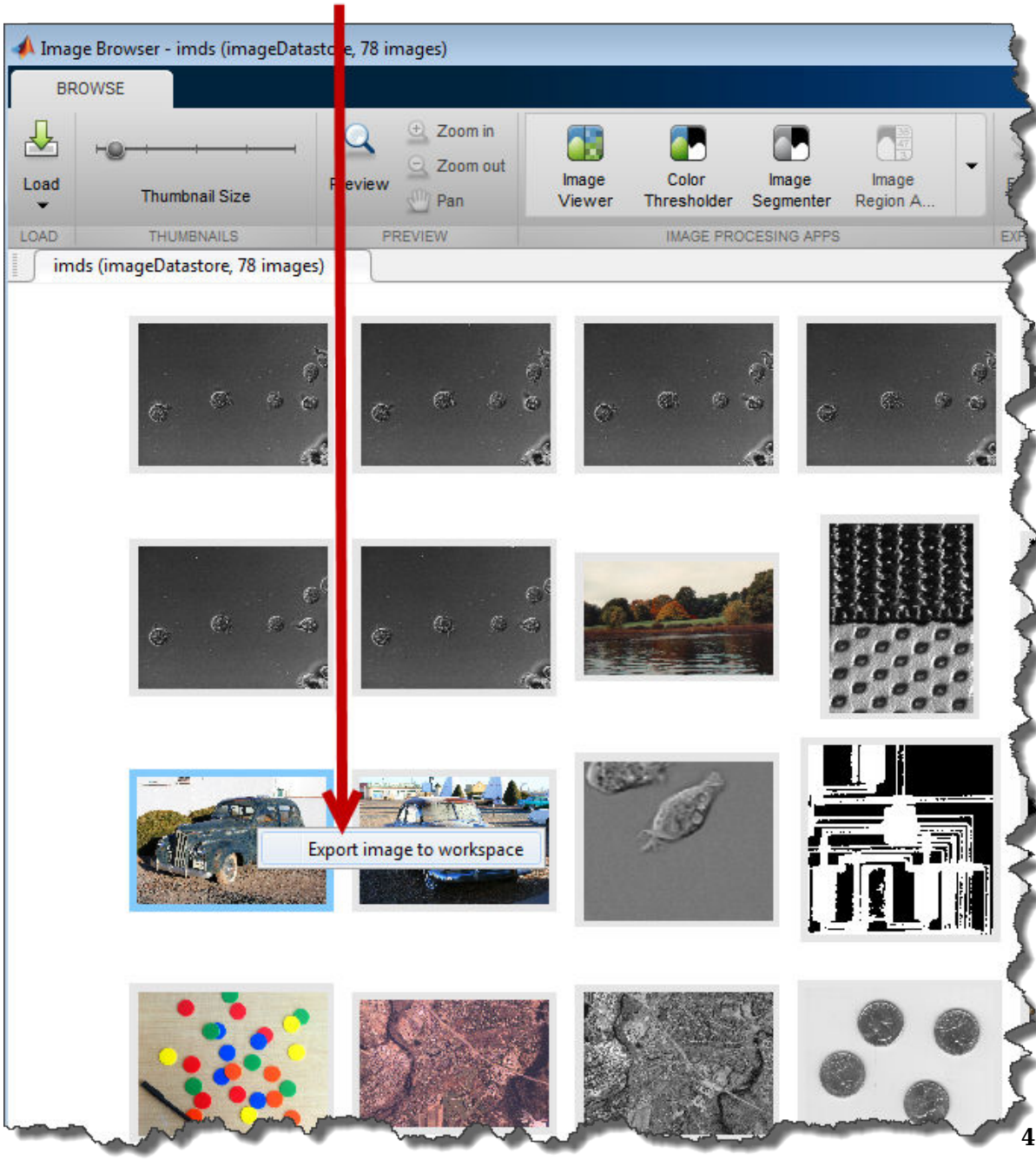


The Image Browser displays thumbnails of all the images in the image datastore.



Save one of the images in the datastore to the workspace. Right-click an image and choose the **Export image to workspace** option. In the Export to workspace dialog box, specify the variable name you want to use for the image.

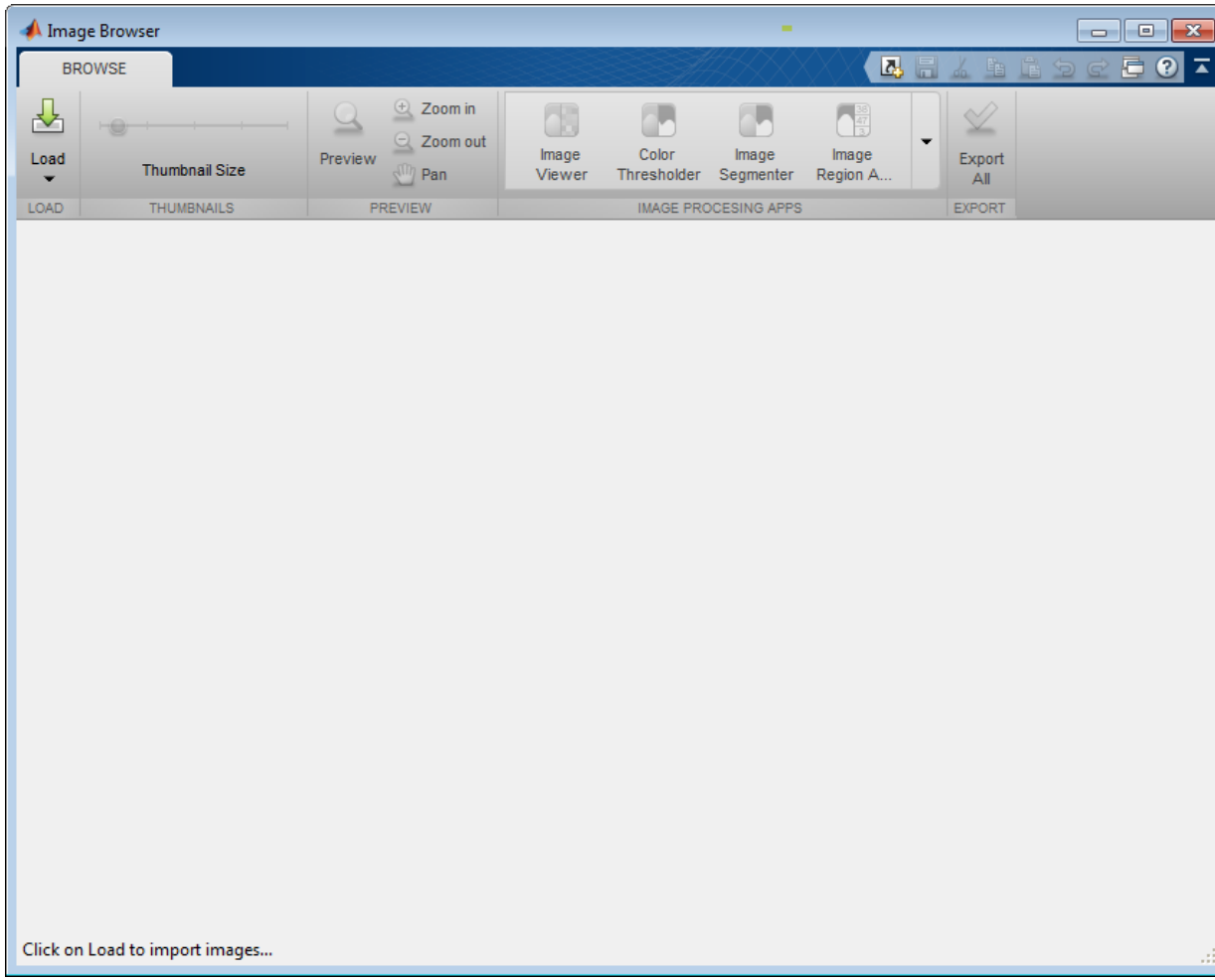
Right-click on an image and choose the **Export image to workspace** option.



Remove Image from Image Browser and Create Datastore

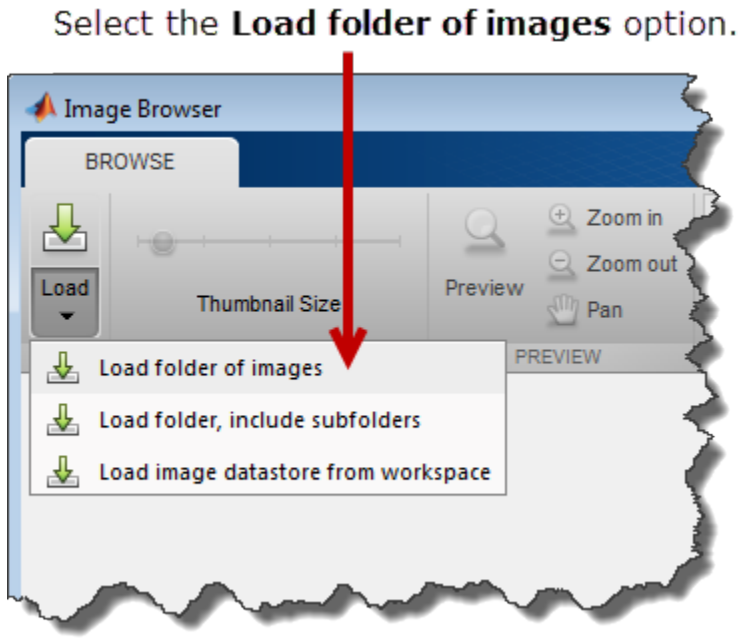
and then save the modified display in an image datastore. Removing images can be useful way of reducing clutter from the Image Browser app display when working with a folder of images that you are going to view in the app more than once. The Image Browser app does not delete the images from the file system—it only removes the thumbnail from the display.

Open the Image Browser app. From the MATLAB Toolstrip, on the Apps tab, in the Image Processing and Computer Vision group, click **Image Browser** . You can also open the app at the command line using the `imageBrowser` command.

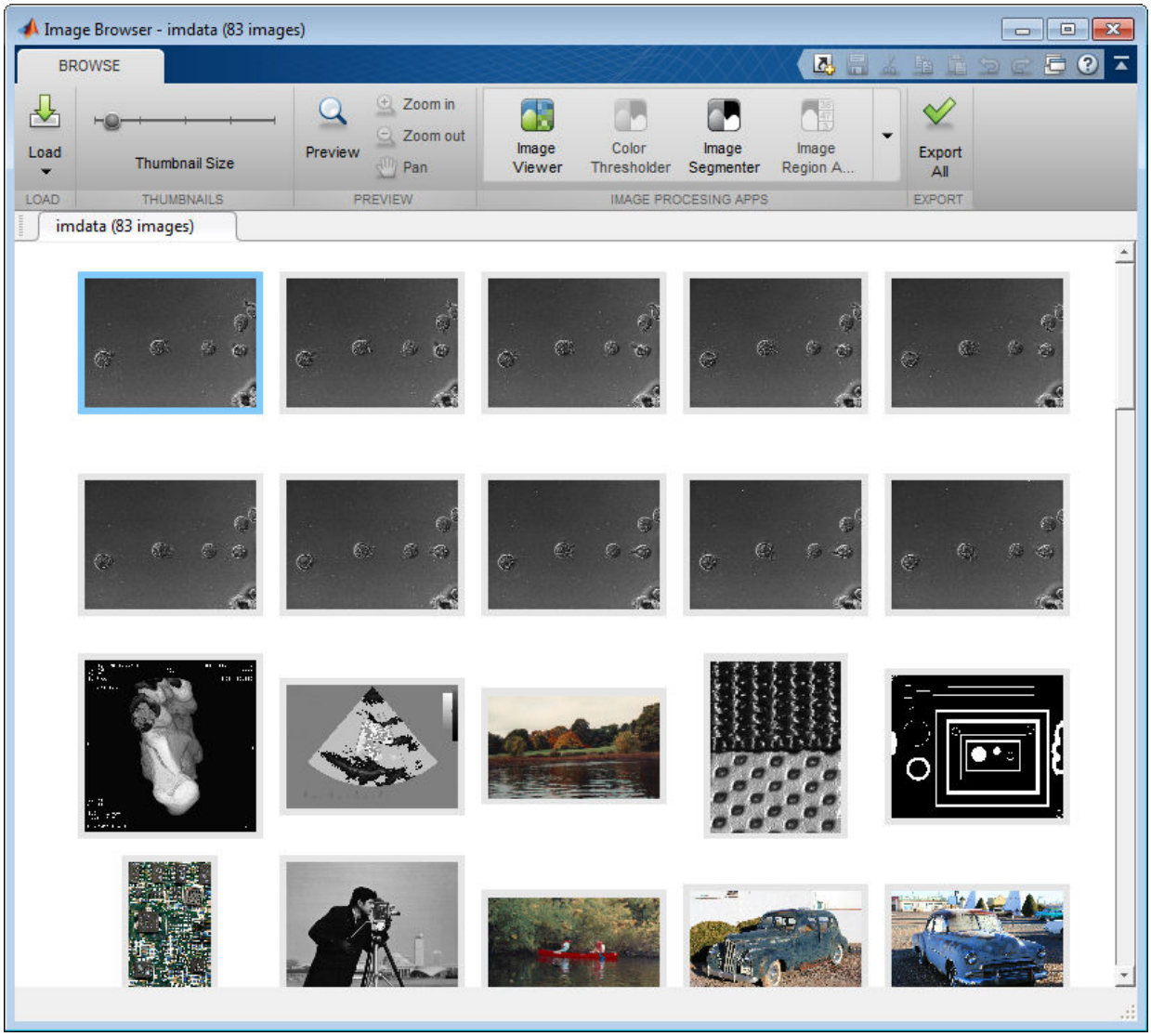


Load images into the app. Click **Load** and select **Load folder of images**. For this example, load all the images in the Image Processing Toolbox sample image folder. In the file selection dialog, navigate to the `imdata` folder. You can also specify a folder name when you open the Image Browser app at the command line.

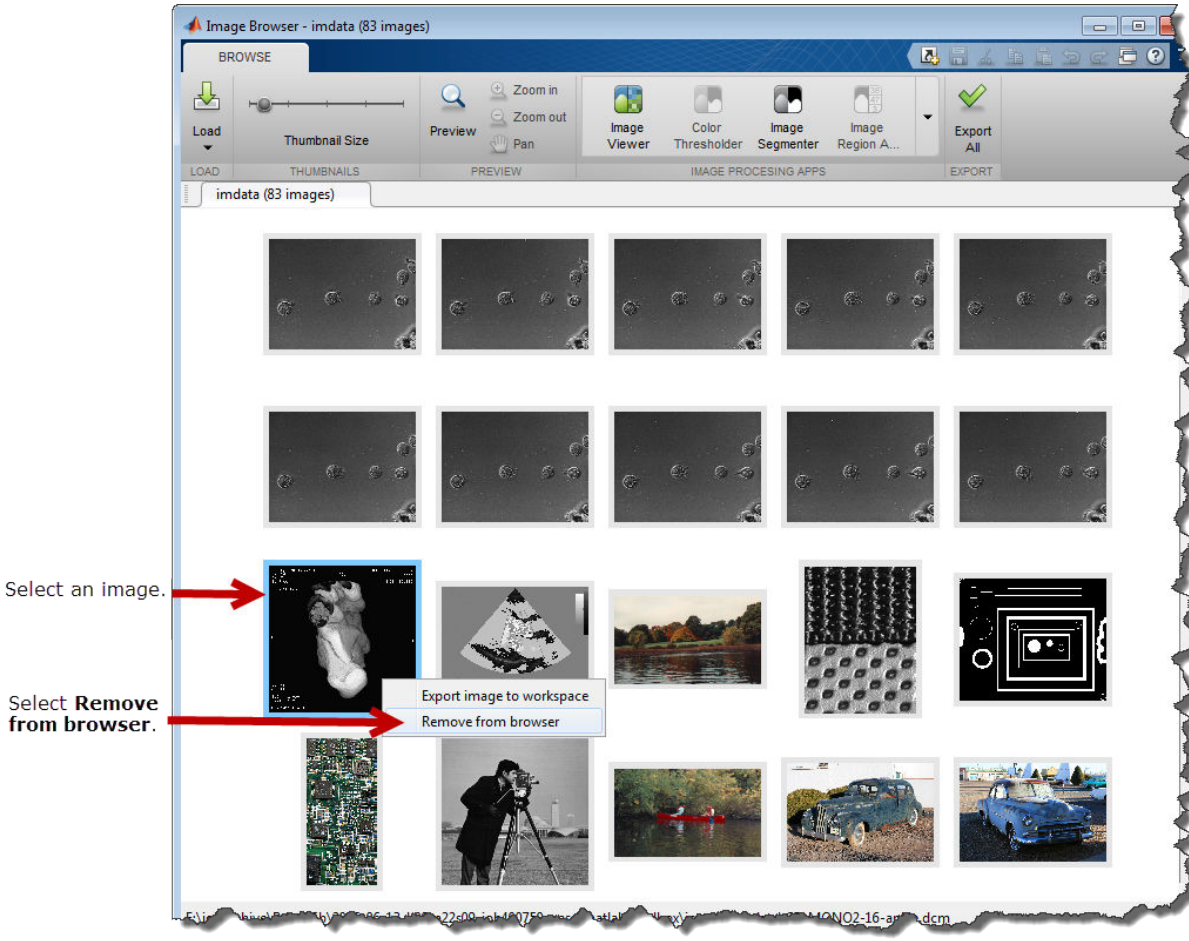
```
imageBrowser(fullfile(matlabroot,'toolbox/images/imdata/'));
```



The Image Browser app displays thumbnails of all the images in the folder.

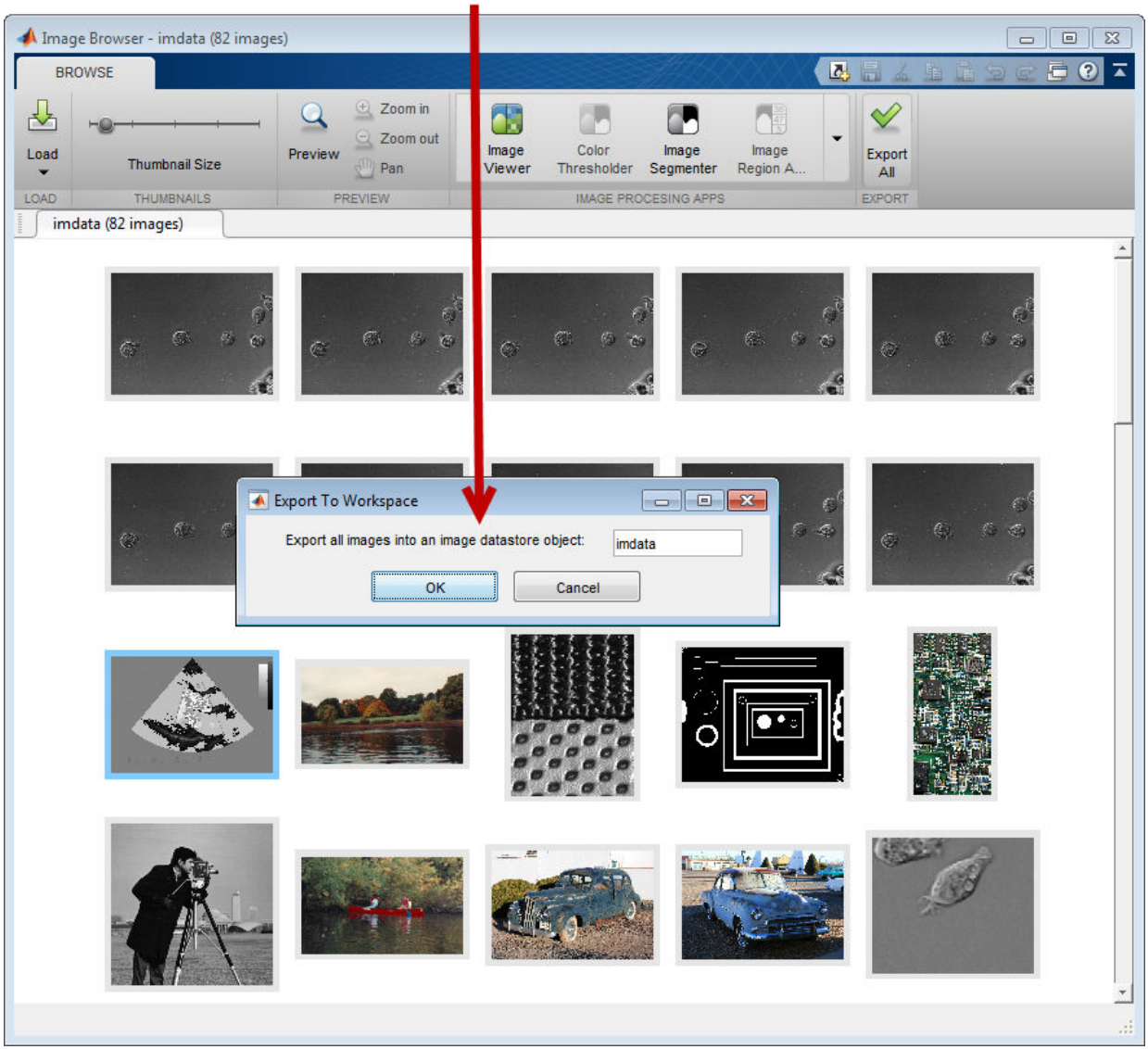


Right-click an image that you want to remove from the Image Browser app display. To remove it, choose the **Remove from browser** option on Image Browser app context menu. The Image Browser app deletes the thumbnail from the display.



Save the modified set of images displayed in the Image Browser in an image datastore. Click **Export All** and specify the variable name you want use for the datastore in the workspace. When you open this image datastore in the future, the Image Browser app displays only the images you saved in the image datastore.

Specify a name for the image datastore and click **OK.**



Interact with Images Using Image Viewer App

In this section...

"Open the Image Viewer App" on page 4-33
"Initial Image Magnification in the Image Viewer App" on page 4-34
"Choose the Colormap Used by the Image Viewer App" on page 4-35
"Import Image Data from the Workspace into the Image Viewer App" on page 4-36
"Export Image Data from the Image Viewer App to the Workspace" on page 4-37
"Save Image Data Displayed in Image Viewer" on page 4-38
"Close the Image Viewer App" on page 4-39
"Print Images Displayed in Image Viewer App" on page 4-40

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:

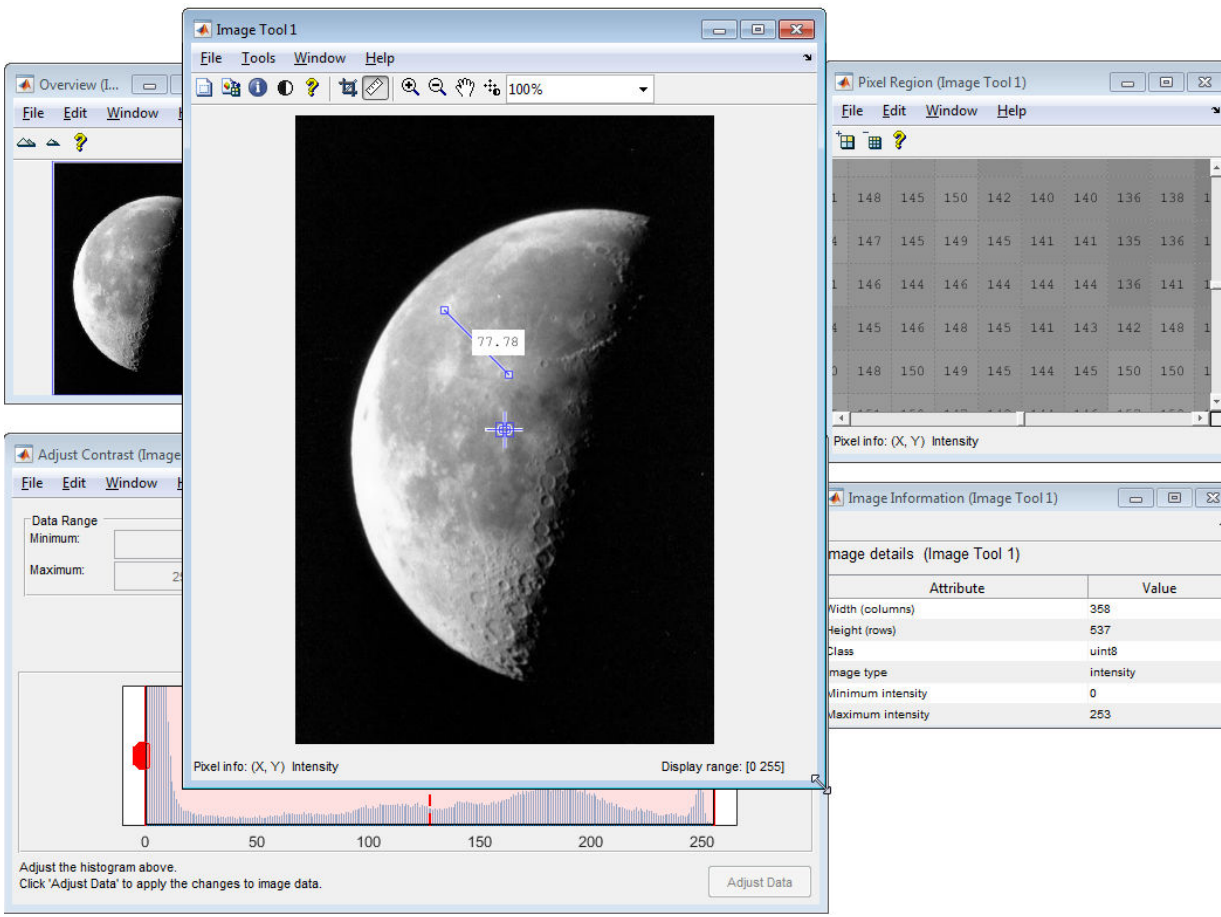
- Pixel Information tool — for getting information about the pixel under the pointer
- Pixel Region tool — for getting information about a group of pixels
- Distance tool — for measuring the distance between two pixels
- Image Information tool — for getting information about image and image file metadata
- 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:

- 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.



Open 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 “Import Image Data from the Workspace into the Image Viewer App” on page 4-36.

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 “Export Image Data from the Image Viewer App to the Workspace” on page 4-37.

Initial Image Magnification in the Image Viewer App

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 `'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 Toolbox Preferences panel. To open the preferences panel, call `iptprefs` or select **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 about specifying an interpolation method, see “Resize an Image with imresize Function” on page 6-2.

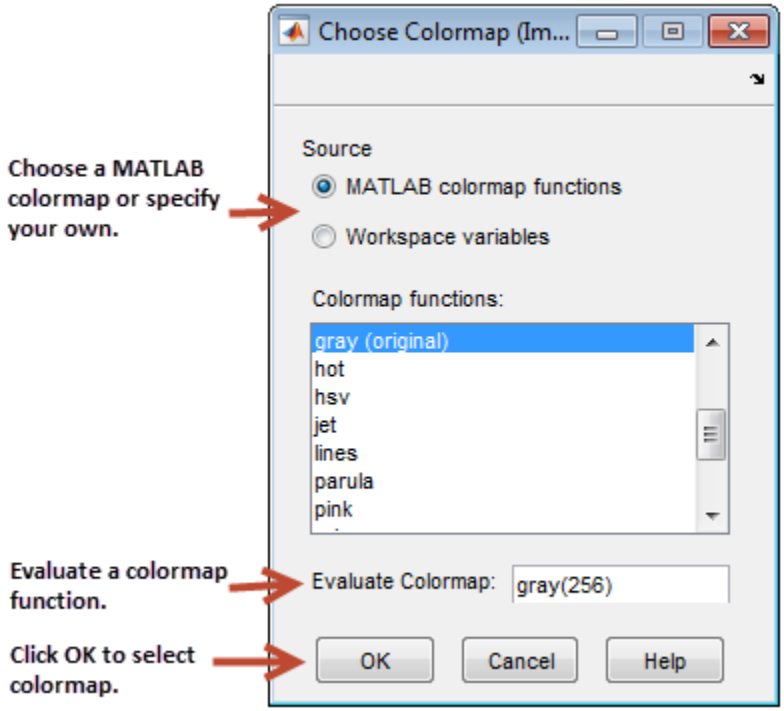
Choose the Colormap Used by the Image Viewer App

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.

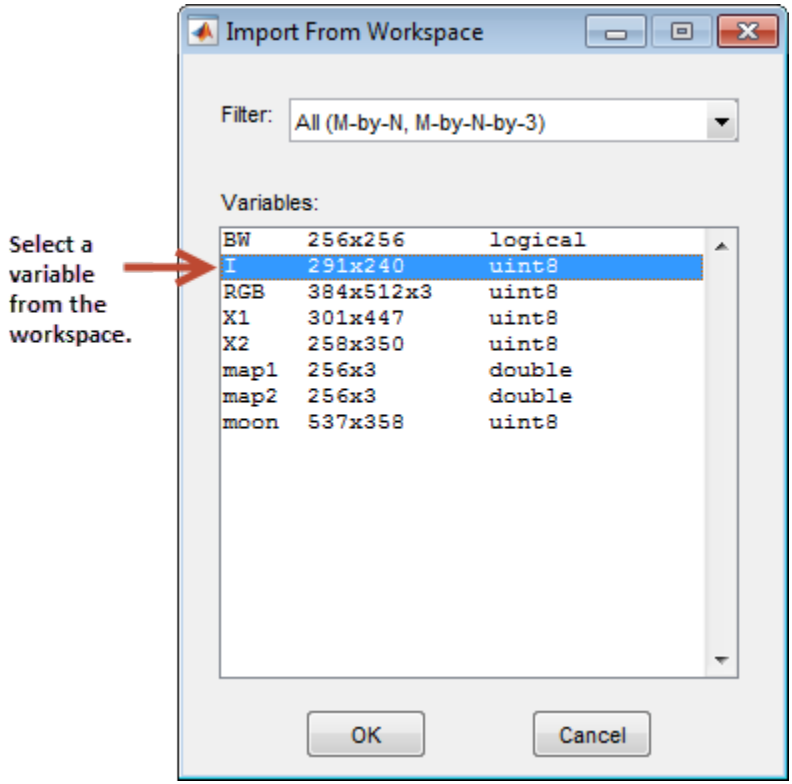
When you choose a colormap, the image updates to use the new map. If you click **OK**, the Image Viewer applies the colormap and closes the Choose Colormap tool. If you click **Cancel**, the image reverts to the previous colormap.



Import Image Data from the Workspace into the Image Viewer App

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.

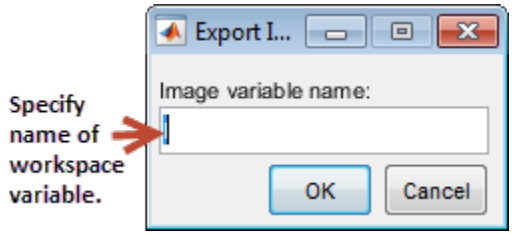
The following figure shows the Import from Workspace dialog box. You can use the Filter menu to limit the images included in the list to certain image types, i.e., binary, indexed, intensity (grayscale), or truecolor.



Export Image Data from the Image Viewer App 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 pre-fills 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

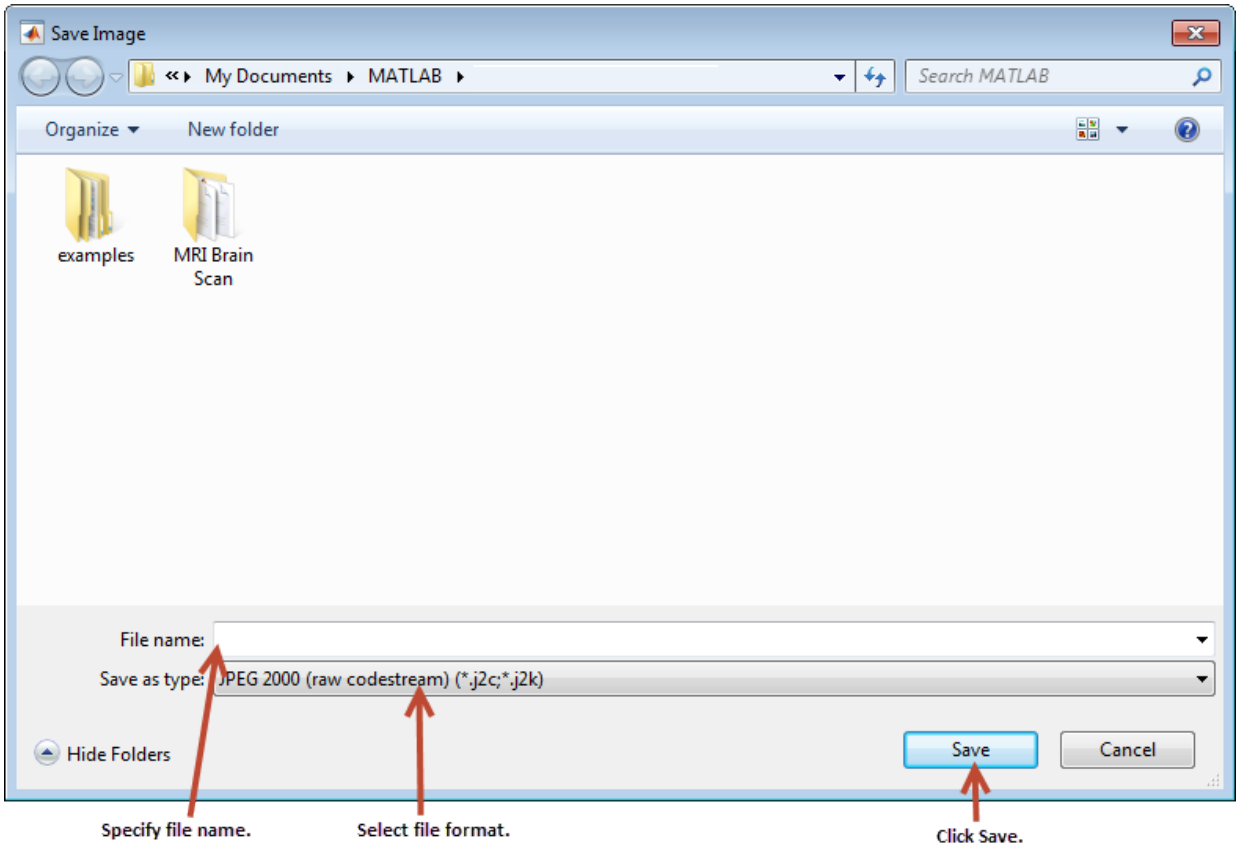
You can also use the `getimage` function to bring image data from the Image Viewer into the MATLAB workspace.

The `getimage` function retrieves the image data (`CData`) from the current image object. You must use the toolbox function `imgca` to get the image object 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);
```

Save 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`.

Close 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 the figure object that contains the app. You can use this object 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 its component graphics 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.

Print 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 4-106 for more information.

Create and Open Reduced Resolution Files

To create a reduced resolution (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 “Perform Block Processing on Image Files in Unsupported Formats” on page 16-17 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.

Open a Reduced Resolution File

To open a reduced resolution (R-Set) file, use the `imtool` function.

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

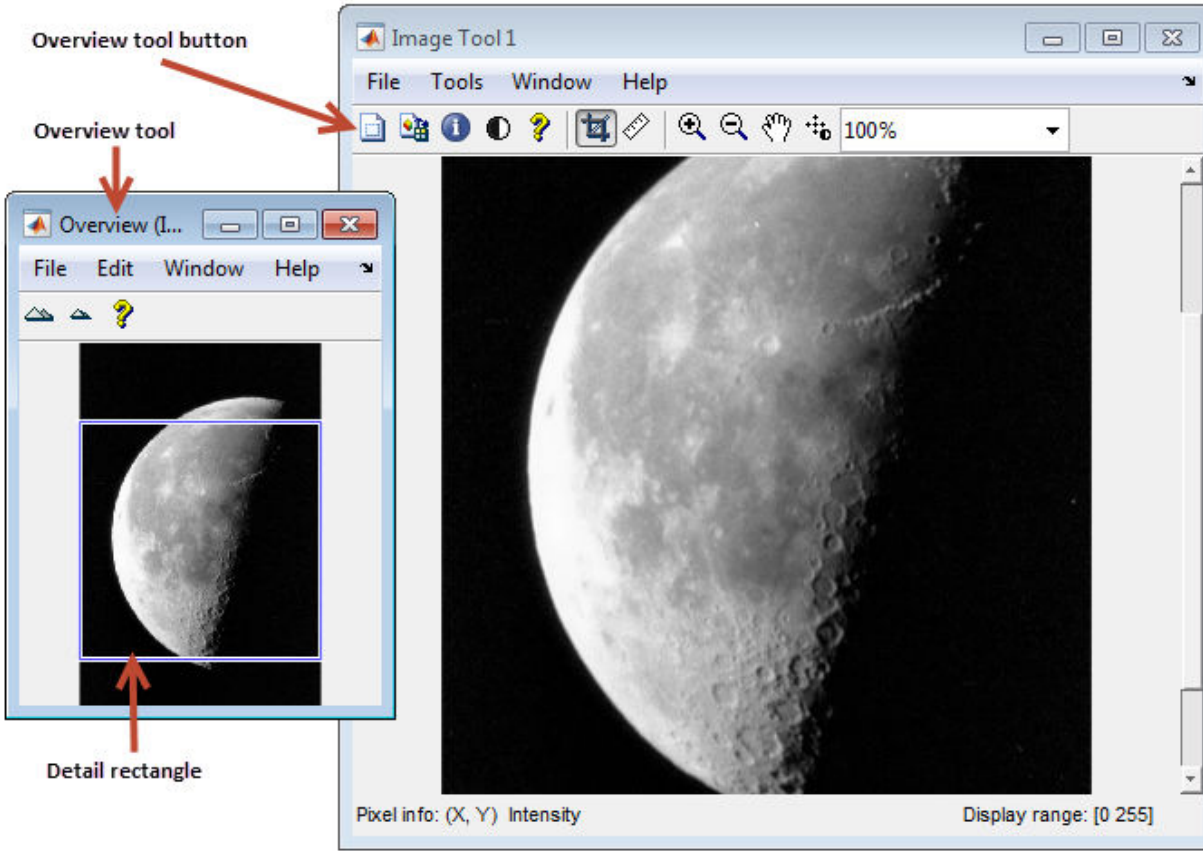
You can also open an R-Set file using the `openrset` function.

Explore Images with Image Viewer App

In this section...
“Explore Images Using the Overview Tool” on page 4-42 “Pan Images Displayed in Image Viewer App” on page 4-45 “Zoom Images in the Image Viewer App” on page 4-45 “Specify Image Magnification in Image Viewer” on page 4-46

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 4-44
- “Moving the Detail Rectangle to Change the Image View” on page 4-44
- “Specifying the Color of the Detail Rectangle” on page 4-44
- “Getting the Position and Size of the Detail Rectangle” on page 4-44
- “Printing the View of the Image in the Overview Tool” on page 4-45

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 4-106 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.

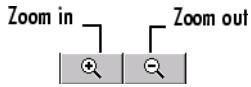
1. Click the **Pan** tool button in the toolbar or select **Pan** from the **Tools** menu. When the Pan tool is active, a check-mark appears next to the Pan selection in the menu.
2. Move the pointer over the image in the Image Viewer, using the mouse. The pointer changes to an open-hand shape .
3. Press 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 4-42.

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 4-46 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 check-mark appears next to the appropriate Zoom selection in the menu.

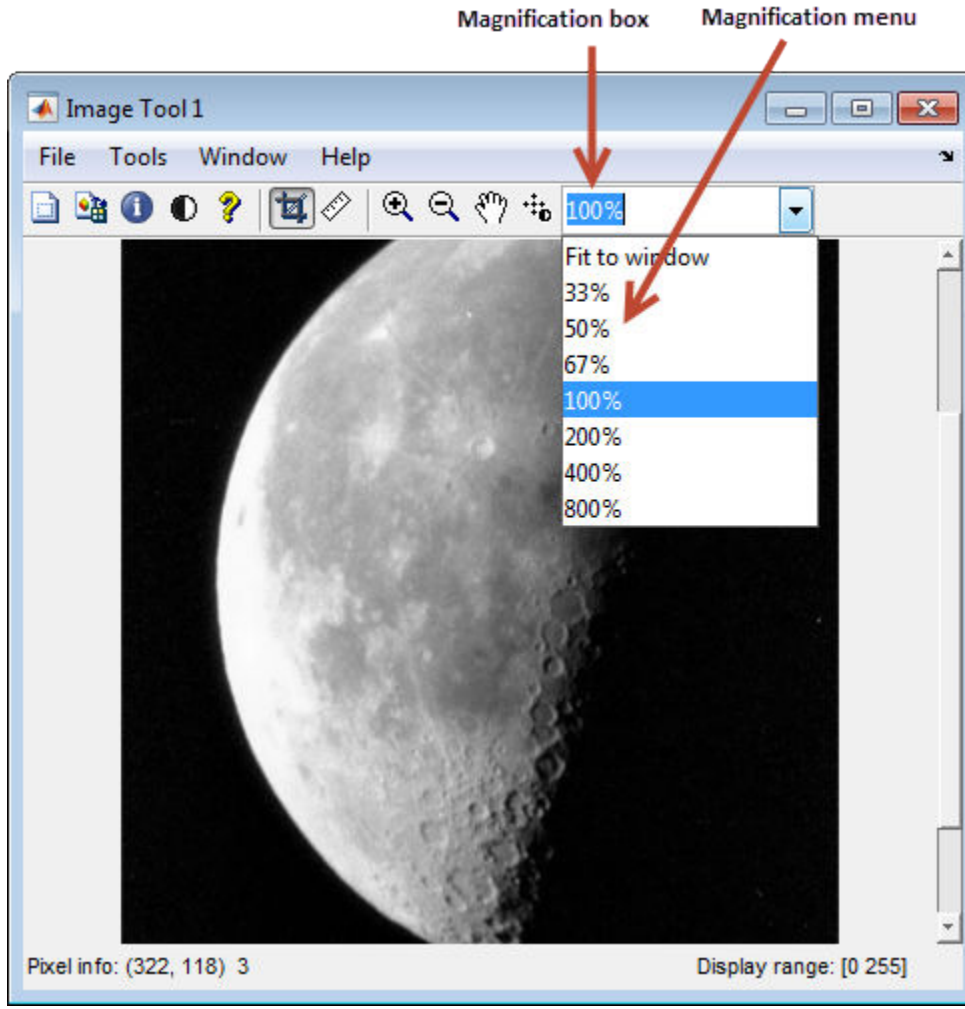


2. Move the pointer over the image you want to zoom in or out on, using the mouse. The pointer changes to the appropriate magnifying glass icon. With each click, the Image Viewer changes the magnification of the image, centering the new view of the image on the spot where you clicked.

   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 4-45 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 4-49 “Determine Pixel Values in an Image Region” on page 4-51 “Determine Image Display Range in Image Viewer” on page 4-54

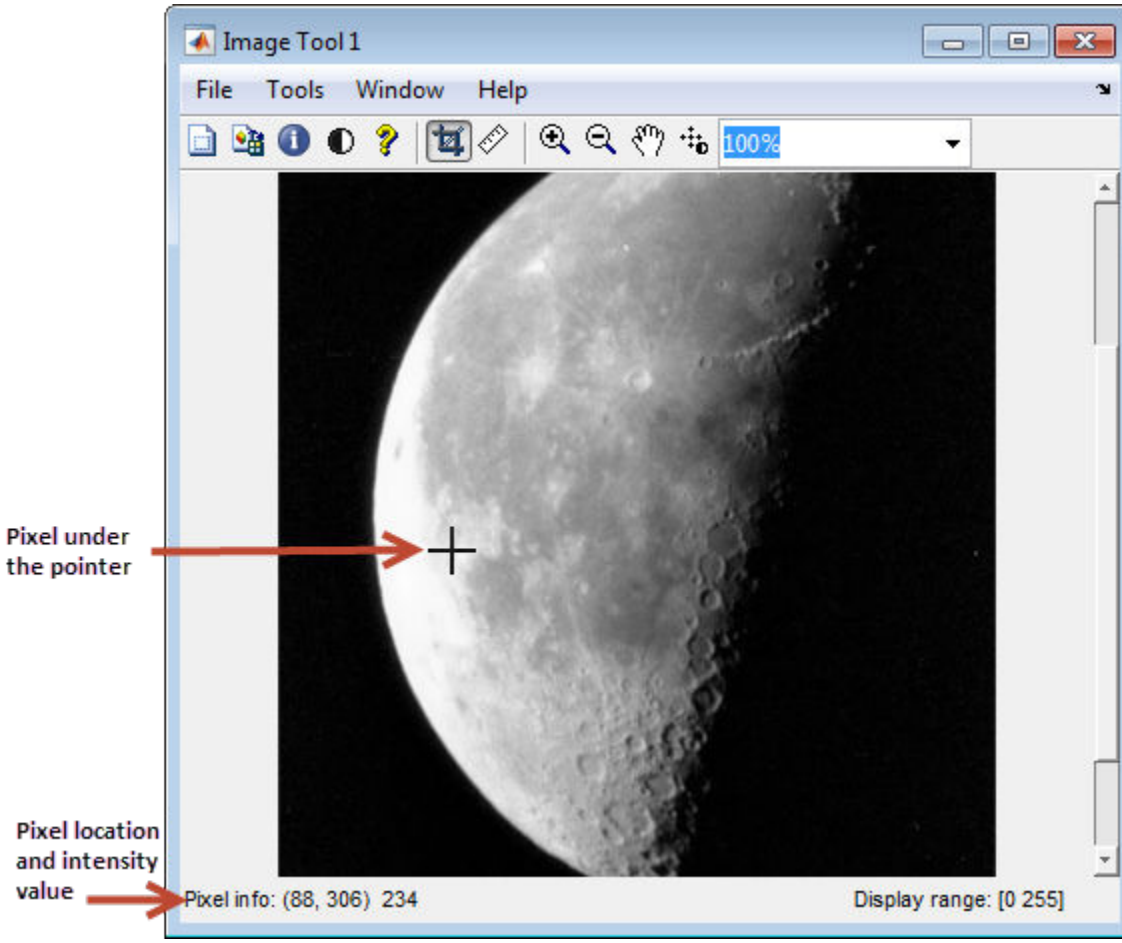
Determine Individual Pixel Values in Image Viewer

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 4-50.



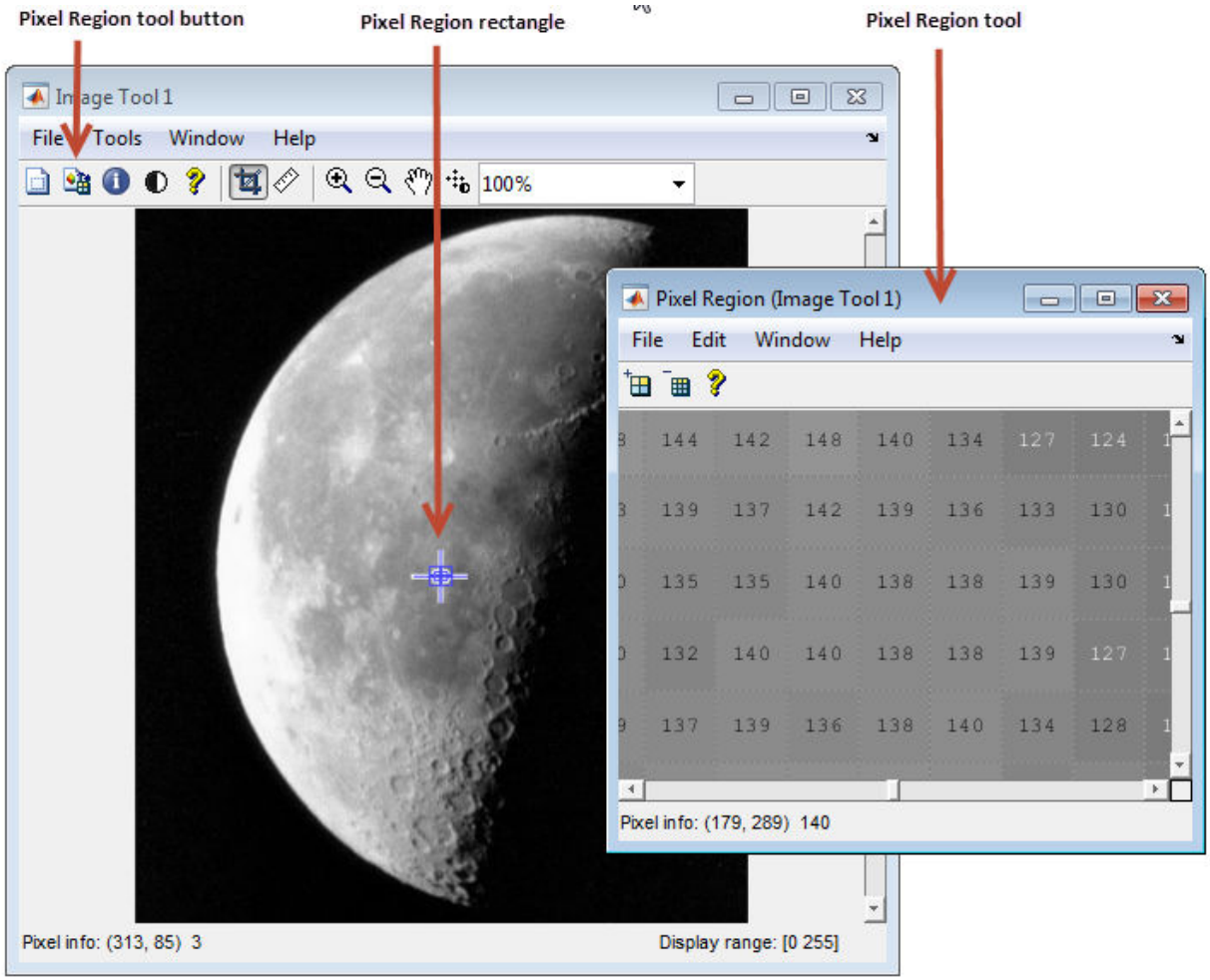
Saving the Pixel Value and Location Information

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 4-52
- "Customizing the View" on page 4-52
- "Determining the Location of the Pixel Region Rectangle" on page 4-53
- "Printing the View of the Image in the Pixel Region Tool" on page 4-54

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 4-53.

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.

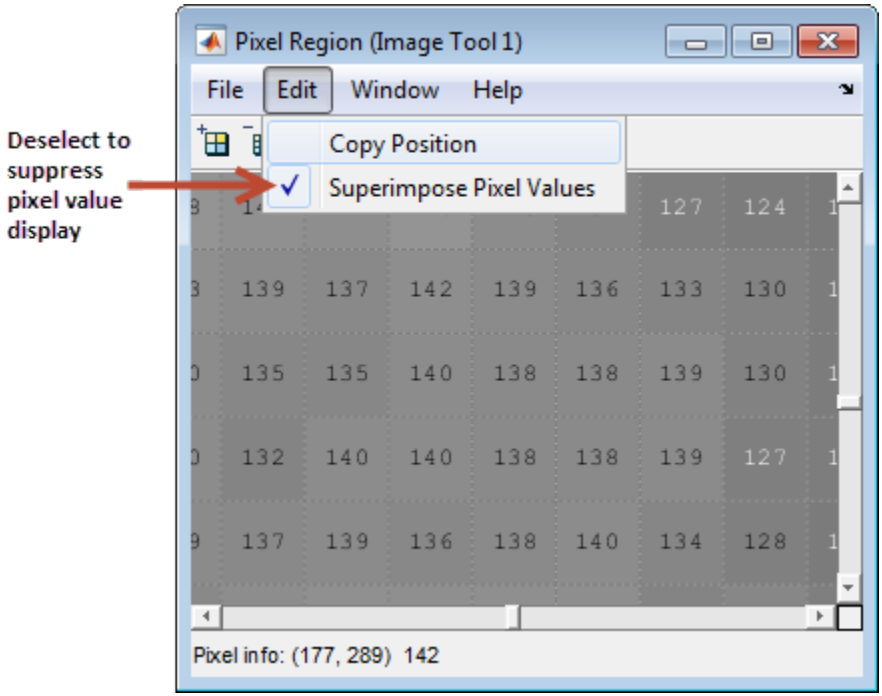
Customizing the View

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

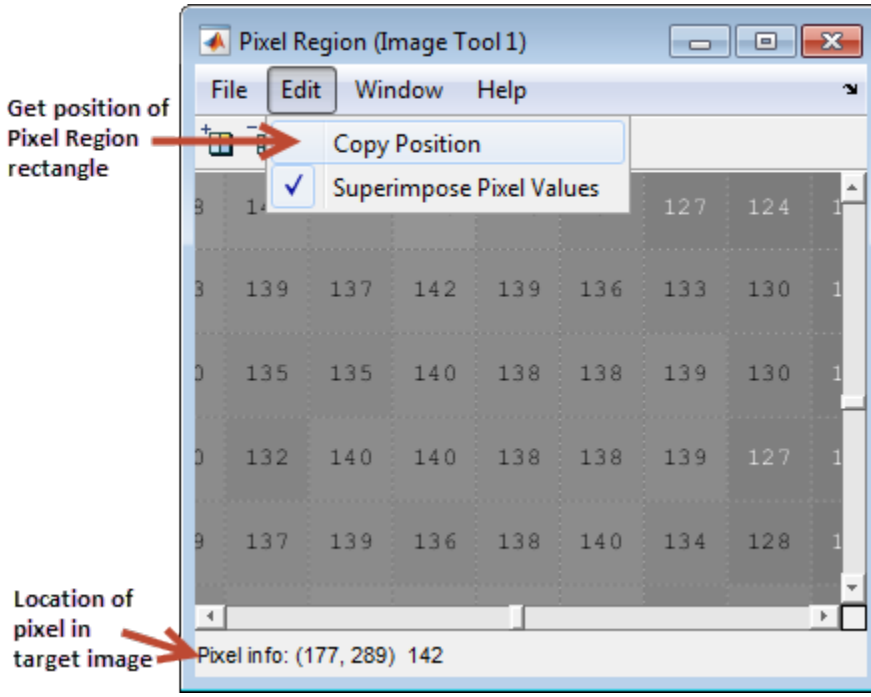
To determine the current location of the pixel region in the target image, you can use the pixel information given at the bottom of the tool. This information includes the *x*- and *y*-coordinates of pixels in the target image coordinate system. When you move the pixel region rectangle over the target 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 4-106 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 `imdisplayrange` from the command line.)

For example, view an image in the Image Viewer.

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

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

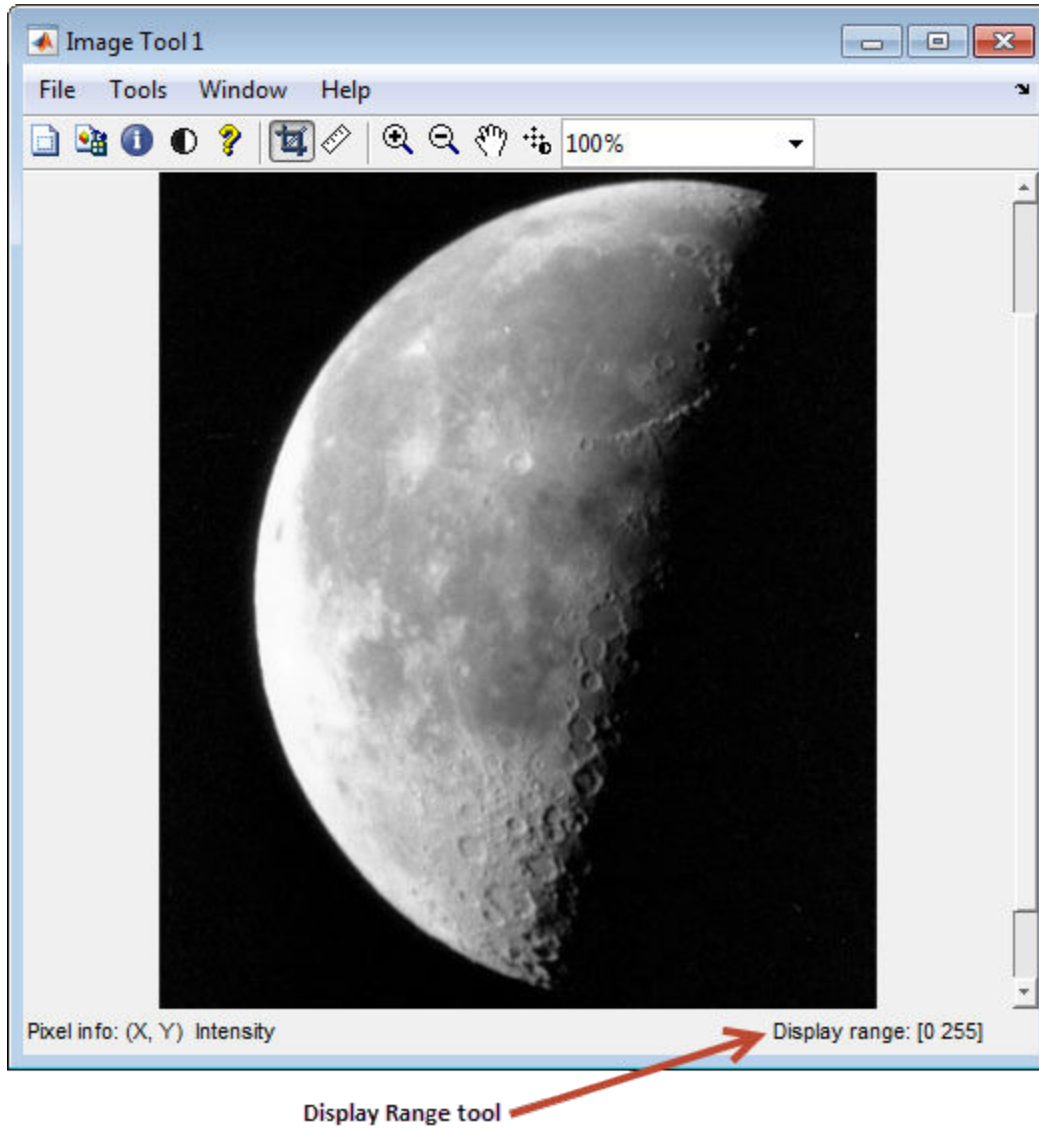


Image Tool 1
File
Tools
Window
Help
100%
Pixel info: (X, Y) Intensity
Display range: [0 255]
Display Range tool

Measure Distance Between Pixels in Image Viewer App

In this section...

"Determine Distance Between Pixels Using Distance Tool" on page 4-57
"Export Endpoint and Distance Data" on page 4-59
"Customize the Appearance of the Distance Tool" on page 4-59

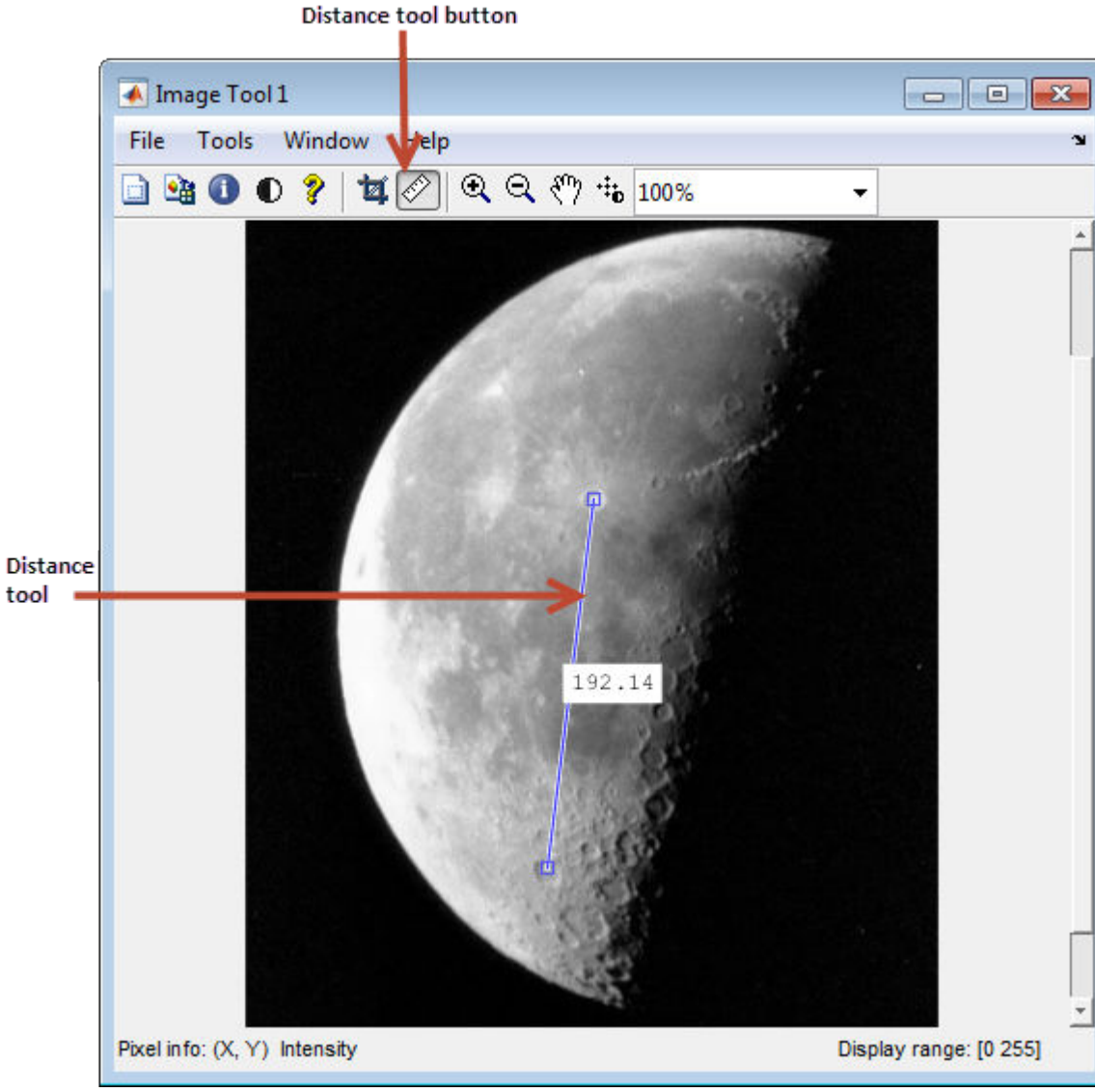
Determine Distance Between Pixels Using Distance Tool

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.

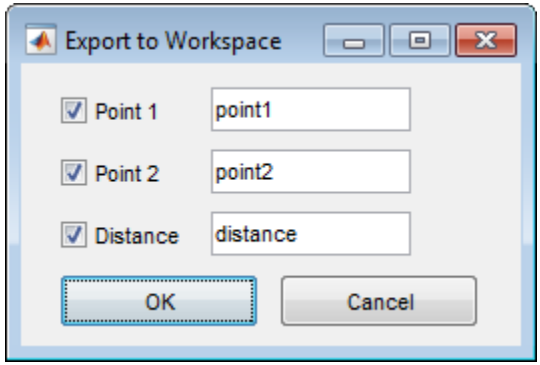
   The Distance tool displays the Euclidean distance between the two endpoints of the line in a label superimposed over the line. The tool specifies the distance in data units determined by the `XData` and `YData` properties, which is pixels, by default.



3 An alternative to using the Distance tool in Image Viewer is to open a figure window using `imshow` and then call `imdistline`. 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 file name 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.

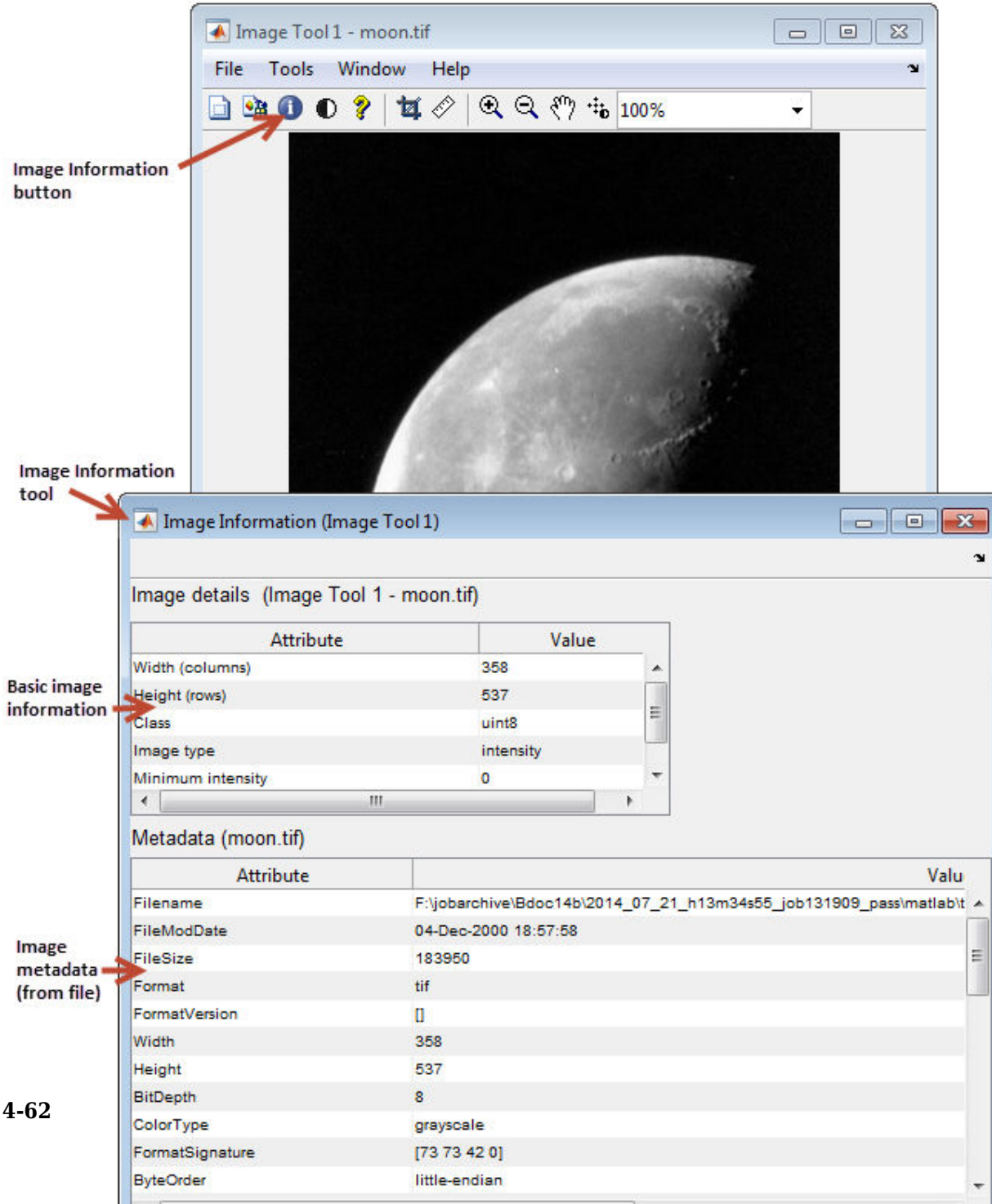


Image Tool 1 - moon.tif
File Tools Window Help
100%
Image Information button
Image Information tool
Image Information (Image Tool 1)
Image details (Image Tool 1 - moon.tif)
Attribute Value
Width (columns) 358
Height (rows) 537
Class uint8
Image type intensity
Minimum intensity 0
Basic image information
Metadata (moon.tif)
Attribute Valu
Filename F:\jobarchive\Bdoc14b\2014_07_21_h13m34s55_job131909_pass\matlab\t
FileModDate 04-Dec-2000 18:57:58
FileSize 183950
Format tif
FormatVersion []
Width 358
Height 537
BitDepth 8
ColorType grayscale
FormatSignature [73 73 42 0]
ByteOrder little-endian
Image metadata (from file)

Adjust Image Contrast in Image Viewer App

In this section...

"Open the Adjust Contrast Tool" on page 4-63
"Adjust Image Contrast Using the Histogram Window" on page 4-65
"Adjust Image Contrast Using Window/Level Tool" on page 4-66
"Make Contrast Adjustments Permanent" on page 4-69

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 4-65.

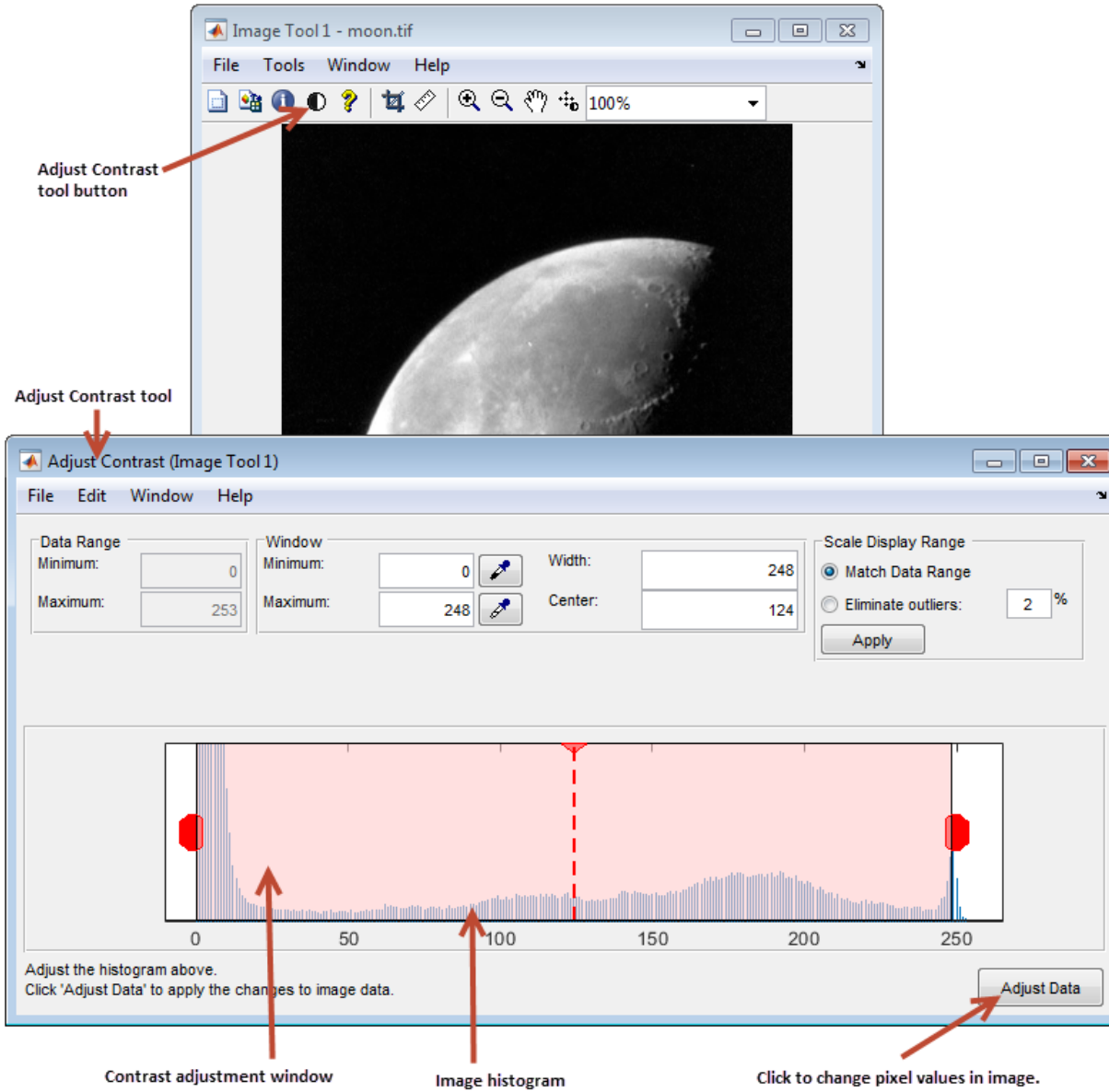


Image Tool 1 - moon.tif
File Tools Window Help
100%
Adjust Contrast tool button
Adjust Contrast tool
Adjust Contrast (Image Tool 1)
File Edit Window Help
Data Range
Minimum: 0
Maximum: 253
Window
Minimum: 0
Maximum: 248
Width: 248
Center: 124
Scale Display Range
Match Data Range
Eliminate outliers: 2 %
Apply
0 50 100 150 200 250
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 4-66.

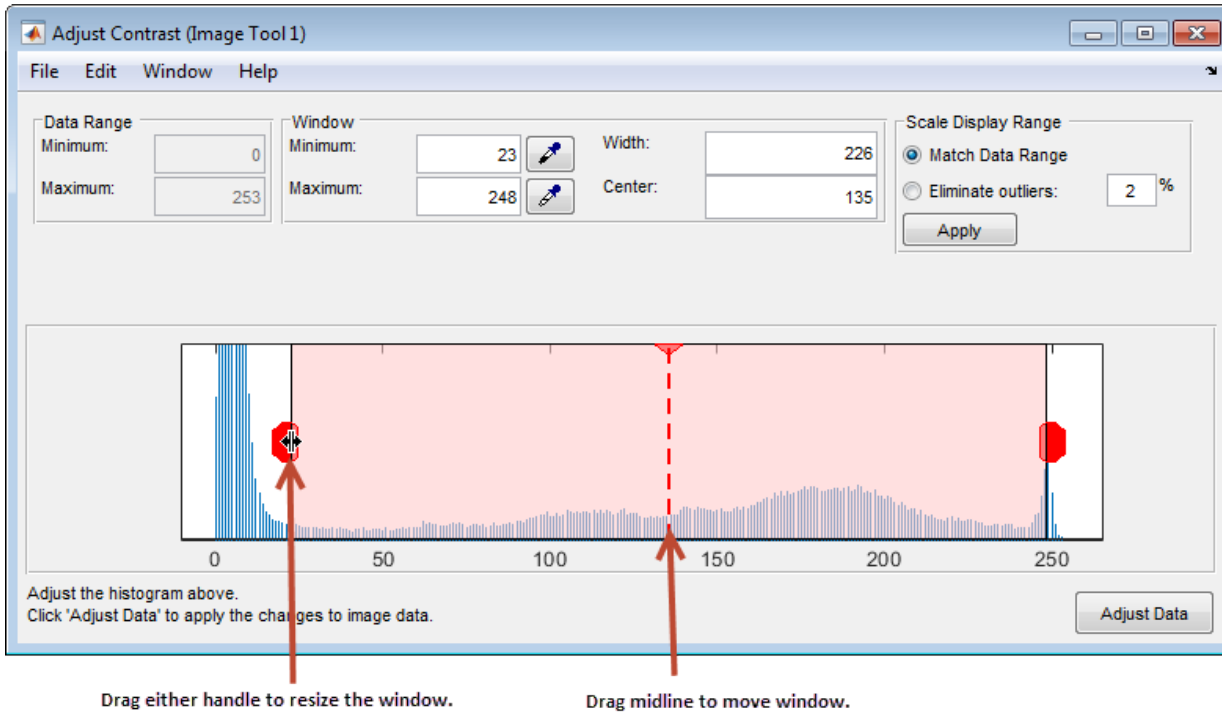
When you close the Adjust Contrast tool, the Window/Level tool remains active. To turn off the Window/Level tool, click the Window/Level button or one of the navigation buttons in the Image Viewer toolbar.

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.



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.
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.

   ```
   imtool(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
Image Tool 1 - I
File
Tools
Window
Help
100%
Fri Apr 30 06:27:24 1993 3D
Ex: 2411
Se: 3
Surface
DFOV 16.0cm
BONE
AIL
JFK IMAGING CENTER
****
M 26 151832
R
I
L
S
kv 120
mA 200
3.0 mm/1.0sp
Tilt: -0.0
W = 1492 L = 906
PSR
Pixel info: (343, 511) 4080
Display range: [32 4080]

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 4-65. 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 Adjust Data 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

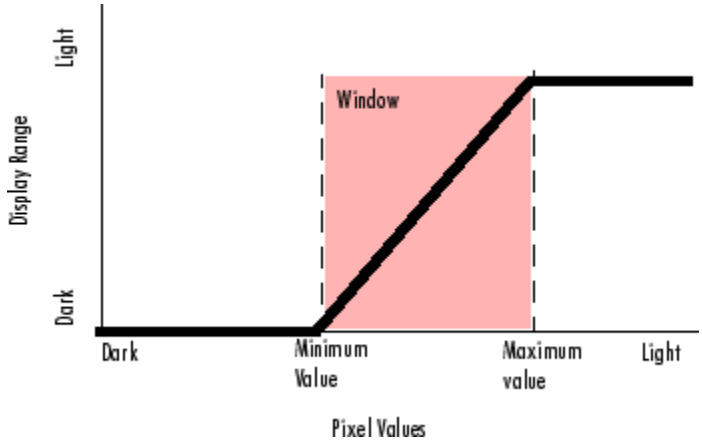
By default, if you close the Image Viewer, it does not save the modified image data. To save these changed values, use the **Save As** option from the Image Viewer **File** menu to store the modified data in a file or use the **Export to Workspace** option to save the modified data in a workspace variable.

Interactive 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 in the Image View app 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 4-63



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.

Crop Image Using Image Viewer App

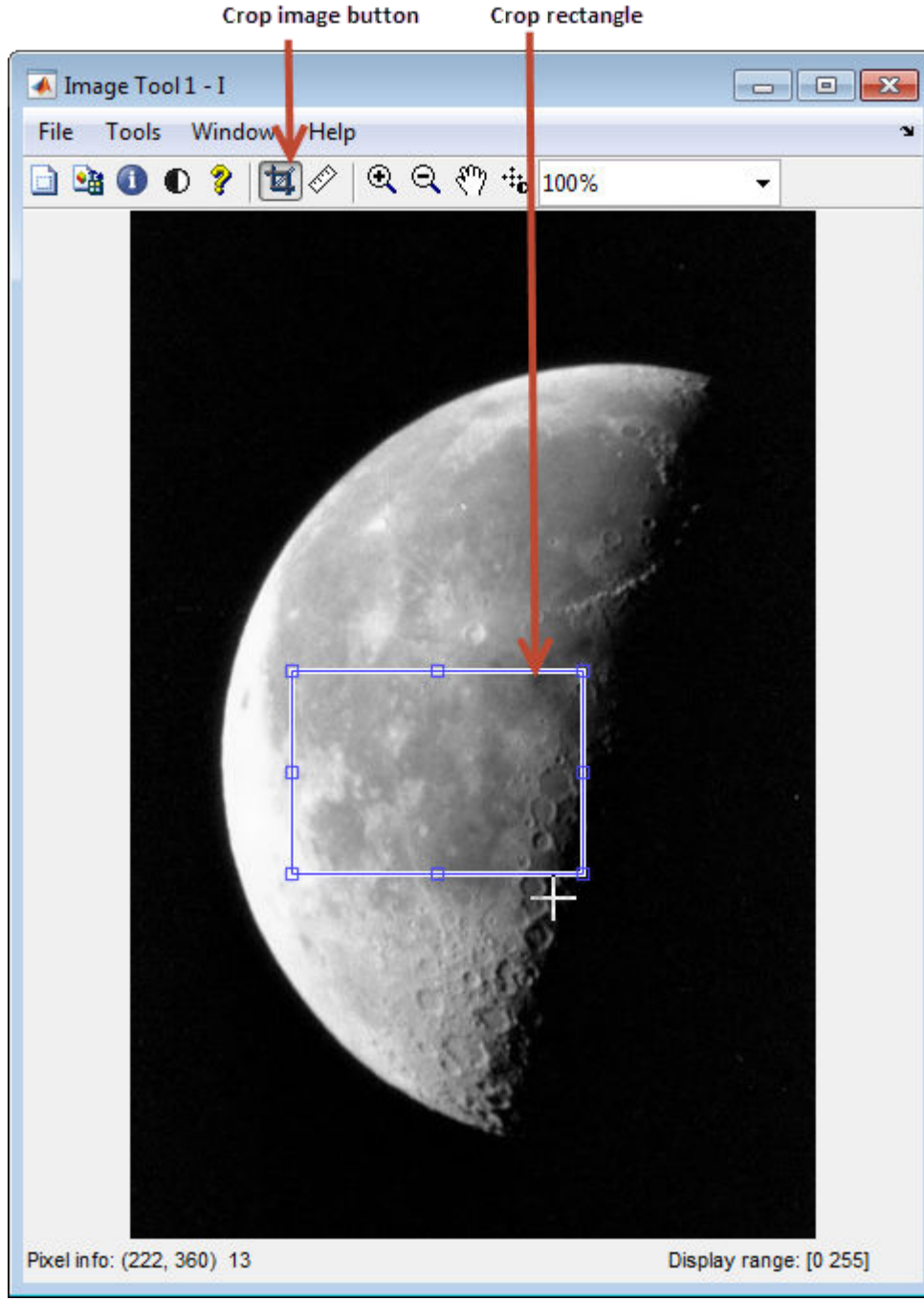
Cropping an image means creating a new image from a part of an original image. To crop an image using the Image Viewer, use the Crop Image tool. To use the Crop Image tool, follow this procedure.

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.


Crop image button
Crop rectangle
Image Tool 1 - I
File
Tools
Window
Help
100%
Pixel info: (222, 360) 13
Display range: [0 255]

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.

Image Tool 1
File
Tools
Window
Help
100%
Pixel info: (X, Y) Intensity
Display range: [0 255]

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

Explore 3-D Volumetric Data with Volume Viewer App

This example shows how to look at and explore 3-D volumetric data using the Volume Viewer app. Volume rendering is highly dependent on defining an appropriate alphamap so that structures in your data that you want to see are opaque and structures that you do not want to see are transparent. To illustrate, the example loads an MRI study of the human head into the Volume Viewer and explores the data using the visualization capabilities of the Volume Viewer.

In this section...
“Load Volume Data into the Volume Viewer” on page 4-77 “View the Volume Data in the Volume Viewer” on page 4-79 “View the Volume Data in the Volume Viewer” on page 4-82 “Refine the View with the Rendering Editor” on page 4-86

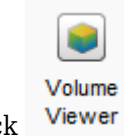
Load Volume Data into the Volume Viewer

This part of the example shows how to load volumetric data into the Volume Viewer.

Load the MRI data of a human head from a MAT-file into the workspace. This operation creates a variable named `D` in your workspace that contains the volumetric data. Use the `squeeze` command to remove the singleton dimension from the data.

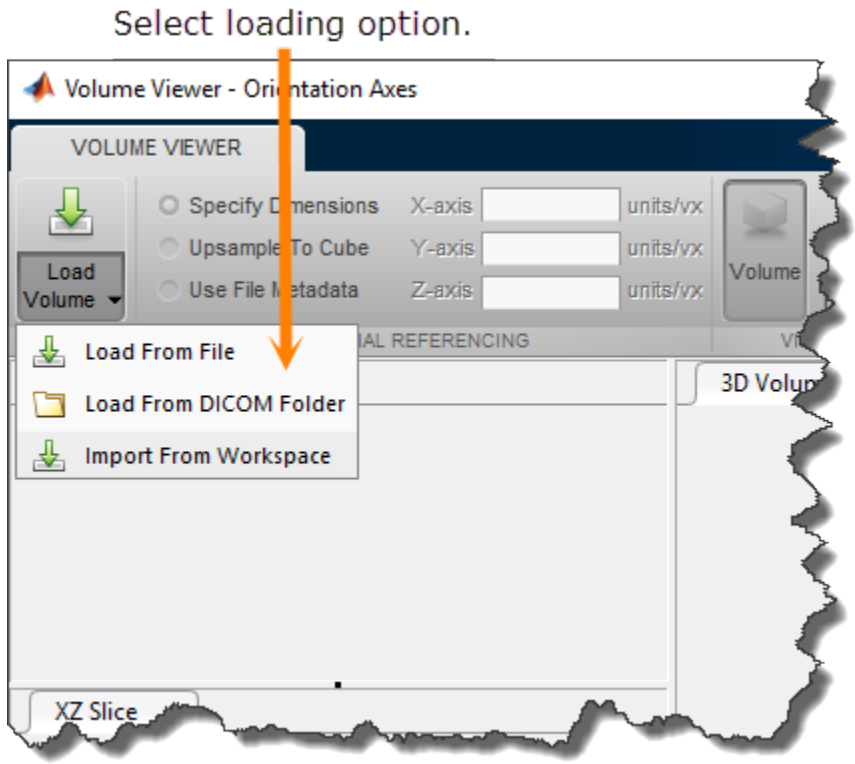
```
load mri

D = squeeze(D);

whos
```

```
Name        Size                 Bytes  Class     Attributes

  D         128x128x27            442368  uint8
  map        89x3                   2136  double
  siz         1x3                     24  double
```

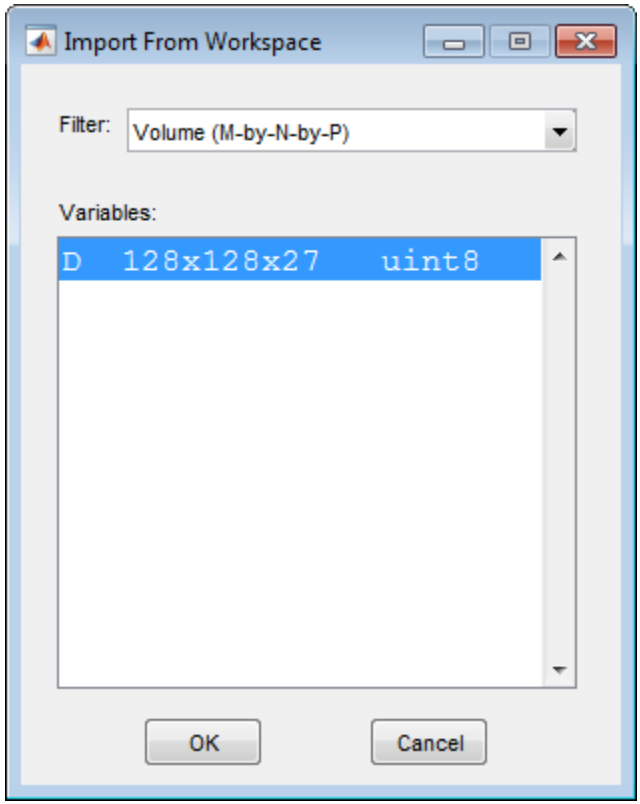
Open the Volume Viewer app. From the MATLAB Tool strip, open the Apps tab and under Image Processing and Computer Vision, click 

. You can also open the app using the `volumeViewer` command.

Load volumetric data into Volume Viewer app. Click **Load Volume**. You can load an image by specifying its file name or load a variable from the workspace. If you have volumetric data in a DICOM format that uses multiple files to represent a volume, you can specify the DICOM folder name. Choose the **Import From Workspace** option because the data is in the workspace.



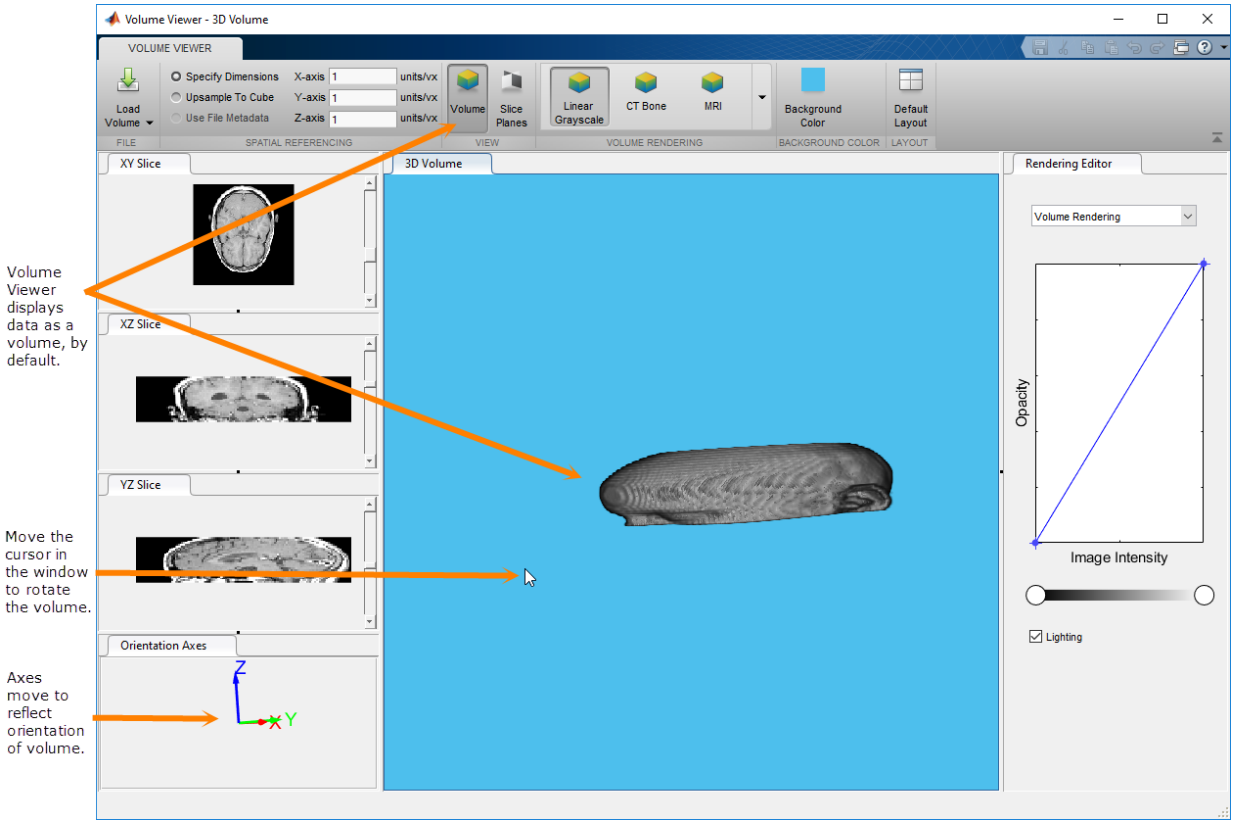
Select the workspace variable in the **Import from Workspace** dialog box and click **OK**.



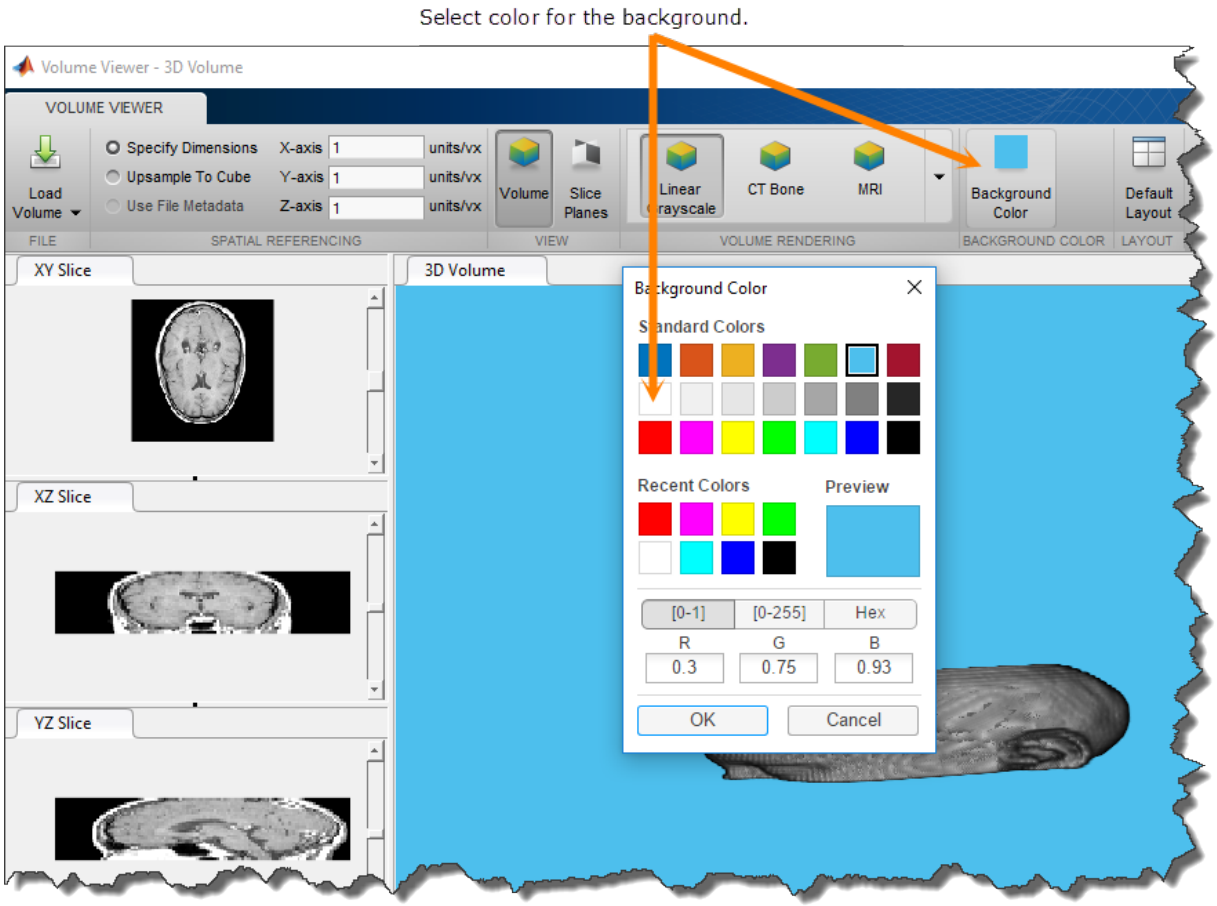
View the Volume Data in the Volume Viewer

In this part of the example, you decide how you want to view your data. The Volume Viewer offers several options.

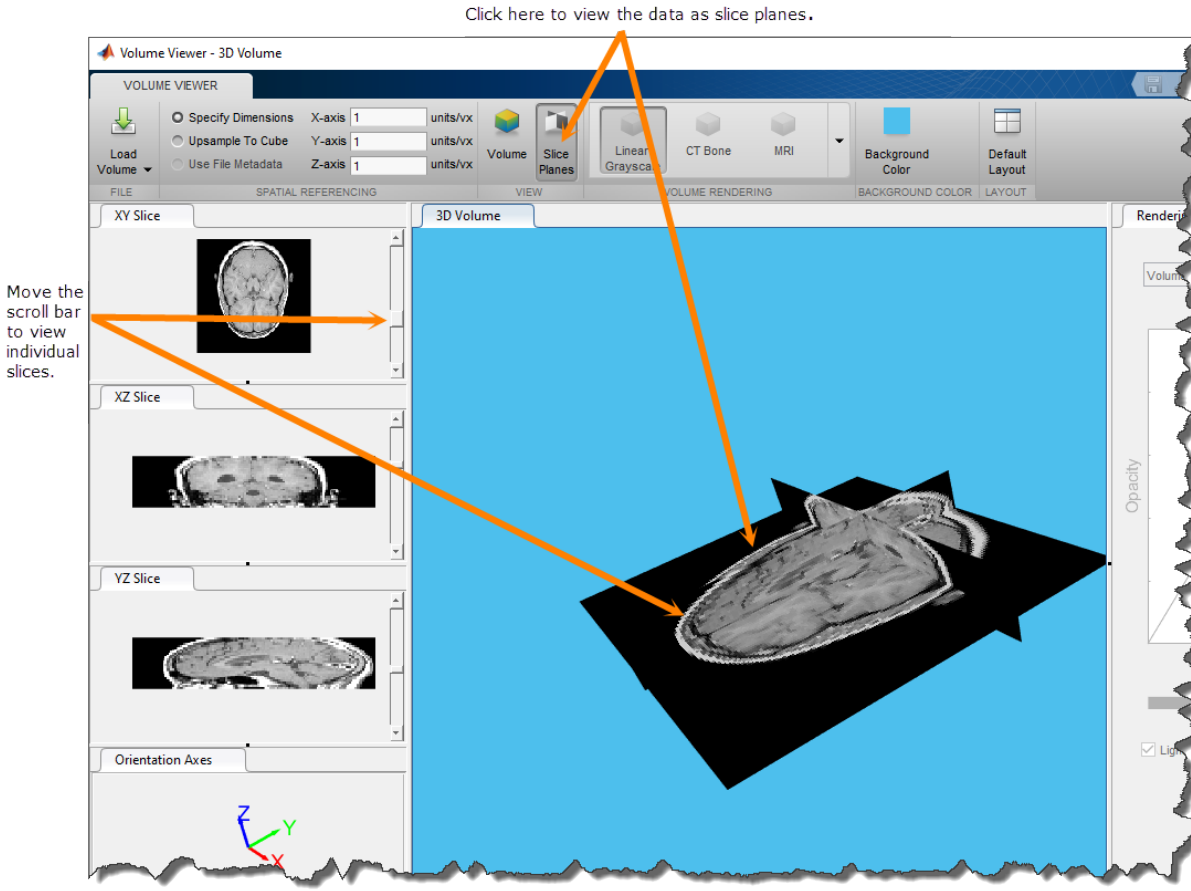
View the volume in the Volume Viewer app. By default, the Volume Viewer displays the data as a volume but you can also view it as slice planes. The MRI data displayed as a volume is recognizable as a human head. To explore the volume, zoom in and out on the image using the mouse wheel or a right-click. You can also rotate the volume by positioning the cursor in the image window, pressing and holding the mouse, and moving the cursor. You are always zooming or rotating around the center of the volume. The axes in the **Orientation Axes** window moves to reflect the spatial orientation of the image as you rotate it.



To change the background color used in the display window, click **Background Color** and select a color.



View the MRI data as a set of slice planes. Click **Slice Planes**. You can also zoom in and rotate this view of the data. Use the scroll bars in the three slice windows to view individual slices in any of the planes.



Continue used Volume Viewer capabilities until you achieve the best view of your data that.

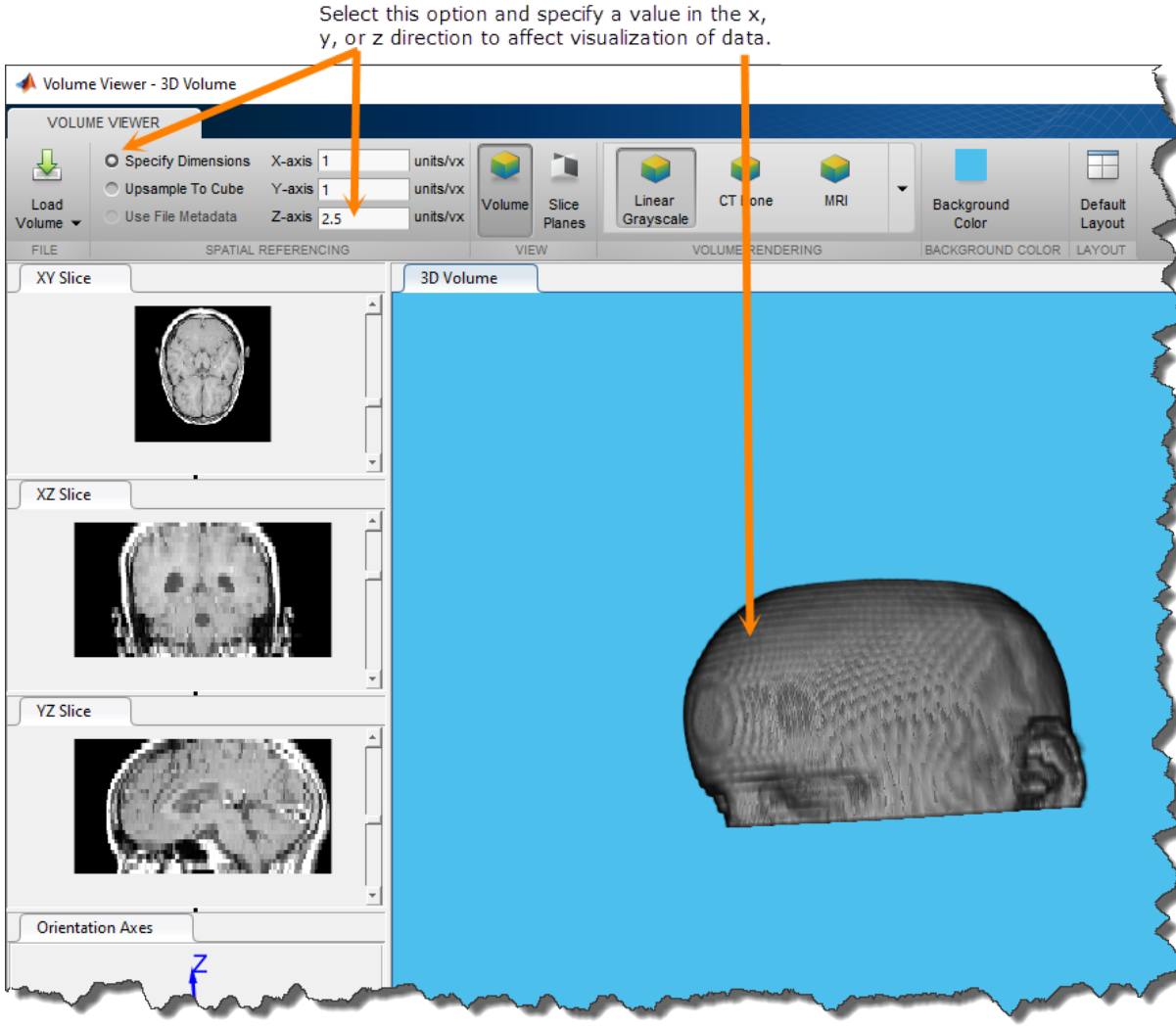
View the Volume Data in the Volume Viewer

In this part of the example, you view the volumetric data in the Volume Viewer app.

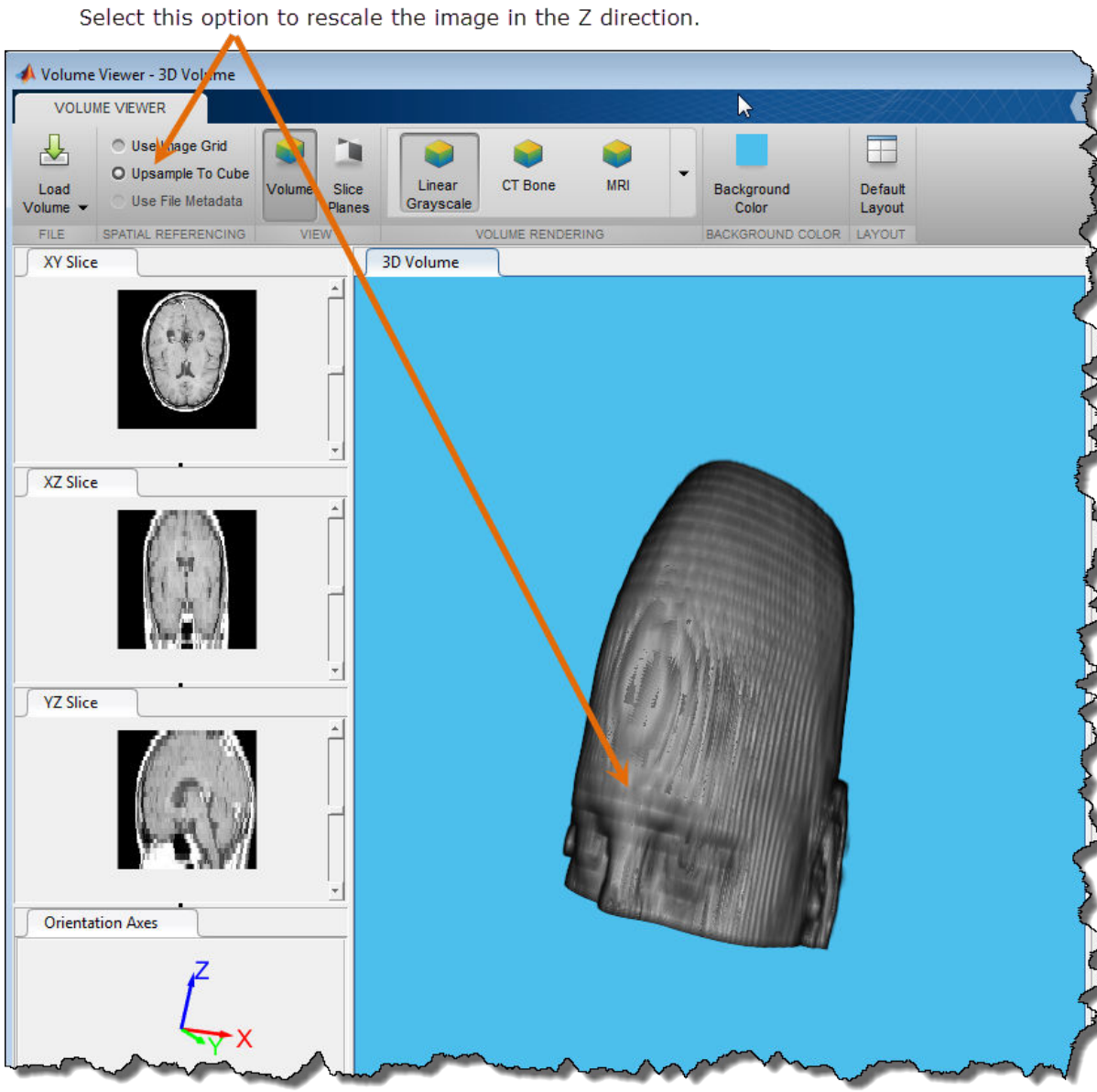
Click **Volume** to return to viewing your data as a volume and use the capabilities of the Volume Viewer to get the best visualization of your data. The Volume Viewer provides

several spatial referencing options that let you modify the view to get a more realistic view of the head volume (it appears flattened in the default view).

- **Specify Dimensions**--You can specify the dimensions in the X, Y, and Z directions.



- **Upsample To Cube**--The Volume Viewer calculates a scale factor that makes the number of samples in each dimension the same as the largest dimension in the volume. This setting can make non-isotropically sampled data appear scaled more correctly.


Select this option to rescale the image in the Z direction.
Volume Viewer - 3D Volume
VOLUME VIEWER
Load Volume
Use Image Grid
Upsample To Cube
Use File Metadata
Volume
Slice Planes
Linear Grayscale
CT Bone
MRI
Background Color
Default Layout
FILE
SPATIAL REFERENCING
VIEW
VOLUME RENDERING
BACKGROUND COLOR
LAYOUT
XY Slice
XZ Slice
YZ Slice
Orientation Axes
3D Volume
Z
X
Y

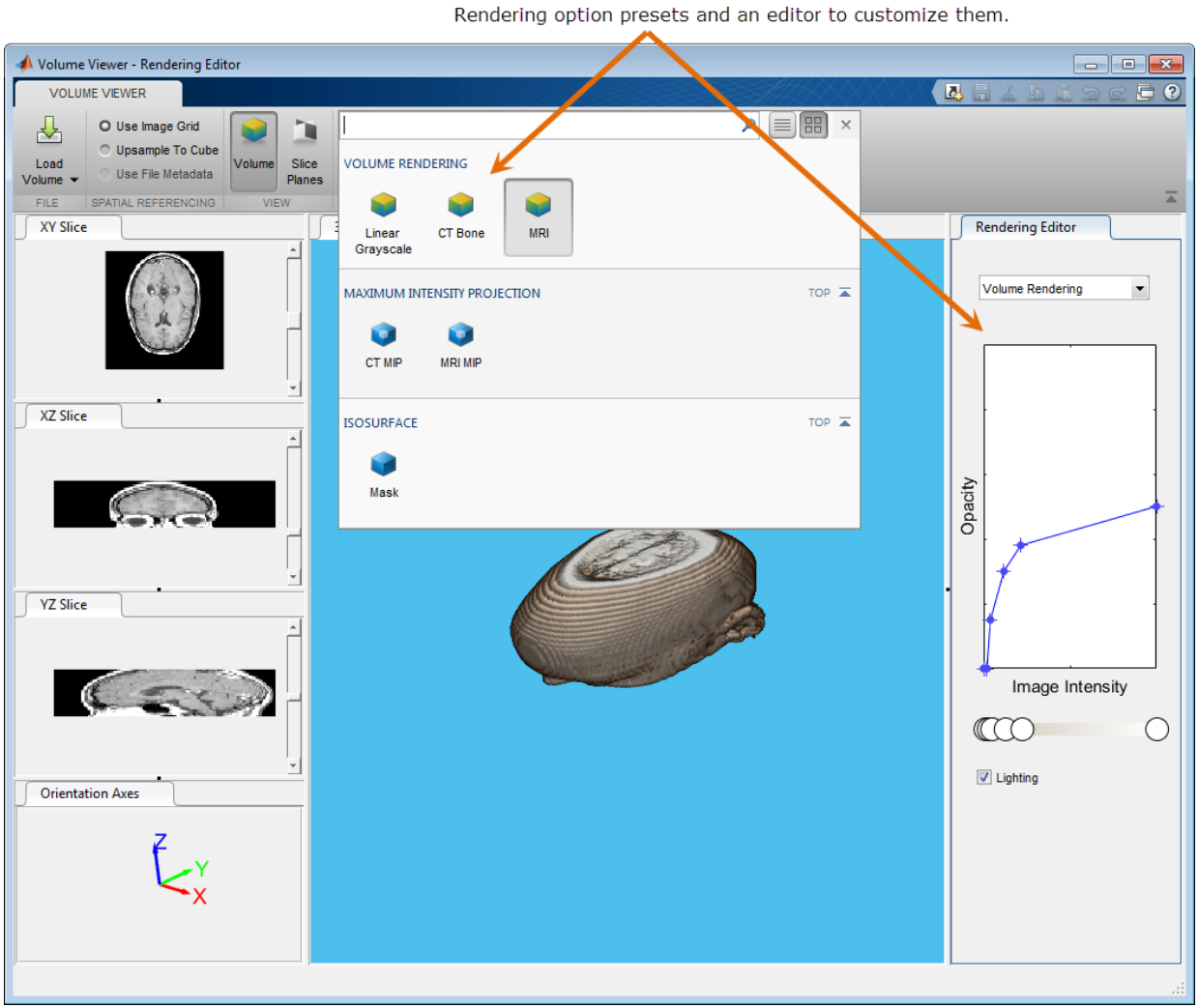
- **Use File Metadata**--If the data file includes resolution data in its metadata, the Volume Viewer uses the metadata and displays the volume true to scale. The Volume Viewer selects the **Use File Metadata** option, by default, if metadata is present.

Refine the View with the Rendering Editor

This part of the example describes how to use the Volume Viewer rendering editor to modify your view of the data.

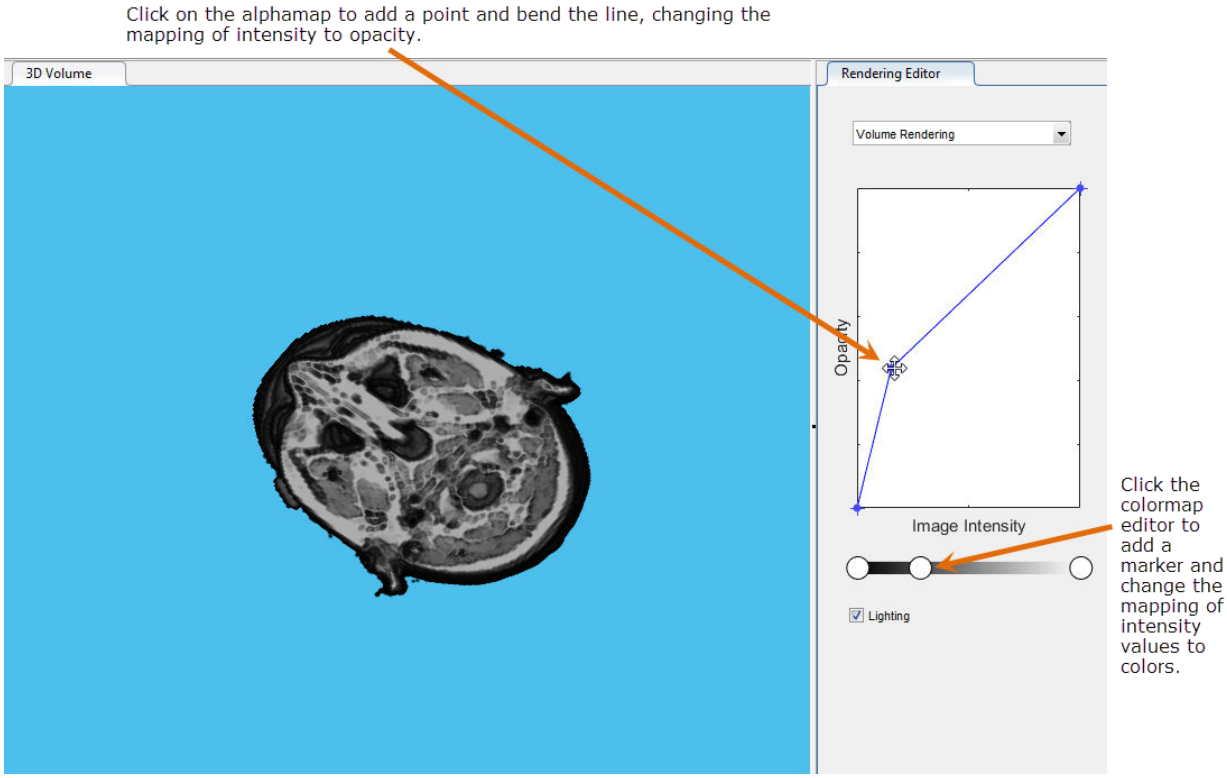
Refine the display, using Volume Viewer rendering options and the **Rendering Editor**.

Volume rendering is highly dependent on defining an appropriate alphamap so that structures you want to see are opaque and structures you don't want to see are transparent. You define the opacity and transparency of voxel values throughout the volume, and choose the color mapping of these intensity values. The Volume Viewer offers a set of rendering presets that automatically set the alphamap and colormap to achieve certain well-defined effects. For example, to set a view that works well with CT bone data, click the **CT Bone** rendering preset. The **Maximum Intensity Projection** (MIP) presets look for the voxel with the highest intensity value for each ray projected through the data. MIP can be useful for revealing the highest intensity structure within a volume. When you select a rendering preset, the Rendering Editor presents the alphamap and colormap with that preset. By default, the Volume Viewer uses a simple linear relationship, but each preset changes the curve of the plot to give certain data value more or less opacity.



Edit the volume rendering in the **Rendering Editor**. The Rendering Editor controls the relationship between voxel intensity, opacity, and color. For example, if you want to lighten up the rendering of the head MRI using **Linear Grayscale**, click in the colormap editor to create a new marker near the white end of the scale and then you can move the marker towards the dark end of the scale. In this way, you expand the number of pixel

values that map to white. You can also manipulate the alphamap to change the opacity mapping. If you click on the line, the Volume Viewer adds a marker



Continue used Volume Viewer capabilities until you achieve the best view of your data that.

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.

View MRI Sequence Using Video Viewer App

This example shows how to view multi-slice volumetric data in the Video Viewer app.

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

```
load mristack
```

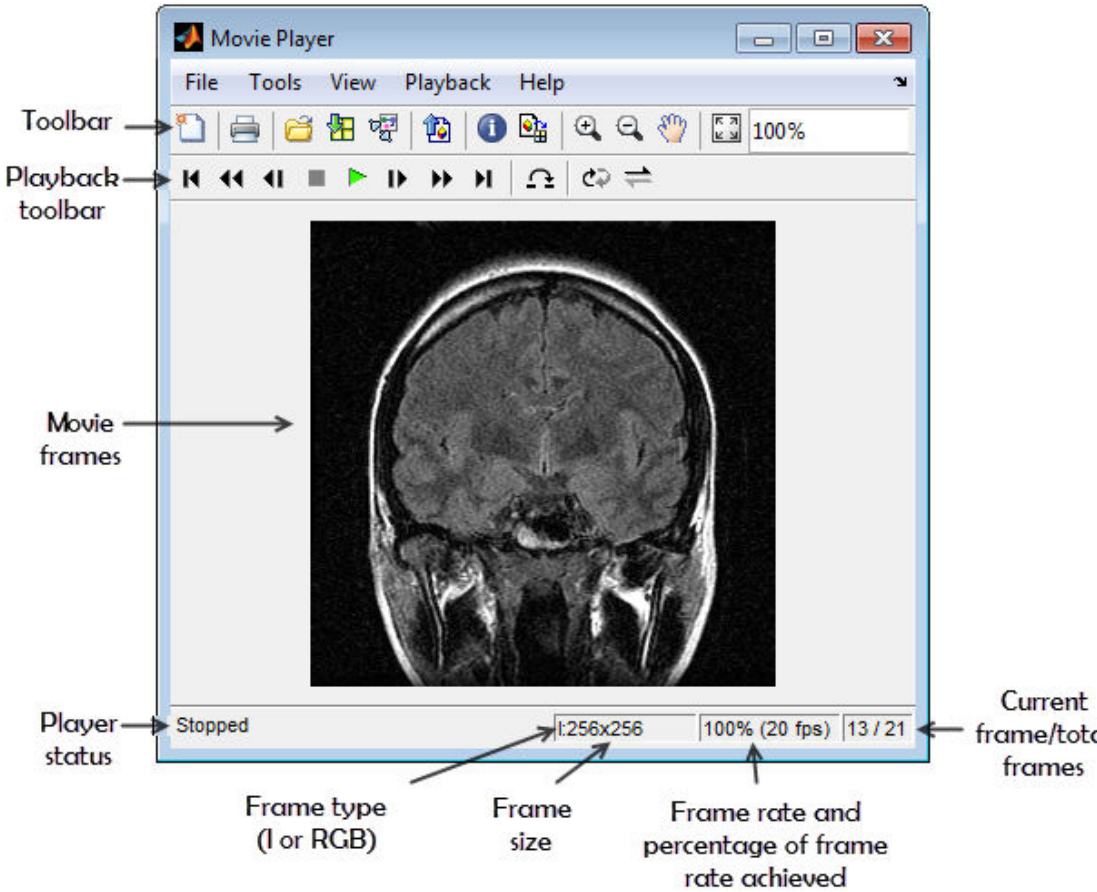
This places a variable named `mristack` in your workspace. The variable is an array of 21 grayscale frames containing MRI images of the brain. Each frame is a 256-by-256 array of `uint8` data.

```
mristack       256x256x21       1276256     uint8
```

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(mristack)
```

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.



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, the Video Viewer cycles through these options and the appearance changes to indicate the current selection. The Video Viewer uses plus (+) and minus (-) signs to indicate playback direction in the Current Frame display. When you play a video backwards, the frame numbers displayed are negative. When you are in AutoReverse mode, the Video Viewer uses plus signs to indicate the forward direction.	**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)

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

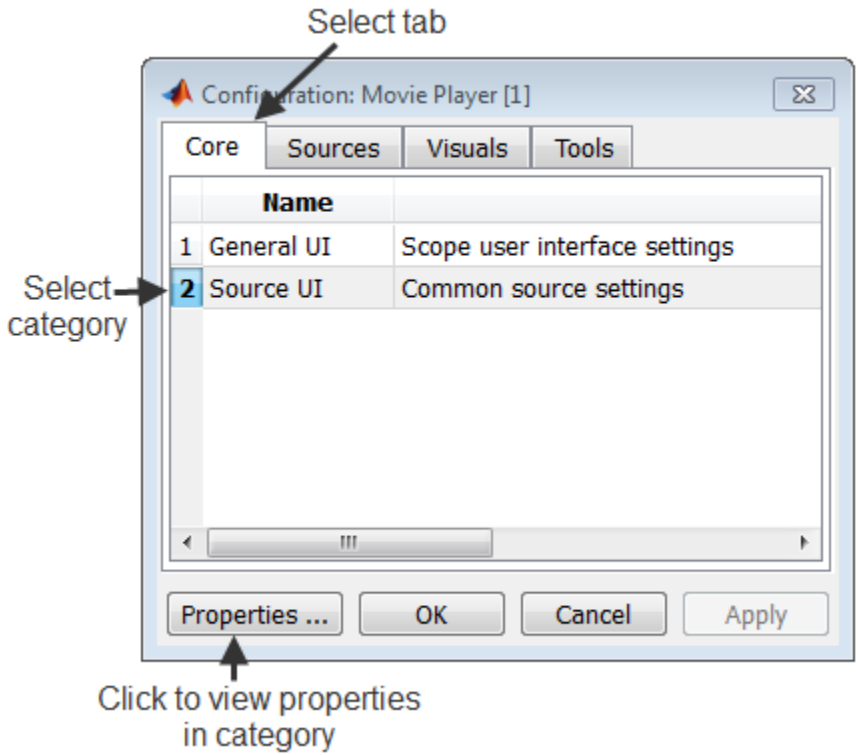
The Video Viewer supports several tools listed in the Tools menu and on the Toolbar that you can use to examine the frames in the image sequence more closely.

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	**Display the full path** 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.

Pane	Option Category	Option Descriptions
Core	Source UI	**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`. **Recently used sources list** parameter — Specifies the number of sources listed in the **File** menu.
Sources	Simulink	**Load Simulink model if not open** check box — You must have Simulink installed. **Connect scope on selection of:** — Signal lines only or signal lines and blocks. You must have Simulink installed.
Sources	File	**Default open file path** parameter — Specify the directory that is displayed in the Connect to File dialog box when you click **File > Open**.
Sources	Workspace	There are no options associated with this selection.
Visuals	Video	There are no options associated with this selection.
Tools	Image Tool	**Open new Image Tool window for each export** check box — Opens a new Image Viewer for each exported frame.
Tools	Pixel Region	There are no options associated with this selection.
Tools	Image Navigation Tools	There are no options associated with this selection.
Tools	Instrumentation Sets	There are no options associated with this selection.

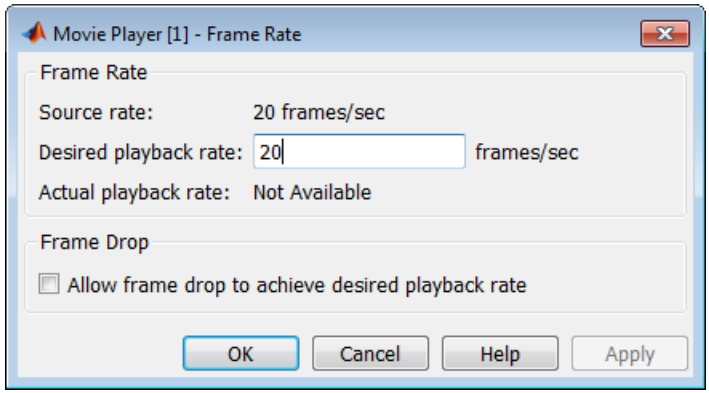
Save Image Viewer App Configuration Settings

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

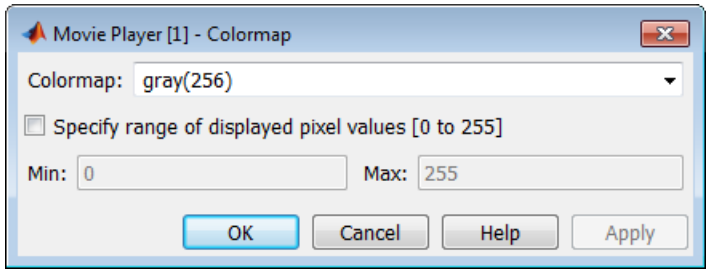
To decrease or increase the playback rate, select **Frame Rate** from the Playback menu, or use the keyboard shortcut **T**. The Frame Rate dialog box displays the frame rate of the source, lets you change the rate at which the Video Viewer app plays the image sequence or video, and displays the actual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

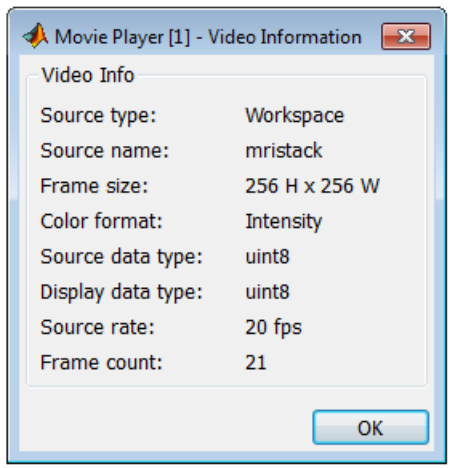


Use the **Colormap** parameter to specify a particular colormap.

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.



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.

```
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 4-98 “Display Grayscale Images” on page 4-99 “Display Binary Images” on page 4-100 “Display Truecolor Images” on page 4-102

If you need help determining what type of image you are working with, see “Image Types in the Toolbox” on page 2-13.

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 “Display Colors” on page 15-2 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

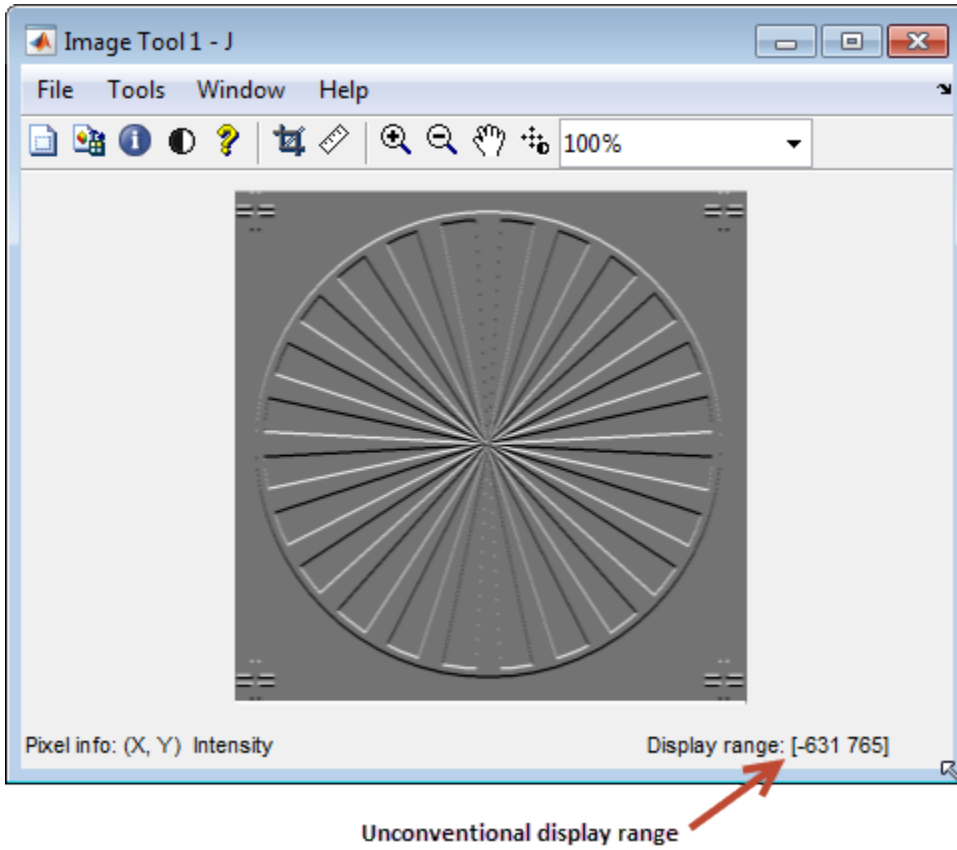
```
imtool(I,'DisplayRange',[low high])
```

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',[]);
```



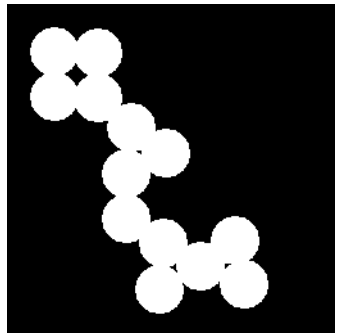
Display Binary Images

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

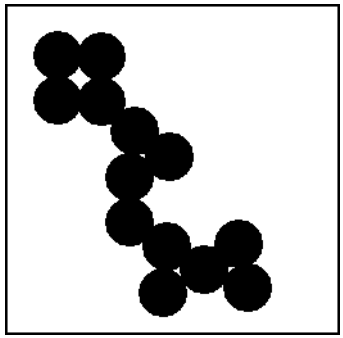
```
BW = imread('circles.png');
imshow(BW)
```



Change Display Colors of Binary Image

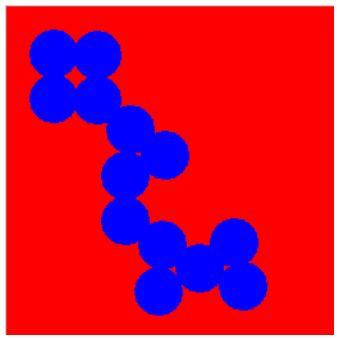
You might prefer to invert binary images when you display them, so that 0 values are displayed as white and 1 values are displayed as black. To do this, use the NOT (~) operator in MATLAB. (In this figure, a box is drawn around the image to show the image boundary.) For example:

```
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 “Display Colors” on page 15-2 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 4-98.

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.

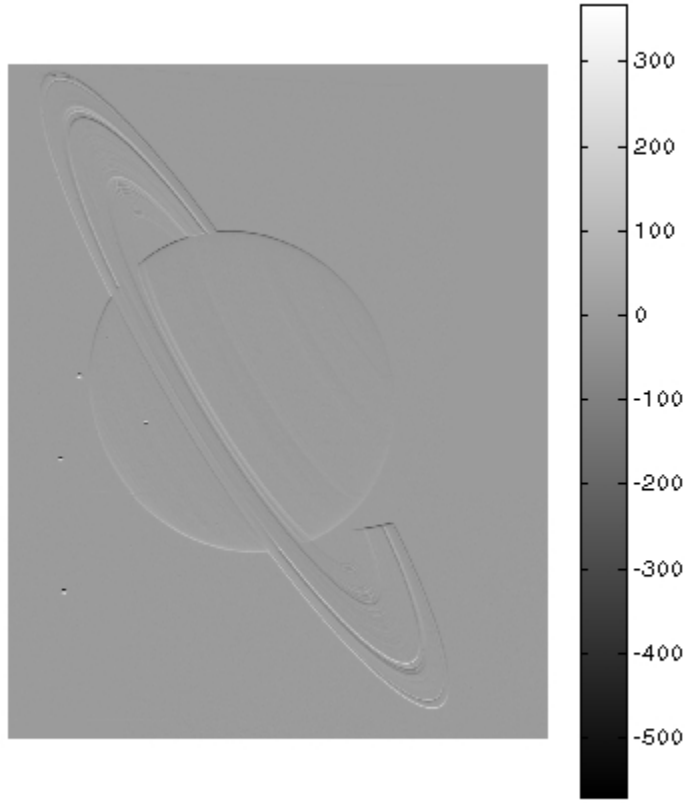
When you add a colorbar to an axes object that contains an image object, the colorbar indicates the data values that the different colors in the image correspond to.

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 4-99.

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

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


300
200
100
0
-100
-200
-300
-400
-500

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 “Write Image Data to File in Graphics Format” on page 3-8 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 non-image elements such as labels, titles, and other annotations.

Graphics Object Properties That Impact Printing

The output reflects the settings of various properties of 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, see “Print Figure from File Menu” (MATLAB). 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.

Manage Display Preferences

In this section...
“Retrieve Values of Toolbox Preferences” on page 4-107 “Set Values of Toolbox Preferences” on page 4-107

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.

5

Building GUIs with Modular Tools

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

- “Build Custom Image Processing Apps Using Modular Interactive Tools” on page 5-2
- “Interactive Modular Tool Workflow” on page 5-9
- “Build App To Display Pixel Information” on page 5-27
- “Build App for Navigating Large Images” on page 5-30
- “Customize Modular Tool Interactivity” on page 5-32
- “Build Image Comparison Tool” on page 5-33
- “Create Your Own Modular Tools” on page 5-37
- “Create Angle Measurement Tool Using ROI Objects” on page 5-39

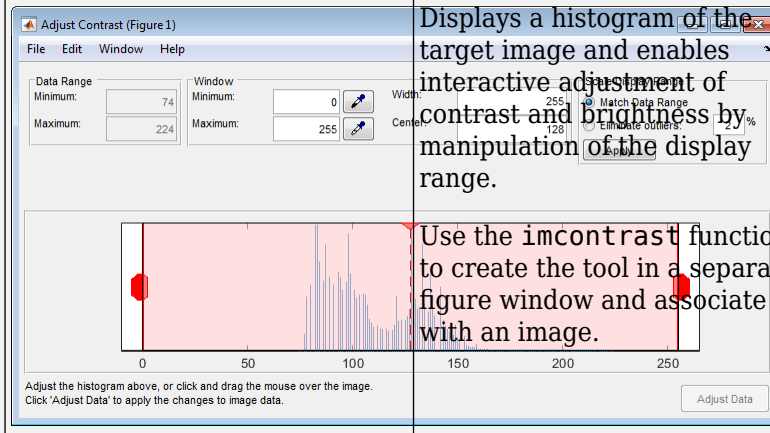
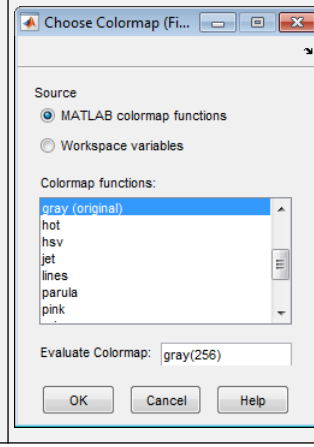
Build Custom Image Processing Apps Using Modular Interactive Tools

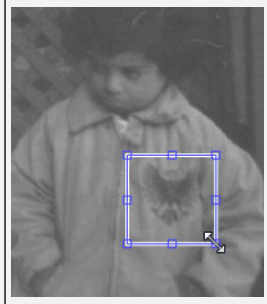
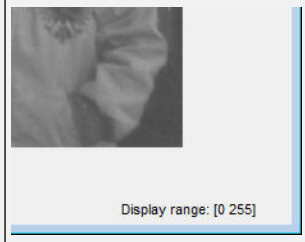
The toolbox includes several modular interactive tools that you can activate from the command line and use with images displayed in a MATLAB figure window, called the *target image* in this documentation. The tools are modular because they can be used independently or in combination to create custom image processing apps.

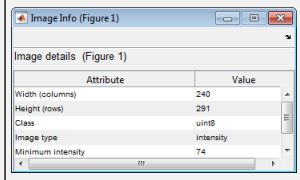
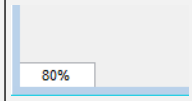
The following table lists the modular tools in alphabetical order. The table includes an illustration of each tool and the function you use to create it.

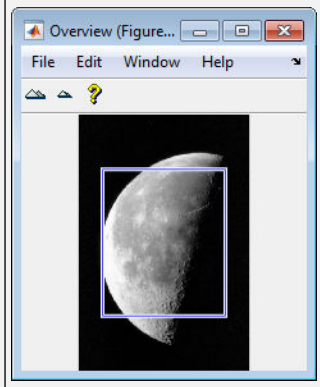
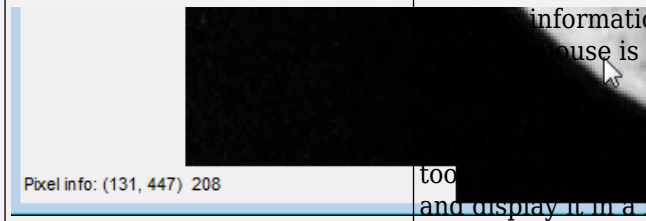
Note The Image Processing Toolbox image viewing and exploration app, Image Tool, uses these modular tools — see “Interact with Images Using Image Viewer App” on page 4-32.

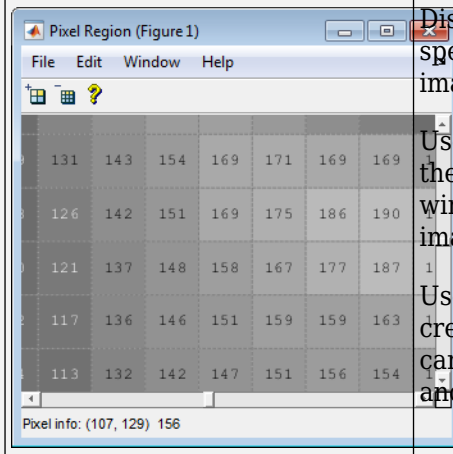
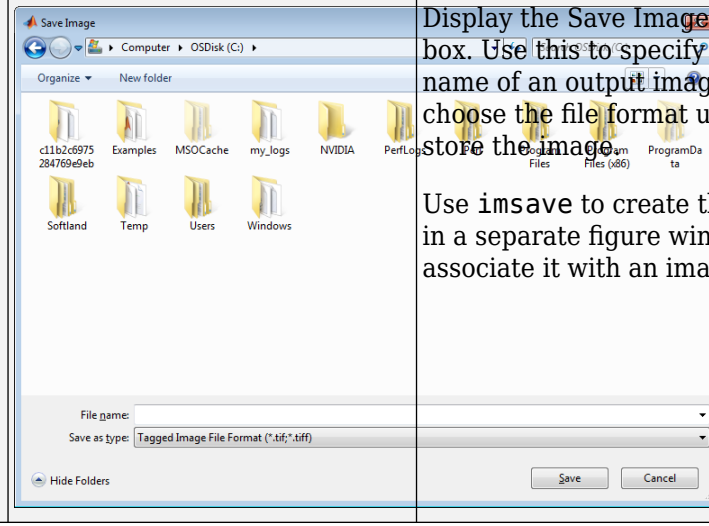
Summary of Modular Tools

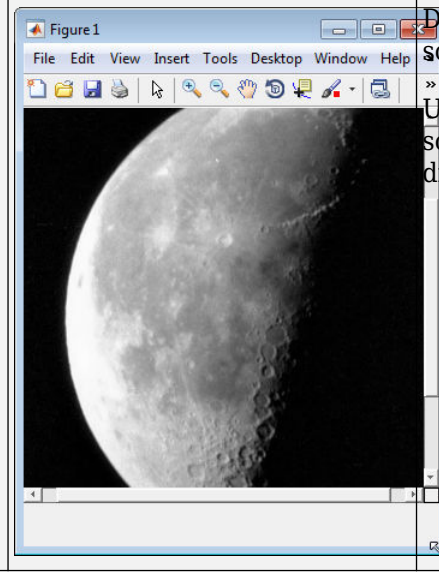
Modular Tool	Example	Description
Adjust Contrast tool		Displays a histogram of the target image and enables interactive adjustment of contrast and brightness by manipulation of the display range. Use the `imcontrast` function to create the tool in a separate figure window and associate it with an image.
Choose Colormap tool		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. Use the `imcolormaptool` function to launch the tool in a separate figure window.

Modular Tool	Example	Description
Crop Image tool		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. Use the `imcrop` function to create the tool and associate it with an image.
Display Range tool		Displays the display range values of the associated image. Use the `imdisplayrange` function to create the tool, associate it with an image, and embed it in a figure or uipanel.
Distance tool		Displays a draggable, resizable line on an image. Superimposed on the line is the distance between the two endpoints of the line. The distance is measured in units specified by the `XData` and `YData` properties, which is pixels by default. Use the `imdistline` function to create the tool and associate it with an image.

Modular Tool	Example	Description
Image Information tool		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. Use the `imageinfo` function to create the tool in a separate figure window and associate it with an image.
Magnification box		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. Use `immagbox` to create the tool, associate it with an image, and embed it in a figure or uipanel. **Note** The target image must be contained in a scroll panel.

Modular Tool	Example	Description
Overview tool		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. Use `imoverview` to create the tool in a separate figure window and associate it with an image. Use `imoverviewpanel` to create the tool in a uipanel that can be embedded within another figure or uipanel. **Note** The target image must be contained in a scroll panel.
Pixel Information tool		information about the ouse is over in the to create the too h an image, and display it in a figure or uipanel. If you want to display only the pixel values, without the **Pixel info** label, use `impixelinfoval`.

Modular Tool	Example	Description
Pixel Region tool		Display pixel values for a specified region in the target image. Use `impixelregion` to create the tool in a separate figure window and associate it with an image. Use `impixelregionpanel` to create the tool as a uipanel that can be embedded within another figure or uipanel.
Save Image tool		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. Use `imsave` to create the tool in a separate figure window and associate it with an image.

Modular Tool	Example	Description
Scroll Panel tool		Display target image in a scrollable panel. Use `imscrollpanel` to add a scroll panel to an image displayed in a figure window.

Interactive Modular Tool Workflow

In this section...
“Display the Target Image in a Figure Window” on page 5-11 “Associate Modular Tools with the Target Image” on page 5-11 “Associate Modular Tools with a Particular Target Image” on page 5-12 “Get Handle to Target Image” on page 5-14 “Specify the Parent of a Modular Tool” on page 5-17 “Position Modular Tools in a GUI” on page 5-21 “Adding Navigation Aids to a GUI” on page 5-23

Using the interactive modular tools typically involves the following steps.

Step	Description	Notes
1	Display the image to be processed (called the target image) in a figure window.	Use the `imshow` function to display the target image, see “Display the Target Image in a Figure Window” on page 5-11.

Step	Description	Notes
2	Create the modular tool, associating it with the target image.	You use the modular tool creation functions to create the tools — see “Build Custom Image Processing Apps Using Modular Interactive Tools” on page 5-2 for a list of available tools. Most of the tools associate themselves with the image in the current axes, by default, but you can specify a specific image object, or a figure, axes, or uipanel object that contains an image. See “Interactive Modular Tool Workflow” on page 5-9. 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 “Specify the Parent of a Modular Tool” on page 5-17 for more information. You might need to specify the position of the graphics objects in the GUI, including the modular tools. See “Position Modular Tools in a GUI” on page 5-21 for more information.
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 “Customize Modular Tool Interactivity” on page 5-32 for more information.

Display the Target Image in a Figure Window

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.

Associate Modular Tools with the Target Image

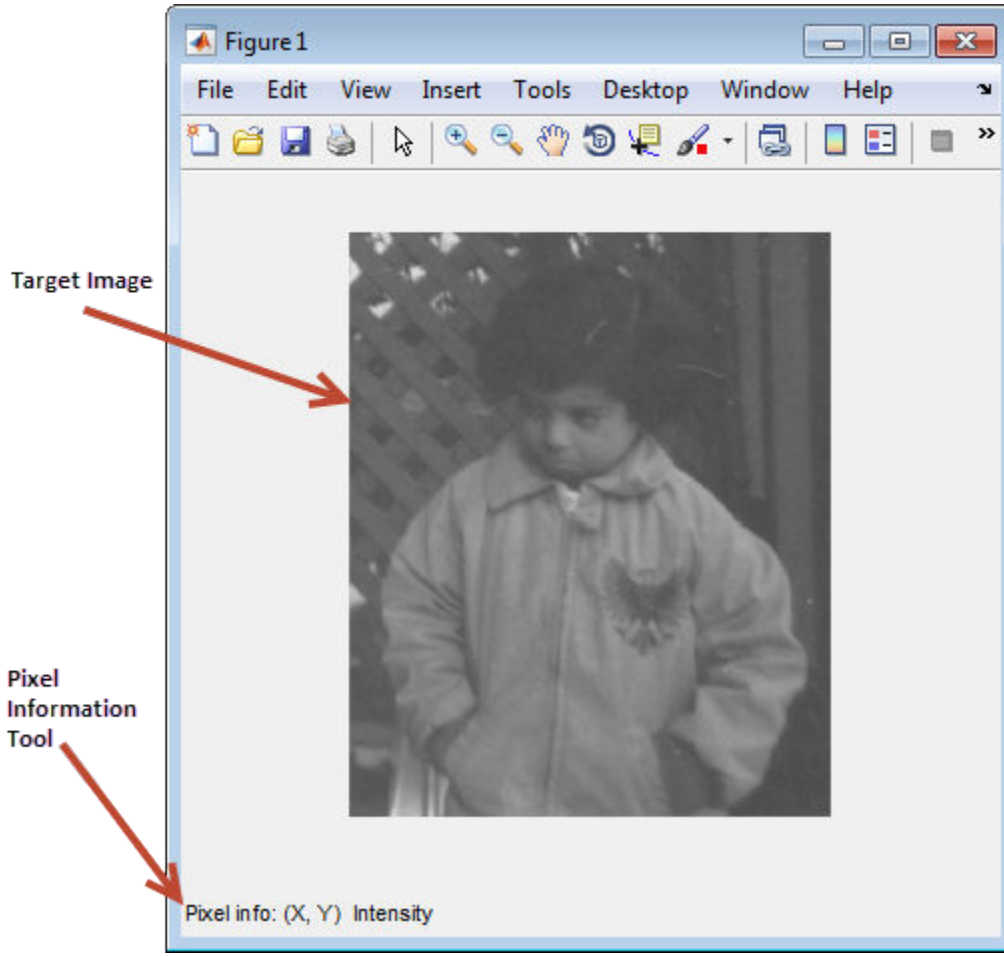
To associate a modular tool with a target image displayed in a MATLAB figure window, you must create the tool using the appropriate tool creation function. You specify the target image as an argument to the tool creation function. The function creates the tool and automatically sets up the interactivity 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 `imdisplayrange` 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.



Associate Modular Tools with a Particular Target Image

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

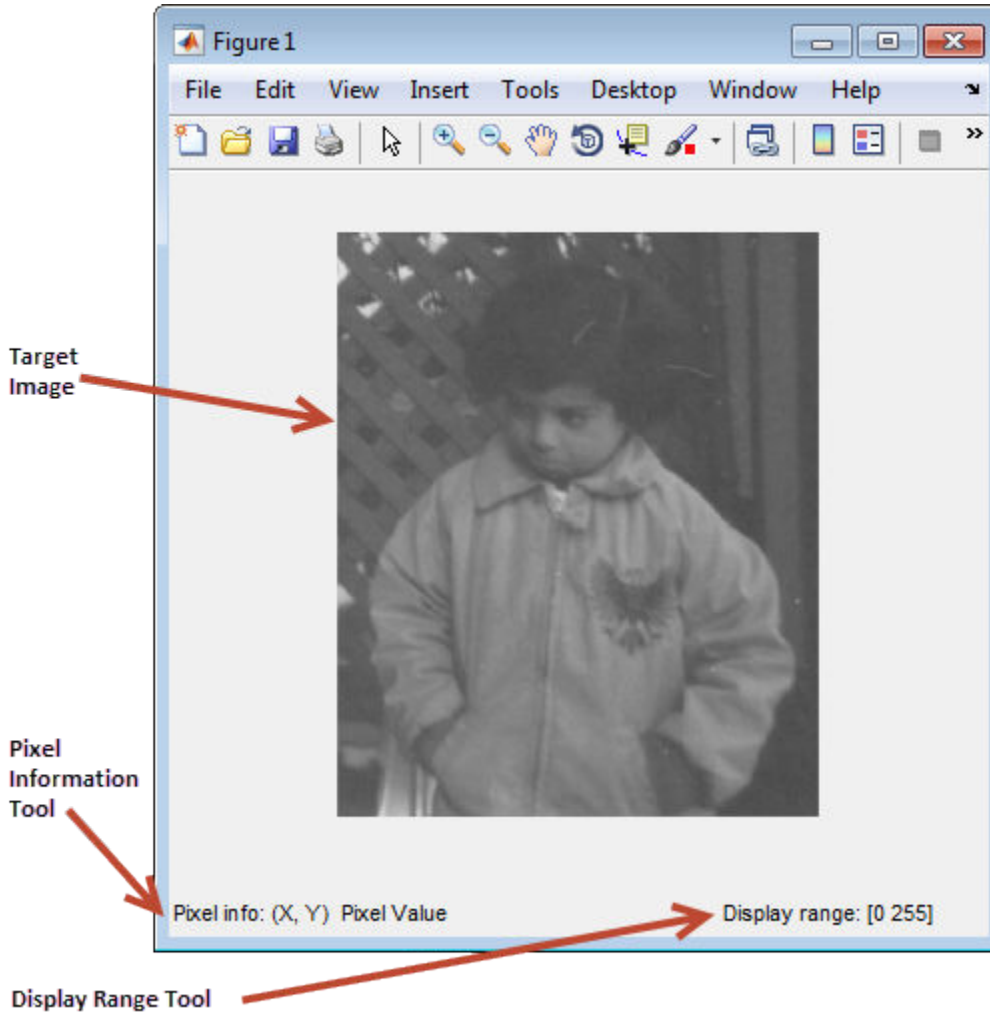
Continuing the example in the previous section, you might want to add the Display Range tool to the figure window that already contains the Pixel Information tool. To do this, call

the `imdisplayrange` function, specifying the target image. You could also have specified the figure, axes, or uipanel object containing the target image.

```
imageobj = imshow('pout.tif');
pixelinfopanelobj = impixelinfo(imageobj);
drangepanelobj = imdisplayrange(imageobj);
```

Note that the tool creation functions return the uipanel objects created by the `impixelinfo` and `imdisplayrange` functions. You can use these objects if you want to change the positioning of the tools. See “Position Modular Tools in a GUI” on page 5-21 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.



Get Handle to Target Image

This example shows several ways to get the handle to the image displayed in a figure window, referred to as the *target image*. This can be useful when creating apps with the modular interactive tools.

Get the handle when you initially display the image in a figure window using the `imshow` syntax that returns a handle.

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

```
himage =
  Image with properties:

           CData: [537x358 uint8]
    CDataMapping: 'scaled'

  Show all properties
```

Get the handle after you have displayed the image in a figure window using the `imhandles` function. You must specify a handle to the figure window as a parameter.

```
himage2 = imhandles(hfig)
```

```
himage2 =
  Image with properties:

           CData: [537x358 uint8]
    CDataMapping: 'scaled'

  Show all properties
```

Specify 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

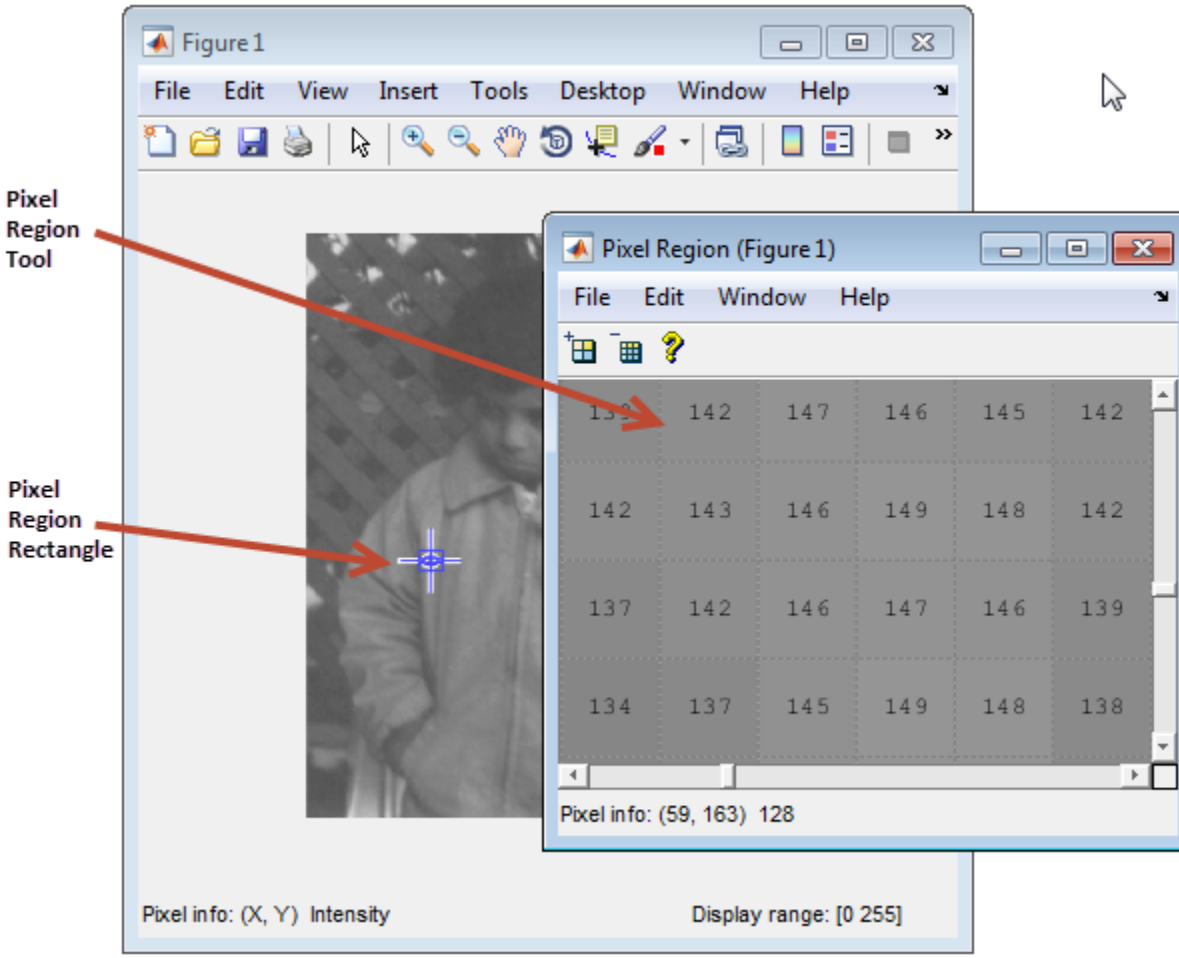
`impixelregion`, open the tools in a separate figure window. To specify a different parent, you must use the `imoverviewpanel` and `impixelregionpanel` functions.

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.

Embed Pixel Region Tool in 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);
```

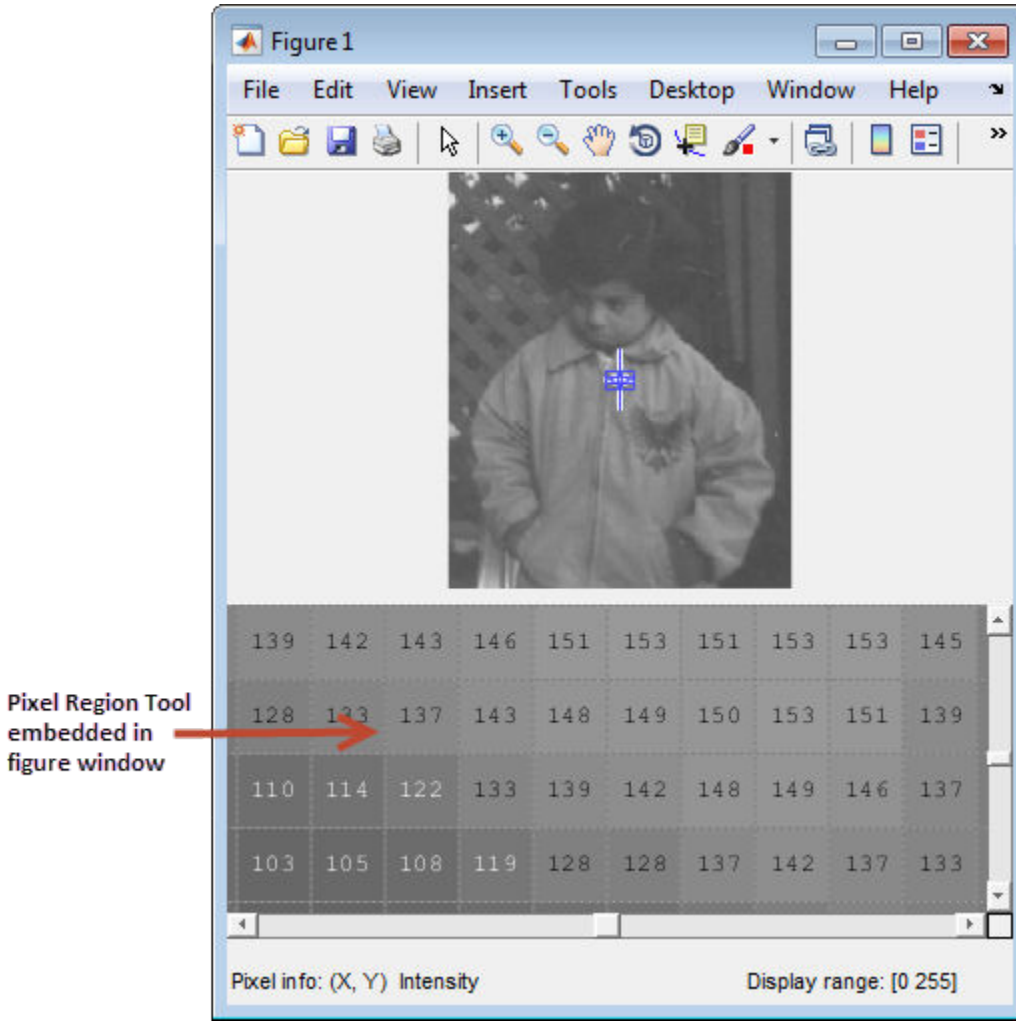


To embed the Pixel Region tool in the same window as the target image, you must specify the target image's parent figure as the parent of the Pixel Region tool when you create it.

The following example creates a figure and an axes object and then positions the objects in the figure to ensure their visibility. See “Position Modular Tools in a GUI” on page 5-21 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.

```
fig = figure;
ax = axes('units','normalized','position',[0 .5 1 .5]);
img = imshow('pout.tif')
pixelinfopanelobj = impixelinfo(img);
drangepanelobj = imdisplayrange(img);
pixregionobj = impixelregionpanel(fig,img);
set(pixregionobj, 'Units','normalized','Position',[0 .08 1 .4]);
```

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



Position 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 graphics objects, such as uipanel objects, you can use properties of the objects to change their default positioning or other characteristics.

For example, in “Specify the Parent of a Modular Tool” on page 5-17, 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.

Specify 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 pages for these 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])
```

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.

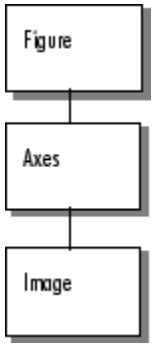
- “Understanding Scroll Panels” on page 5-23 — Adding a scroll panel to an image display changes the relationship of the graphics objects used in the display. This section provides some essential background.
- “Build App for Navigating Large Images” on page 5-30 — This section shows how to add a scroll 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.

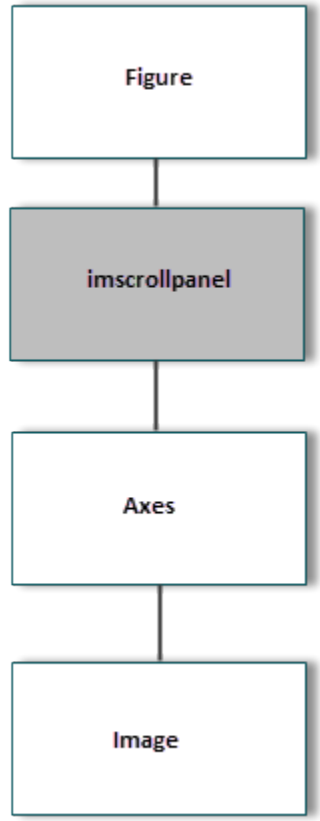


Object Hierarchy of Image Displayed in a Figure

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.

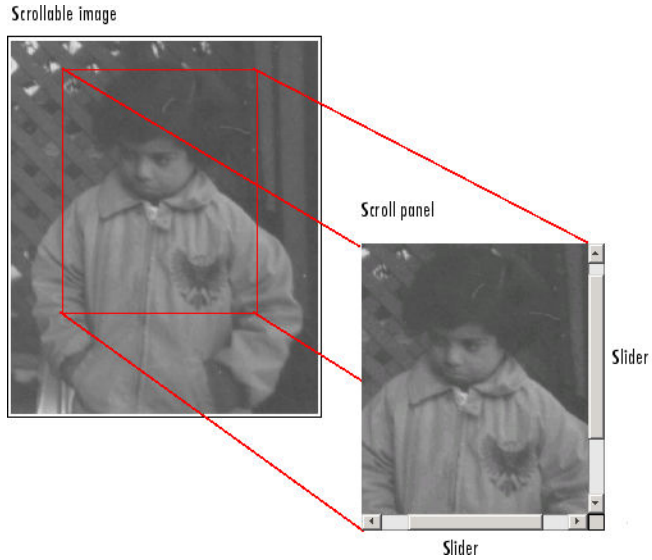
```
hfig = figure('Toolbar','none',...
              'Menubar', 'none');
himage = imshow('concordaerial.png');
hpanel = imscrollpanel(hfig,himage);
```

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

Build App To Display Pixel Information

This example shows how to create a simple app that provides information about pixels and features in an image using modular pixel information tools. Because the standard figure window zoom tools are not compatible with the toolbox modular tools, the example suppresses the toolbar and menu bar in the figure window.

Create a function that accepts an image as an argument and displays the image in a figure window with a Pixel Information tool, Display Range tool, Distance tool, and Pixel Region tool.

```
function my_pixinfotool(im)
% Create figure, setting up properties
fig = 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
ax = axes('Units','normalized',...
           'Position',[0 .5 1 .5]);

% Display image in the axes
img = imshow(im);

% Add Distance tool, specifying axes as parent
distool = imdistline(ax);

% Add Pixel Information tool, specifying image as parent
pixinfo = impixelinfo(img);

% Add Display Range tool, specifying image as parent
drange = imdisplayrange(img);

% Add Pixel Region tool panel, specifying figure as parent
% and image as target
pixreg = impixelregionpanel(fig,img);

% Reposition the Pixel Region tool to fit in the figure
% window, leaving room for the Pixel Information and
```

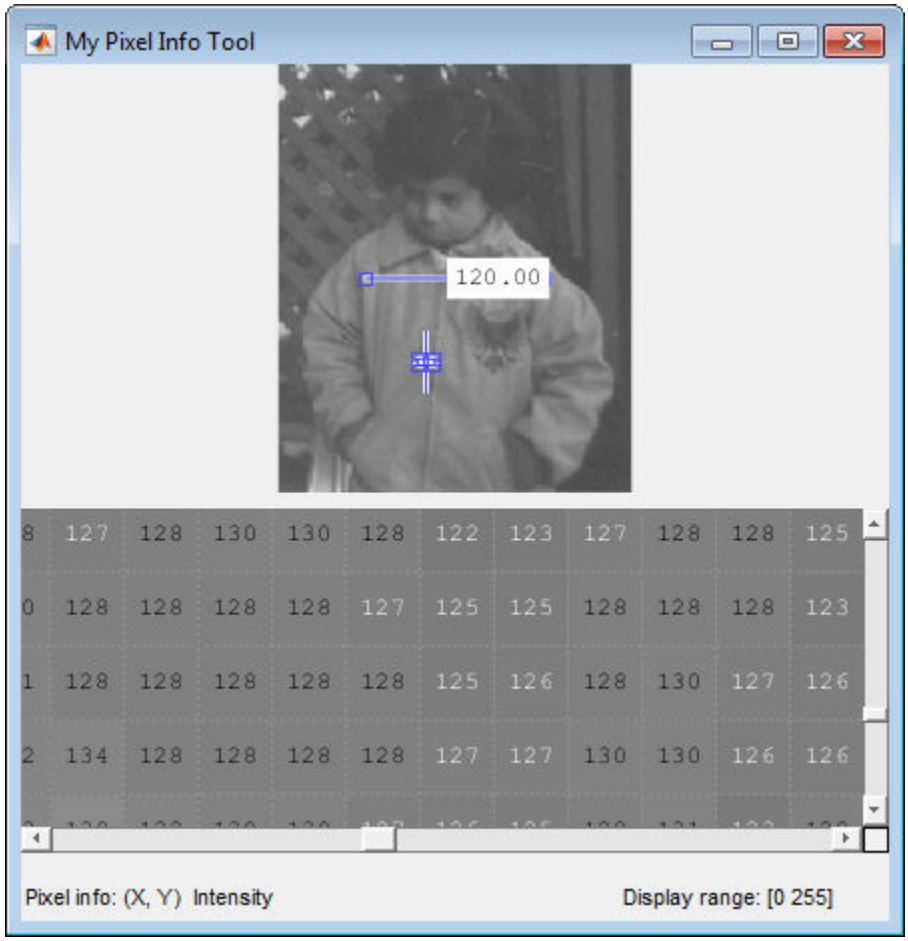
```
% Display Range tools.
set(pixreg, 'units','normalized','position',[0 .08 1 .4])
```

Read an image into the workspace.

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

Use the app to display the image with pixel information tools. 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.

```
my_pixinfotool(pout)
```


My Pixel Info Tool
120.00
8 127 128 130 130 128 122 123 127 128 128 125
0 128 128 128 128 127 125 125 128 128 128 123
1 128 128 128 128 128 125 126 128 130 127 126
2 134 128 128 128 128 127 127 130 130 126 126
Pixel info: (X, Y) Intensity
Display range: [0 255]

Build App for Navigating Large Images

This example shows how to build an app for navigating large images using modular navigation tools. 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.

Create a function that accepts an image as an argument and displays the image in a figure window with scroll bars, an Overview tool, and a Magnification box.

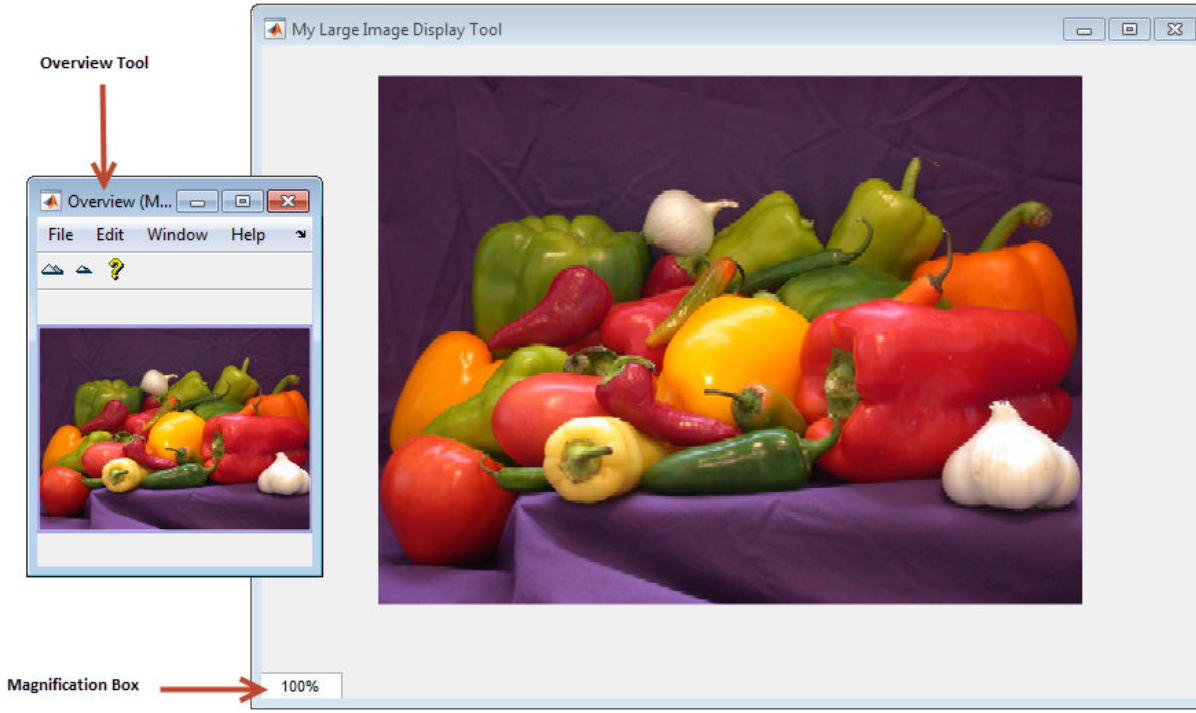
```
function my_large_image_display(im)

% 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);
```

Read an image into the workspace.

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

Use the app to display the image with navigation aids.

```
my_large_image_display(big_image)
```



Customize Modular Tool Interactivity

When you create a modular tool and associate it with a target image, the tool automatically makes the necessary connections to the target image to do its job. For example, the Pixel Information tool sets up a connection to the target image so that it can display the location and value of the pixel currently under the 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 the indicate the portion of the image that is now visible.
- **Overview tool to the target image** — If a user moves the detail rectangle in the Overview tool, the portion of the target image visible in the scroll panel is updated.

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

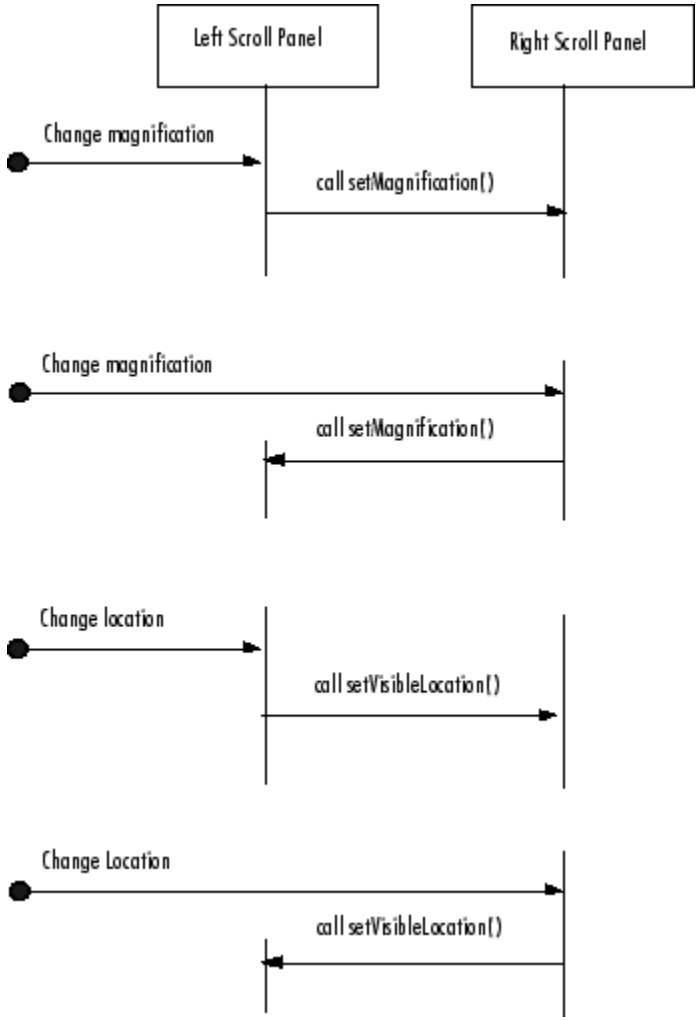
To illustrate how to use callbacks to make the connections required for interactions between tools, this example uses the Scroll Panel API to build a simple image comparison GUI. This custom tool displays two images side by side in scroll panels that are synchronized in location and magnification. The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

% 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.

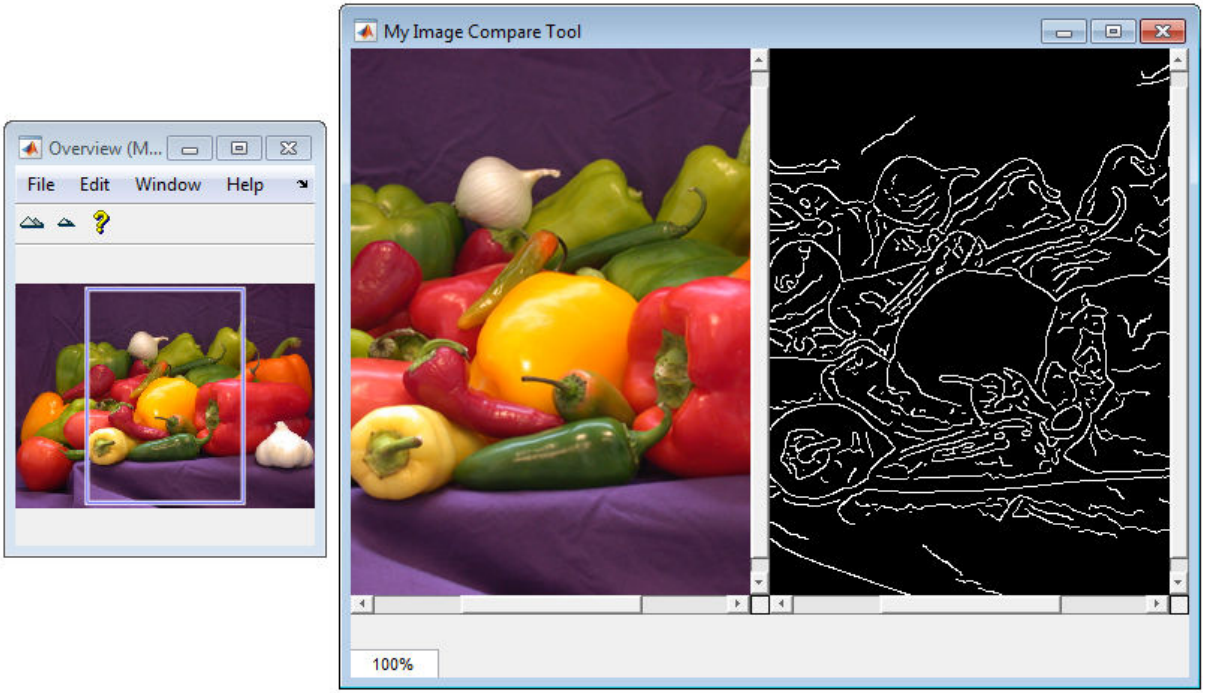


Scroll Panel Connections in Custom Image Comparison Tool

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.



Create Your Own Modular Tools

Because the toolbox uses an open architecture for the modular interactive tools, you can extend the toolbox by creating your own modular interactive tools, using 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 Using ROI Objects” on page 5-39 for an example.

The following table lists these utility functions in alphabetical order. See the function's reference page for more detailed information.

Utility Function	Description
`getimagemodel`	Retrieve image model object from image object.
`getrangefromclass`	Get default display range of image, based on its class
`imagemodel`	Access to properties of an image relevant to its display
`imattributes`	Return information about image attributes
`imellipse`	Create draggable, resizable ellipse
`imfreehand`	Create draggable freehand region
`imgca`	Get current axes containing an image
`imgcf`	Get most recent current figure containing an image
`imgetfile`	Display Open Image dialog box
`imhandles`	Get all image objects
`imline`	Create draggable, resizable line
`impoint`	Create draggable point
`impoly`	Create draggable, resizable polygon
`imputfile`	Display Save Image dialog box
`imrect`	Create draggable, resizable rectangle
`iptaddcallback`	Add function handle to a callback list

Utility Function	Description
`iptcheckconn`	Check validity of connectivity argument
`iptcheckhandle`	Check validity of object argument
`iptcheckmap`	Check validity of colormap argument
`iptgetapi`	Get application programmer interface (API) for an object
`iptGetPointerBehavior`	Retrieve pointer behavior from graphics object
`ipticondir`	Return names of directories containing IPT and MATLAB icons
`iptnum2ordinal`	Convert positive integer to ordinal value
`iptPointerManager`	Install mouse pointer manager in figure
`iptremovecallback`	Delete function handle from callback list
`iptSetPointerBehavior`	Store pointer behavior in graphics object
`iptwindowalign`	Align figure windows
`narginchk`	Check number of input arguments
`validateattributes`	Check validity of input argument
`validatestring`	Check validity of character vector arguments

Create Angle Measurement Tool Using ROI Objects

This example shows how to create an angle measurement tool using modular tools and ROI objects. The example displays an image in a figure window and overlays a simple angle measurement tool over the image. When you move the lines in the angle measurement tool, the function calculates the angle formed by the two lines and displays the angle in a title.

Create a function that accepts an image as an argument and displays an angle measurement tool over the image in a figure window. This code includes a second function used as a callback function that calculates the angle and displays the angle in the figure.

```
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 % Display image
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)
```

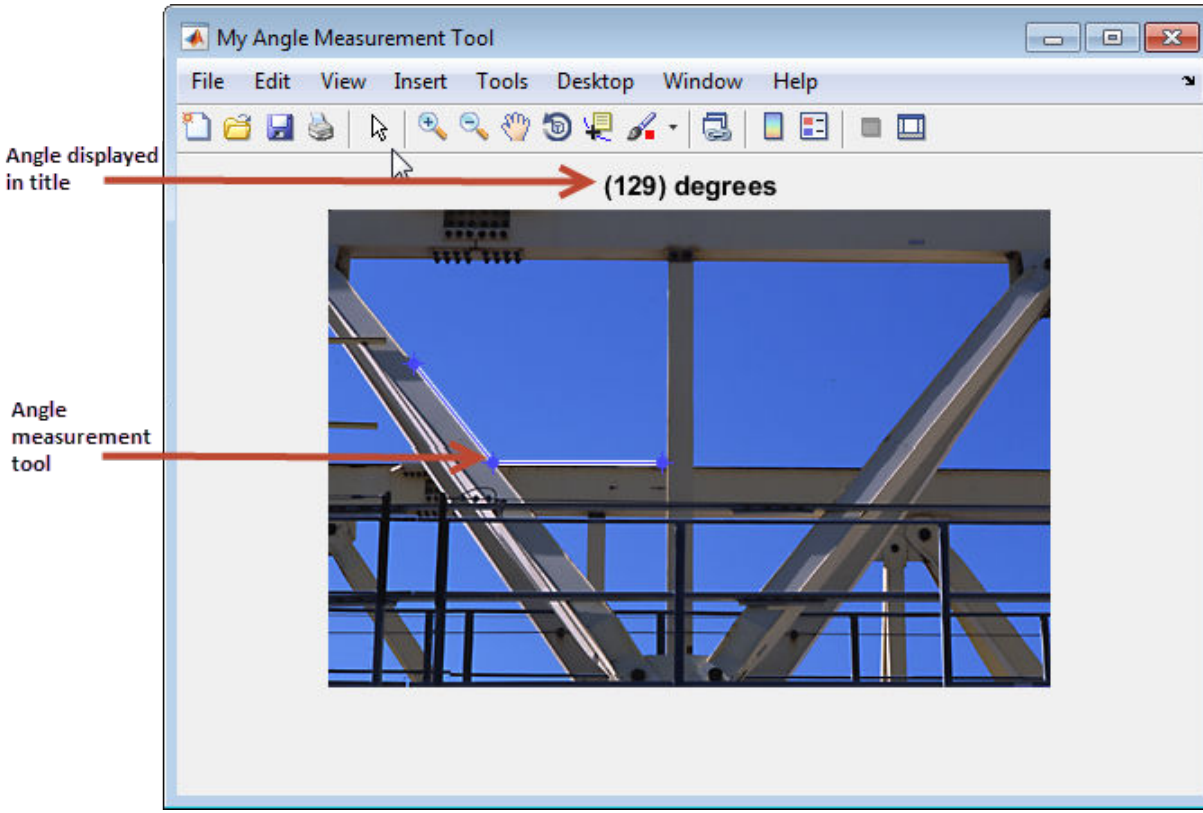
```
% set up callback to update angle in title.
api.addNewPositionCallback(@updateAngle);
fcn = makeConstrainToRectFcn('impoly',get(gca,'XLim'),get(gca,'YLim'));
api.setPositionConstraintFcn(fcn);
%

% 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))
```

Read image into the workspace.

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

Open the angle measurement tool, specifying the image as an argument. 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.

```
my_angle_measurement_tool(I);
```



See Also

More About

- “Create Your Own Modular Tools” on page 5-37

Geometric Transformations

A geometric transformation (also known as a spatial transformation) modifies the spatial relationship between pixels in an image, mapping pixel locations in a 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 with imresize Function” on page 6-2
- “Rotate an Image” on page 6-6
- “Crop an Image” on page 6-9
- “Translate an Image using imtranslate Function” on page 6-11
- “2-D and 3-D Geometric Transformation Process Overview” on page 6-14
- “Matrix Representation of Geometric Transformations” on page 6-19
- “Specify Fill Values in Geometric Transformation Output” on page 6-27
- “Perform Simple 2-D Translation Transformation” on page 6-30
- “N-Dimensional Spatial Transformations” on page 6-35
- “Register Two Images Using Spatial Referencing to Enhance Display” on page 6-37
- “Create a Gallery of Transformed Images” on page 6-44
- “Exploring a Conformal Mapping” on page 6-65
- “Exploring Slices from a 3-Dimensional MRI Data Set” on page 6-79
- “Padding and Shearing an Image Simultaneously” on page 6-88

Resize an Image with imresize Function

This example shows how to resize an image using the `imresize` function.

Specify the Magnification Value

Read an image into the workspace.

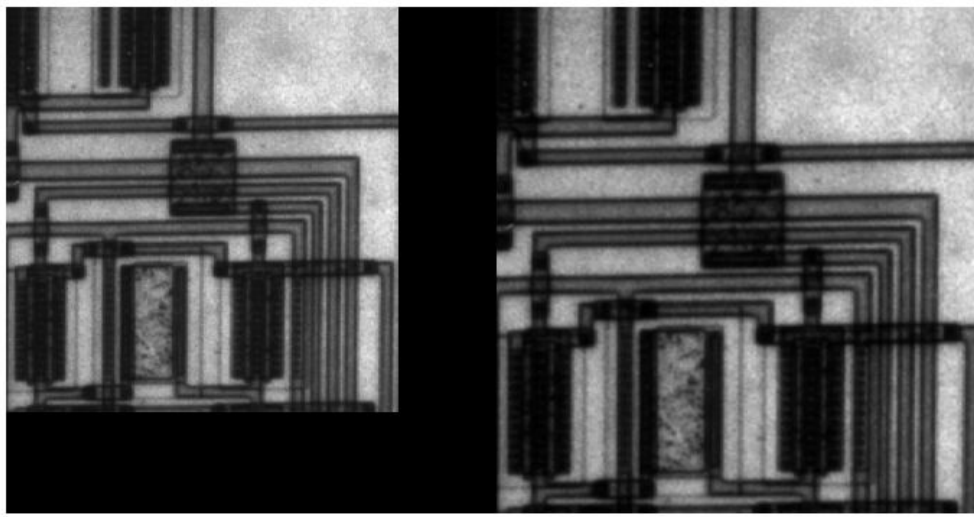
```
I = imread('circuit.tif');
```

Resize the image, using the `imresize` function. In this example, you specify a magnification factor. To enlarge an image, specify a magnification factor greater than 1.

```
J = imresize(I,1.25);
```

Display the original image next to the enlarged version.

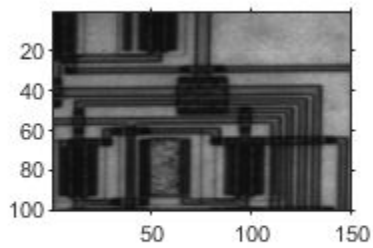
```
figure
imshowpair(I,J,'montage')
axis off
```



Specify the Size of the Output Image

Resize the image again, this time specifying the desired size of the output image, rather than a magnification value. Pass `imresize` 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. To perform the resizing required for multi-resolution processing, use `impyramid`.

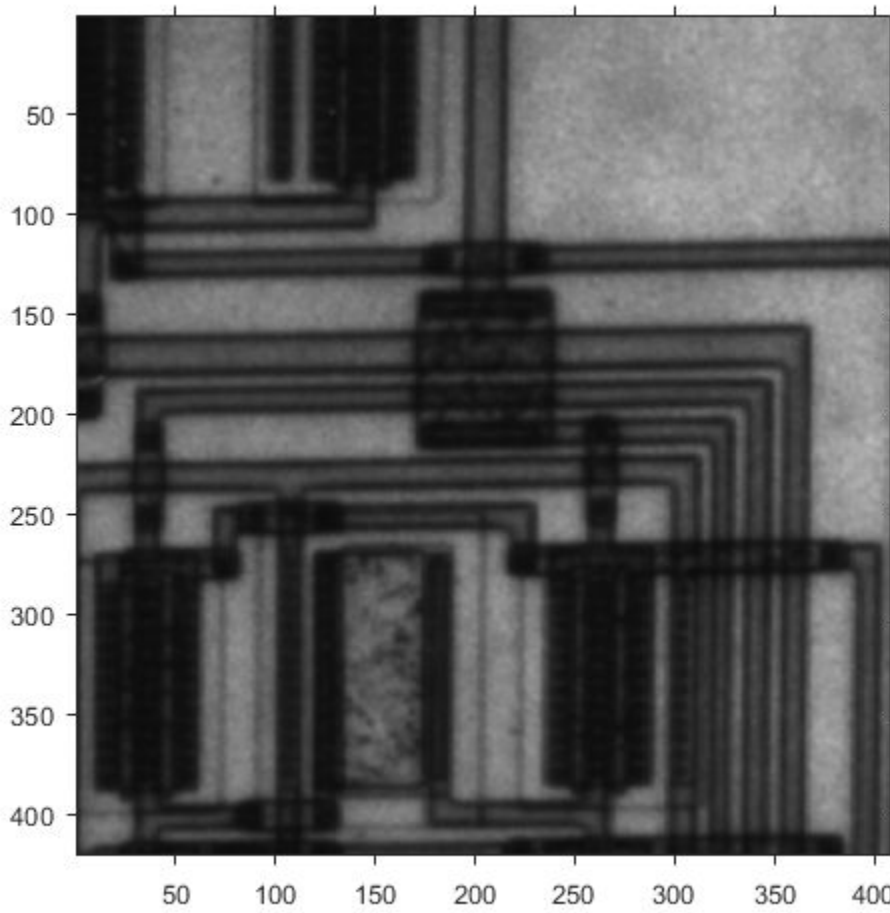
```
K = imresize(I,[100 150]);
figure, imshow(K)
```



Specify the Interpolation Method

Resize the image again, this time specifying the interpolation method. When you enlarge an image, the output image contains more pixels than the original image. `imresize` uses interpolation to determine the values of these pixels, computing a weighted average of some set of pixels in the vicinity of the pixel location. `imresize` bases the weightings on the distance each pixel is from the point. By default, `imresize` uses bicubic interpolation, but you can specify other interpolation methods or interpolation kernels. See the `imresize` reference page for a complete list. You can also specify your own custom interpolation kernel. This example use bilinear interpolation.

```
L = imresize(I,1.5,'bilinear');
figure, imshow(L)
```

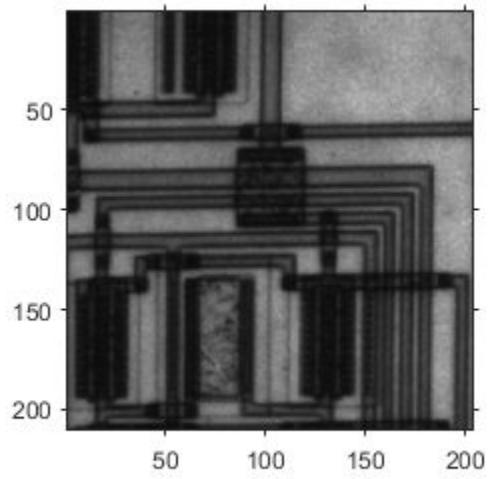


Prevent Aliasing When Shrinking an Image

Resize the image again, this time shrinking the image. When you reduce the size of an image, you lose some of the original pixels because there are fewer pixels in the output image. This can introduce artifacts, such as aliasing. The aliasing that occurs as a result of size reduction normally appears as stair-step patterns (especially in high-contrast images), or as moire (ripple-effect) patterns 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. Even with antialiasing turned on, resizing can introduce artifacts because information is always lost when you reduce the size of an image.

```
M = imresize(I,.75,'Antialiasing',false);
figure, imshow(M)
```



Rotate an Image

This example shows how to rotate an image using 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, the image rotates counterclockwise; if you specify a negative rotation angle, the image rotates clockwise.

By default, the output image is 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.

By default, `imrotate` uses nearest-neighbor interpolation 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.

Rotate an Image Counterclockwise

Read an image into the workspace.

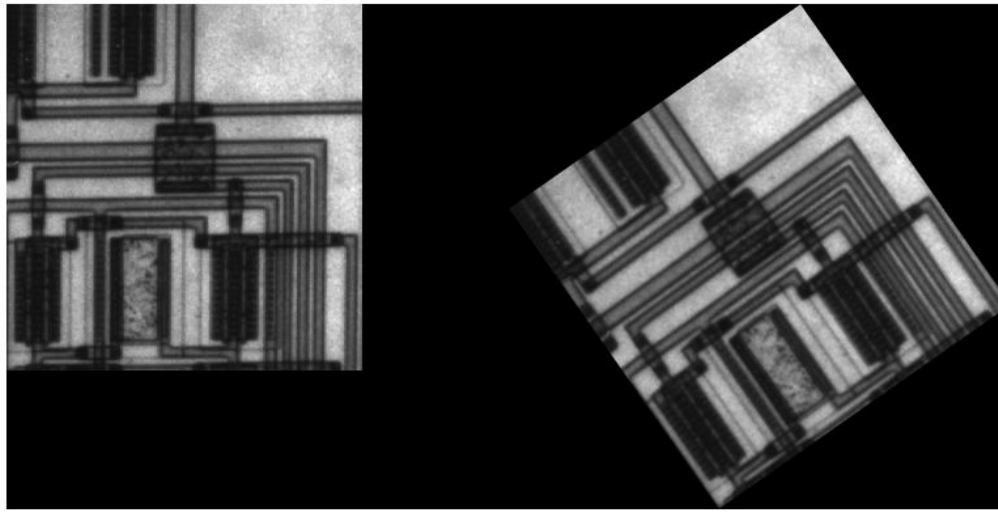
```
I = imread('circuit.tif');
```

Rotate the image 35 degrees counterclockwise. In this example, specify bilinear interpolation.

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

Display the original image and the rotated image.

```
figure
imshowpair(I,J,'montage')
```



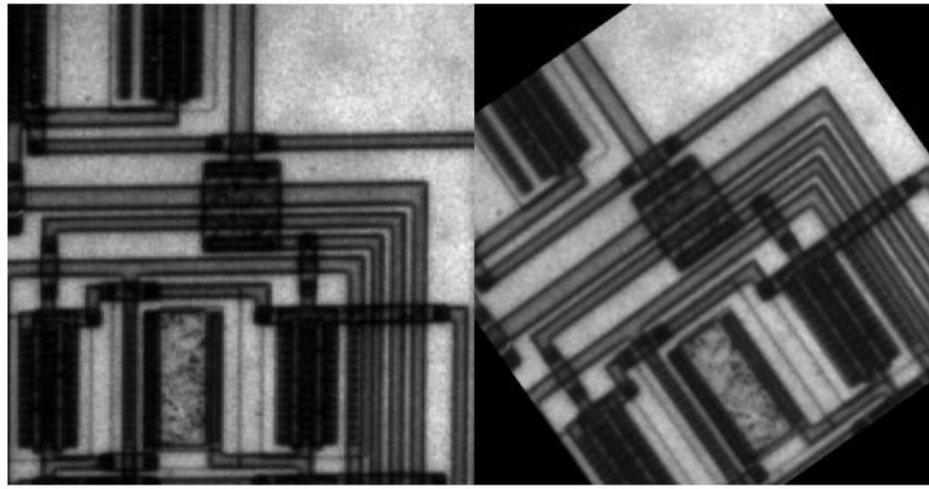
Crop a Rotated Image

Rotate the original image 35 degrees counterclockwise, specifying that the rotated image be cropped to the same size as the original image.

```
K = imrotate(I,35,'bilinear','crop');
```

Display the original image and the new image.

```
figure
imshowpair(I,K,'montage')
```



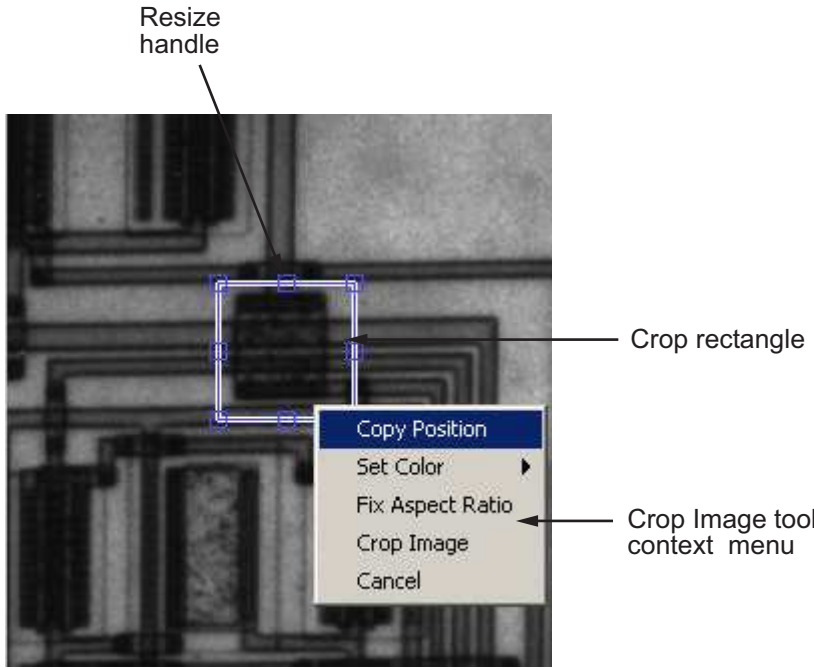
Crop an Image

Note You can also crop an image interactively using the Image Tool — see “Crop Image Using Image Viewer App” on page 4-72.

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 using imtranslate Function

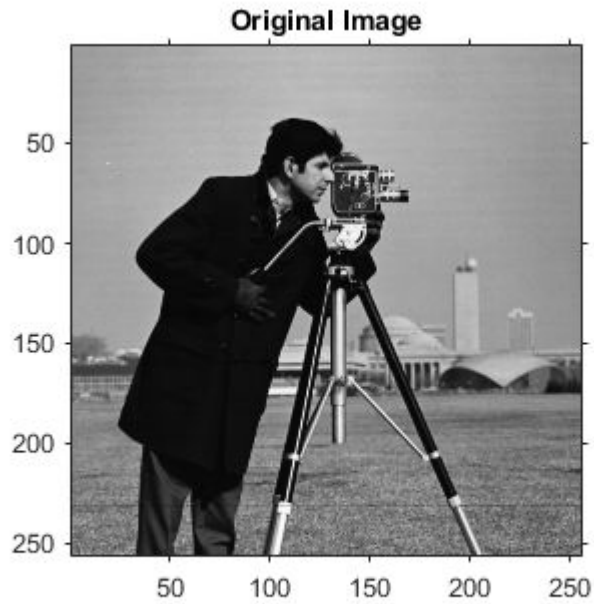
This example shows how to perform a translation operation on an image using the `imtranslate` function. A translation operation shifts an image by a specified number of pixels in either the *x*- or *y*-direction, or both.

Read an image into the workspace.

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

Display the image. The size of the image is 256-by-256 pixels. By default, `imshow` displays the image with the upper right corner at (0,0).

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



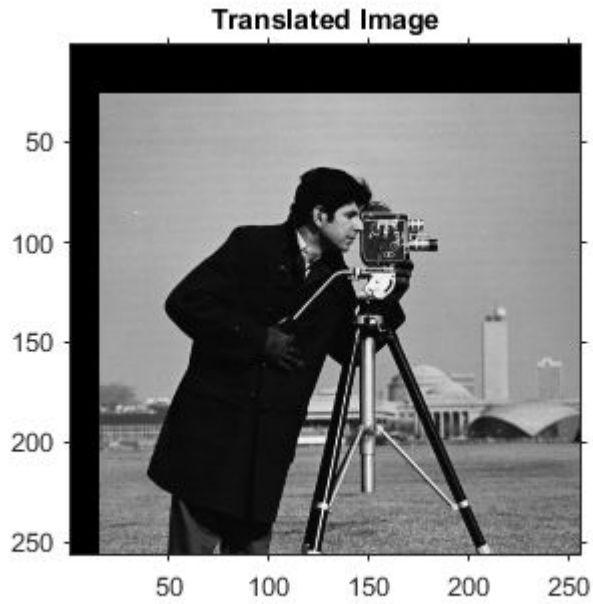
Translate the image, shifting the image by 15 pixels in the *x*-direction and 25 pixels in the *y*-direction. Note that, by default, `imtranslate` displays the translated image within the

boundaries (or limits) of the original 256-by-256 image. This results in some of the translated image being clipped.

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

Display the translated image. The size of the image is 256-by-256 pixels.

```
figure
imshow(J)
title('Translated Image')
```

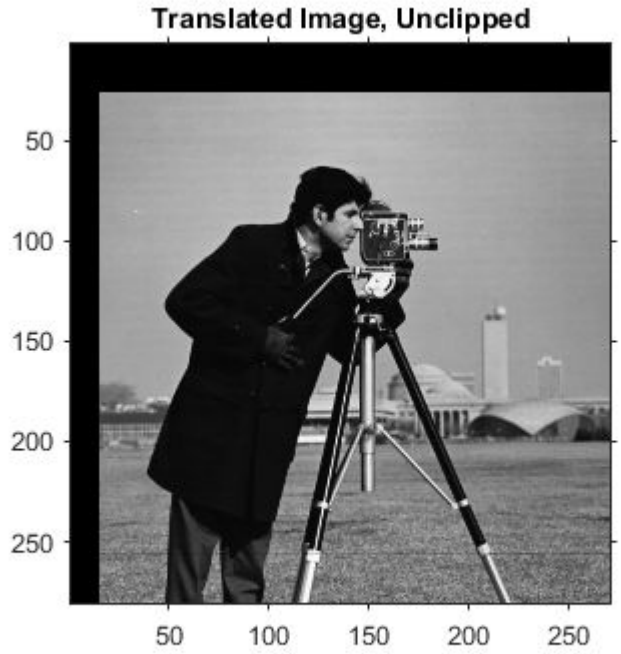


Use the `'OutputView'` parameter set to `'full'` to prevent clipping the translated image. The size of the new image is 281-by-271 pixels.

```
K = imtranslate(I,[15, 25],'OutputView','full');
```

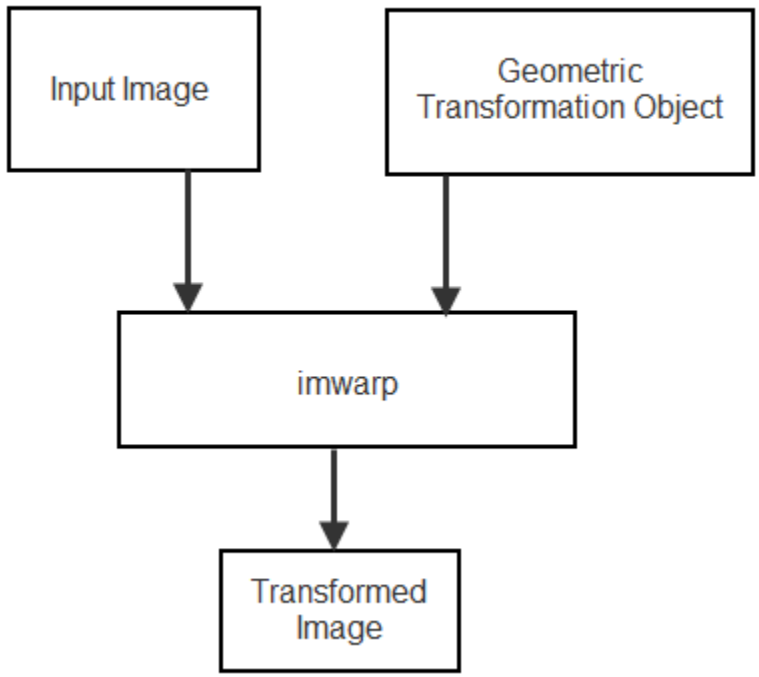
Display the translated image.

```
figure
imshow(K)
title('Translated Image, Unclipped')
```



2-D and 3-D Geometric Transformation Process Overview

To perform a 2-D or 3-D geometric transformation, first create a geometric transformation object that stores information about the transformation. Then, pass the image to be transformed and the geometric transformation object to the `imwarp` function. You optionally can provide spatial referencing information about the input image to `imwarp`.



Create Geometric Transformation Object

Different types of geometric transformation objects store different information about the transformation.

- The `affine2d`, `projective2d`, and `affine3d` objects store a transformation matrix.

- The `geometricTransform2d` and `geometricTransform3d` objects store an inverse point-wise mapping function, and optionally a forward point-wise mapping function.
- Other transformation types, including the `LocalWeightedMeanTransformation2D`, `PiecewiseLinearTransformation2D`, and `PolynomialTransformation2D` objects, store a mapping between control point pairs.

There are several ways to create a geometric transformation object.

Approach to Create Tranformation Object	`affine2d`	`affine3d`	`projective2d`	`geometricTransform2d` / `geometricTransform3d`	Other Transformation Objects
“Define Transformation Matrix” on page 6-15	X	X	X		
“Define Custom Point-Wise Mapping Function” on page 6-16				X	
“Estimate Transformation from Control Point Pairs” on page 6-17	X		X		X
“Estimate Transformation Using Similarity Optimization” on page 6-17	X	X			
“Estimate Transformation Using Phase Correlation” on page 6-18	X				

Define Transformation Matrix

If you know the transformation matrix for the geometric transformation you want to perform, then you can create a `affine2d`, `projective2d`, or `affine3d` geometric transformation object directly. For more information about creating a transformation matrix, see “Matrix Representation of Geometric Transformations” on page 6-19.

The following example defines the transformation matrix for a 2-D 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
```

Define Custom Point-Wise Mapping Function

If you have an inverse point-wise mapping function, then you can define a custom 2-D and 3-D geometric transformation using the `geometricTransform2d` and the `geometricTransform3d` objects respectively.

The following example specifies an inverse mapping function that accepts and returns 2-D points in packed (*x*,*y*) format. Then, the example creates a `geometricTransform2d` custom geometric transformation object.

```
inversefn = @(c) [c(:,1)+c(:,2),c(:,1).^2]
```

```
inversefn = function_handle with value:
    @(c)[c(:,1)+c(:,2),c(:,1).^2]
```

```
tform = geometricTransform2d(inversefn)
```

```
tform =
  geometricTransform2d with properties:

        InverseFcn: @(c)[c(:,1)+c(:,2),c(:,1)-c(:,2)]
        ForwardFcn: []
    Dimensionality: 2
```

Similarly, the following example creates a `geometricTransform3d` custom geometric transformation object using the inverse mapping function. The example specifies an inverse mapping function that accepts and returns 3-D points in packed (*x*,*y*,*z*) format.

```
inversefn = @(c)[c(:,1)+c(:,2),c(:,1)-c(:,2),c(:,3).^2]
```

```
inversefn = function_handle with value:
    @(c)[c(:,1)+c(:,2),c(:,1)-c(:,2),c(:,3).^2]
```

```
tform = geometricTransform3d(inversefn)
```

```
tform = 

  geometricTransform3d with properties:

        InverseFcn: @(c)[c(:,1)+c(:,2),c(:,1)-c(:,2),c(:,3).^2]
        ForwardFcn: []
    Dimensionality: 3
```

Estimate Transformation from Control Point Pairs

You can create a geometric transformation object by passing two sets of control point pairs to the `fitgeotrans` function. The `fitgeotrans` function automatically 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 `affine2d` geometric transformation object.

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

tform_cpp = fitgeotrans(movingPoints,fixedPoints,'affine')
```

```
tform_cpp = 

  affine2d with properties:

                 T: [3x3 double]
    Dimensionality: 2
```

Estimate Transformation Using Similarity Optimization

If you have a fixed image and a moving image that are slightly misaligned, then you can use the `imregtform` function to estimate an affine geometric transformation that aligns the images. `imregtform` optimizes the mean squares or Mattes mutual information similarity metric of the two images, using a regular step gradient descent or one-plus-one evolutionary optimizer. For more information, see “Create an Optimizer and Metric for Intensity-Based Image Registration” on page 7-34.

Estimate Transformation Using Phase Correlation

If you have a fixed image and a moving image that are severely misaligned, then you can use the `imregcorr` function to estimate an affine geometric transformation that improves the image alignment. You can refine the resulting transformation by using similarity optimization.

Perform the Geometric Transformation

After you define the transformation in a geometric transformation object, perform the transformation by using the `imwarp` function. When calling the function, specify the image to be transformed and the geometric transformation object.

`imwarp` uses the geometric transformation to map coordinates in the output image to the corresponding coordinates in the input image (inverse mapping). Then, `imwarp` uses the coordinate mapping to interpolate pixel values within the input image and compute the output pixel value.

See Also

`affine2d` | `affine3d` | `fitgeotrans` | `geometricTransform2d` | `geometricTransform3d` | `imwarp` | `projective2d`

Related Examples

- “Perform Simple 2-D Translation Transformation” on page 6-30

More About

- “Matrix Representation of Geometric Transformations” on page 6-19
- “N-Dimensional Spatial Transformations” on page 6-35

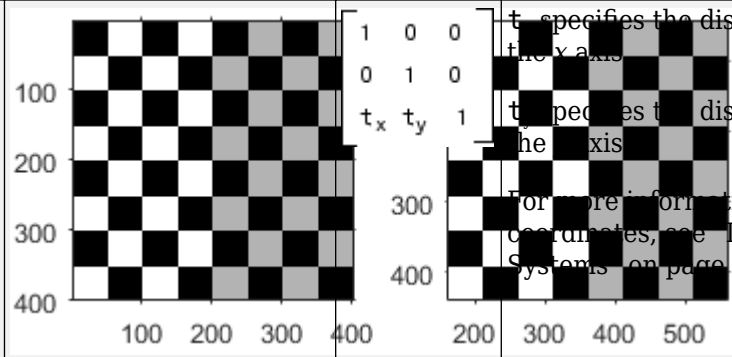
Matrix Representation of Geometric Transformations

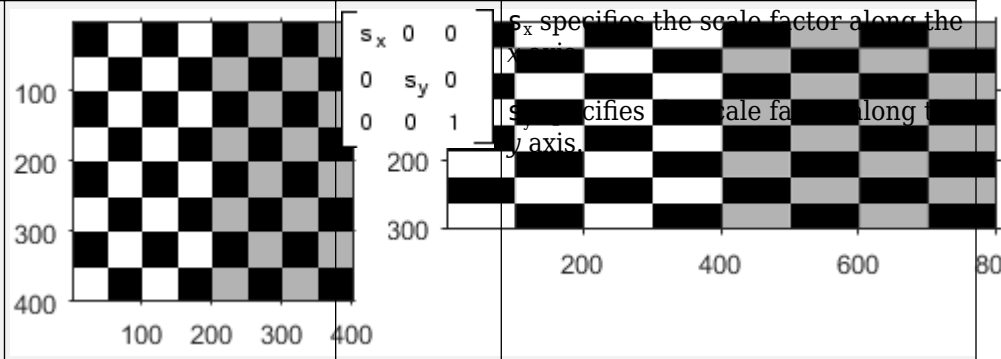
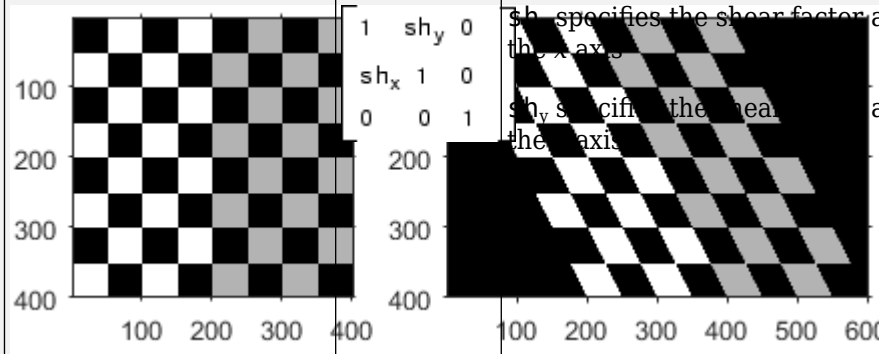
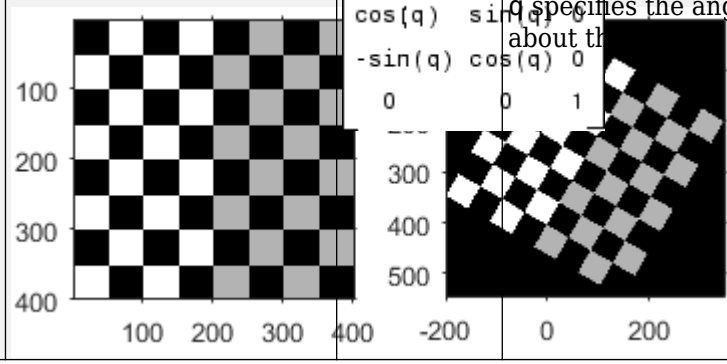
You can use a geometric transformation matrix to perform a global transformation of an image. First, define a transformation matrix and use it to create a geometric transformation object. Then, apply a global transformation to an image by calling `imwarp` with the geometric transformation object. For an example, see “Perform Simple 2-D Translation Transformation” on page 6-30.

2-D Affine Transformations

The table lists 2-D affine transformations with the transformation matrix used to define them. For 2-D affine transformations, the last column must contain [0 0 1] homogeneous coordinates.

Use the transformation matrix to create an `affine2d` geometric transformation object.

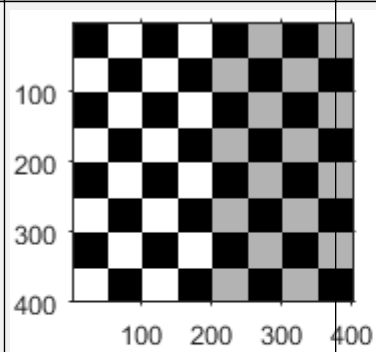
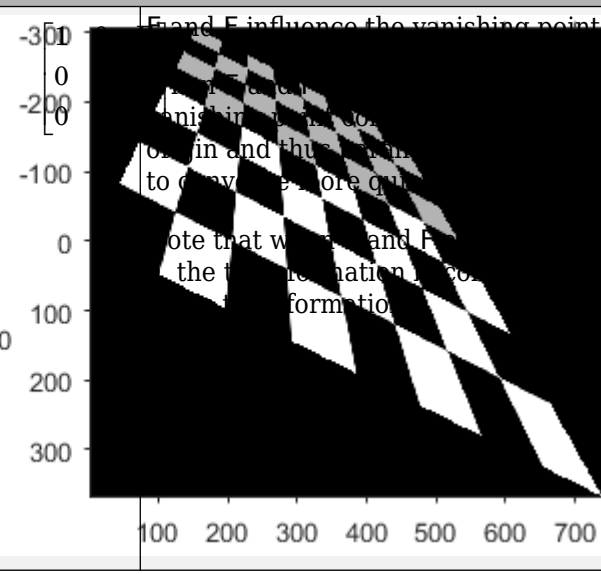
2-D Affine Transformation	Example (Original and Transformed Image)	Transformation Matrix
Translation		$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$ t_x specifies the displacement along the x axis. t_y specifies the displacement along the y axis. For more information about pixel coordinates, see “Image Coordinate Systems” on page 2-3.

2-D Affine Transformation	Example (Original and Transformed Image)	Transformation Matrix
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 about the

2-D Projective Transformations

Projective transformation enables the plane of the image to tilt. Parallel lines can converge towards a vanishing point, creating the appearance of depth.

The transformation is a 3-by-3 matrix. Unlike affine transformations, there are no restrictions on the last column of the transformation matrix.

2-D Projective Transformation	Example	Transformation Matrix
Tilt		$\begin{bmatrix} 1 & 0 & 0 \\ [illegible] & [illegible] & [illegible] \\ [illegible] & [illegible] & [illegible] \end{bmatrix}$ 

Projective transformations are frequently used to register images that are out of alignment. If you have two images that you would like to align, first select control point pairs using `cpselect`. Then, fit a projective transformation matrix to control point pairs using `fitgeotrans` and setting the `transformationType` to `'projective'`. This automatically creates a `projective2d` geometric transformation object. The transformation matrix is stored as a property in the `projective2d` object. The transformation can then be applied to other images using `imwarp`.

Create Composite 2-D Affine Transformations

You can combine multiple transformations into a single matrix using matrix multiplication. The order of the matrix multiplication matters.

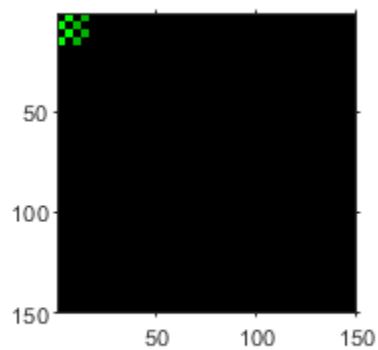
This example shows how to create a composite of 2-D translation and rotation transformations.

Create a checkerboard image that will undergo transformation. Also create a spatial reference object for the image.

```
cb = checkerboard(4,2);
cb_ref = imref2d(size(cb));
```

To illustrate the spatial position of the image, create a flat background image. Overlay the checkerboard over the background, highlighting the position of the checkerboard in green.

```
background = zeros(150);
imshowpair(cb,cb_ref,background,imref2d(size(background)))
```



Create a translation matrix, and store it as an `affine2d` geometric transformation object. This translation will shift the image horizontally by 100 pixels.

```
T = [1 0 0;0 1 0;100 0 1];
tform_t = affine2d(T);
```

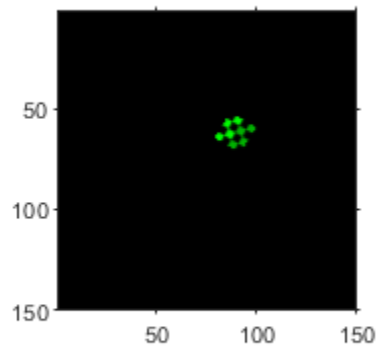
Create a rotation matrix, and store it as an `affine2d` geometric transformation object. This translation will rotate the image 30 degrees clockwise about the origin.

```
R = [cosd(30) sind(30) 0;-sind(30) cosd(30) 0;0 0 1];
tform_r = affine2d(R);
```

Translation Followed by Rotation

Perform translation first and rotation second. In the multiplication of the transformation matrices, the translation matrix T is on the left, and the rotation matrix R is on the right.

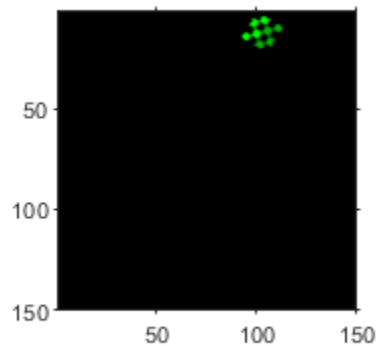
```
TR = T*R;
tform_tr = affine2d(TR);
[out,out_ref] = imwarp(cb,cb_ref,tform_tr);
imshowpair(out,out_ref,background,imref2d(size(background)))
```



Rotation Followed by Translation

Reverse the order of the transformations: perform rotation first and translation second. In the multiplication of the transformation matrices, the rotation matrix R is on the left, and the translation matrix T is on the right.

```
RT = R*T;
tform_rt = affine2d(RT);
[out,out_ref] = imwarp(cb,cb_ref,tform_rt);
imshowpair(out,out_ref,background,imref2d(size(background)))
```



Notice how the spatial position of the transformed image is different than when translation was followed by rotation.

3-D Affine Transformations

The following table lists the 3-D affine transformations with the transformation matrix used to define them. Note that in the 3-D case, there are multiple matrices, depending on how you want to rotate or shear the image. The last column must contain [0 0 0 1].

Use the transformation matrix to create an `affine3d` geometric transformation object.

3-D Affine Transformation	Transformation Matrix		
Translation	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$		

3-D Affine Transformation	Transformation Matrix		
Scale	$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$		
Shear	x,y shear: $x' = x + az$ $y' = y + bz$ $z' = z$ $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a & b & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	x,z shear: $x' = x + ay$ $y' = y$ $z' = z + cy$ $\begin{bmatrix} 1 & 0 & 0 & 0 \\ a & 1 & c & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	y, z shear: $x' = x$ $y' = y + bx$ $z' = z + cx$ $\begin{bmatrix} 1 & b & c & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Rotation	About x axis: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(a) & \sin(a) & 0 \\ 0 & -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	About y axis: $\begin{bmatrix} \cos(a) & 0 & -\sin(a) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(a) & 0 & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	About z axis: $\begin{bmatrix} \cos(a) & \sin(a) & 0 & 0 \\ -sin(a) & \cos(a) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

For N-D affine transformations, the last column must contain `[zeros(N,1); 1]`. `imwarp` does not support transformations of more than three dimensions.

See Also

`fitgeotrans` | `imwarp`

Related Examples

- “Perform Simple 2-D Translation Transformation” on page 6-30

More About

- “2-D and 3-D Geometric Transformation Process Overview” on page 6-14

Specify Fill Values in Geometric Transformation Output

This example shows how to specify the fill values used by `imwarp` when it performs a geometric transformation. 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 display as black. Using the `FillValues` parameter, you can specify a different color. If the image being transformed is a grayscale image, specify a scalar value that specifies a shade of gray. 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.

Read image into workspace. This example uses a color image.

```
rgb = imread('onion.png');
```

Create the transformation matrix. This matrix defines a translation transformation.

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

Create the geometric transformation object. This example creates an affine2d object.

```
tform_translate = affine2d(xform)
```

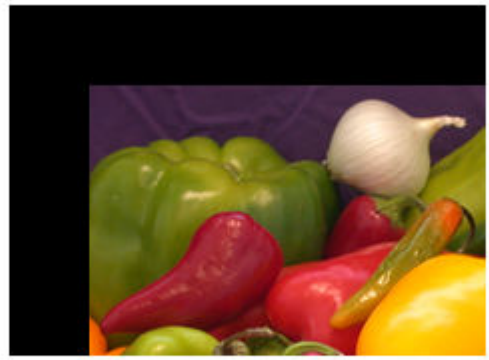
```
tform_translate = 
  affine2d with properties:

                 T: [3x3 double]
    Dimensionality: 2
```

Create a 2D referencing object. This object specifies aspects of the coordinate system of the output space so that the area needing fill values is visible. By default, `imwarp` sizes the output image to be just large enough to contain the entire transformed image but not the entire output coordinate space.

```
Rout = imref2d(size(rgb));
Rout.XWorldLimits(2) = Rout.XWorldLimits(2)+40;
Rout.YWorldLimits(2) = Rout.YWorldLimits(2)+40;
Rout.ImageSize = Rout.ImageSize+[40 40];
```

Perform the transformation with the `imwarp` function.

```
cb_rgb = imwarp(rgb,tform_translate,'OutputView',Rout);
figure, imshow(cb_rgb)
```



Now perform the transformation, this time specifying a fill value.

```
cb_fill = imwarp(rgb,tform_translate,'FillValues',[187;192;57],...
    'OutputView',Rout);

figure, imshow(cb_fill)
```

See Also

`affine2d` | `imref2d` | `imwarp`

More About

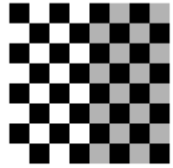
- “2-D and 3-D Geometric Transformation Process Overview” on page 6-14
- “Matrix Representation of Geometric Transformations” on page 6-19

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.

```
cb_ref = imref2d(size(cb))
```

```
cb_ref 
  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 3-by-3 transformation matrix, called `T` in this example, that defines the transformation. In this matrix, `T(3,1)` specifies the number of pixels to shift the image in the horizontal direction and `T(3,2)` specifies the number of pixels to shift the image in the vertical direction.

```
T = [1 0 0; 0 1 0; 20 30 1]
```

```
T = 3×3

     1     0     0
     0     1     0
    20    30     1
```

Create a geometric transformation object that defines the translation you want to perform. Because translation transformations are a special case of the affine transformation, the example uses an `affine2d` geometric transformation object to represent translation. Create an `affine2d` object by passing the 3-by-3 transformation matrix, `T`, to the `affine2d` constructor.

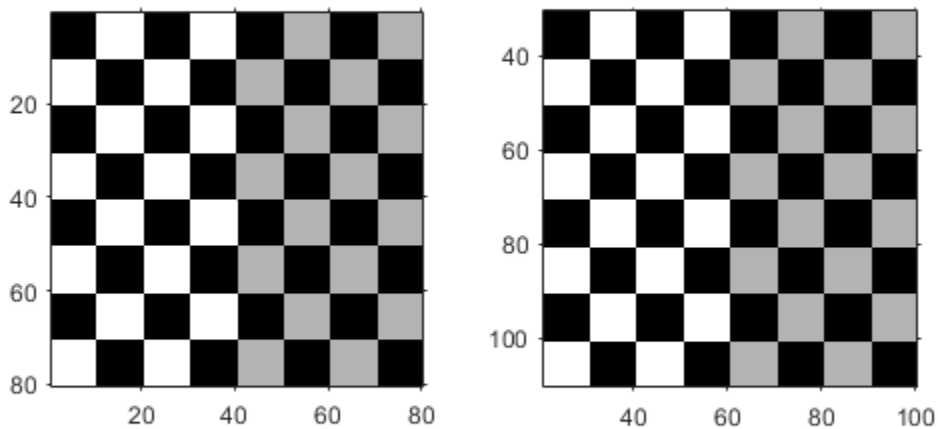
```
tform = affine2d(T);
```

Perform the transformation. Call the `imwarp` function specifying the image you want to transform and the geometric transformation object. `imwarp` returns the transformed image, `cb_translated`. This example also returns the optional spatial referencing object, `cb_translated_ref`, which contains spatial referencing information about the transformed image.

```
[cb_translated,cb_translated_ref] = imwarp(cb,tform);
```

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.

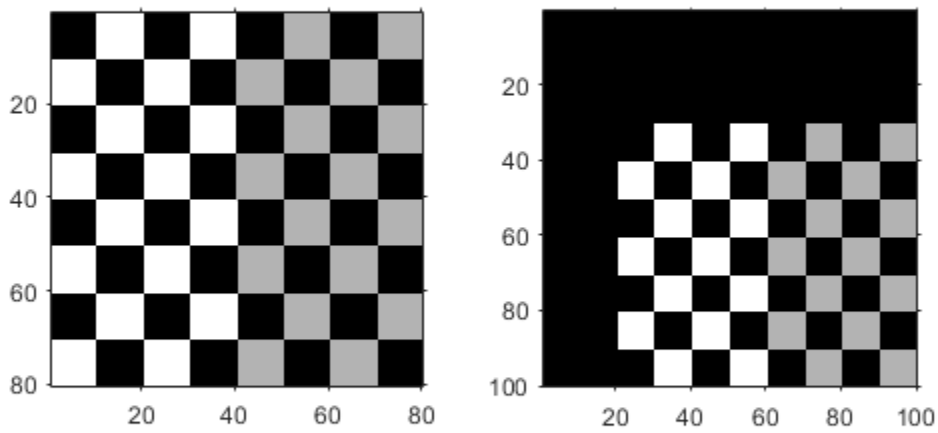
```
figure;
subplot(1,2,1);
imshow(cb,cb_ref);
subplot(1,2,2);
imshow(cb_translated,cb_translated_ref)
```



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

```
cb_translated_ref = cb_ref;
cb_translated_ref.XWorldLimits(2) = cb_translated_ref.XWorldLimits(2)+20;
cb_translated_ref.YWorldLimits(2) = cb_translated_ref.YWorldLimits(2)+20;
```

```
[cb_translated,cb_translated_ref] = imwarp(cb,tform,'OutputView',cb_translated_ref);

figure, subplot(1,2,1);
imshow(cb,cb_ref);
subplot(1,2,2);
imshow(cb_translated,cb_translated_ref)
```



See Also

affine2d | imref2d | imwarp

More About

- “2-D and 3-D Geometric Transformation Process Overview” on page 6-14
- “Matrix Representation of Geometric Transformations” on page 6-19

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`
- `makeresampler`
- `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 point-wise data.

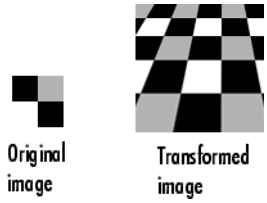
You can use `tformarray` to work with arbitrary-dimensional array transformations. The arrays do not need to have the same dimensions. The output can have either a lower or higher number of dimensions than the input. For example, if you are sampling 3-D data on a 2-D slice or manifold, the input array 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.

You can create a resampling structure using the `makeresampler` function 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.

The following example uses `imtransform` to perform a projective transformation of a checkerboard image, and `makeresampler` to create a resampling structure with a standard interpolation method.

```
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 = makeresampler('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.

See Also

More About

- “2-D and 3-D Geometric Transformation Process Overview” on page 6-14

Register Two Images Using Spatial Referencing to Enhance Display

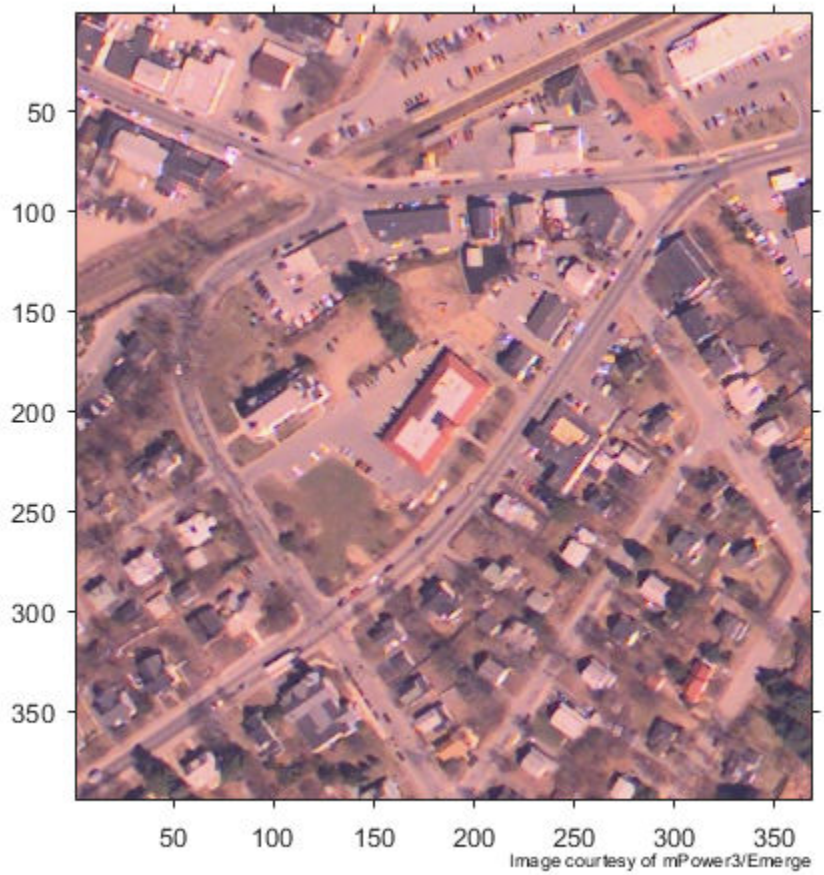
This example shows how to use spatial referencing objects to understand the spatial relationship between two images in image registration and display them effectively. This example brings one of the images, called the `moving` image, into alignment with the other image, called the `fixed` image.

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');
```



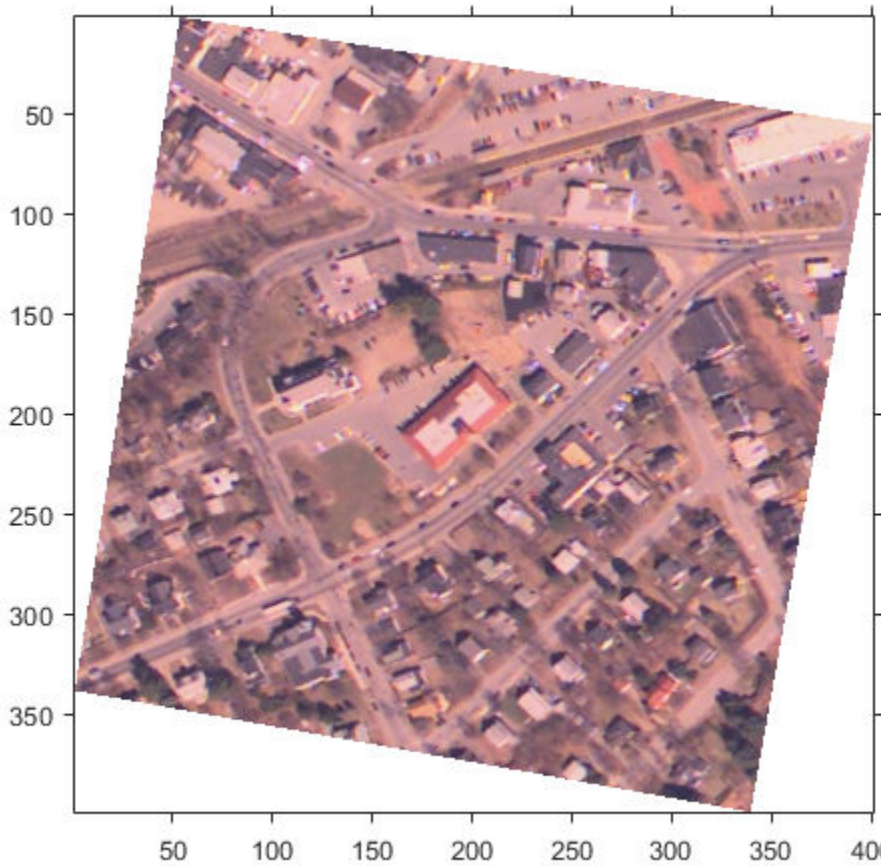
Load a MAT-file that contains preselected control points for the `fixed` and `moving` images and create a geometric transformation fit to the control points, using `fitgeotrans` .

```
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

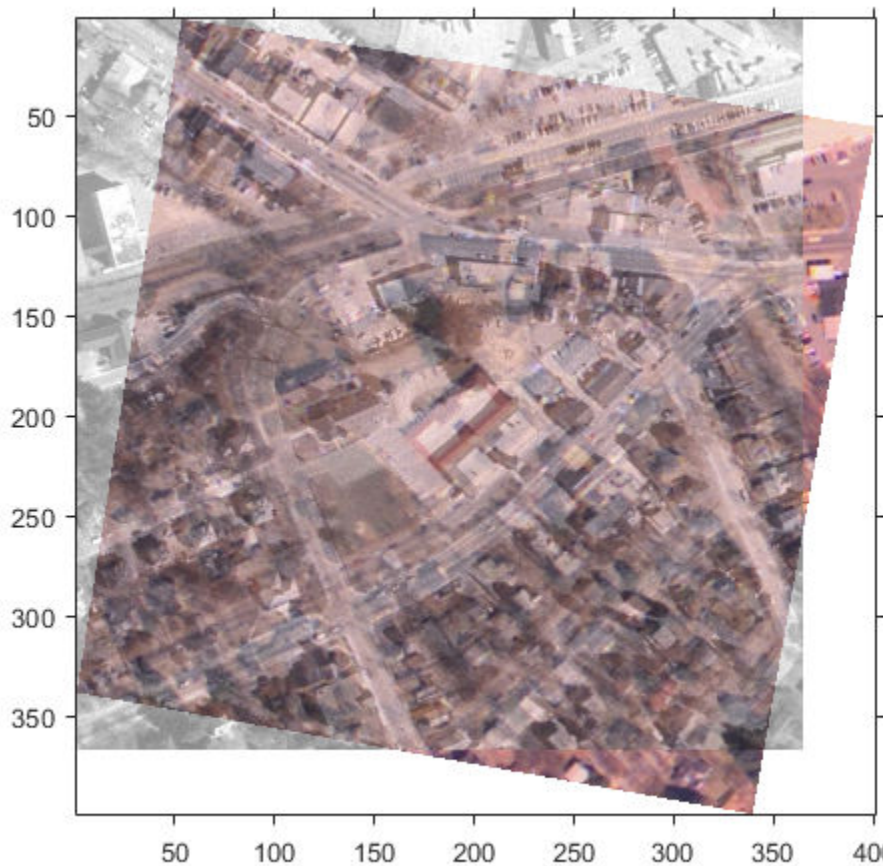
specify a fill value (white), which will help when displaying the `fixed` image over the transformed `moving` image, to check registration. Notice that the full content of the geometrically transformed `moving` image is present, now called `registered` . Also note that there are no blank rows or columns.

```
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.

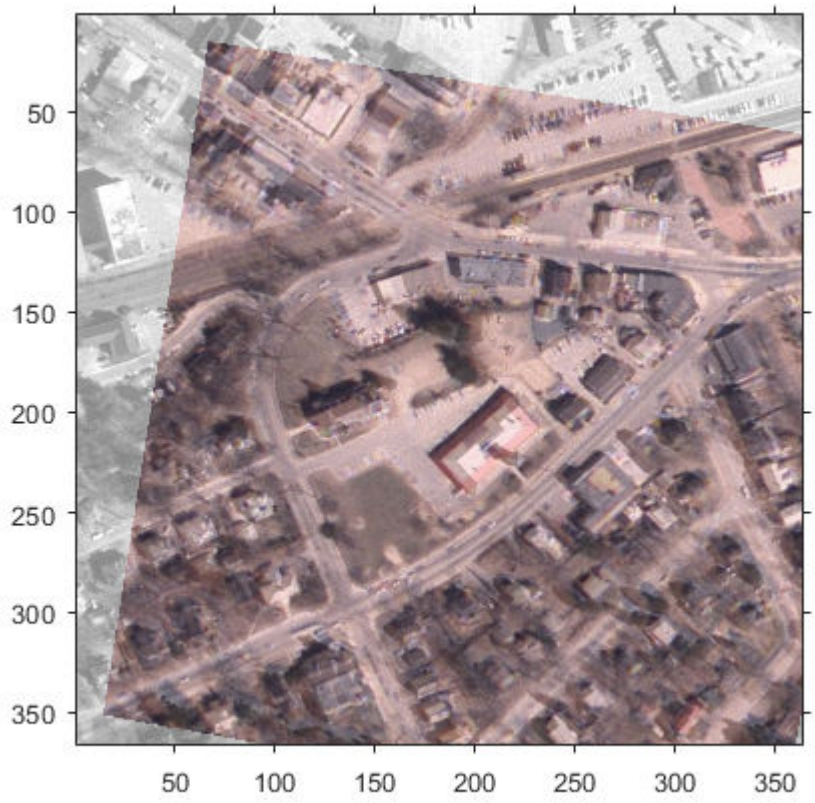
```
figure, imshowpair(fixed,registered,'blend');
```



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

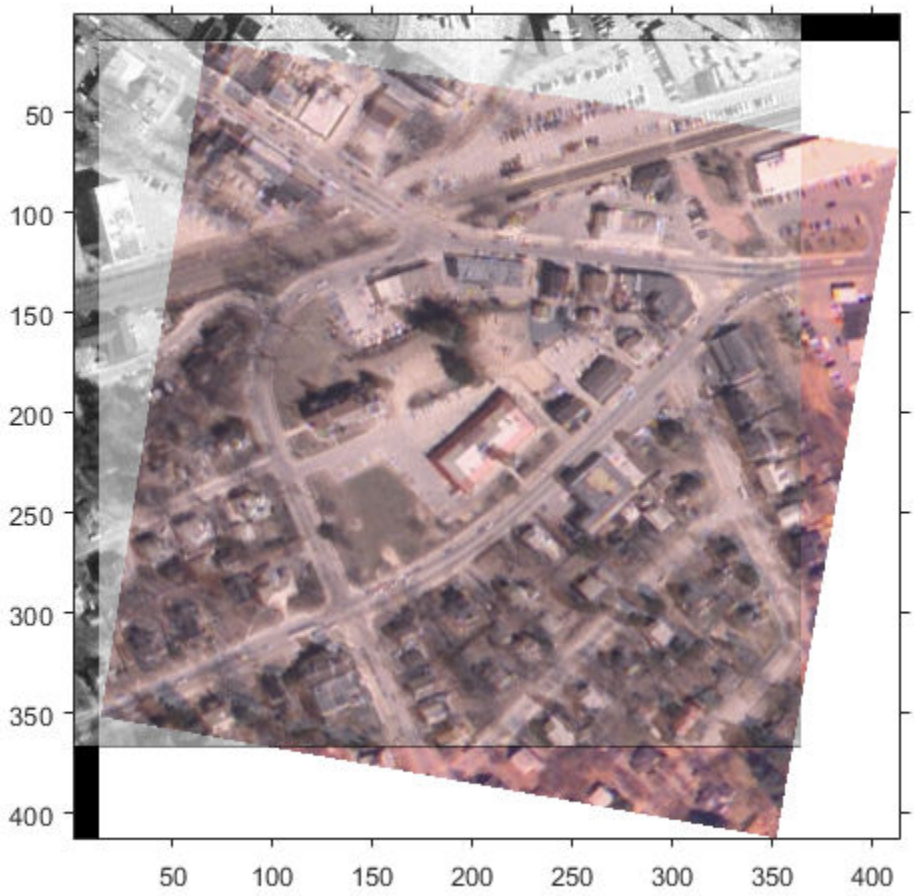
image appears registered with the fixed image but areas of the registered image that would extrapolate beyond the extent of the fixed image are discarded. To do this, create a default spatial referencing object that specifies the size and location of the `fixed` image, and use `imwarp's 'OutputView'` 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');
```



Clean up.

```
iptsetpref('ImshowAxesVisible','off')
```

Create a Gallery of Transformed Images

This example shows many properties of geometric transformations by applying different transformations to a checkerboard image.

Overview

A two-dimensional geometric transformation is a mapping that associates each point in a Euclidean plane with another point in a Euclidean plane. In these examples, the geometric transformation is defined by a rule that tells how to map the point with Cartesian coordinates (x,y) to another point with Cartesian coordinates (u,v). A checkerboard pattern is helpful in visualizing a coordinate grid in the plane of the input image and the type of distortion introduced by each transformation.

Image 1: Create Checkerboard

`checkerboard` produces an image that has rectangular tiles and four unique corners, which makes it easy to see how the checkerboard image gets distorted by geometric transformations.

After you have run this example once, try changing the image `I` to your favorite image.

```
sqsize = 60;
I = checkerboard(sqsize,4,4);
nrows = size(I,1);
ncols = size(I,2);
fill = 0.3;

imshow(I)
title('Original')
```

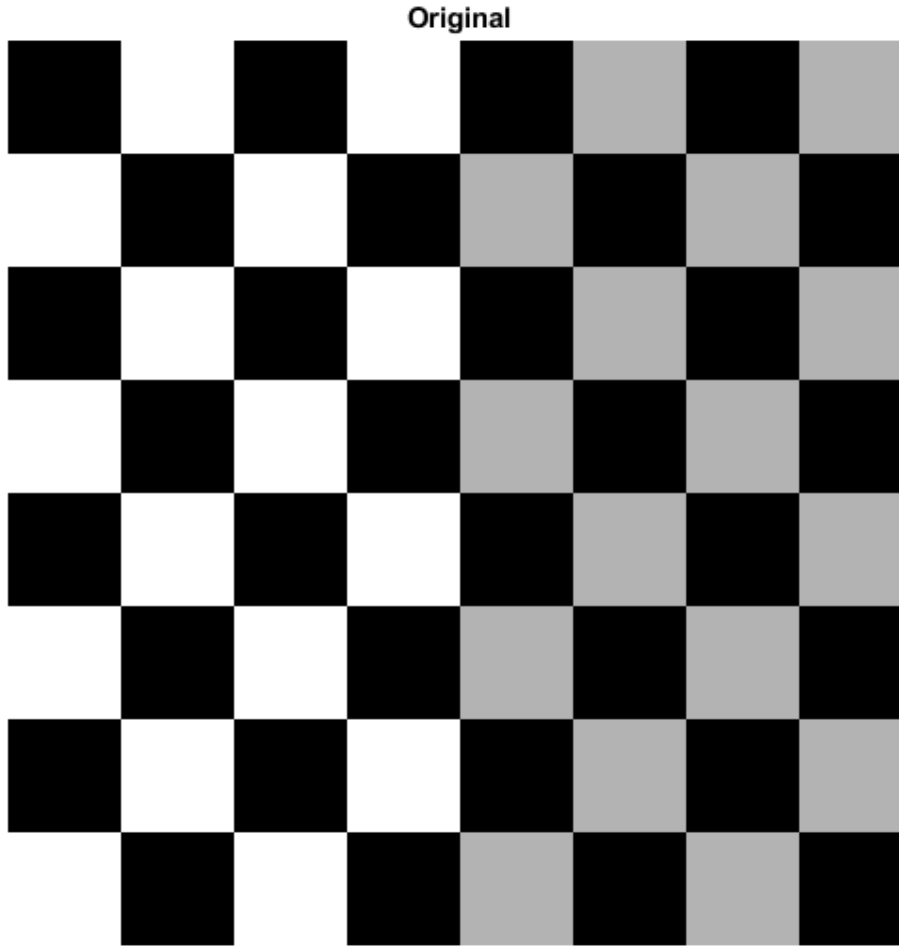


Image 2: Apply Nonreflective Similarity to Checkerboard

Nonreflective similarity transformations may include a rotation, a scaling, and a translation. Shapes and angles are preserved. Parallel lines remain parallel. Straight lines remain straight.

For a nonreflective similarity,

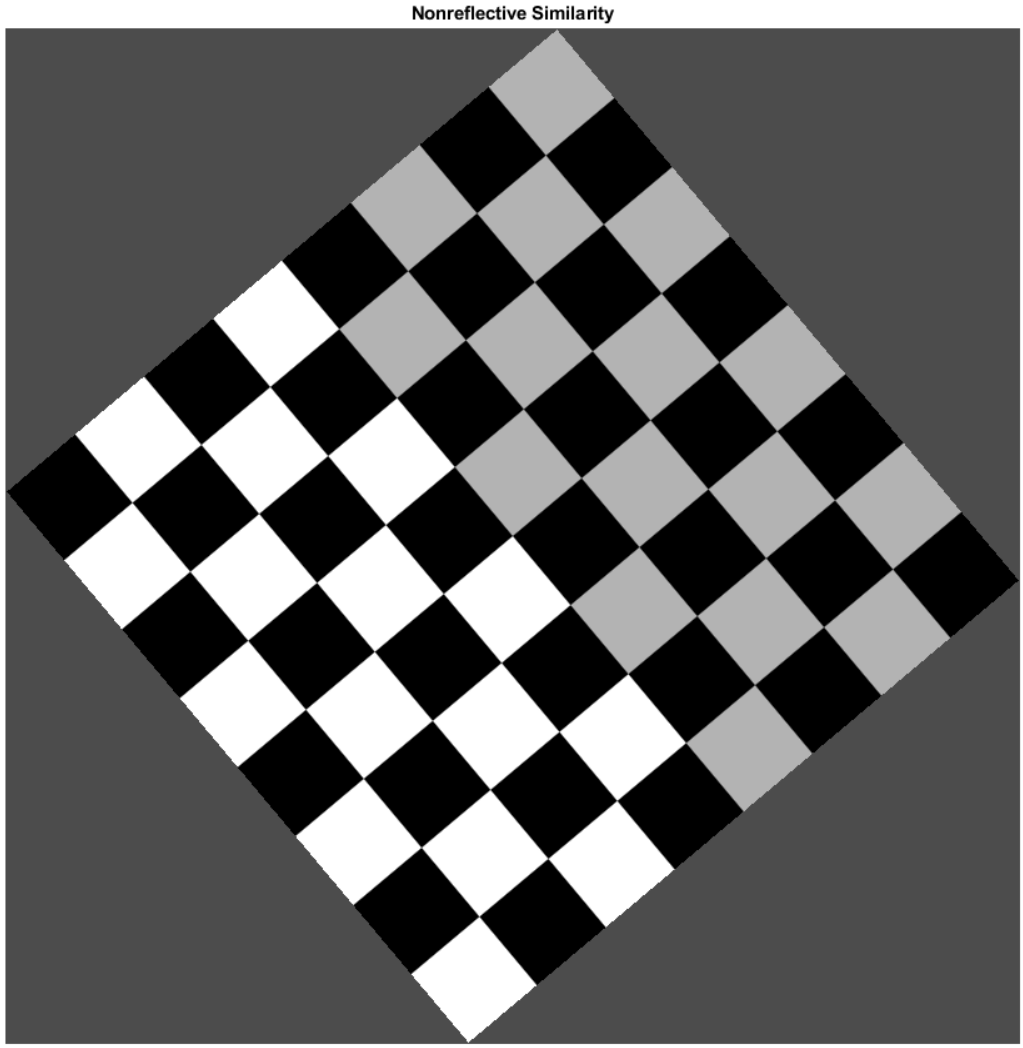
$$[u \;\; v] = [x \;\; y \;\; 1]\,T$$

T is a 3-by-3 matrix that depends on 4 parameters.

```
% Try varying these 4 parameters.
scale = 1.2;       % scale factor
angle = 40*pi/180; % rotation angle
tx = 0;            % x translation
ty = 0;            % y translation

sc = scale*cos(angle);
ss = scale*sin(angle);

T = [ sc -ss  0;
      ss  sc  0;
      tx  ty  1];
```

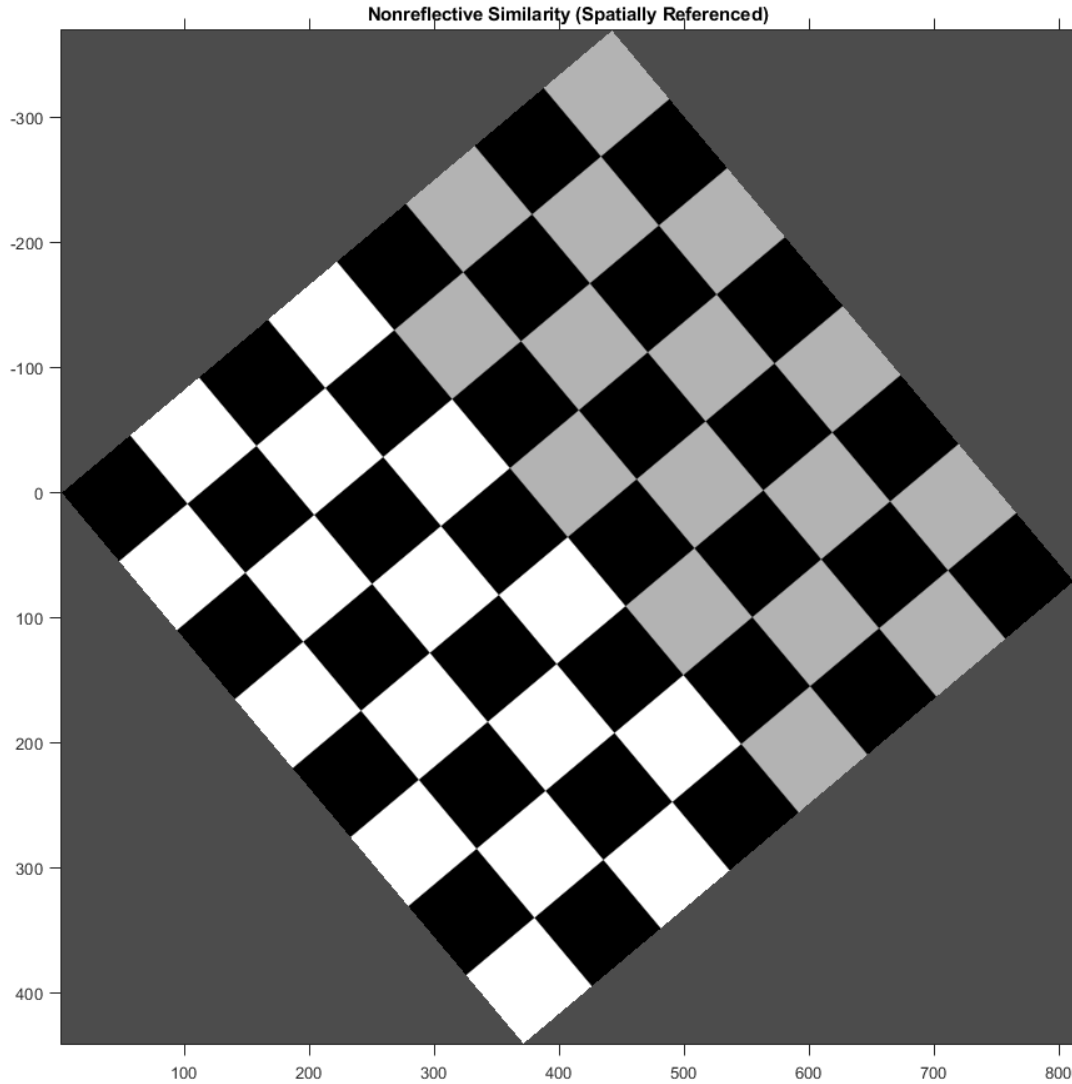
Since nonreflective similarities are a subset of affine transformations, create an `affine2d` object using:

```
t_nonsim = affine2d(T);
I_nonreflective_similarity = imwarp(I,t_nonsim,'FillValues',fill);

imshow(I_nonreflective_similarity);
title('Nonreflective Similarity')
```



If you change either `tx` or `ty` to a non-zero value, you will notice that it has no effect on the output image. If you want to see the coordinates that correspond to your transformation, including the translation, include spatial referencing information:

```
[I_nonreflective_similarity,RI] = imwarp(I,t_nonsim,'FillValues',fill);

imshow(I_nonreflective_similarity,RI)
axis on
title('Nonreflective Similarity (Spatially Referenced)')
```



Notice that passing the output spatial referencing object `RI` from `imwarp` reveals the translation. To specify what part of the output image you want to see, use the 'OutputView' name-value pair in the `imwarp` function.

Image 3: Apply Similarity to Checkerboard

In a similarity transformation, similar triangles map to similar triangles. Nonreflective similarity transformations are a subset of similarity transformations.

For a similarity, the equation is the same as for a nonreflective similarity:

$$[u \;\; v] = [x \;\; y \;\; 1]\,T$$

T is a 3-by-3 matrix that depends on 4 parameters plus an optional reflection.

```
% Try varying these parameters.
scale = 1.5;        % scale factor
angle = 10*pi/180; % rotation angle
tx = 0;            % x translation
ty = 0;            % y translation
a = -1;            % -1 -> reflection, 1 -> no reflection

sc = scale*cos(angle);
ss = scale*sin(angle);

T = [   sc   -ss  0;
      a*ss  a*sc  0;
        tx    ty  1];
```

Since similarities are a subset of affine transformations, create an `affine2d` object using:

```
t_sim = affine2d(T);
```

As in the translation example above, retrieve the output spatial referencing object `RI` from the `imwarp` function, and pass `RI` to `imshow` to reveal the reflection.

```
[I_similarity,RI] = imwarp(I,t_sim,'FillValues',fill);

imshow(I_similarity,RI)
axis on
title('Similarity')
```

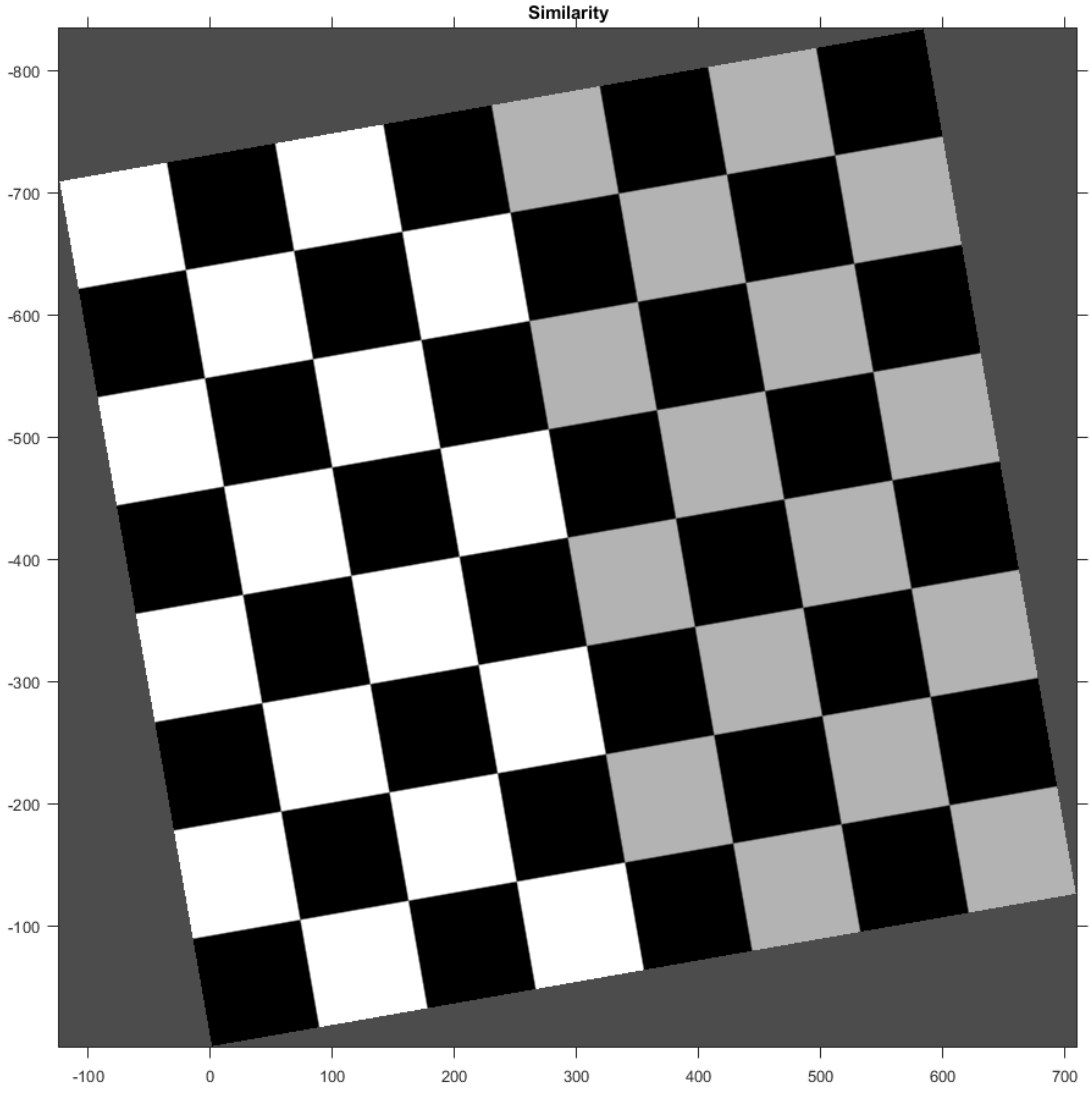


Image 4: Apply Affine Transformation to Checkerboard

In an affine transformation, the x and y dimensions can be scaled or sheared independently and there may be a translation, a reflection, and/or a rotation. Parallel lines

remain parallel. Straight lines remain straight. Similarities are a subset of affine transformations.

For an affine transformation, the equation is the same as for a similarity and nonreflective similarity:

$$[u \;\; v] = [x \;\; y \;\; 1]\,T$$

T is 3-by-3 matrix, where all six elements of the first and second columns can be different. The third column must be [0;0;1].

```
% Try varying the definition of T.
T = [1  0.3  0;
     1    1  0;
     0    0  1];
t_aff = affine2d(T);
I_affine = imwarp(I,t_aff,'FillValues',fill);

imshow(I_affine)
title('Affine')
```

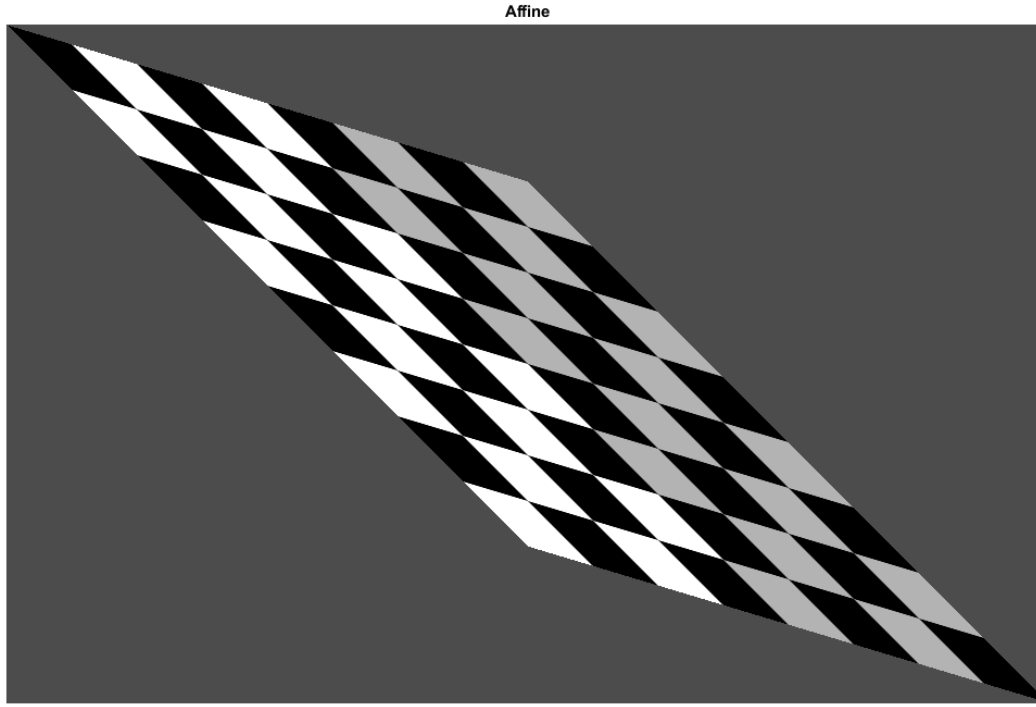


Image 5: Apply Projective Transformation to Checkerboard

In a projective transformation, quadrilaterals map to quadrilaterals. Straight lines remain straight but parallel lines do not necessarily remain parallel. Affine transformations are a subset of projective transformations.

For a projective transformation:

$$\begin{bmatrix} up & vp & wp \end{bmatrix} = \begin{bmatrix} x & y & w \end{bmatrix} T$$

$$u = \frac{up}{wp}$$

$$v = \frac{vp}{wp}$$

T is a 3-by-3 matrix, where all nine elements can be different.

$$T = \begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix}$$

The above matrix equation is equivalent to these two expressions:

$$u = \frac{Ax + By + C}{Gx + Hy + I}$$

$$v = \frac{Dx + Ey + F}{Gx + Hy + I}$$

Try varying any of the nine elements of T.

```
T = [1  0  0.002;
     1  1  0.0002;
     0  0  1   ];
t_proj = projective2d(T);

I_projective = imwarp(I,t_proj,'FillValues',fill);
imshow(I_projective)
title('Projective')
```

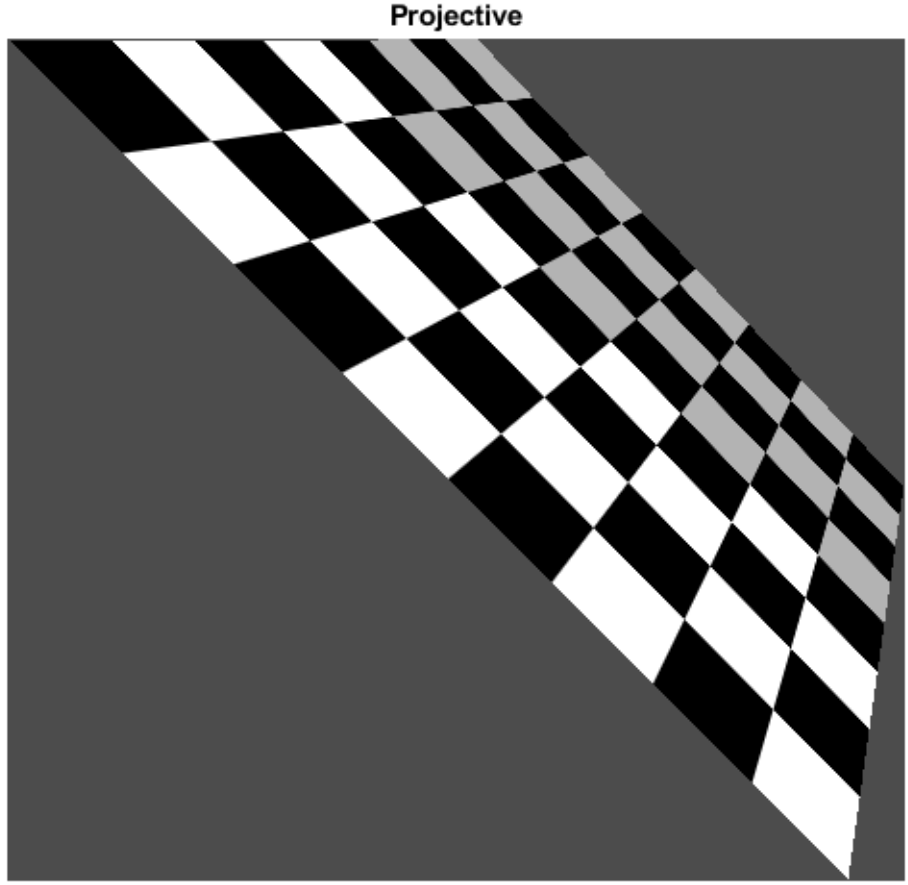


Image 6: Apply Piecewise Linear Transformation to Checkerboard

In a piecewise linear transformation, affine transformations are applied separately to regions of the image. In this example, the top-left, top-right, and bottom-left points of the checkerboard remain unchanged, but the triangular region at the lower-right of the image is stretched so that the bottom-right corner of the transformed image is 50% further to the right and 20% lower than the original coordinate.

```
movingPoints = [0 0; 0 nrows; ncols 0; ncols nrows;];
fixedPoints  = [0 0; 0 nrows; ncols 0; ncols*1.5 nrows*1.2];
t_piecewise_linear = fitgeotrans(movingPoints,fixedPoints,'pwl');

I_piecewise_linear = imwarp(I,t_piecewise_linear,'FillValues',fill);
imshow(I_piecewise_linear)
title('Piecewise Linear')
```

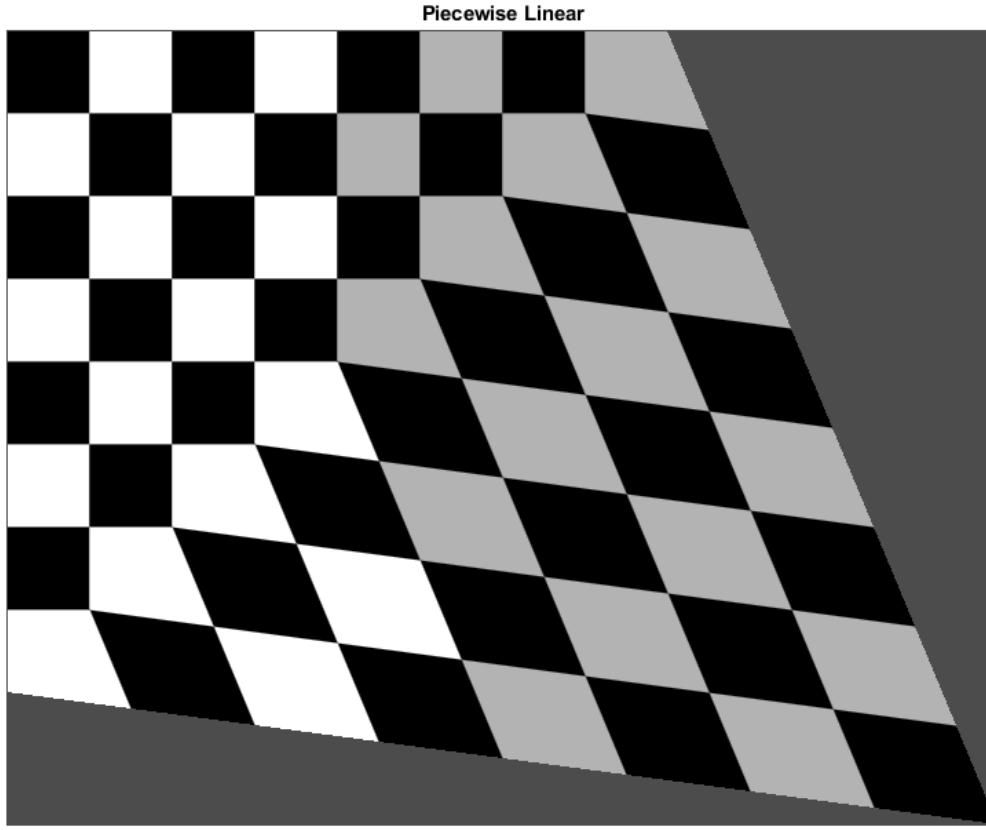


Image 7: Apply Sinusoidal Transformation to Checkerboard

This example and the following two examples show how you can create an explicit mapping to associate each point in a regular grid (xi,yi) with a different point (ui,vi). This mapping is stored in a `geometricTranform2d` object, which used by `imwarp` to transform the image.

In this sinusoidal transformation, the x-coordinate of each pixel is unchanged. The y-coordinate of each row of pixels is shifted up or down following a sinusoidal pattern.

```
a = ncols/12; % Try varying the amplitude of the sinusoid
ifcn = @(xy) [xy(:,1), xy(:,2) + a*sin(2*pi*xy(:,1)/nrows)];
tform = geometricTransform2d(ifcn);

I_sinusoid = imwarp(I,tform,'FillValues',fill);
imshow(I_sinusoid);
title('Sinusoid')
```

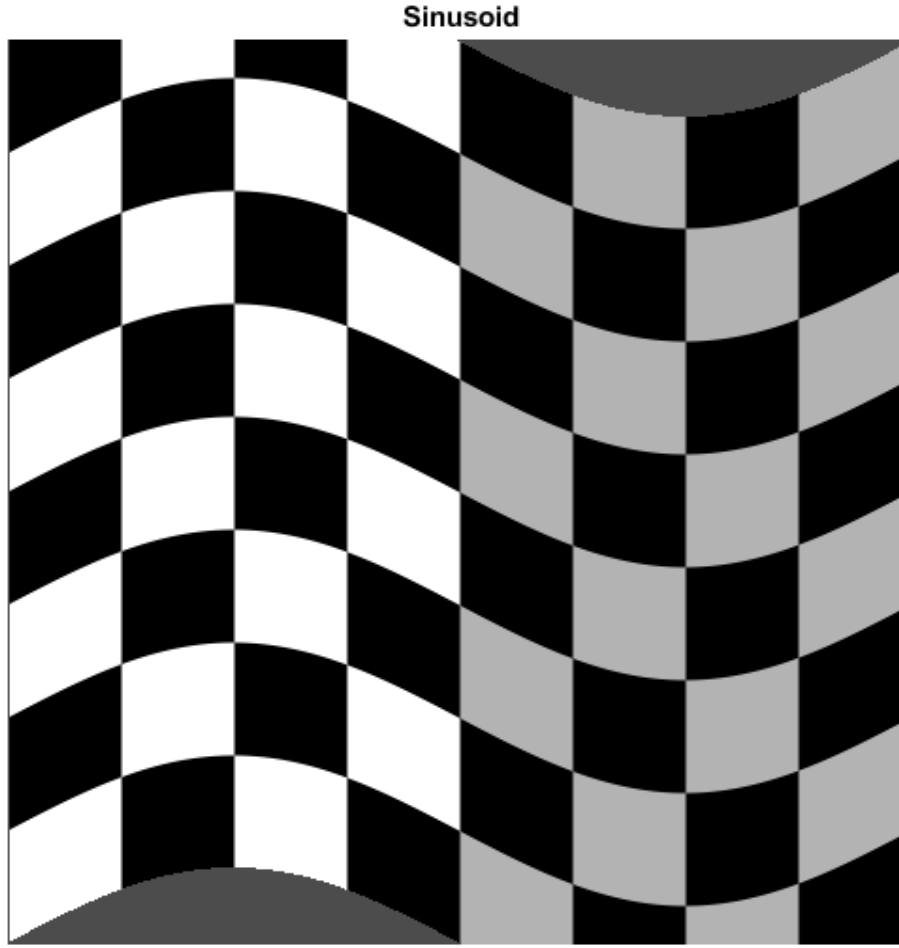


Image 8: Apply Barrel Transformation to Checkerboard

Barrel distortion perturbs an image radially outward from its center. Distortion is greater farther from the center, resulting in convex sides.

First, define a function that maps pixel indices to distance from the center. Use the `meshgrid` function to create arrays of the x-coordinate and y-coordinate of each pixel, with the origin in the upper-left corner of the image.

```
[xi,yi] = meshgrid(1:ncols,1:nrows);
```

Shift the origin to the center of the image. Then, convert the Cartesian x- and y-coordinates to cylindrical angle (`theta`) and radius (`r`) coordinates using the `cart2pol` function. `r` changes linearly as distance from the center pixel increases.

```
xt = xi - ncols/2;
yt = yi - nrows/2;
[theta,r] = cart2pol(xt,yt);
```

Define the amplitude, `a`, of the cubic term. This parameter is adjustable. Then, add a cubic term to `r` so that `r` changes nonlinearly with distance from the center pixel.

```
a = 1; % Try varying the amplitude of the cubic term.
rmax = max(r(:));
s1 = r + r.^3*(a/rmax.^2);
```

Convert back to the Cartesian coordinate system. Shift the origin back to the upper-right corner of the image.

```
[ut,vt] = pol2cart(theta,s1);
ui = ut + ncols/2;
vi = vt + nrows/2;
```

Store the mapping between (`xi`,`yi`) and (`ui`,`vi`) in a `geometricTranform2d` object. Use `imwarp` to transform the image according to the pixel mapping.

```
ifcn = @(c) [ui(:) vi(:)];
tform = geometricTransform2d(ifcn);

I_barrel = imwarp(I,tform,'FillValues',fill);
imshow(I_barrel)
title('Barrel')
```

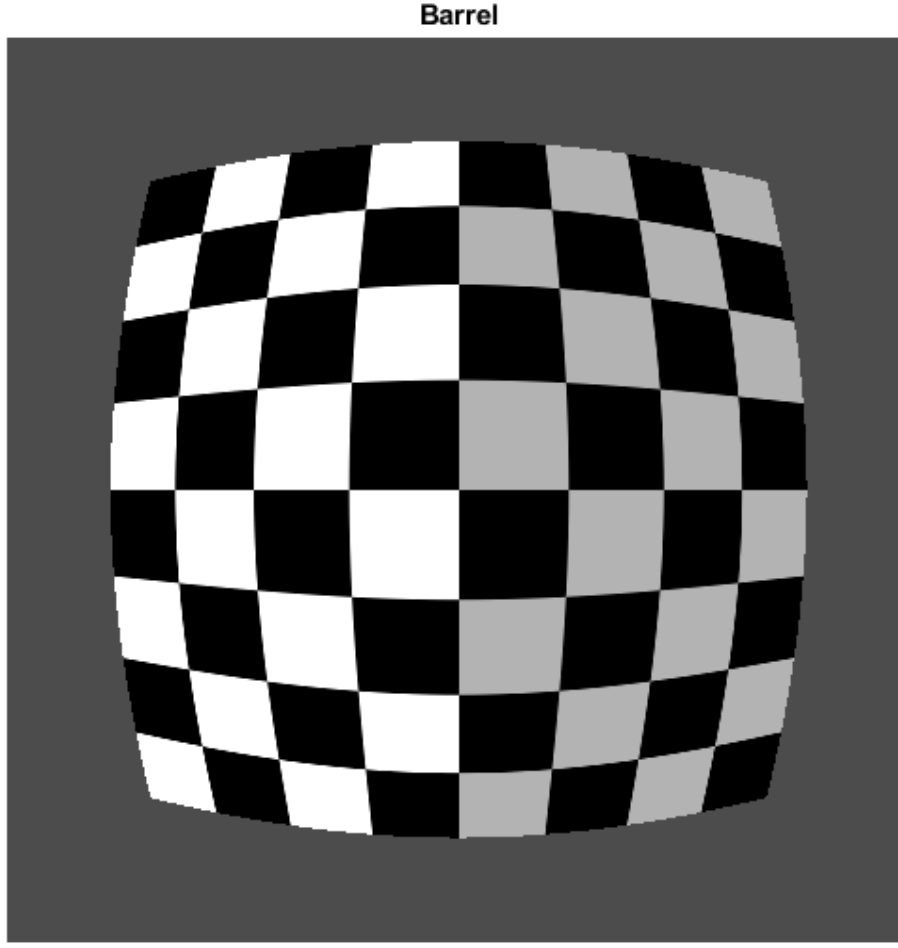


Image 9: Apply Pin Cushion Transformation to Checkerboard

Pin-cushion distortion is the inverse of barrel distortion because the cubic term has a negative amplitude. Distortion is still greater farther from the center but the distortion appears as concave sides.

You can begin with the same `theta` and `r` values as for the barrel transformation. Define a different amplitude, b, of the cubic term. This parameter is adjustable. Then, subtract a cubic term to `r` so that `r` changes nonlinearly with distance from the center pixel.

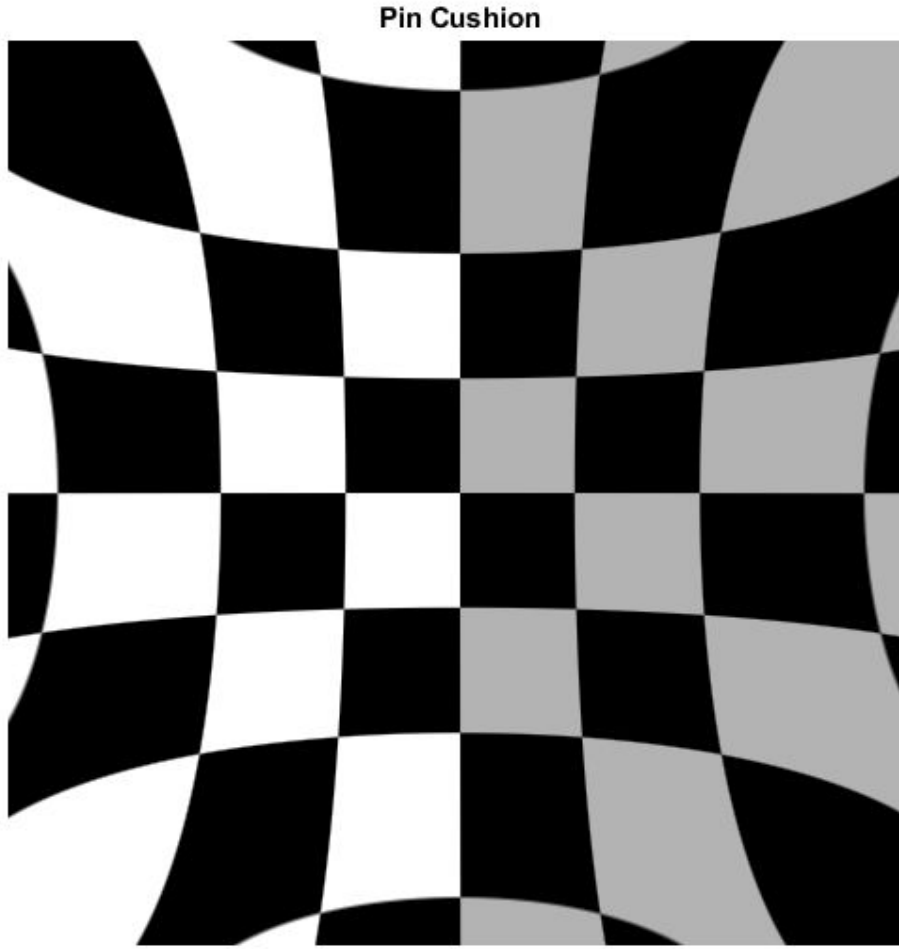
```
b = 0.4; % Try varying the amplitude of the cubic term.
s = r - r.^3*(b/rmax.^2);
```

Convert back to the Cartesian coordinate system. Shift the origin back to the upper-right corner of the image.

```
[ut,vt] = pol2cart(theta,s);
ui = ut + ncols/2;
vi = vt + nrows/2;
```

Store the mapping between (`xi`,`yi`) and (`ui`,`vi`) in a `geometricTranform2d` object. Use `imwarp` to transform the image according to the pixel mapping.

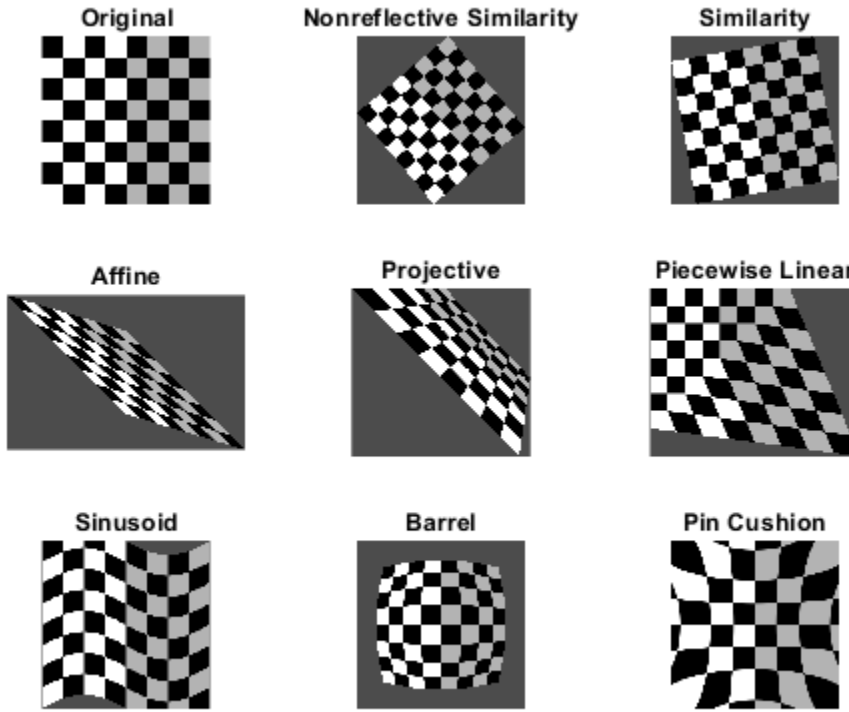
```
ifcn = @(c) [ui(:) vi(:)];
tform = geometricTransform2d(ifcn);
I_pin = imwarp(I,tform,'FillValues',fill);
imshow(I_pin)
title('Pin Cushion')
```



Summary: Display All Geometric Transformations of Checkerboard

```
figure
subplot(3,3,1),imshow(I),title('Original')
subplot(3,3,2),imshow(I_nonreflective_similarity),title('Nonreflective Similarity')
subplot(3,3,3),imshow(I_similarity),title('Similarity')
```

```
subplot(3,3,4),imshow(I_affine),title('Affine')
subplot(3,3,5),imshow(I_projective),title('Projective')
subplot(3,3,6),imshow(I_piecewise_linear),title('Piecewise Linear')
subplot(3,3,7),imshow(I_sinusoid),title('Sinusoid')
subplot(3,3,8),imshow(I_barrel),title('Barrel')
subplot(3,3,9),imshow(I_pin),title('Pin Cushion')
```



Note that `subplot` changes the scale of the images being displayed.

See Also

Functions
`checkerboard` | `fitgeotrans` | `imwarp` | `makeresampler` | `tformarray`

Objects
`LocalWeightedMeanTransformation2D` | `PiecewiseLinearTransformation2D` | `PolynomialTransformation2D` | `affine2d` | `projective2d`

More About

- “Matrix Representation of Geometric Transformations” on page 6-19
- “2-D and 3-D Geometric Transformation Process Overview” on page 6-14

Exploring a Conformal Mapping

This example shows how to explore a conformal mapping. Geometric image transformations are useful in understanding a conformal mapping that is important in fluid-flow problems, and the mapping itself can be used to transform imagery for an interesting special effect.

Step 1: Select a Conformal Transformation

Conformal transformations, or mappings, have many important properties and uses. One property relevant to image transformation is the preservation of local shape (except sometimes at isolated points).

This example uses a 2-D conformal transformation to warp an image. The mapping from output to input, `g: R^2 -> R^2`, is defined in terms of a complex analytic function `G: C -> C`, where

```
G(z) = (z + 1/z) / 2.
```

We define `g` via a direct correspondence between each point `(x,y)` in `R^2` (the Euclidean plane) and the point `z = x + i*y` in `C` (the complex plane),

```
g(x,y) = (Re(w),Im(w)) = (u,v)
```

where

```
w = u + i*v = G(x + i*y).
```

This conformal mapping is important in fluid mechanics because it transforms lines of flow around a circular disk (or cylinder, if we add a third dimension) to straight lines. (See pp. 340-341 in Strang, Gilbert, Introduction to Applied Mathematics, Wellesley-Cambridge Press, Wellesley, MA, 1986.)

A note on the value of complex variables: although we could express the definition of `g` directly in terms of `x` and `y`, that would obscure the underlying simplicity of the transformation. This disadvantage would come back to haunt us in Step 3 below. There, if we worked purely in real variables, we would need to solve a pair of simultaneous nonlinear equations instead of merely applying the quadratic formula!

Step 2: Warp an Image Using the Conformal Transformation

We start by loading the peppers image, extracting a 300-by-500 subimage, and displaying it.

```
A = imread('peppers.png');
A = A(31:330,1:500,:);
figure
imshow(A)
title('Original Image','FontSize',14)
```

Then use `maketform` to make a custom `tform` struct with a handle to function `conformalInverse` as its `INVERSE_FCN` argument:

```
conformal = maketform('custom', 2, 2, [], @conformalInverse, []);
```

To view `conformalInverse` use:

```
type conformalInverse.m
```

```
function U = conformalInverse(X, ~)
% conformalInverse Inverse conformal transformation.
%
```

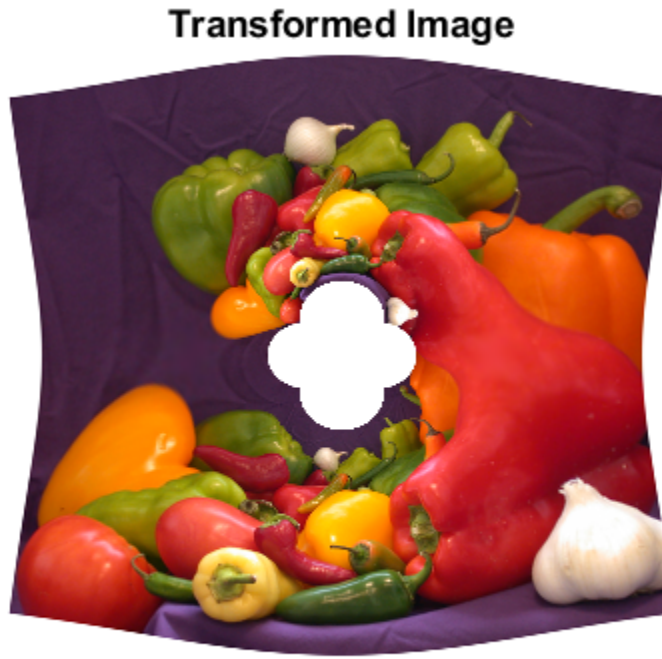
```
% Supports conformal transformation example, ConformalMappingImageExample.m
% ("Exploring a Conformal Mapping").

% Copyright 2005-2013 The MathWorks, Inc.

Z = complex(X(:,1),X(:,2));
W = (Z + 1./Z)/2;
U(:,2) = imag(W);
U(:,1) = real(W);
```

Horizontal and vertical bounds are needed for mapping the original and transformed images to the input and output complex planes. Note that the proportions in `uData` and `vData` match the height-to-width ratio of the original image (3/5).

```
uData = [ -1.25   1.25];  % Bounds for REAL(w)
vData = [  0.75  -0.75];  % Bounds for IMAG(w)
xData = [ -2.4    2.4 ];  % Bounds for REAL(z)
yData = [  2.0   -2.0 ];  % Bounds for IMAG(z)
```

We apply `imtransform` using the `SIZE` parameter to ensure an aspect ratio that matches the proportions in `xData` and `yData` (6/5), and view the result.

```
B = imtransform( A, conformal, 'cubic', ...
                'UData', uData,'VData', vData,...
                'XData', xData,'YData', yData,...
                'Size', [300 360], 'FillValues', 255 );
figure
imshow(B)
title('Transformed Image','FontSize',14)
```



Compare the original and transformed images. Except that the edges are now curved, the outer boundary of the image is preserved by the transformation. Note that each feature from the original image appears twice in the transformed image (look at the various peppers). And there is a hole in the middle of the transformed image with four regular cusps around its edges.

In fact, every point in the input w-plane is mapped to two points in the output z-plane, one inside the unit circle and one outside. The copies inside the unit circle are much smaller than those outside. It's clear that the cusps around the central hole are just the copies of the four image corners that mapped inside the unit circle.

Step 3: Construct Forward Transformations

If the transformation created with `maketform` has a forward function, then we can apply `tformfwd` to regular geometric objects (in particular, to rectangular grids and uniform

arrays of circles) to obtain further insight into the transformation. In this example, because `G` maps two output points to each input point, there is no unique forward transformation. But we can proceed if we are careful and work with two different forward functions.

Letting `w = (z + 1/z)/2` and solving the quadratic equation that results,

```
z^2 + 2*w*z + 1 = 0,
```

we find that

```
z = w +/- sqrt{(w^2 - 1).
```

The positive and the negative square roots lead to two separate forward transformations. We construct the first using `maketform` and a handle to the function, `conformalForward1`.

```
t1 = maketform('custom', 2, 2, @conformalForward1, [], []);
```

To view `conformalForward1` use:

```
type conformalForward1.m
```

```
function X = conformalForward1(U, ~)
% conformalForward1 Forward transformation with positive square root.
%
% Supports conformal transformation example, ConformalMappingImageExample.m
% ("Exploring a Conformal Mapping").

% Copyright 2005-2013 The MathWorks, Inc.

W = complex(U(:,1),U(:,2));
Z = W + sqrt(W.^2 - 1);
X(:,2) = imag(Z);
X(:,1) = real(Z);
```

We construct the second transformation with another function that is identical to `conformalForward1` except for a sign change.

```
t2 = maketform('custom', 2, 2, @conformalForward2, [], []);
```

```
type conformalForward2.m
```

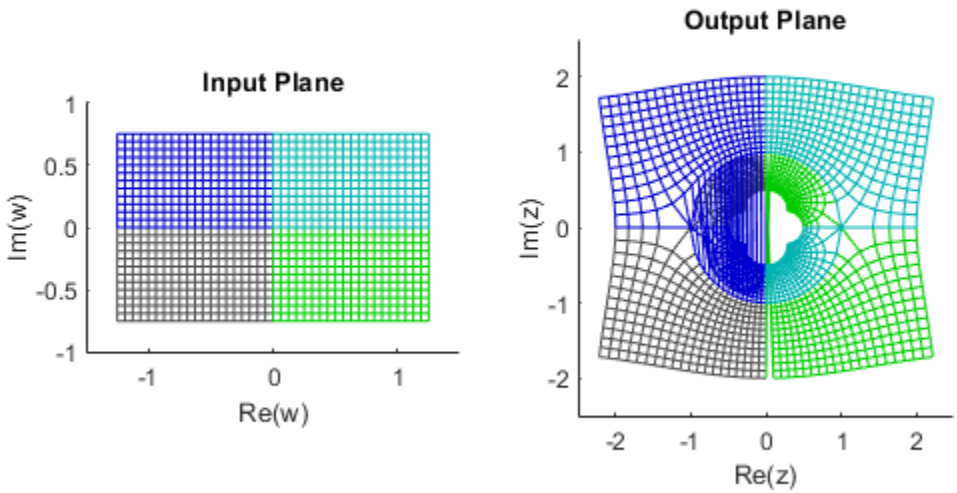
```
function X = conformalForward2(U, ~)
```

```
% conformalForward2 Forward transformation with negative square root.
%
% Supports conformal transformation example, ConformalMappingImageExample.m
% ("Exploring a Conformal Mapping").

% Copyright 2005-2013 The MathWorks, Inc.

W = complex(U(:,1),U(:,2));
Z = W - sqrt(W.^2 - 1);
X(:,2) = imag(Z);
X(:,1) = real(Z);
```

Step 4: Explore the Mapping Using Grid Lines

With the two forward transformations, we can illustrate the mapping of a grid of lines, using additional helper functions.

```
f3 = figure('Name','Conformal Transformation: Grid Lines');
axIn  = conformalSetupInputAxes( subplot(1,2,1));
axOut = conformalSetupOutputAxes(subplot(1,2,2));
conformalShowLines(axIn, axOut, t1, t2)

% Reduce wasted vertical space in figure
f3.Position = [1 1 1 0.7] .* f3.Position;
```

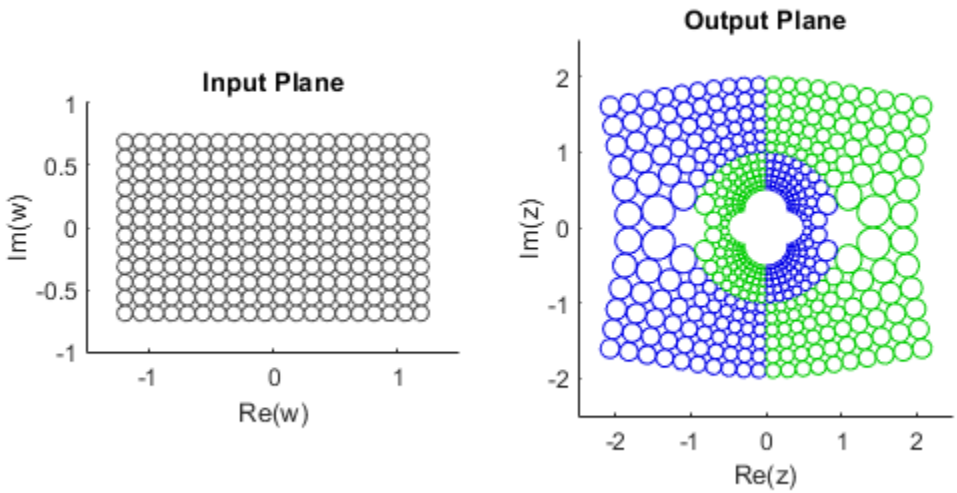


You can see that the grid lines are color-coded according to their quadrants in the input plane before and after the transformations. The colors also follow the transformed grids to the output planes. Note that each quadrant transforms to a region outside the unit circle and to a region inside the unit circle. The right-angle intersections between grid lines are preserved under the transformation -- evidence of the shape-preserving property of conformal mappings -- except for the points at +1 and -1 on the real axis.

Step 5: Explore the Mapping Using Packed Circles

Under a conformal transformation, small circles should remain nearly circular, changing only in position and size. Again applying the two forward transformations, this time we map a regular array of uniformly-sized circles.

```
f4 = figure('Name','Conformal Transformation: Circles');
axIn  = conformalSetupInputAxes( subplot(1,2,1));
axOut = conformalSetupOutputAxes(subplot(1,2,2));
conformalShowCircles(axIn, axOut, t1, t2)

% Reduce wasted vertical space in figure
f4.Position = [1 1 1 0.7] .* f4.Position;
```



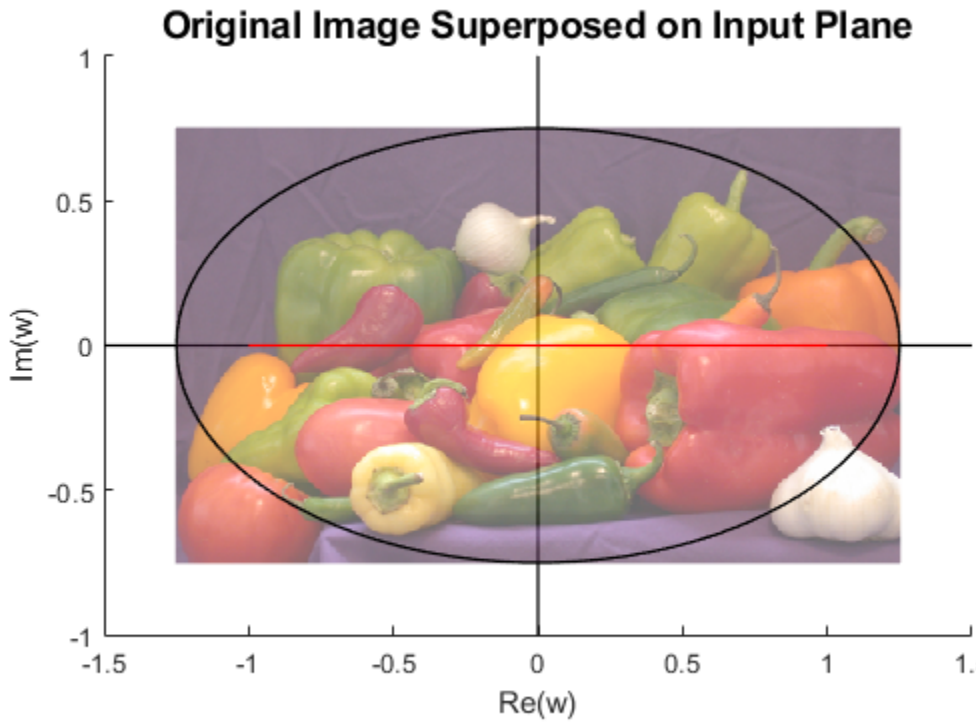
You can see that the transform to a circle packing where tangencies have been preserved. In this example, the color coding indicates use of the positive (green) or negative (blue) square root of `w^2 - 1`. Note that the circles change dramatically but that they remain circles (shape-preservation, once again).

Step 6: Explore the Mapping Using Images

To further explore the conformal mapping, we can place the input and transformed images on the pair of axes used in the preceding examples and superpose a set of curves as well.

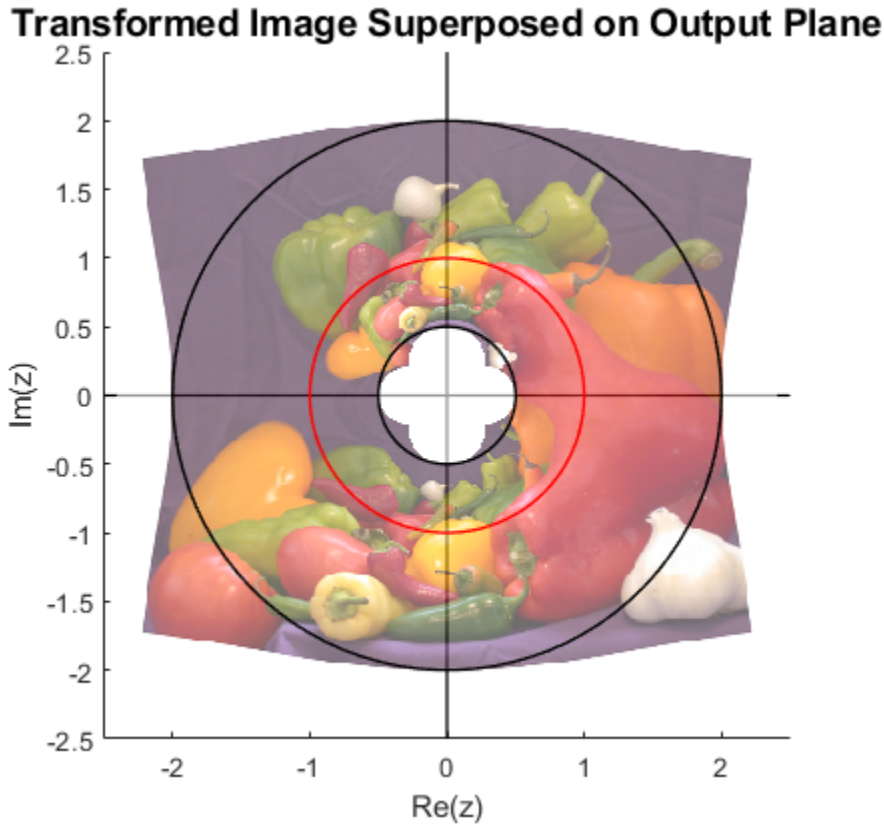
First we display the input image, rendered semi-transparently, over the input axes of the conformal map, along with a black ellipse and a red line along the real axis.

```
figure
axIn = conformalSetupInputAxes(axes);
conformalShowInput(axIn, A, uData, vData)
title('Original Image Superposed on Input Plane','FontSize',14)
```



Next we display the output image over the output axes of the conformal map, along with two black circles and one red circle. Again, the image is semi-transparent.

```
figure
axOut = conformalSetupOutputAxes(axes);
conformalShowOutput(axOut, B, xData, yData)
title('Transformed Image Superposed on Output Plane','FontSize',14)
```



MATLAB® graphics made it easy to shift and scale the original and transformed images to superpose them on the input (`w-`) and output (`z-`) planes, respectively. The use of semi-transparency makes it easier to see the ellipse, line, and circles. The ellipse in the w-plane has intercepts at 5/4 and -5/4 on the horizontal axis and 3/4 and -3/4 on the vertical axis. `G` maps two circles centered on the origin to this ellipse: the one with radius 2 and the one with radius 1/2. And, as shown in red, `G` maps the unit circle to the interval [-1 1] on the real axis.

Step 7: Obtain a Special Effect by Masking Parts of the Output Image

If the inverse transform function within a custom `tform` struct returns a vector filled with `NaN` for a given output image location, then `imtransform` (and also `tformarray`) assign

the specified fill value at that location. In this step we repeat Step 1, but modify our inverse transformation function slightly to take advantage of this feature.

```
type conformalInverseClip.m
```

```
function U = conformalInverseClip( X, ~)
% conformalInverseClip Inverse conformal transformation with clipping.
%
% This is a modification of conformalInverse in which points in X
% inside the circle of radius 1/2 or outside the circle of radius 2 map to
% NaN + i*NaN.
%
% Supports conformal transformation example, ConformalMappingImageExample.m
% ("Exploring a Conformal Mapping").

% Copyright 2000-2013 The MathWorks, Inc.

Z = complex(X(:,1),X(:,2));
W = (Z + 1./Z)/2;
q = 0.5 <= abs(Z) & abs(Z) <= 2;
W(~q) = complex(NaN,NaN);
U(:,2) = imag(W);
U(:,1) = real(W);
```

This is the same as the function defined in Step 2, except for the two additional lines:

```
q = 0.5 <= abs(Z) & abs(Z) <= 2;
W(~q) = complex(NaN,NaN);
```

which cause the inverse transformation to return `NaN` at any point not between the two circles with radii of 1/2 and 2, centered on the origin. The result is to mask that portion of the output image with the specified fill value.

```
ring = maketform('custom', 2, 2, [], @conformalInverseClip, []);
Bring = imtransform( A, ring, 'cubic',...
                    'UData', uData,  'VData', vData,...
                    'XData', [-2 2], 'YData', yData,...
                    'Size', [400 400], 'FillValues', 255 );
figure
imshow(Bring)
title('Transformed Image With Masking','FontSize',14);
```

The result is identical to our initial transformation except that the outer corners and inner cusps have been masked away to produce a ring effect.

Step 8: Repeat the Effect on a Different Image

Applying the "ring" transformation to an image of winter greens (hemlock and alder berries) leads to an aesthetic special effect.

Load the image `greens.jpg`, which already has a 3/5 height-to-width ratio, and display it.

```
C = imread('greens.jpg');
figure
imshow(C)
title('Winter Greens Image','FontSize',14);
```

Transform the image and display the result, this time creating a square output image.

```
D = imtransform( C, ring, 'cubic',...
                 'UData', uData, 'VData', vData,...
                 'XData', [-2 2], 'YData', [-2 2],...
                 'Size', [400 400], 'FillValues', 255 );
figure
imshow(D)
title('Transformed and Masked Winter Greens Image','FontSize',14);
```

Notice that the local shapes of objects in the output image are preserved. The alder berries stayed round!

Exploring Slices from a 3-Dimensional MRI Data Set

This example shows how to explore a volume of data by extracting slices through a three-dimensional MRI data set using `imtransform` and `tformarray` functions.

Step 1: Load and View Horizontal MRI

This example uses the MRI data set that comes with MATLAB® and that is used in the help examples for both `montage` and `immovie`. Loading `mri.mat` adds two variables to the workspace: `D` (128-by-128-by-1-by-27, class uint8) and a grayscale colormap, `map` (89-by-3, class double).

`D` comprises 27 128-by-128 horizontal slices from an MRI data scan of a human cranium. Values in `D` range from 0 through 88, so the colormap is needed to generate a figure with a useful visual range. The dimensionality of `D` makes it compatible with `montage`. The first two dimensions are spatial. The third dimension is the color dimension, with size 1 because it indexes into the color map. (`size(D,3)` would be 3 for an RGB image sequence.) The fourth dimension is temporal (as with any image sequence), but in this particular case it is also spatial. So there are three spatial dimensions in `D` and we can use `imtransform` or `tformarray` to convert the horizontal slices to sagittal slices (showing the view from the side of the head) or coronal (frontal) slices (showing the view from the front or back of the head).

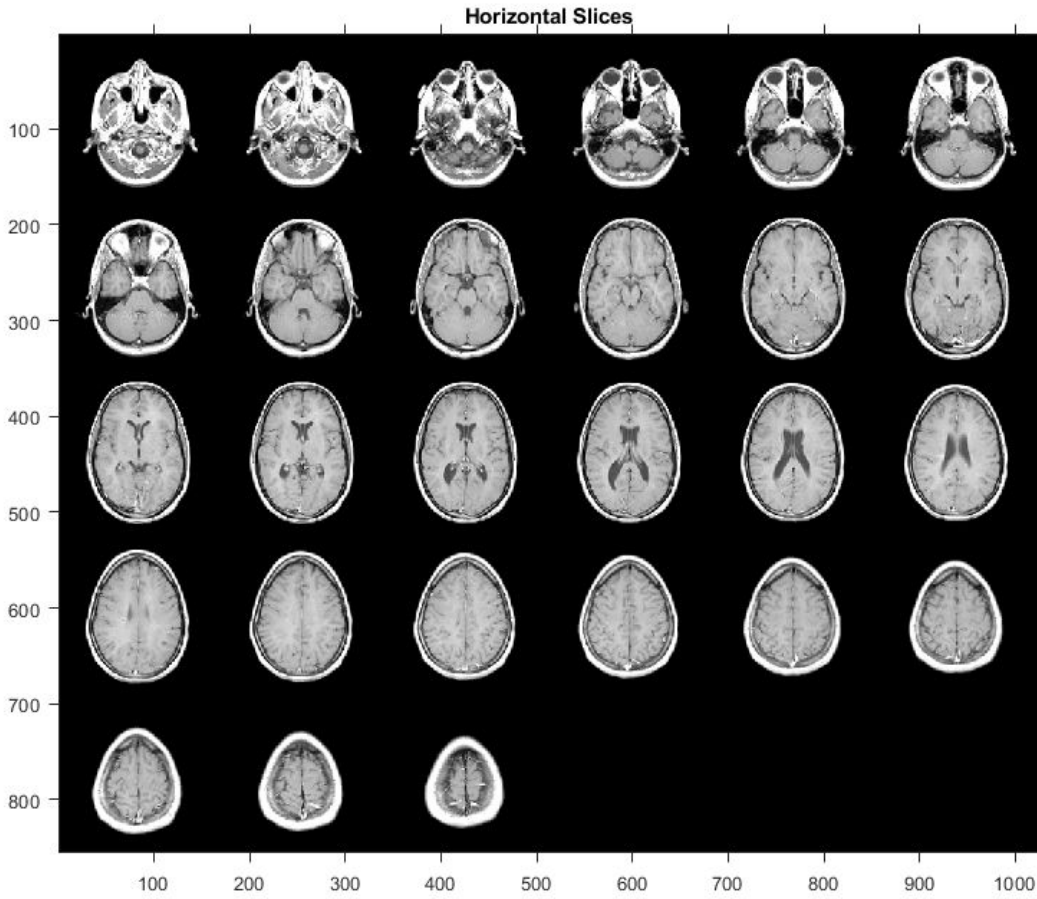
The spatial dimensions of `D` are ordered as follows:

- Dimension 1: Front to back of head (rostral/anterior to caudal/posterior)
- Dimension 2: Left to right of head
- Dimension 4: Bottom to top of head (inferior to superior).

An important factor is that the sampling intervals are not the same along the three dimensions: samples along the vertical dimension (4) are spaced 2.5 times more widely than along the horizontal dimensions.

Load the MRI data set and view the 27 horizontal slices as a montage.

```
load mri;
montage(D,map)
title('Horizontal Slices');
```



Step 2: Extract Sagittal Slice from Horizontal Slices Using IMTRANSFORM

We can construct a mid-sagittal slice from the MRI data by taking a subset of `D` and transforming it to account for the different sampling intervals and the spatial orientation of the dimensions of `D`.

The following statement extracts all the data needed for a midsagittal slice.

```
M1 = D(:,64,:,:); size(M1)
```

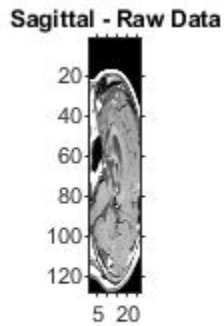
```
ans = 1×4

   128     1     1    27
```

However we cannot view `M1` as an image because it is 128-by-1-by-1-by-27. `reshape` (or `squeeze`) can convert `M1` into a 128-by-27 image that is viewable with `imshow`.

```
M2 = reshape(M1,[128 27]); size(M2)
```

```
ans = 1×2

   128    27
```

```
figure, imshow(M2,map);
title('Sagittal - Raw Data');
```



The dimensions in `M2` are ordered as follows:

- Dimension 1: Front to back of head (rostral to caudal)
- Dimension 2: Bottom to top of head (inferior to superior).

We can obtain a much more satisfying view by transforming `M2` to change its orientation and increase the sampling along the vertical (inferior-superior) dimension by a factor of 2.5 -- making the sampling interval equal in all three spatial dimensions. We could do this in steps starting with a transpose, but the following affine transformation enables a single-step transformation and more economical use of memory.

```
T0 = maketform('affine',[0 -2.5; 1 0; 0 0]);
```

The upper 2-by-2 block of the matrix passed to maketform, `[0 -2.5;1 0]`, combines the rotation and scaling. After transformation we have:

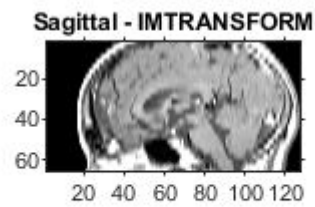
- Dimension 1: Top to bottom of head (superior to inferior).
- Dimension 2: Front to back of head (rostral to caudal)

The call

imtransform(M2,T0,'cubic')

would suffice to apply `T` to `M2` and provide good resolution while interpolating along the top to bottom direction. However, there is no need for cubic interpolation in the front to back direction, since no resampling will occur along (output) dimension 2. Therefore we specify nearest-neighbor resampling in this dimension, with greater efficiency and identical results.

```
R2 = makeresampler({'cubic','nearest'},'fill');
M3 = imtransform(M2,T0,R2);
figure, imshow(M3,map);
title('Sagittal - IMTRANSFORM')
```



Step 3: Extract Sagittal Slice from the Horizontal Slices Using TFORMARRAY

In this step we obtain the same result as step 2, but use `tformarray` to go from three spatial dimensions to two in a single operation. Step 2 does start with an array having three spatial dimensions and end with an array having two spatial dimensions, but intermediate two-dimensional images (`M1` and `M2`) pave the way for the call to `imtransform` that creates `M3`. These intermediate images are not necessary if we use `tformarray` instead of `imtransform`. `imtransform` is very convenient for 2-D to 2-D

transformations, but `tformarray` supports N-D to M-D transformations, where M need not equal N.

Through its `TDIMS_A` argument, `tformarray` allows us to define a permutation for the input array. Since we want to create an image with:

- Dimension 1: Superior to inferior (original dimension 4, reversed)
- Dimension 2: Caudal to rostral (original dimension 1)

and extract just a single sagittal plane via the original dimension 2, we specify `tdims_a` = [4 1 2]. We create a `tform` via composition starting with a 2-D affine transformation `T1` that scales the (new) dimension 1 by a factor of -2.5 and adds a shift of 68.5 to keep the array coordinates positive. The second part of the composite is a custom transformation `T2` that extracts the 64th sagittal plane using a very simple `INVERSE_FCN`.

```
T1 = maketform('affine',[-2.5 0; 0 1; 68.5 0]);
inverseFcn = @(X,t) [X repmat(t.tdata,[size(X,1) 1])];
T2 = maketform('custom',3,2,[],inverseFcn,64);
Tc = maketform('composite',T1,T2);
```

Note that `T2` and `Tc` take a 3-D input to a 2-D input.

We use the same approach to resampling as before, but include a third dimension.

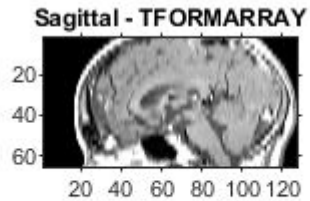
```
R3 = makeresampler({'cubic','nearest','nearest'},'fill');
```

`tformarray` transforms the three spatial dimensions of `D` to a 2-D output in a single step. Our output image is 66-by-128, with the original 27 planes expanding to 66 in the vertical (inferior-superior) direction.

```
M4 = tformarray(D,Tc,R3,[4 1 2],[1 2],[66 128],[],0);
```

The result is identical to the previous output of `imtransform`.

```
figure, imshow(M4,map);
title('Sagittal - TFORMARRAY');
```



Step 4: Create and Display Sagittal Slices

We create a 4-D array (the third dimension is the color dimension) that can be used to generate an image sequence that goes from left to right, starts 30 planes in, skips every other plane, and has 35 frames in total. The transformed array has:

- Dimension 1: Top to bottom (superior to inferior)
- Dimension 2: Front to back (rostral to caudal)
- Dimension 4: Left to right.

As in the previous step, we permute the input array using `TDIMS_A = [4 1 2]`, again flipping and rescaling/resampling the vertical dimension. Our affine transformation is the same as the T1 above, except that we add a third dimension with a (3,3) element of 0.5 and (4,3) element of -14 chosen to map 30, 32, ... 98 to 1, 2, ..., 35. This centers our 35 frames on the mid-sagittal slice.

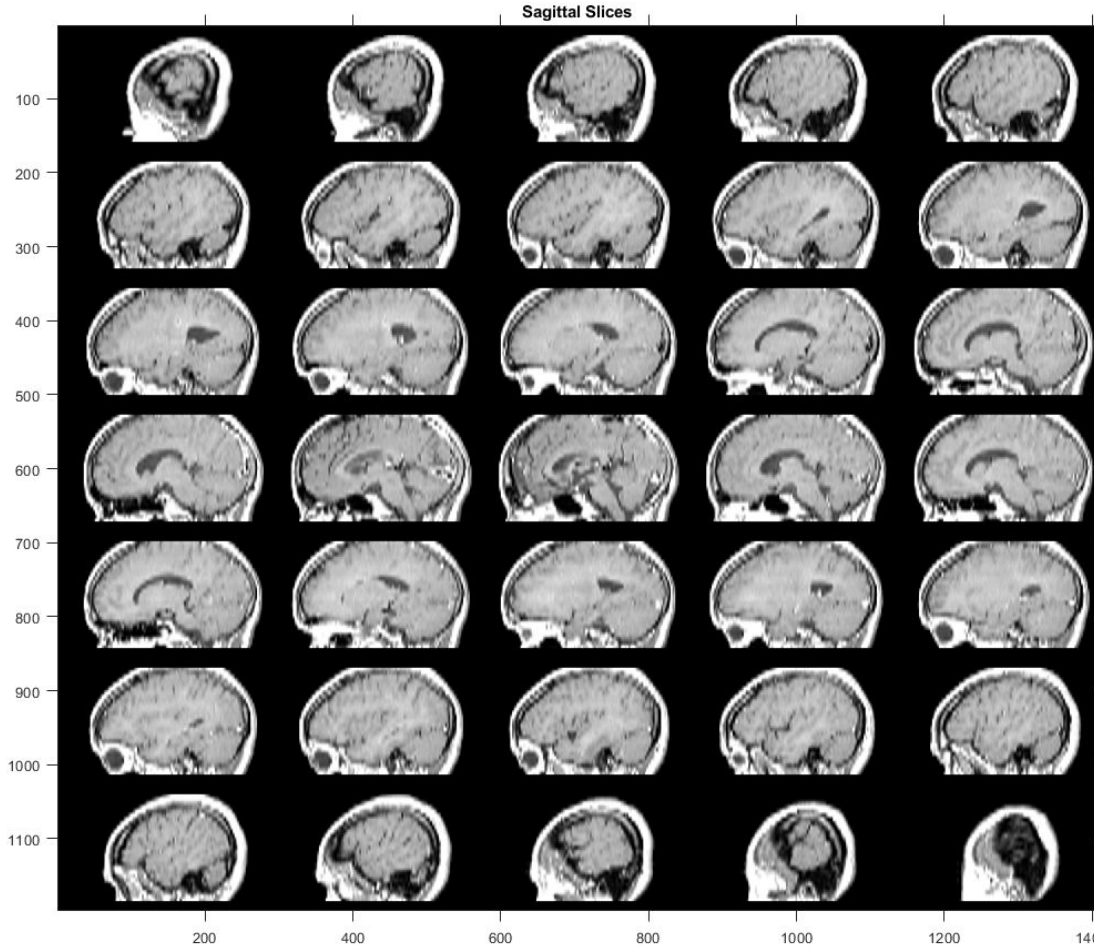
```
T3 = maketform('affine',[-2.5 0 0; 0 1 0; 0 0 0.5; 68.5 0 -14]);
```

In our call to `tformarray`, `TSIZE_B = [66 128 35]` now includes the 35 frames in the 4th, left-to-right dimension (which is the third transform dimension). The resampler remains the same.

```
S = tformarray(D,T3,R3,[4 1 2],[1 2 4],[66 128 35],[],0);
```

View the sagittal slices as a montage (padding the array slightly to separate the elements of the montage).

```
S2 = padarray(S,[6 0 0 0],0,'both');
figure, montage(S2,map)
title('Sagittal Slices');
```



Step 5: Create and Display Coronal Slices

Constructing coronal slices is almost the same as constructing sagittal slices. We change `TDIMS_A` from `[4 1 2]` to `[4 2 1]`. We create a series of 45 frames, starting 8 planes in and moving from back to front, skipping every other frame. The dimensions of the output array are ordered as follows:

- Dimension 1: Top to bottom (superior to inferior)

- Dimension 2: Left to right
- Dimension 4: Back to front (caudal to rostral).

```
T4 = maketform('affine',[-2.5 0 0; 0 1 0; 0 0 -0.5; 68.5 0 61]);
```

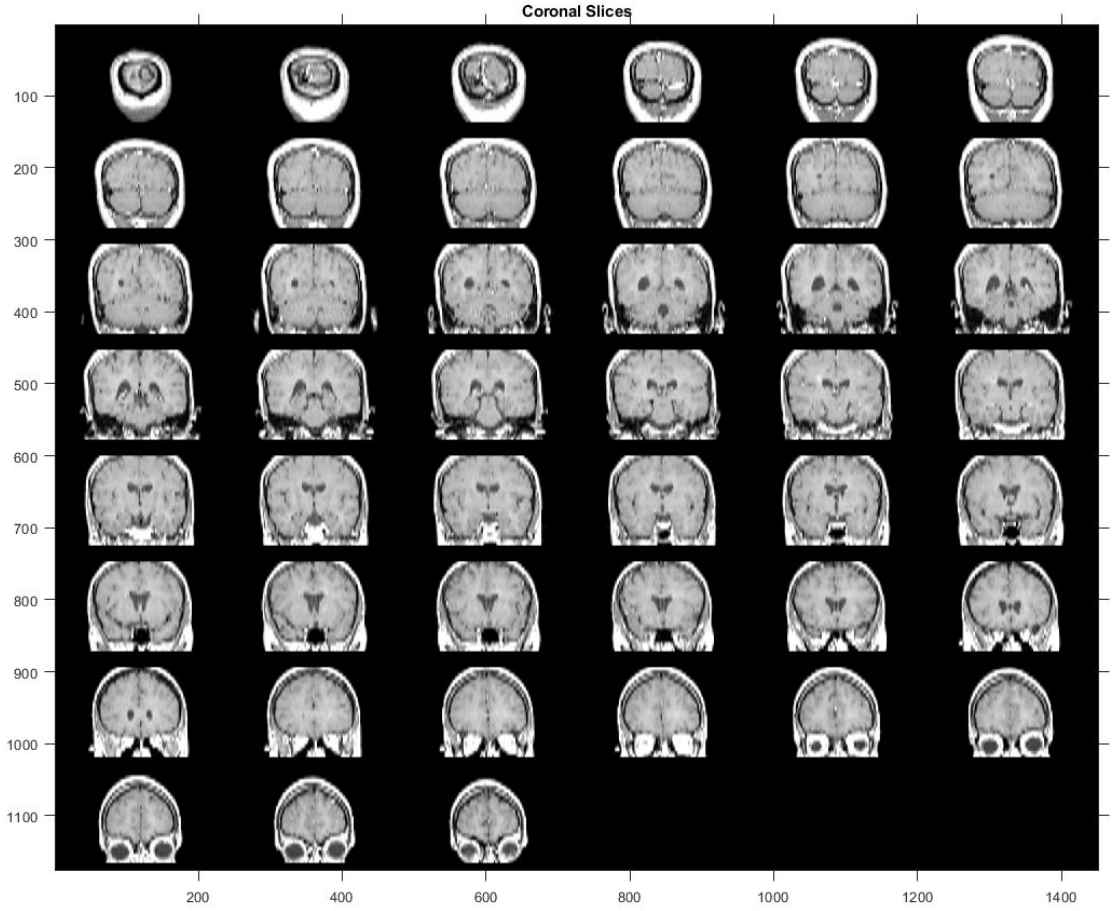
In our call to `tformarray`, `TSIZE_B` = [66 128 48] specifies the vertical, side-to-side, and front-to-back dimensions, respectively. The resampler remains the same.

```
C = tformarray(D,T4,R3,[4 2 1],[1 2 4],[66 128 45],[],0);
```

Note that all array permutations and flips in steps 3, 4, and 5 were handled as part of the `tformarray` operation.

View the coronal slices as a montage (padding the array slightly to separate the elements of the montage).

```
C2 = padarray(C,[6 0 0 0],0,'both');
figure, montage(C2,map)
title('Coronal Slices');
```


Coronal Slices
100
200
300
400
500
600
700
800
900
1000
1100
200
400
600
800
1000
1200
1400

Padding and Shearing an Image Simultaneously

This example shows how to construct a `tform` struct that represents a simple shear transformation and then applies it to an image. We explore how the transformation affects straight lines and circles, and then use it as a vehicle to explore the various options for image padding that can be used with `imtransform` and `tformarray`.

Step 1: Transform an Image Using Simple Shear

In two dimensions, a simple shear transformation that maps a pair of input coordinates `[u v]` to a pair of output coordinates `[x y]` has the form

$$x = u + a * v$$

$$y = v$$

where a is a constant.

Any simple shear is a special case of an affine transformation. You can easily verify that

$$[x \;\; y \;\; 1] = [u \;\; v \;\; 1] * \begin{bmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

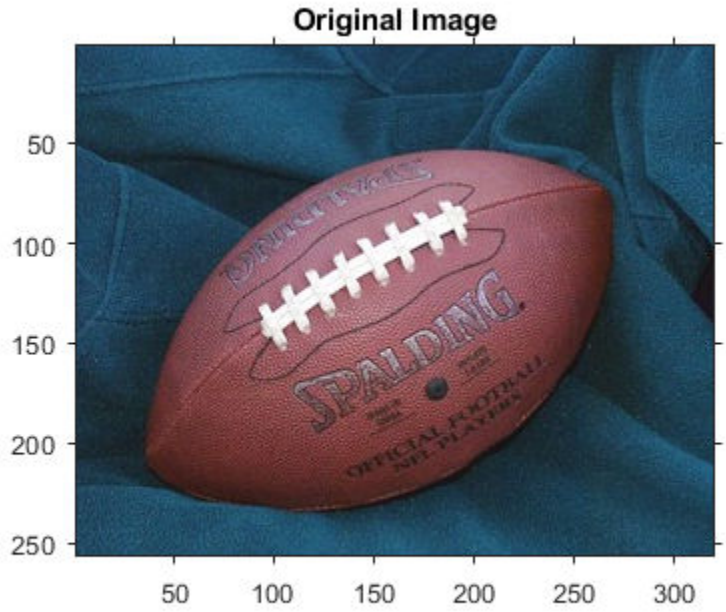
yields the values for x and y that you received from the first two equations.

Setting a = 0.45, we construct an affine `tform` struct using `maketform`.

```
a = 0.45;
T = maketform('affine', [1 0 0; a 1 0; 0 0 1] );
```

We select, read, and view and image to transform.

```
A = imread('football.jpg');
h1 = figure; imshow(A); title('Original Image');
```



We choose a shade of orange as our fill value.

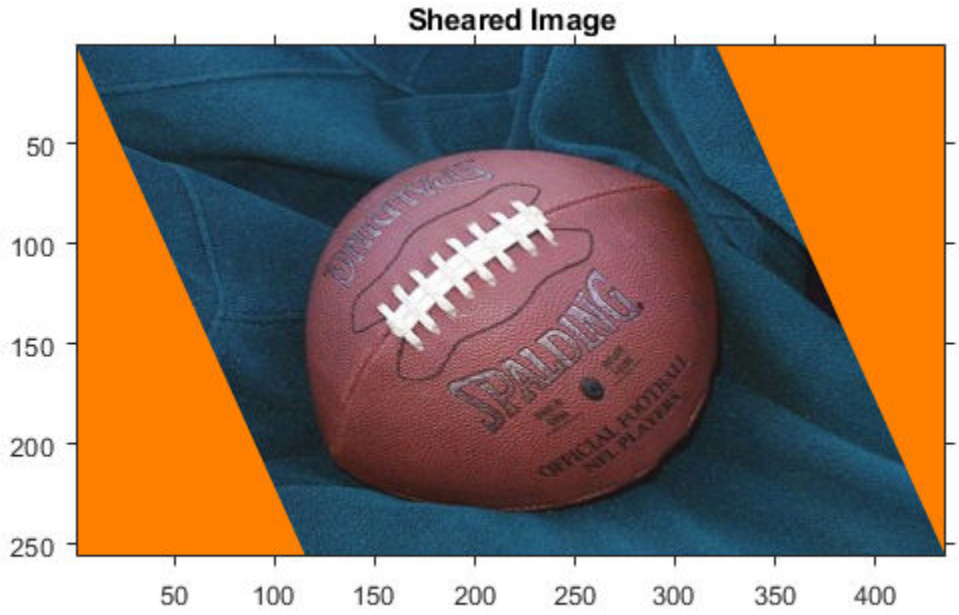
```
orange = [255 127 0]';
```

We are ready to use `T` to transform `A`. We could call `imtransform` as follows:

```
B = imtransform(A,T,'cubic','FillValues',orange);
```

but this is wasteful since we would apply cubic interpolation along both columns and rows. (With our pure shear transform, we really only need to interpolate along each row.) Instead, we create and use a resampler that applies cubic interpolation along the rows but simply uses nearest neighbor interpolation along the columns, then call `imtransform` and display the result.

```
R = makeresampler({'cubic','nearest'},'fill');
B = imtransform(A,T,R,'FillValues',orange);
h2 = figure; imshow(B);
title('Sheared Image');
```

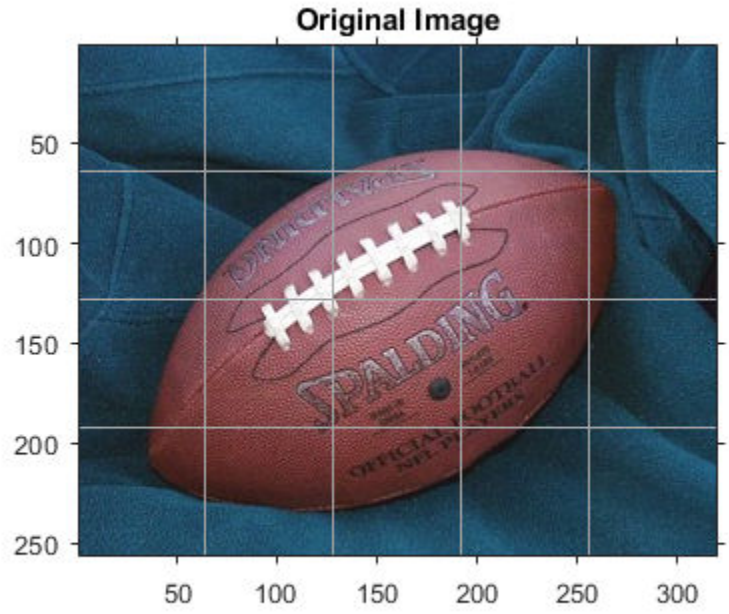


Step 2: Explore the Transformation

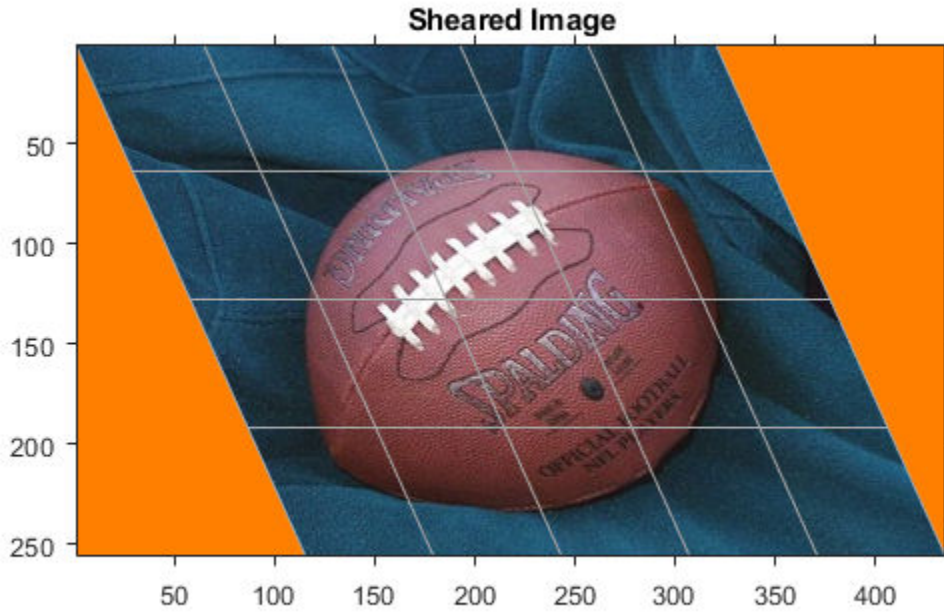
Transforming a grid of straight lines or an array of circles with `tformfwd` is a good way to understand a transformation (as long as it has both forward and inverse functions).

Define a grid of lines covering the original image, and display it over the image Then use `tformfwd` to apply the pure shear to each line in the grid, and display the result over the sheared image.

```
[U,V] = meshgrid(0:64:320,0:64:256);
[X,Y] = tformfwd(T,U,V);
gray = 0.65 * [1 1 1];

figure(h1);
hold on;
line(U, V, 'Color',gray);
line(U',V','Color',gray);
```

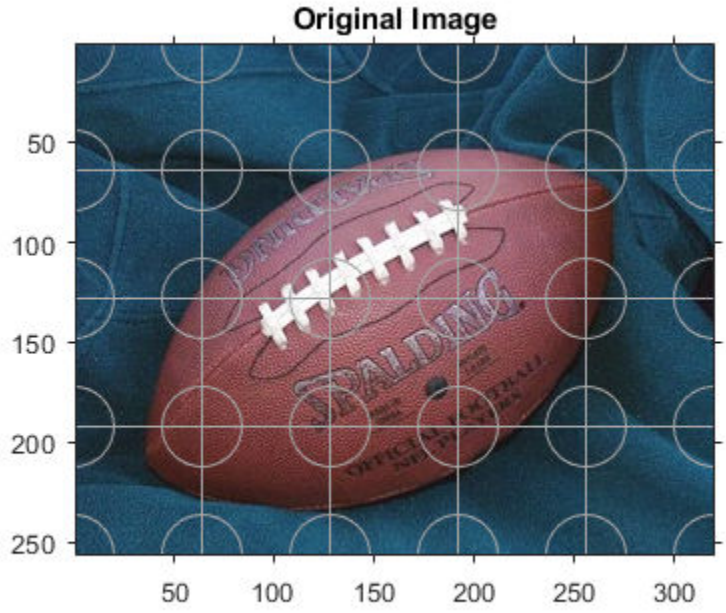


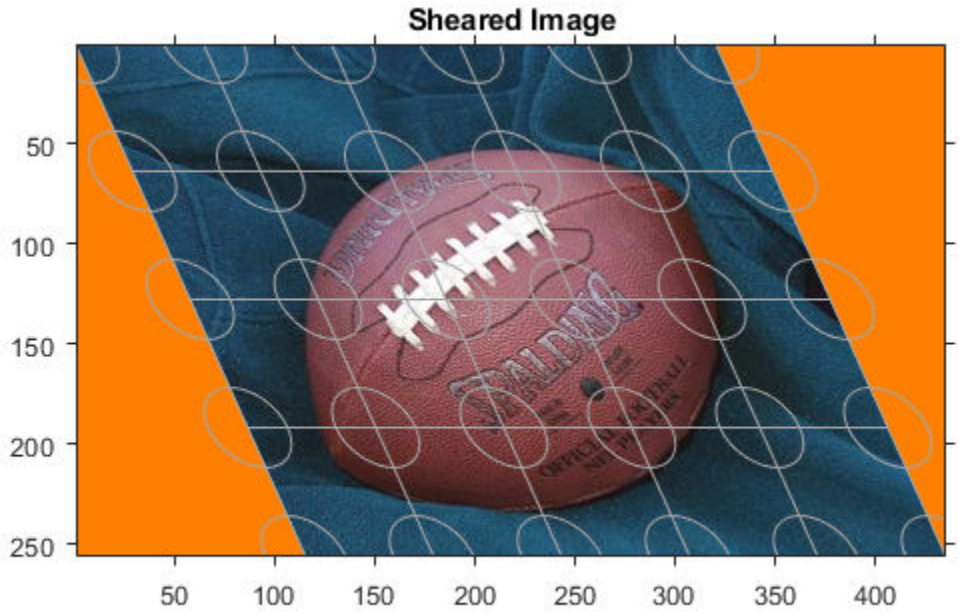

```
figure(h2);
hold on;
line(X, Y, 'Color',gray);
line(X',Y','Color',gray);
```



You can do the same thing with an array of circles.

```
gray = 0.65 * [1 1 1];
for u = 0:64:320
    for v = 0:64:256
        theta = (0 : 32)' * (2 * pi / 32);
        uc = u + 20*cos(theta);
        vc = v + 20*sin(theta);
        [xc,yc] = tformfwd(T,uc,vc);
        figure(h1); line(uc,vc,'Color',gray);
        figure(h2); line(xc,yc,'Color',gray);
    end
end
```


Original Image
50
100
150
200
250
50
100
150
200
250
300

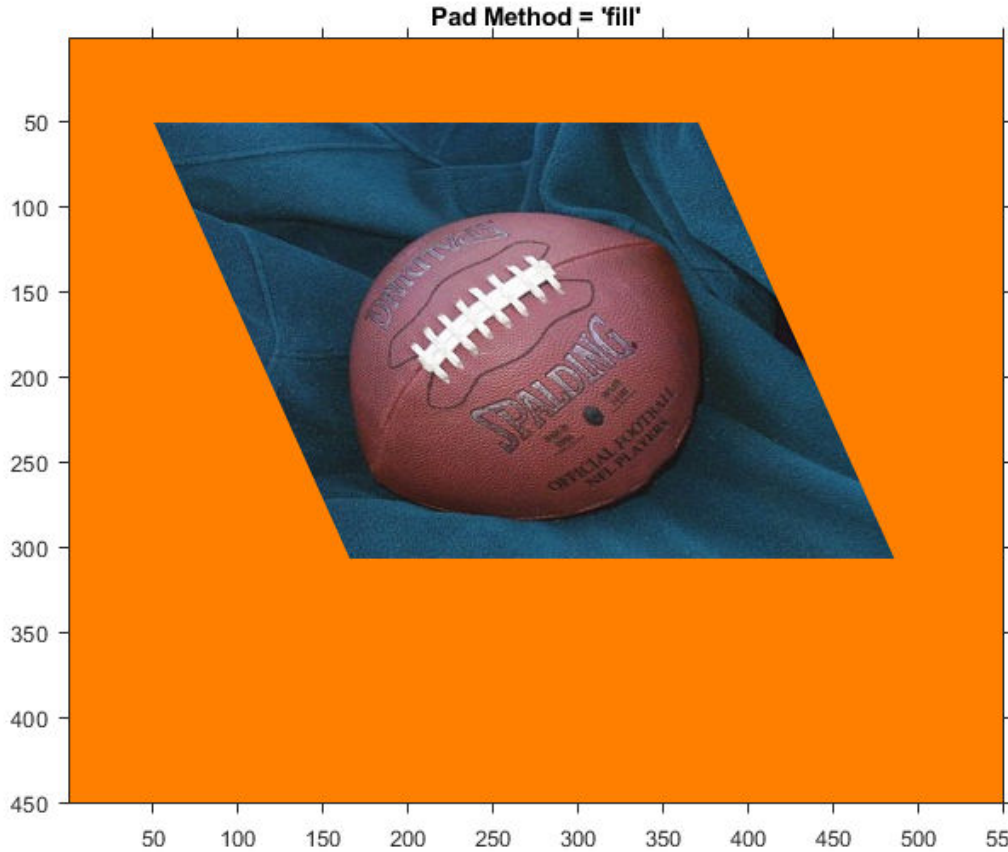


Step 3: Compare the 'fill', 'replicate', and 'bound' Pad Methods

When we applied the shear transformation, `imtransform` filled in the orange triangles to the left and right, where there was no data. That's because we specified a pad method of `'fill'` when calling `makeresampler`. There are a total of five different pad method choices (`'fill'`, `'replicate'`, `'bound'`, `'circular'`, and `'symmetric'`). Here we compare the first three.

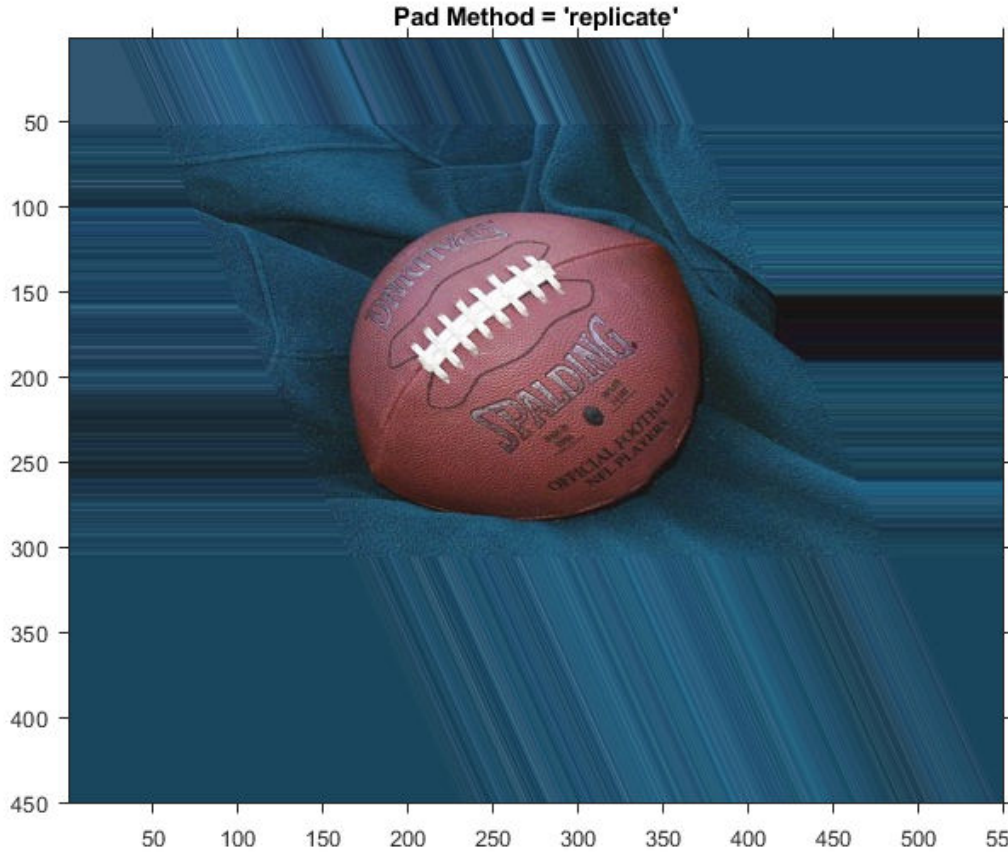
First, to get a better look at how the `'fill'` option worked, use the `'XData'` and `'YData'` options in `imtransform` to force some additional space around the output image.

```
R = makeresampler({'cubic','nearest'},'fill');

Bf = imtransform(A,T,R,'XData',[-49 500],'YData',[-49 400],...
                 'FillValues',orange);

figure, imshow(Bf);
title('Pad Method = ''fill''');
```



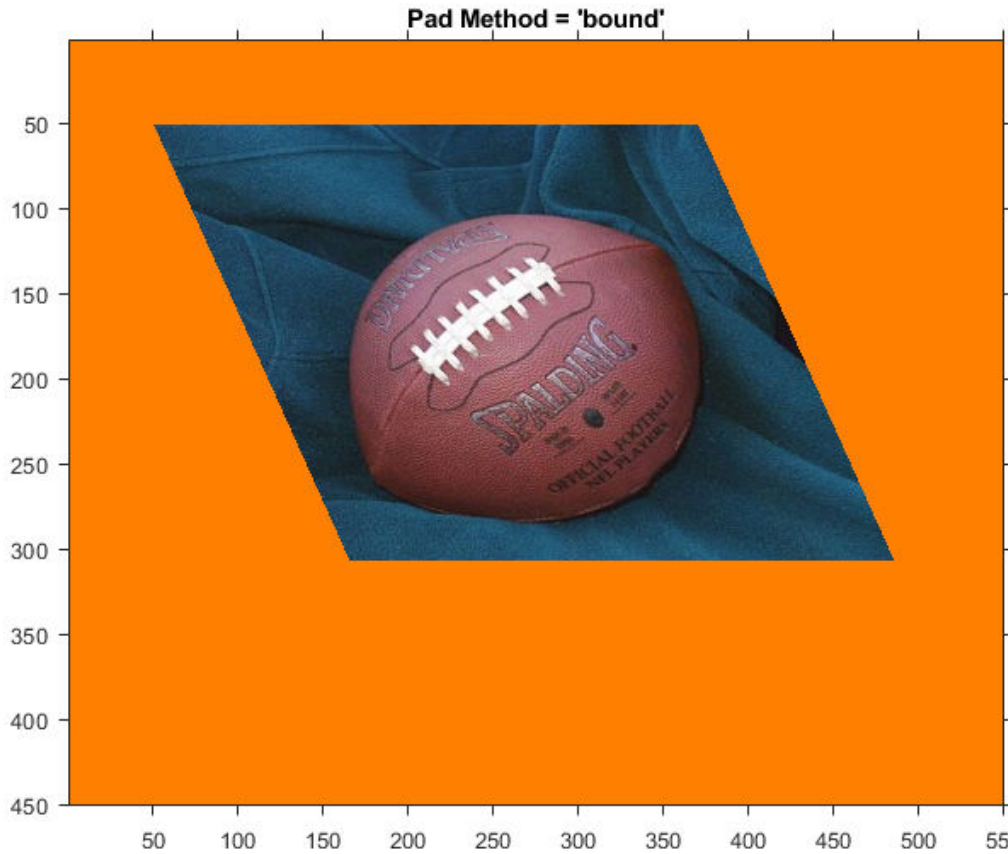
Now, try the `'replicate'` method (no need to specify fill values in this case).

```
R = makeresampler({'cubic','nearest'},'replicate');
Br = imtransform(A,T,R,'XData',[-49 500],'YData', [-49 400]);

figure, imshow(Br);
title('Pad Method = ''replicate''');
```



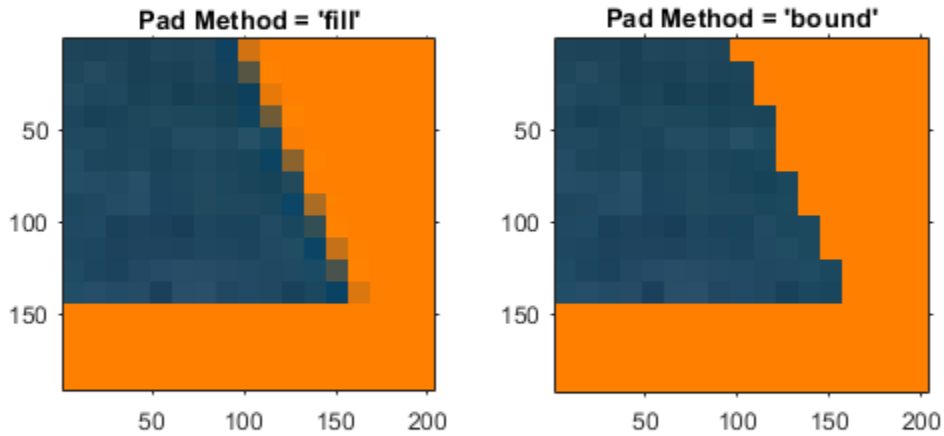
And try the `'bound'` method.

```
R = makeresampler({'cubic','nearest'}, 'bound');
Bb = imtransform(A,T,R,'XData',[-49 500],'YData',[-49 400],...
                 'FillValues',orange);
figure, imshow(Bb);
title('Pad Method = ''bound''');
```



Results with `'fill'` and `'bound'` look very similar, but look closely and you'll see that the edges are smoother with `'fill'`. That's because the input image is padded with the fill values, then the cubic interpolation is applied across the edge, mixing fill and image values. In contrast, `'bound'` recognizes a strict boundary between the inside and outside of the input image. Points falling outside are filled. Points falling inside are interpolated, using replication when they're near the edge. A close up look helps show this more clearly. We choose `XData` and `YData` to bracket a point near the lower right corner of the image, in the output image space, the resize with `'nearest'` to preserve the appearance of the individual pixels.

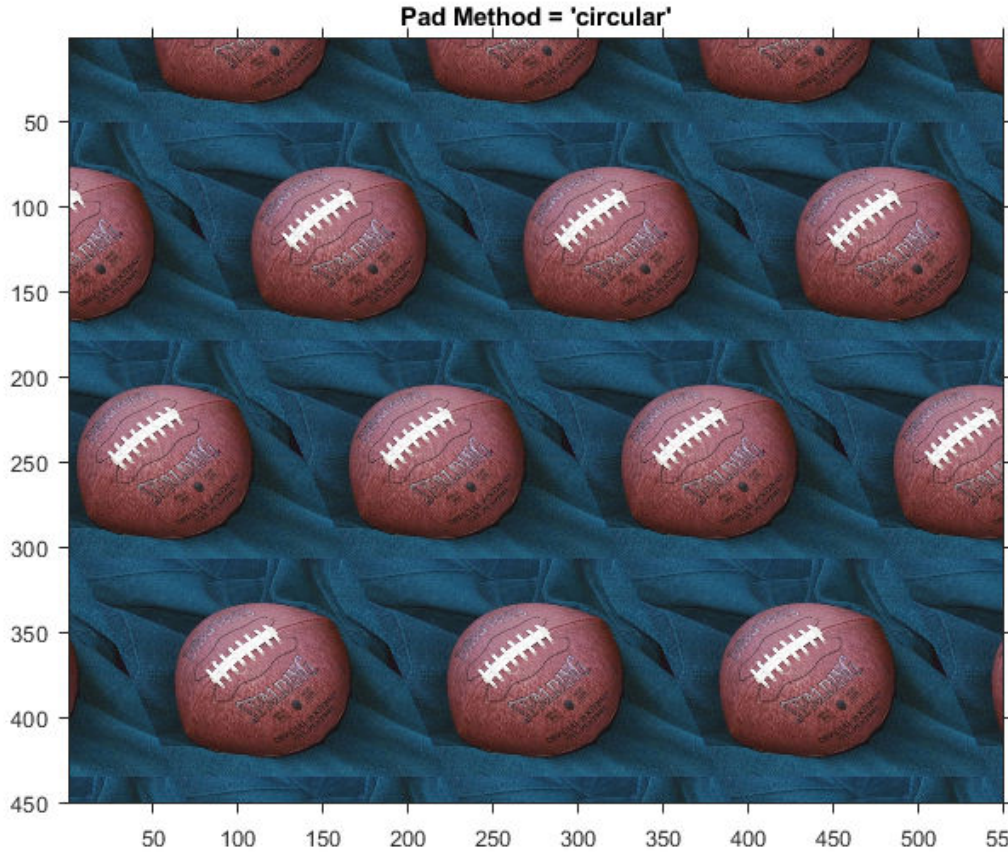
```
R = makeresampler({'cubic','nearest'},'fill');
Cf = imtransform(A,T,R,'XData',[423 439],'YData',[245 260],...
                 'FillValues',orange);

R = makeresampler({'cubic','nearest'},'bound');
Cb = imtransform(A,T,R,'XData',[423 439],'YData',[245 260],...
                 'FillValues',orange);

Cf = imresize(Cf,12,'nearest');
Cb = imresize(Cb,12,'nearest');

figure;
subplot(1,2,1); imshow(Cf); title('Pad Method = ''fill''');
subplot(1,2,2); imshow(Cb); title('Pad Method = ''bound''');
```



Step 4: Exercise the 'circular' and 'symmetric' Pad Methods

The remaining two pad methods are `'circular'` (circular repetition in each dimension) and `'symmetric'` (circular repetition of the image with an appended mirror image). To show more of the pattern that emerges, we redefine the transformation to cut the scale in half.

```
Thalf = maketform('affine',[1 0; a 1; 0 0]/2);

R = makeresampler({'cubic','nearest'},'circular');
Bc = imtransform(A,Thalf,R,'XData',[-49 500],'YData',[-49 400],...
                 'FillValues',orange);
figure, imshow(Bc);
title('Pad Method = ''circular''');
```




```
R = makeresampler({'cubic','nearest'},'symmetric');
Bs = imtransform(A,Thalf,R,'XData',[-49 500],'YData',[-49 400],...
                 'FillValues',orange);
figure, imshow(Bs);
title('Pad Method = ''symmetric''');
```

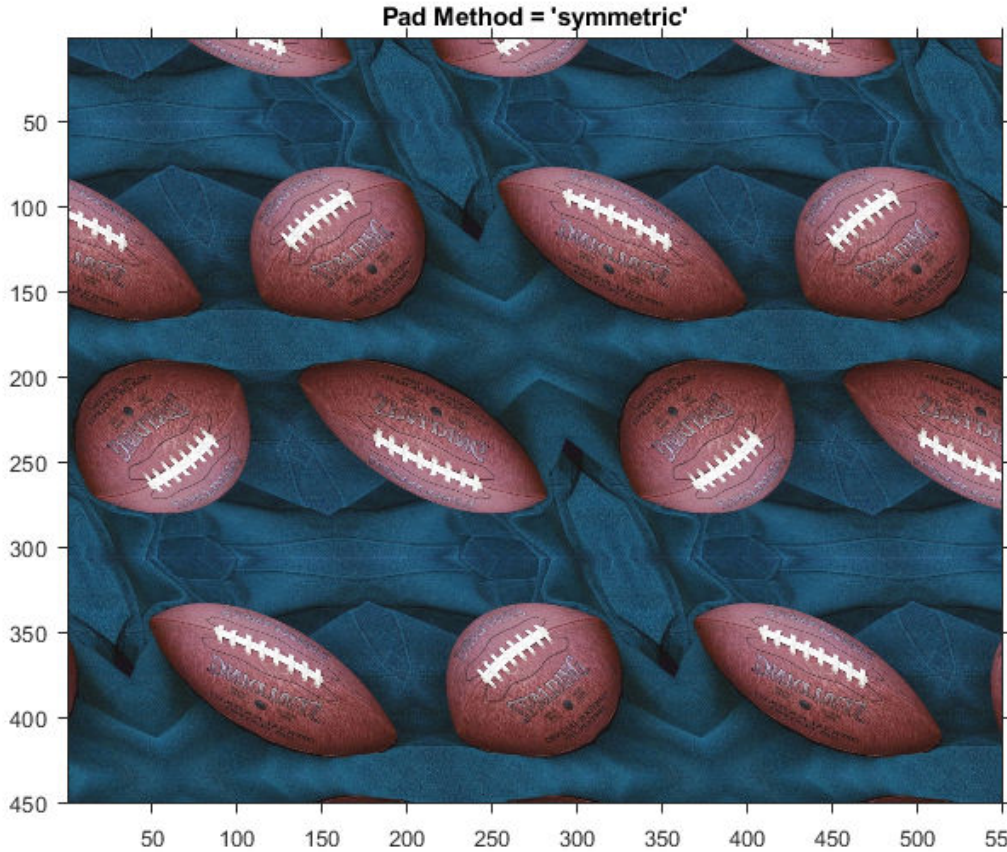

Pad Method = 'symmetric'
50
100
150
200
250
300
350
400
450
50
100
150
200
250
300
350
400
450
500
550

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.

- “Approaches to Registering Images” on page 7-3
- “Register Images Using the Registration Estimator App” on page 7-9
- “Load Images into Registration Estimator App” on page 7-18
- “Tune Registration Settings in Registration Estimator App” on page 7-22
- “Export the Results from Registration Estimator App” on page 7-26
- “Techniques Supported by Registration Estimator App” on page 7-28
- “Intensity-Based Automatic Image Registration” on page 7-31
- “Create an Optimizer and Metric for Intensity-Based Image Registration” on page 7-34
- “Use Phase Correlation as Preprocessing Step in Registration” on page 7-36
- “Register Multimodal MRI Images” on page 7-42
- “Register Multimodal 3-D Medical Images” on page 7-56
- “Registering an Image Using Normalized Cross-Correlation” on page 7-65
- “Control Point Registration” on page 7-72
- “Geometric Transformation Types for Control Point Registration” on page 7-75
- “Control Point Selection Procedure” on page 7-77
- “Start the Control Point Selection Tool” on page 7-80
- “Find Visual Elements Common to Both Images” on page 7-83
- “Select Matching Control Point Pairs” on page 7-88
- “Export Control Points to the Workspace” on page 7-95
- “Find Image Rotation and Scale” on page 7-98
- “Find Image Rotation and Scale Using Automated Feature Matching” on page 7-103
- “Use Cross-Correlation to Improve Control Point Placement” on page 7-110

- “Register an Aerial Photograph to a Digital Orthophoto” on page 7-111
- “Registering an Aerial Photo to an Orthophoto” on page 7-116

Approaches to Registering Images

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

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 medical images captured with different diagnostic modalities, such as MRI and SPECT. Image registration enables 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.

Image Processing Toolbox offers three image registration approaches: an interactive Registration Estimator app, intensity-based automatic image registration, and control point registration. Computer Vision System Toolbox offers automated feature detection and matching.

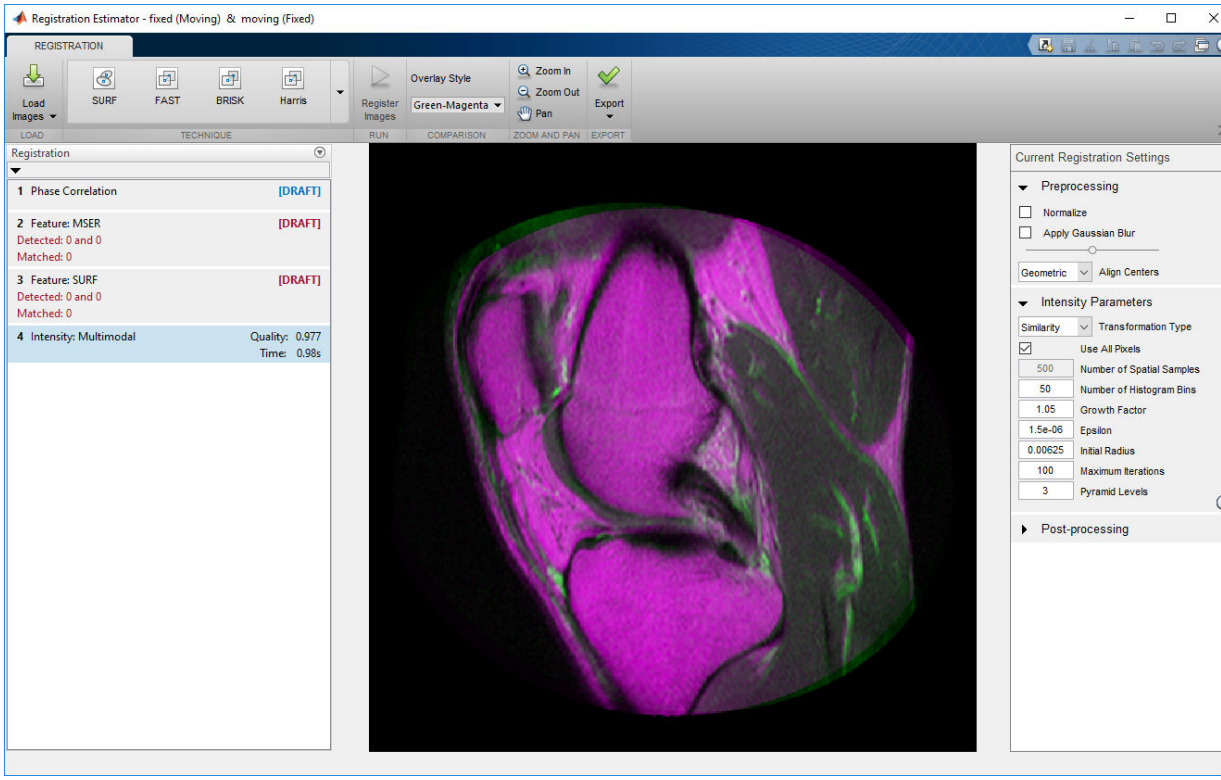
Capability	**“Registration Estimator App” on page 7-4**	**“Intensity-Based Automatic Image Registration” on page 7-5**	**“Control Point Registration” on page 7-6**	**“Automated Feature Detection and Matching” on page 7-7 (requires Computer Vision System Toolbox)**
Interactive registration	X			
Automated intensity-based registration	X	X		

Capability	“Registration Estimator App” on page 7-4	“Intensity-Based Automatic Image Registration” on page 7-5	“Control Point Registration” on page 7-6	“Automated Feature Detection and Matching” on page 7-7 (requires Computer Vision System Toolbox)
Automated feature detection	X			X
Manual feature selection			X	
Automated feature matching	X		X	X
Nonrigid transformation	X	X	X	
Fully automatic workflow		X		X
Supports 3-D images		X		

Registration Estimator App

Registration Estimator app enables you to register 2-D images interactively. You can compare different registration techniques, tune settings, and visualize the registered image. The app provides a quantitative measure of quality, and it returns the registered image and the transformation matrix. The app also generates code with your selected registration technique and settings, so you can apply an identical transformation to multiple images.

Registration Estimator app offers six feature-based techniques, three intensity-based techniques, and one nonrigid registration technique. For a more detailed comparison of the available techniques, see “Techniques Supported by Registration Estimator App” on page 7-28.



Intensity-Based Automatic Image Registration

“Intensity-Based Automatic Image Registration” on page 7-31 maps pixels in each image based on relative intensity patterns. You can register both monomodal and multimodal image pairs, and you can register 2-D and 3-D images. This approach is useful for:

- Registering a large collection of images
- Automated registration

To register images using an intensity-based technique, use `imregister` and specify the type of geometric transformation to apply to the moving image. `imregister` iteratively adjusts the transformation to optimize the similarity of the two images.

Alternatively, you can estimate a localized displacement field and apply a nonrigid transformation to the moving image using `imregdemons`.

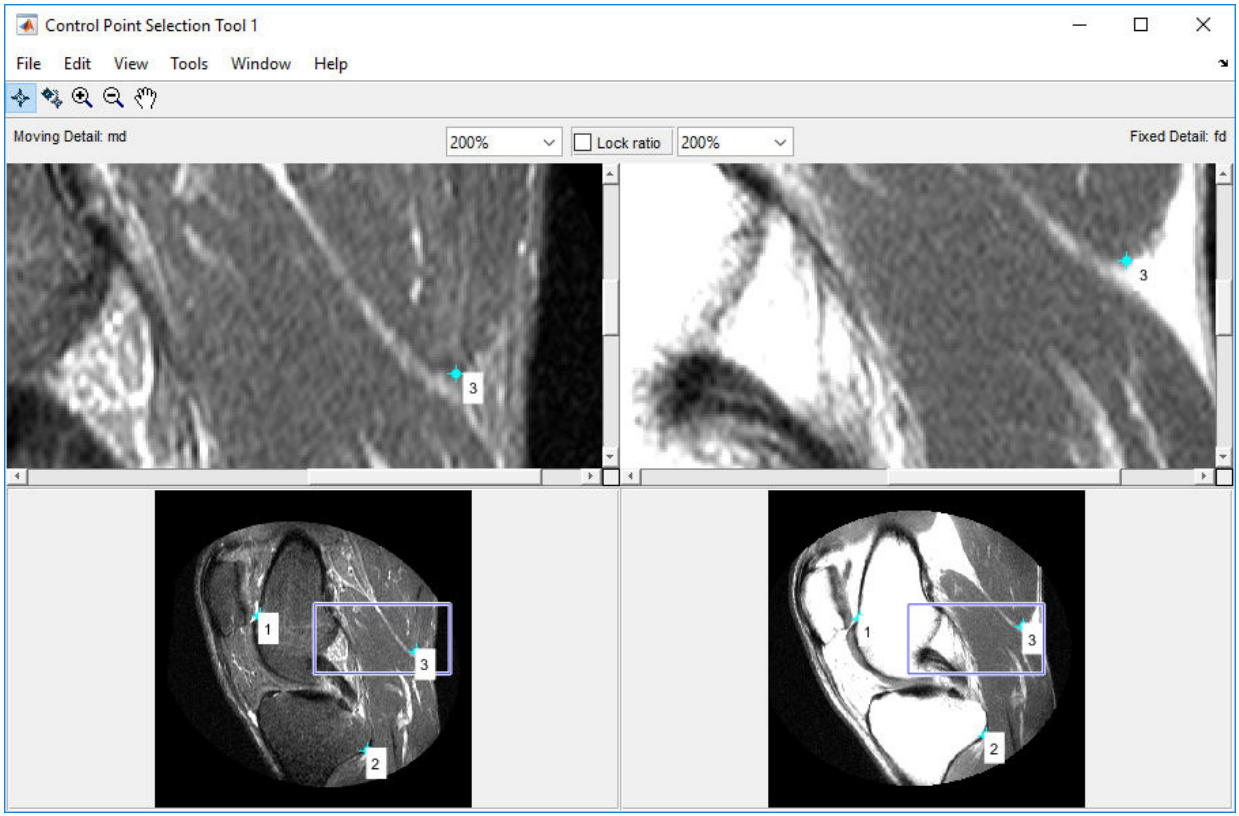
Control Point Registration

“Control Point Registration” on page 7-72 enables you to select common features in each image manually. Control point registration is useful when:

- You want to prioritize the alignment of specific features, rather than the entire set of features detected using automated feature detection. For example, when registering two medical images, you can focus the alignment on desired anatomical features and disregard matched features that correspond to less informative anatomical structures.
- Images have repeated patterns that provide an unclear mapping using automated feature matching. For example, photographs of buildings with many windows, or aerial photographs of gridded city streets, have many similar features that are challenging to map automatically. In this case, manual selection of control point pairs can provide a clearer mapping of features, and thus a better transformation to align the feature points.

Control point registration can apply many types of transformations to the moving image. Global transformations, which act on the entire image uniformly, include affine, projective, and polynomial geometric transformations. Nonrigid transformations, which act on local regions, include piecewise linear and local weighted mean transformations.

Use the Control Point Selection Tool to select control points. Start the tool with `cpselect`.



Automated Feature Detection and Matching

Automated “Feature Detection and Extraction” (Computer Vision System Toolbox) detects features such as corners and blobs, matches corresponding features in the moving and fixed images, and estimates a geometric transform to align the matched features.

For an example, see “Find Image Rotation and Scale Using Automated Feature Matching” on page 7-103. You must have Computer Vision System Toolbox to use this method.

Note The Registration Estimator app offers six feature-based techniques to register a single pair of images. However, the app does not provide an automated workflow to register multiple images.

See Also

`imregister` | `imwarp`

Related Examples

- “Register Images Using the Registration Estimator App” on page 7-9
- “Register Multimodal MRI Images” on page 7-42
- “Register an Aerial Photograph to a Digital Orthophoto” on page 7-111

Register Images Using the Registration Estimator App

This example shows how to use the **Registration Estimator** app to align a pair of images. The Registration Estimator offers several registration techniques using feature-based, intensity-based, and nonrigid registration algorithms. The following example shows one possible path to registering images with the app.

In this section...
“Load Images into Registration Estimator App” on page 7-9 “Obtain Initial Registration Estimate” on page 7-11 “Refine the Registration Settings” on page 7-12 “Export Registration Results” on page 7-16

Load Images into Registration Estimator App

Create two misaligned images in the workspace. In this example, the moving image `J` is generated by rotating the fixed image `I` clockwise 30 degrees.

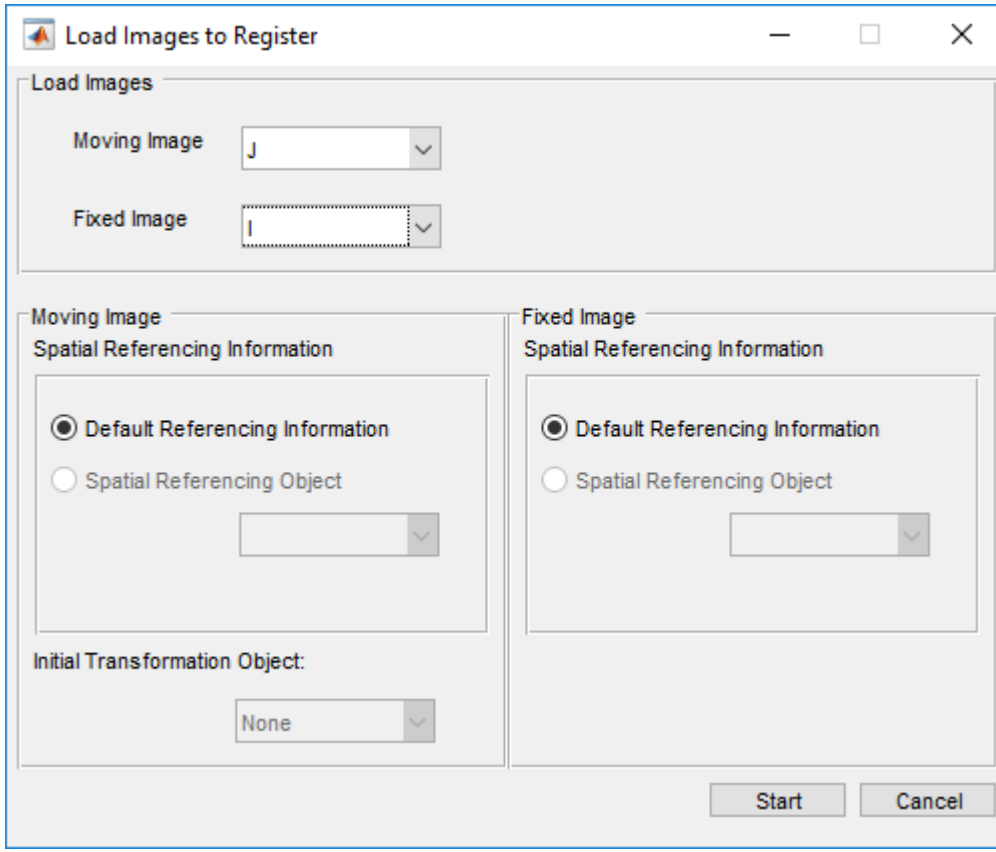
```
I = imread('cameraman.tif');
J = imrotate(I,-30);
```

Open the Registration Estimator app. From the MATLAB Toolstrip, open the Apps tab and under Image Processing and Computer Vision, click **Registration Estimator** . You can also open the Registration Estimator from the command line:

```
registrationEstimator
```

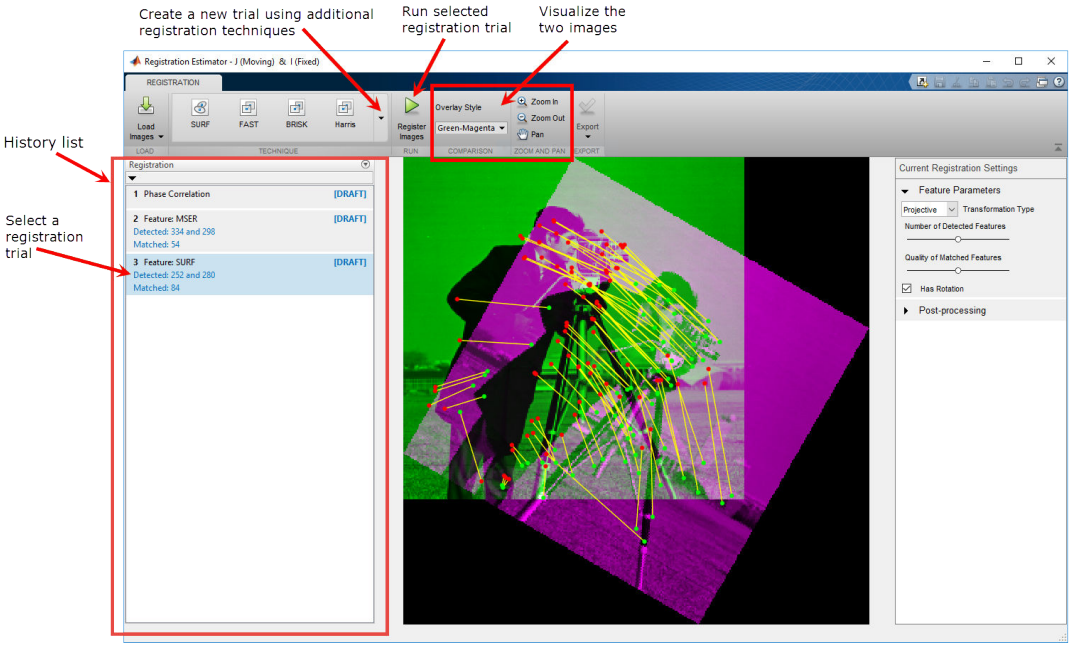
Load the images into the Registration Estimator app. Click **Load Images**, and select the `Load from Workspace` option. In the Moving Image dialog box, select the image to transform. In the Fixed Image dialog box, select the image with the target orientation. There is no spatial referencing information or initial transformation object for the images in this example. To continue, click **Start**.

For more information, see “Load Images into Registration Estimator App” on page 7-18.



The app displays an overlay of the images. The default Green-Magenta color scheme shows the fixed image in green and the moving image in magenta. The overlay looks gray in areas where the two images have similar intensity. You can explore additional overlay styles to help you visualize the results of the registration.

Three registration trials are preloaded in the history list: `Phase Correlation`, `Feature: MSER`, and `Feature: SURF`. When you click a feature-based technique in the history list, the app displays a set of red and green dots connected by yellow lines. These points are the matched features used to align the images.

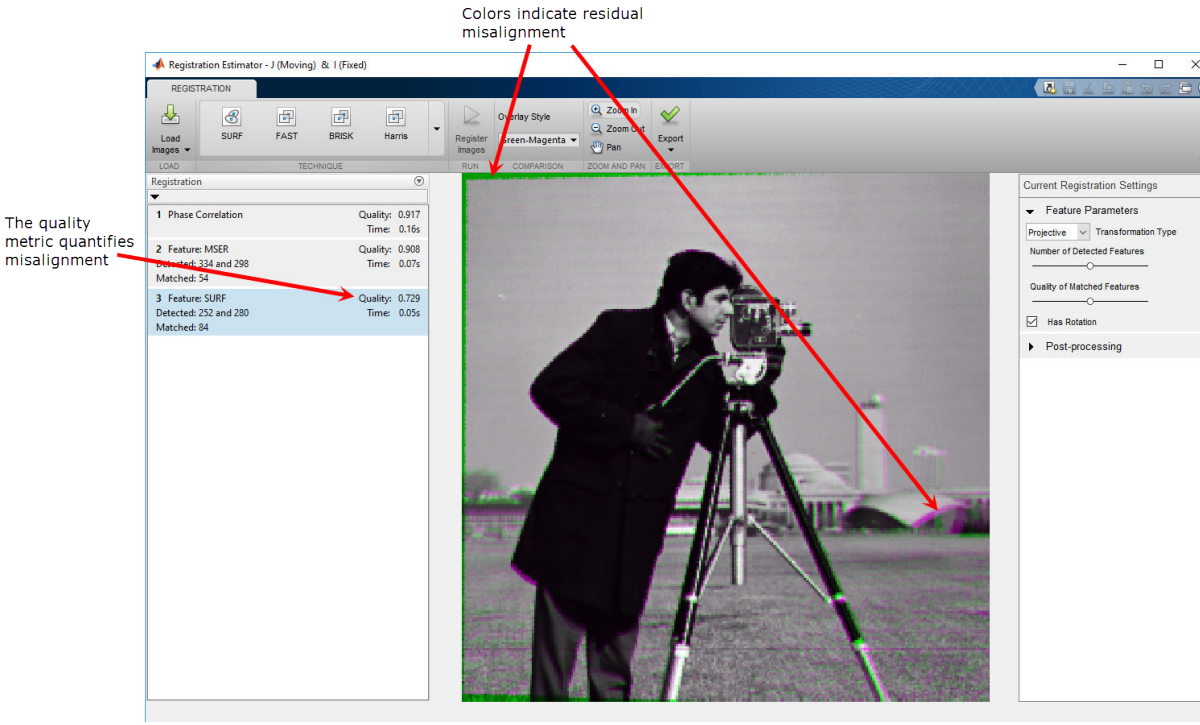


Obtain Initial Registration Estimate

Run the registration trials with default settings. Click each preloaded trial in the history list, then click **Register Images**.

After the registration finishes, the image overlay is updated with the new results. It shows a quality metric and computation time for each trial in the history list. The quality metric, based loosely on `ssim`, provides an overall estimate of registration quality. Different registration techniques and settings can yield similar quality scores but show error in different regions of the image. Visual inspection can confirm which registration technique is the most acceptable.

Note Due to randomness in the registration optimizer, the quality metric, registered image, and geometric transformation can vary slightly between trials despite identical registration settings.



To use a different registration technique, select it from the **Technique** menu. For more information, see “Techniques Supported by Registration Estimator App” on page 7-28.

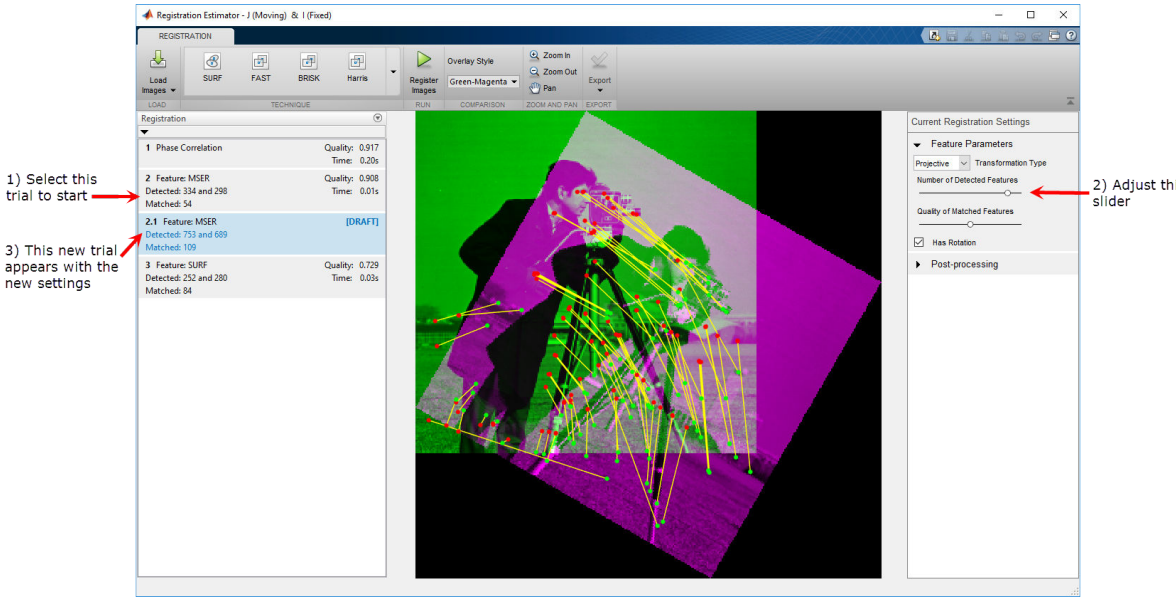
Refine the Registration Settings

Now that you have an initial registration estimate, adjust registration settings to improve the quality of the alignment. For more information on available settings, see “Tune Registration Settings in Registration Estimator App” on page 7-22.

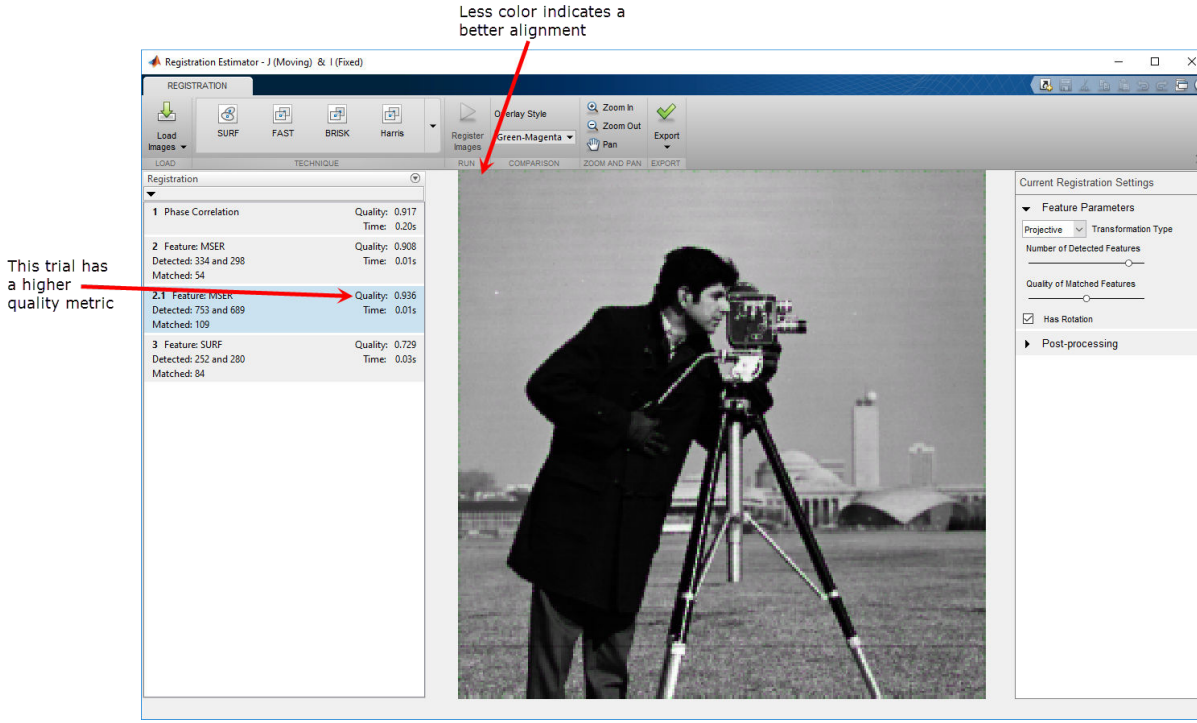
Continuing with this example, adjust the settings of the MSER trial. Try increasing the number of detected features and the quality of matched features independently to see if either improves the quality of the registration.

To increase the number of detected features, click the `Feature: MSER` trial, numbered 2, in the history list. In the right panel, move the **Number of Detected Features** slider to the right to increase the number of features. Observe that these new settings generate

a trial draft numbered 2.1 in the history list. The image overlay also shows more matched features, as expected.

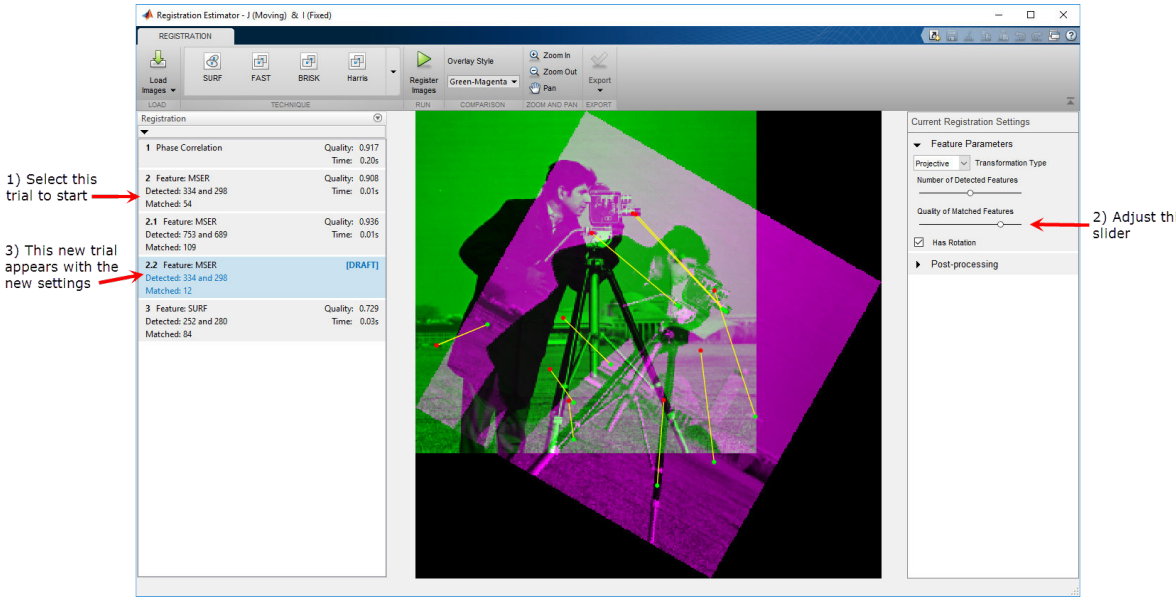


To run the registration with these settings, click **Register Images**.

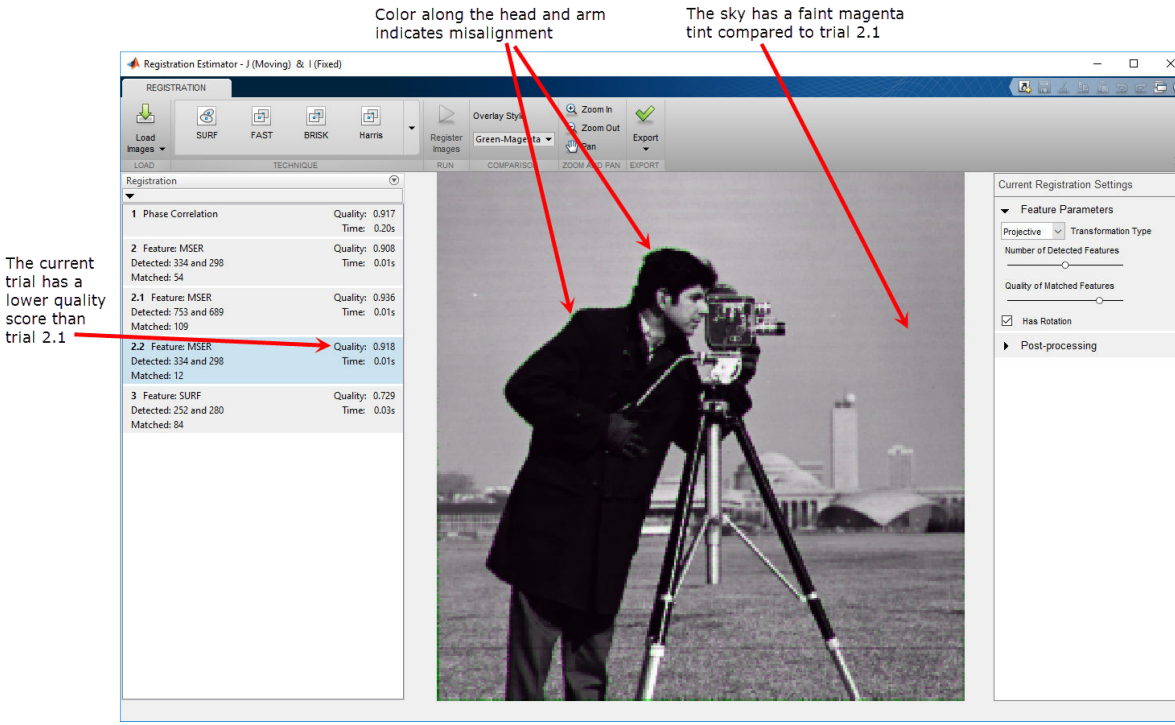


In the updated overlay, notice how there is less error along the edge of the image than there was for the MSER trial with default settings. There is less of a magenta tint to the overall image using the new settings. The quality metric has also improved.

To see the effect of increasing the quality of matched features, click the Feature: MSER trial 2 (not 2.1) in the history list. In the right panel, move the **Quality of Matched Features** slider to the right to increase the quality. Another trial draft, numbered 2.2, appears in the history list. The image overlay shows a smaller number of high quality matched points.



To see the registration with these settings, click **Register Images**.



Compared to the other MSER trials, this trial has more error along the left side of the head and arm, but less error along the edge of the image. Although the quality metric has increased with these new settings, you can use subjective analysis to determine which settings are more appropriate for your application.

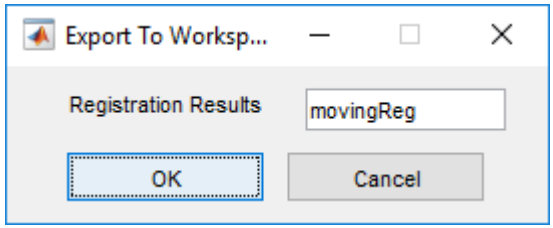
Explore different combinations of number of detected features and quality of matched pairs. If you know the conditions under which the images were obtained, then you can select a different transformation type or clear the **Has Rotation** option. Post-processing using nonrigid transformations is available for advanced workflows.

Export Registration Results

When you find an acceptable registration, export the registered image and the geometric transformation to the workspace.

In this example, trial 2.1 has the highest quality and no severe regions of misalignment, so select this trial to export. Click trial 2.1 in the history list, then click **Export** and select

Export Images. In the Export to Workspace dialog box, assign a name to the registration output. The output is a structure that contains the final registered image and the geometric transformation.



You can use the registration results to apply a similar registration to multiple frames in an image sequence. To learn more, see “Export the Results from Registration Estimator App” on page 7-26.

Load Images into Registration Estimator App

You can load images into Registration Estimator app from file or from the workspace. You can also provide optional spatial referencing information and an optional initial geometric transformation.

In this section...

"Load Images from File or from the Workspace" on page 7-18
"Provide Spatial Referencing Information" on page 7-20
"Provide an Initial Geometric Transformation" on page 7-21

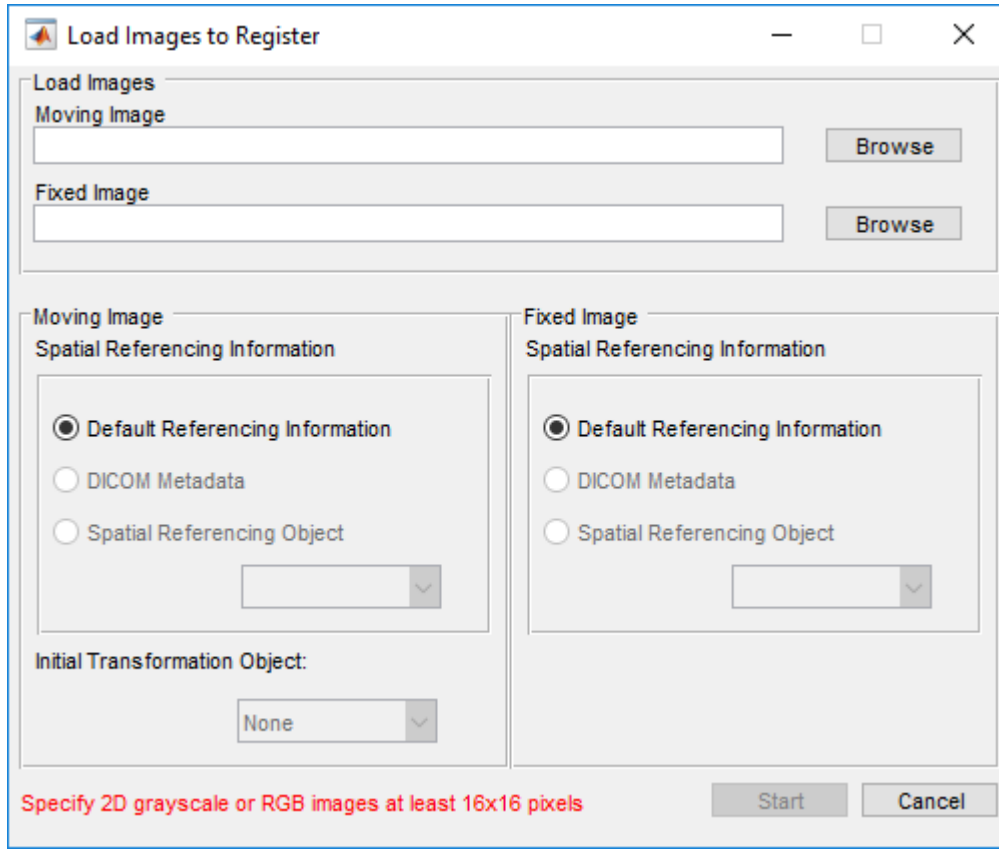
Load Images from File or from the Workspace

You can load images into Registration Estimator app from file or from the workspace.

Access both methods by clicking the **Load Images** icon

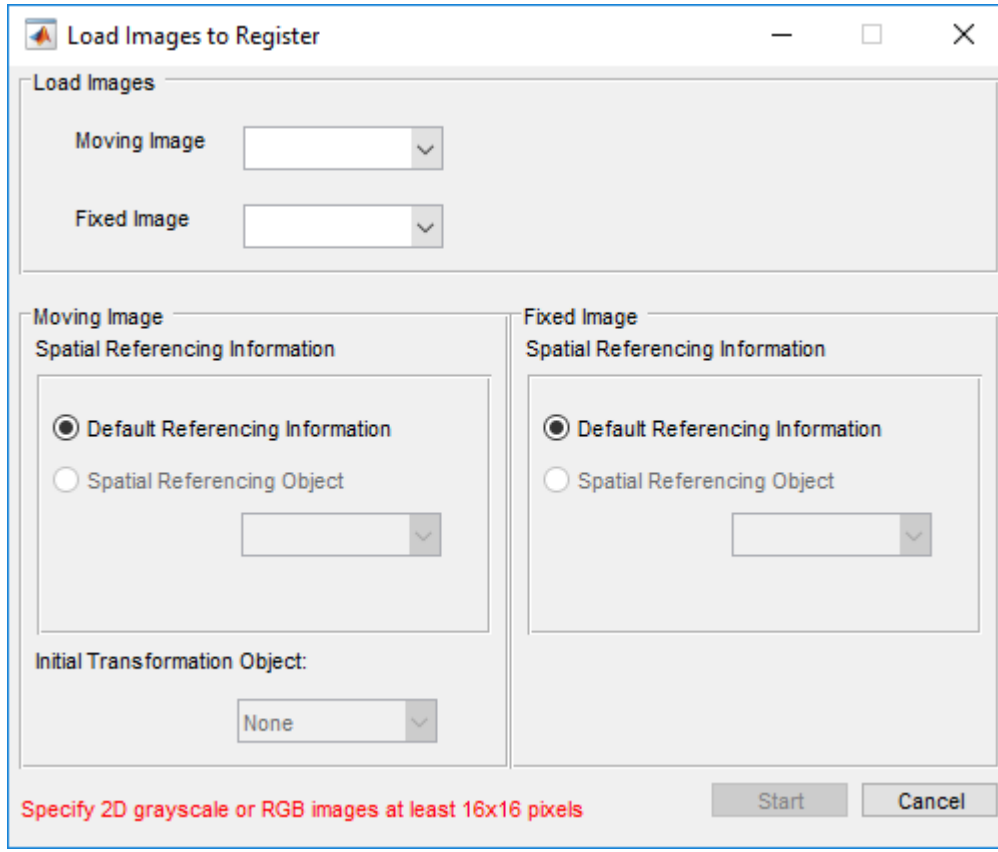
in the Registration Estimator app.

- To load images from a file, click **Load Images**, and select the `Load from file` option.



In the dialog box, select the file path of the moving and fixed images. Only certain file types are permitted: BMP, JPG, JPEG, TIF, TIFF, PNG, and DCM.

- To load variable from the workspace, click **Load Images** and select the `Load from workspace` option.



In the dialog box, select two images from your workspace. You can work with a wider range of file formats by loading images from file into your workspace using `imread`. Registration Estimator app supports any workspace image that was loaded using `imread`. You can also read DICOM images into the workspace using `dicomread`.

The Registration Estimator app only supports 2-D images. Before continuing, all RGB images are converted to grayscale using the `rgb2gray` function.

Provide Spatial Referencing Information

If you have 2-D spatial referencing objects in your workspace, or if you load DICOM images from a file, you can provide optional spatial referencing information. Spatial

referencing information is useful if you want to orient the images to a world coordinate system. For more information about spatial referencing objects, see `imref2d`.

Note If you load DICOM images into the workspace using `dicomread`, spatial referencing information in the metadata is no longer associated with the image data. To preserve spatial referencing information with DICOM images, either load the images from file or create an `imref2d` object from the image metadata. For more information about DICOM metadata, see “Read Metadata from DICOM Files” on page 3-12.

If you do not have spatial referencing information, the Spatial Referencing Object and DICOM Metadata radio buttons are inactive.

Provide an Initial Geometric Transformation

You can provide an optional initial geometric transformation using `affine2d` and `projective2d` geometric transformation objects in your workspace. An initial geometric transformation is useful if you are processing a batch of images with similar initial misalignment. Once the first moving image has been registered, you can export the geometric transformation to the workspace and apply the transformation to other images in the series. See “Export the Results from Registration Estimator App” on page 7-26.

If you do not have a geometric transformation object in your workspace, the Initial Transformation Object selection box is inactive.

See Also

Functions
`dicomread` | `imread`

Classes
`affine2d` | `imref2d` | `projective2d`

Related Examples

- “Register Images Using the Registration Estimator App” on page 7-9

Tune Registration Settings in Registration Estimator App

Adjust settings in the Registration Estimator app based on your registration technique.

Note Due to randomness in the registration optimizer, the quality metric, registered image, and geometric transformation can vary slightly between trials despite identical registration settings.

In this section...

"Geometric Transformations Supported by Registration Estimator App" on page 7-22
"Feature-Based Registration Settings" on page 7-23
"Intensity-Based Registration Settings" on page 7-24
"Nonrigid and Post-Processing Settings" on page 7-25

Geometric Transformations Supported by Registration Estimator App

All feature-based and intensity-based registration techniques allow you to set the transformation type. For more details about each type of transformation matrix, see "Matrix Representation of Geometric Transformations" on page 6-19.

- *Translation* transformations preserve the size and orientation of the image. Each pixel in the image is displaced the same amount in the same direction.
- *Rigid* transformations include rotation and translation. Rigid transformations preserve length.

  Note Although reflection is a type of rigid transformation, Registration Estimator app does not support reflection.
- *Similarity* transformations include isotropic scaling, rotation, and translation. Similarity transformations preserve shape, but not size. When used with a featured-based registration technique, at least two matched pairs of points are required.
- *Affine* transformations include shear and all supported similarity transformations. Affine transformations preserve parallel lines, but not necessarily angles between lines

or distances between points. When used with a featured-based registration technique, at least three matched pairs of points are required.

- *Projective* transformations allow tilting in addition to all supported affine transformations. When used with a featured-based registration technique, at least four matched pairs of points are required.

Registration Technique	Translation	Rigid	Similarity	Affine	Projective
All Feature-Based Techniques			X	X	X
Monomodal Intensity	X	X	X	X	
Multimodal Intensity	X	X	X	X	
Phase Correlation	X	X	X		

Feature-Based Registration Settings

Feature-based registration allows you to adjust three settings in addition to the geometric transformation type:

- Number of Detected Features. The transformation type determines the minimum number of matched features required to perform a registration. Similarity transformations require two or more matched features. Affine transformations require three or more matched features. Projective transformations require four or more matched features.
- Quality of matched features. The quality value is a combination of matched features options.
- Rotation. By default, feature-based registration allows the moving image to rotate. However, some imaging scenarios, such as stereoscopy, produce images with identical rotation. If your images have the same rotation, clearing this option can improve the accuracy of the registration.

Intensity-Based Registration Settings

All intensity-based registration techniques allow you to select the geometric transformation type. Additional settings are available depending on the registration technique.

Monomodal and multimodal intensity-based registration provide three common settings:

- Normalize. This option scales the pixel values of both images to the same dynamic range.
- Apply Gaussian blur. Smoothing the images with a Gaussian blur can help the optimizer find the global maximum or minimum of the solution surface. However, smoothing changes the shape of the surface, and over-smoothing can shift the position of the extrema. Large amounts of blurring are useful when the images are severely misaligned at the start of the registration, to help the optimizer search the correct basin of attraction. Small amounts of blurring are useful when the images start with close alignment.
- Align centers. This option provides an initial transformation that aligns the world coordinates of the centers of the two images. The `geometric` option aligns the geometric centers, based on the spatial referencing information of the images. The `center of mass` option aligns the centers of mass, calculated from the weighted mean of pixel intensities.

Monomodal registration enables you to adjust the properties of the regular step gradient descent optimizer. For more information about the properties of this optimizer, see `RegularStepGradientDescent`.

Multimodal registration enables you to adjust the properties of the one plus one evolutionary optimizer. For more information about the properties of this optimizer, see `OnePlusOneEvolutionary`.

Phase correlation enables you to choose to window the frequency-domain representation of the images. Windowing increases the stability of registration results. If the common features you are trying to align in your images are oriented along the edges, clearing this option can improve registration results. For more information about using phase correlation to transform an image, see `imregcorr`.

Nonrigid and Post-Processing Settings

Every registration technique in Registration Estimator app allows for nonrigid transformations to refine the registration fit locally. For more information about estimating a displacement field for nonrigid transformations, see `imregdemons`.

The nonrigid settings available in the Registration Estimator app are:

- Number of iterations. This value is the number of iterations on each pyramid level.
- Pyramid levels. The value represents the number of Gaussian pyramid reduction levels. The maximum number of pyramid levels depends on the size of each dimension in the images. For example, when the shortest dimension of the fixed and moving images is 256 pixels, at most eight pyramid levels can be used. For more information about pyramid reduction, see `impyramid`.
- Smoothing. The value represents the standard deviation of Gaussian smoothing and remains the same at each pyramid level. Values are in the range [0.5, 3]. Larger values result in smoother output displacement fields. Smaller values result in more localized deformation in the output displacement field.

Note Although isotropic scaling and shearing are nonrigid transformations from a mathematical perspective, these transformations act globally on an image. Enable scaling and shearing in the Registration Estimator app by selecting an affine or projective transformation type, not by applying a nonrigid transformation.

See Also

`imregcorr` | `imregdemons`

Related Examples

- “Register Images Using the Registration Estimator App” on page 7-9

More About

- “Techniques Supported by Registration Estimator App” on page 7-28
- “Matrix Representation of Geometric Transformations” on page 6-19

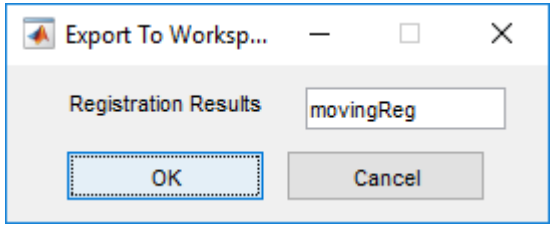
Export the Results from Registration Estimator App

When you find an acceptable registration, export the results. You can use the exported results to apply similar registration to other frames in an image sequence. There are two options to export the results:

- Export the registered image and the geometric transformation to the workspace. Apply an identical geometric transformation to other images using `imwarp`.
- Generate a function with the desired registration technique and settings. Call this function to register other images using the same settings.

Export Results to the Workspace

To export the registration results to the workspace, click the desired trial in the history list, then click **Export** and select **Export Images**. In the Export to Workspace dialog box, assign a name to the registration output. The output is a structure that contains the final registered image, the spatial referencing object, and the geometric transformation used for the registration.



Generate a Function

To generate MATLAB code that registers images using the desired registration technique and settings, click the corresponding trial in the history list, then click **Export**. Select the **Generate Function** option. The app opens the MATLAB editor containing a function with the autogenerated code. To save the code, click **Save** in the MATLAB editor.

Note If you generate a function using a feature-based registration technique, you must have the Computer Vision System Toolbox to run the function. You do not need the Computer Vision System Toolbox to run functions generated using intensity-based and nonrigid registration techniques.

The generated function accepts a moving and a fixed image as inputs. The function returns a structure that contains the final registered image, the spatial referencing object, and the geometric transformation of the registered image.

Note If you generate a function using a feature-based registration technique, the output structure contains two additional fields: the moving matched features and the fixed matched features.

See Also

`imwarp`

Related Examples

- “Register Images Using the Registration Estimator App” on page 7-9

More About

- “2-D and 3-D Geometric Transformation Process Overview” on page 6-14

Techniques Supported by Registration Estimator App

Feature-Based Registration

Feature-based registration techniques automatically detect distinct image features such as sharp corners, blobs, or regions of uniform intensity. The moving image undergoes a single global transformation to provide the best alignment of corresponding features with the fixed image.

FAST detects corner features, especially in scenes of human origin such as streets and indoor rooms. FAST supports single-scale images and point-tracking.

MinEigen also detects corner features. MinEigen supports single-scale images and point-tracking.

Harris also detects corner features, using a more efficient algorithm than MinEigen. Harris supports single-scale images and point-tracking.

BRISK also detects corner features. Unlike the preceding algorithms, BRISK supports changes in scale and rotation, and point-tracking.

SURF detects blobs in images and supports changes in scale and rotation.

MSER detects regions of uniform intensity. MSER supports changes in scale and rotation, and is more robust to affine transformations than the other feature-based algorithms.

Note An automated feature-based workflow exists in the Computer Vision System Toolbox. This workflow includes feature detection, extraction, and matching, followed by transform estimation. For more information, see “Find Image Rotation and Scale Using Automated Feature Matching” on page 7-103.

Intensity-Based Registration

Intensity-based registration techniques correlate image intensity in the spatial or frequency domain. The moving image undergoes a single global transformation to maximize the correlation of its intensity with the intensity of the fixed image.

Monomodal intensity registers images with similar brightness and contrast that are captured on the same type of scanner or sensor. For example, use monomodal intensity to register MRI scans taken of similar subjects using the same imaging sequence.

Multimodal intensity registers images with different brightness and contrast. These images can come from two different types of devices, such as two camera models or two types of medical imaging systems (such as CT and MRI). These images can also come from a single device. For example, use multimodal intensity to register images taken with the same camera using different exposure settings, or to register MRI images acquired during a single session using different imaging sequences.

Phase correlation registers images in the frequency domain. Like multimodal intensity, phase correlation is invariant to image brightness. Phase correlation is more robust to noise than the other intensity-based registration techniques.

Note Phase correlation provides better results when the aspect ratio of each image is square.

Nonrigid Registration

Nonrigid registration applies nonglobal transformations to the moving image. Nonrigid transformations generate a displacement field, in which each pixel location in the fixed image is mapped to a corresponding location in the moving image. The moving image is then warped according to the displacement field and resampled using linear interpolation. For more information about estimating a displacement field for nonrigid transformations, see `imregdemons`.

See Also

Related Examples

- “Register Images Using the Registration Estimator App” on page 7-9

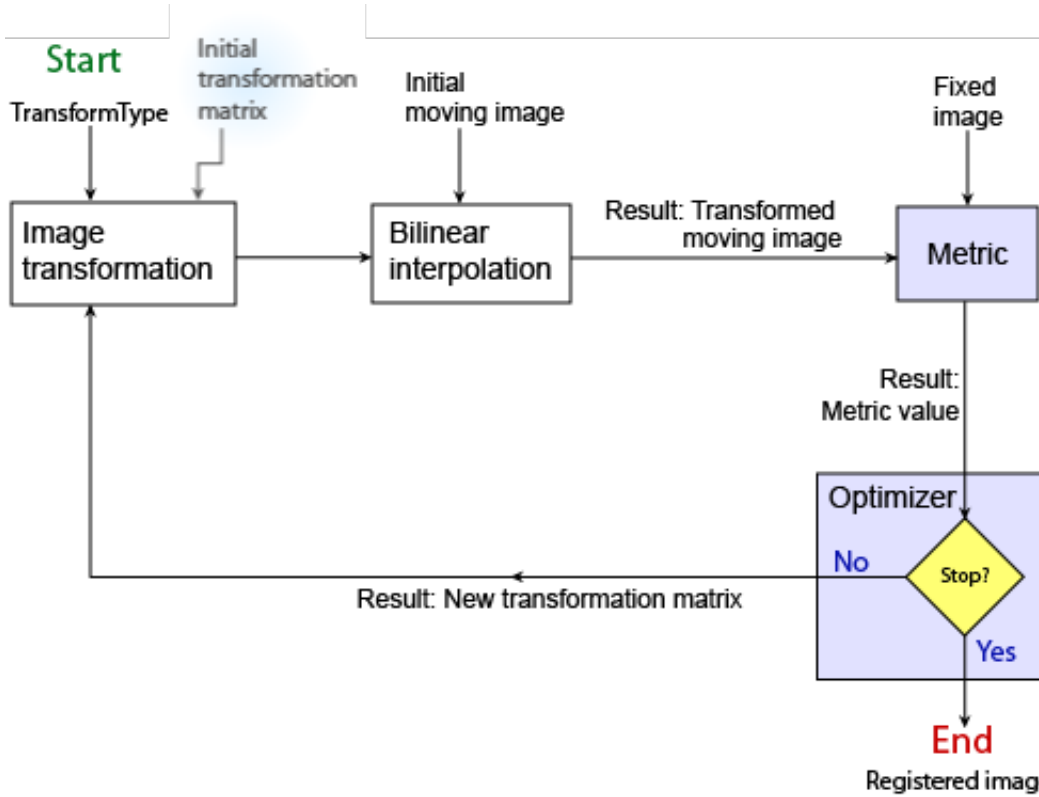
Intensity-Based Automatic Image Registration

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 aligns the misaligned image (called the *moving image*) 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 stops when it reaches a point of diminishing returns or when it reaches 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` or `dicomread`.
2. Create the optimizer and metric. See "Create an Optimizer and Metric for Intensity-Based Image Registration" on page 7-34.
3. Register the images with `imregister`.
4. View the results with `imshowpair` or save a copy of an image showing the results with `imfuse`.

See Also

Related Examples

- “Register Multimodal MRI Images” on page 7-42

More About

- “Use Phase Correlation as Preprocessing Step in Registration” on page 7-36
- “Approaches to Registering Images” on page 7-3

Create an Optimizer and Metric for Intensity-Based Image Registration

You can pass an image similarity metric and an optimizer technique to `imregister`. An image similarity metric takes two images and returns a scalar value that describes how similar the images are. The optimizer you pass to imregister defines the methodology for minimizing or maximizing the similarity metric.

`imregister` supports two similarity metrics:

- Mattes mutual information
- Mean squared error

In addition, `imregister` supports two techniques for optimizing the image metric:

- One-plus-one evolutionary
- Regular step gradient descent

You can pass any combination of metric and optimizer to `imregister`, but some pairs are better suited for some image classes. Refer to the table for help choosing an appropriate starting point.

Capture Scenario	Metric	Optimizer
Monomodal	`MeanSquares`	`RegularStepGradientDescent`
Multimodal	`MattesMutualInformation`	`OnePlusOneEvolutionary`

Use `imregconfig` to create the default metric and optimizer for a capture scenario in one step. For example, the following command returns the optimizer and metric objects suitable for registering monomodal images.

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

Alternatively, you can create the objects individually. This enables you to create alternative combinations to address specific registration issues. The following code creates the same monomodal optimizer and metric combination.

```
optimizer = registration.optimizer.RegularStepGradientDescent();
metric = registration.metric.MeanSquares();
```

Getting good results from optimization-based image registration can require modifying optimizer or metric settings. For an example of how to modify and use the metric and optimizer with `imregister`, see “Register Multimodal MRI Images” on page 7-42.

See Also

`imregconfig` | `imregister`

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 condition.

Read an image that will be the reference image in the registration.

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

Create an unregistered image by deliberately distorting this image using rotation, isotropic scaling, and shearing in the *y* direction.

```
theta = 170;
rot = [cosd(theta) sind(theta) 0;...
    sind(theta) cosd(theta) 0;...
    0 0 1];
sc = 2.3;
scale = [sc 0 0; 0 sc 0; 0 0 1];
sh = 0.5;
shear = [1 sh 0; 0 1 0; 0 0 1];

tform = affine2d(shear*scale*rot);
moving = imwarp(fixed,tform);
```

Add noise to the image, and display the result.

```
moving = moving + uint8(10*rand(size(moving)));
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 being aligned with the original image, `fixed`. But some misalignment remains. `imregcorr` can handle rotation and scale distortions well, but not shear distortion.

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

```
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 that `imregister` has achieved a very accurate registration, given the good initial condition provided by `imregcorr`.

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

See Also

`imregconfig` | `imregcorr` | `imregister` | `imwarp`

Register Multimodal MRI Images

This example shows how you can use `imregister` to automatically align two magnetic resonance (MRI) images 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 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.

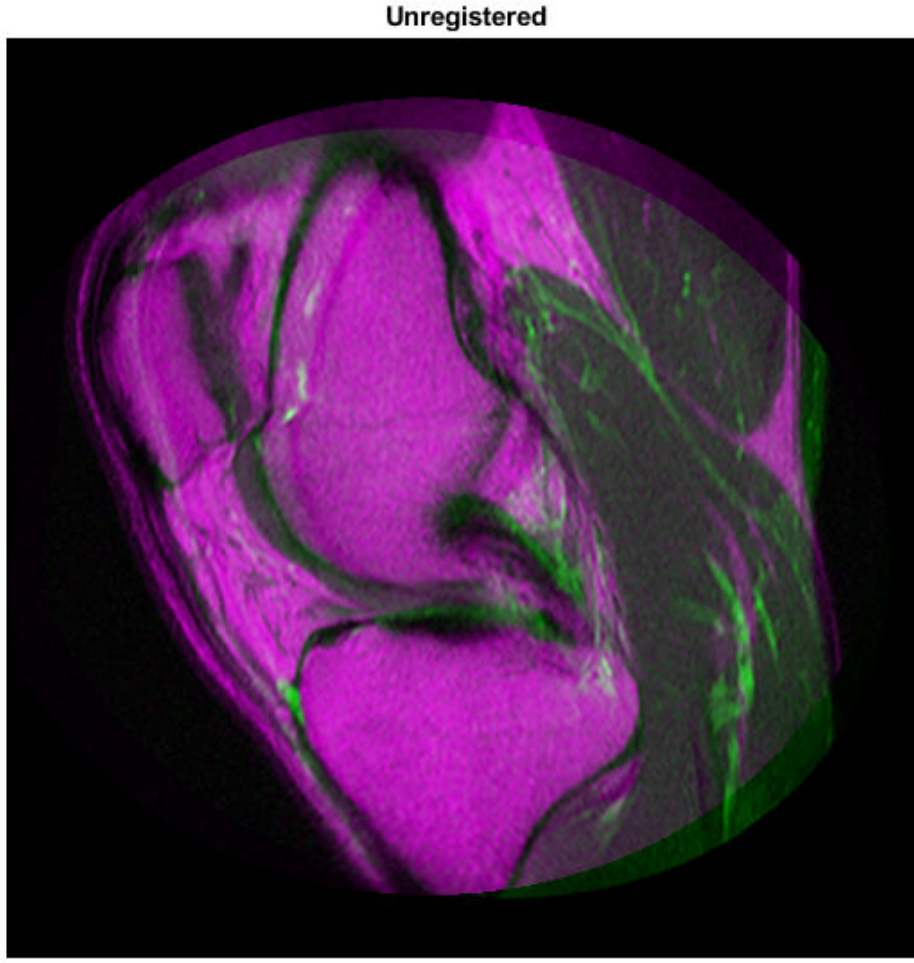
```
fixed = dicomread('knee1.dcm');
moving = dicomread('knee2.dcm');
```

The `imshowpair` function is useful to visualize 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.

```
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.

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



Step 2: Set up the Initial Registration

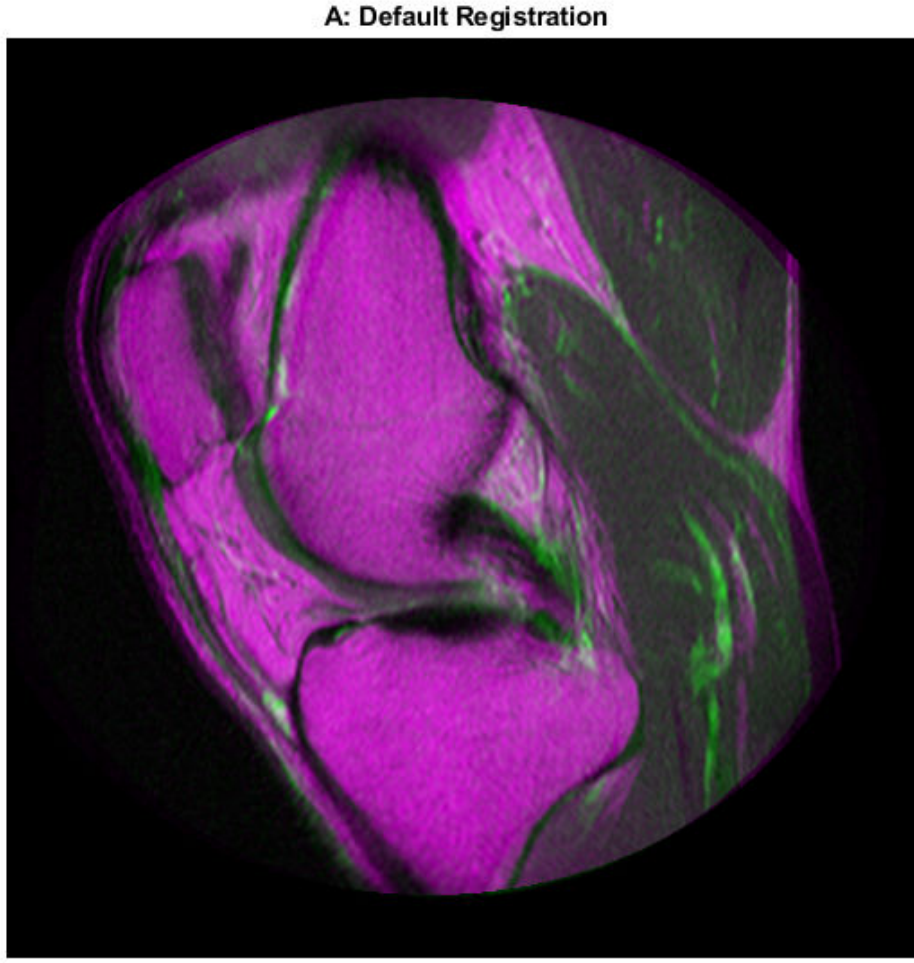
The `imregconfig` function makes it easy to pick the correct optimizer and metric configuration to use with `imregister`. These two images have different intensity distributions, which suggests a multimodal configuration.

```
[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);

imshowpair(movingRegisteredDefault,fixed)
title('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)
```

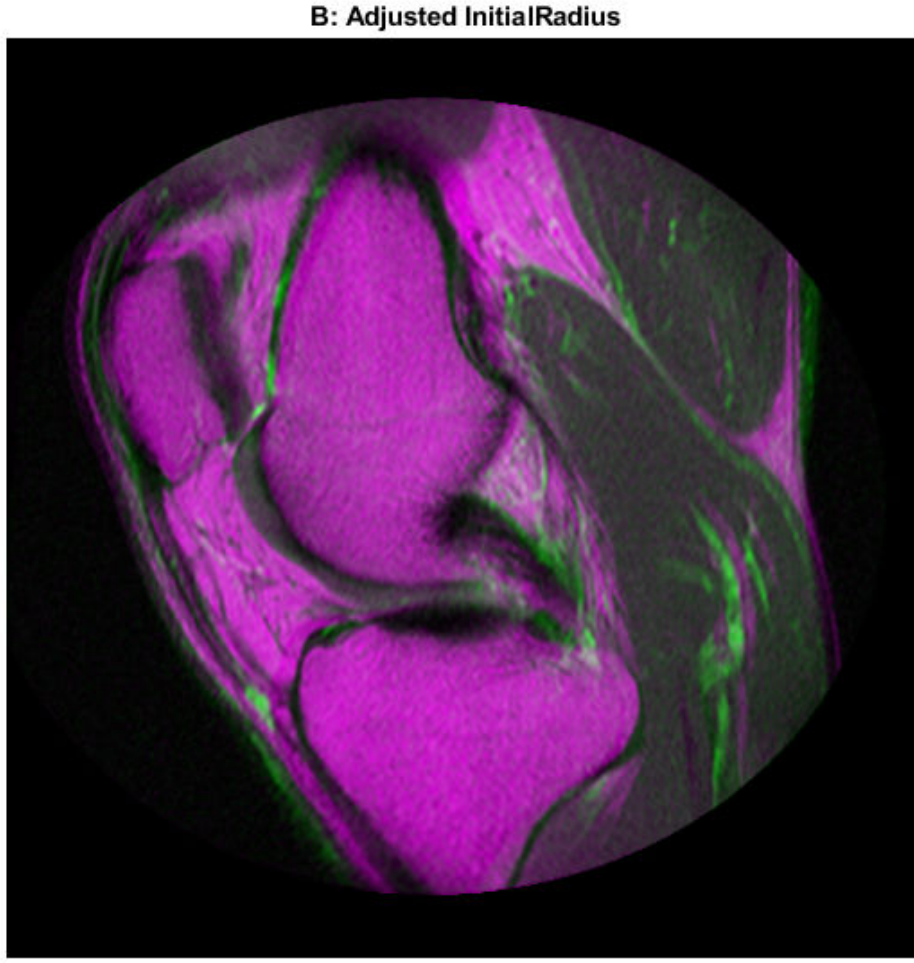
```
  registration.optimizer.OnePlusOneEvolutionary

  Properties:
         GrowthFactor: 1.050000e+00
              Epsilon: 1.500000e-06
        InitialRadius: 6.250000e-03
    MaximumIterations: 100
```

```
disp(metric)
```

```
  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, `InitialRadius` is a good first parameter to adjust. Start by reducing the default value of `InitialRadius` by a scale factor of 3.5.

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

movingRegisteredAdjustedInitialRadius = imregister(moving,fixed,'affine',optimizer,met

imshowpair(movingRegisteredAdjustedInitialRadius,fixed)
title('B: Adjusted InitialRadius')
```

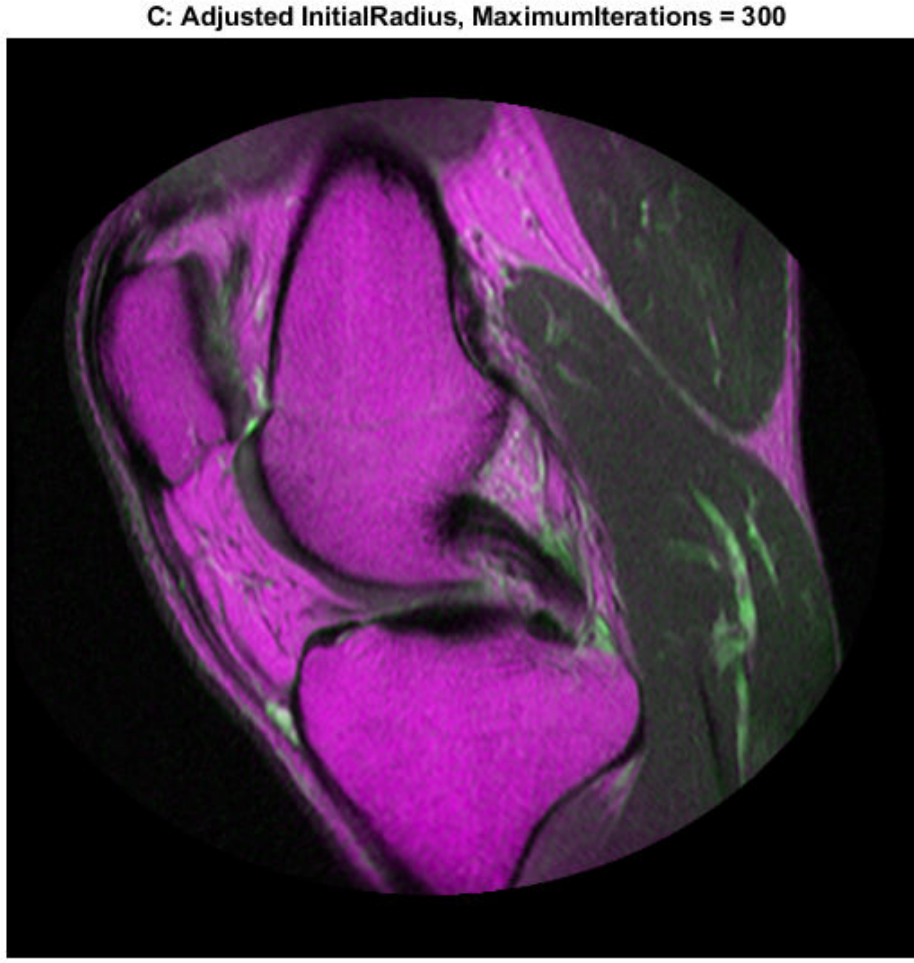


Adjusting `InitialRadius` had a positive impact. There is a noticeable improvement in the alignment of the images at the top and right edges.

The `MaximumIterations` property of the optimizer controls the maximum number of iterations that the optimizer will be allowed to take. Increasing `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,m

imshowpair(movingRegisteredAdjustedInitialRadius300,fixed)
title('C: Adjusted InitialRadius, MaximumIterations = 300')
```



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.

For more information about the multimodal optimizer and metric parameters that can be tuned, see the reference pages for the `OnePlusOneEvolutionary` optimizer and the `MattesMutualInformation` metric.

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 an 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.

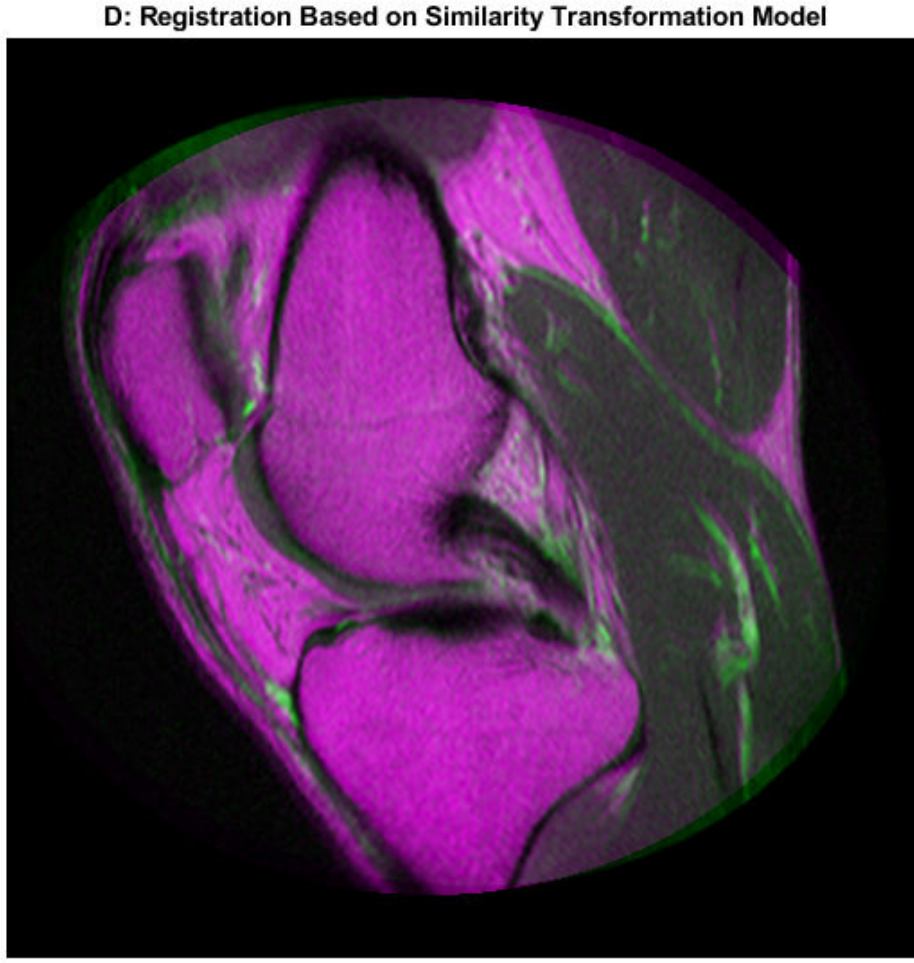
```
tformSimilarity = imregtform(moving,fixed,'similarity',optimizer,metric);
```

Because the registration is being solved in the default 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);

imshowpair(movingRegisteredRigid, fixed)
title('D: Registration Based on Similarity Transformation Model')
```



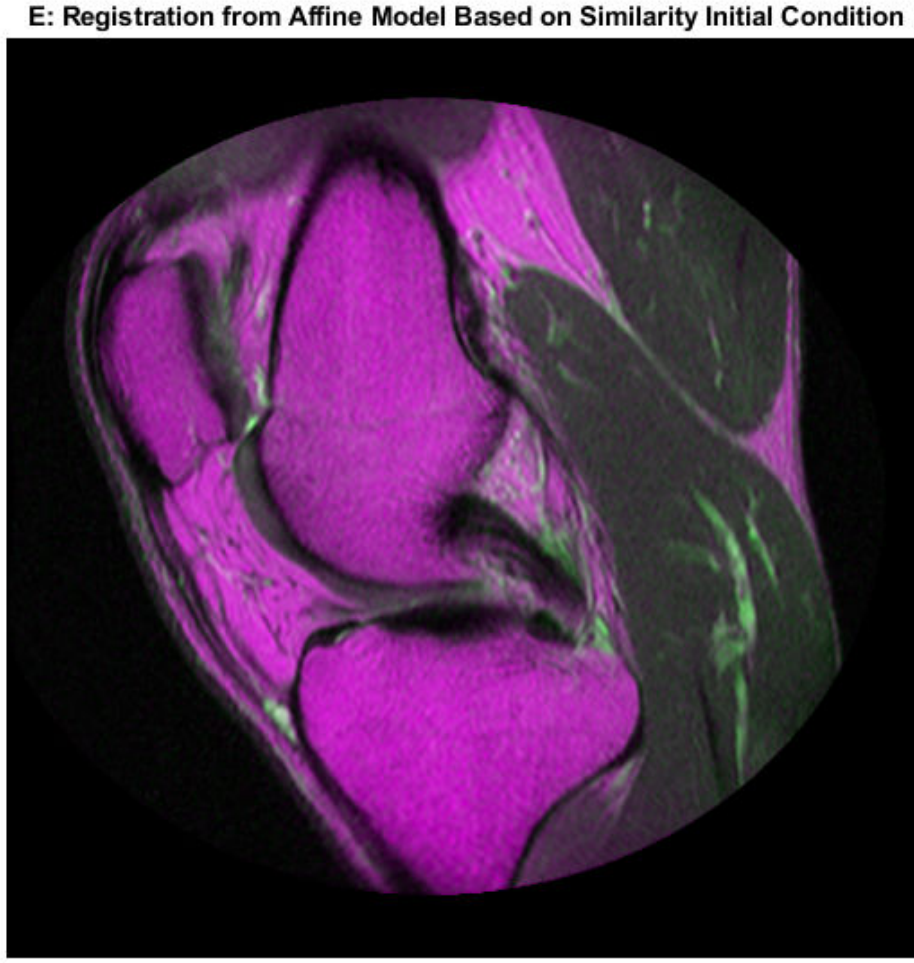
The `'T'` property of the output geometric transformation defines the transformation matrix that maps points in moving to corresponding points in fixed.

```
tformSimilarity.T
```

```
ans = 3×3

    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);

imshowpair(movingRegisteredAffineWithIC,fixed)
title('E: 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.

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 E) 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.)

See Also

`MattesMutualInformation` | `MeanSquares` | `OnePlusOneEvolutionary` | `RegularStepGradientDescent` | `imref2d` | `imregconfig` | `imregister` | `imwarp`

Register Multimodal 3-D Medical Images

This example shows how you can use `imregister`, `imregtform` and `imwarp` to automatically align two volumetric datasets: a CT image and a T1 weighted MR image collected from the same patient at different times. Unlike some other techniques, `imregister` and `imregtform` do not find features or use control points. Intensity-based registration is often well-suited for medical and remotely sensed imagery.

The 3-D CT and MRI datasets used in this example were provided by Dr. Michael Fitzpatrick as part of The Retrospective Image Registration Evaluation (RIRE) Dataset.

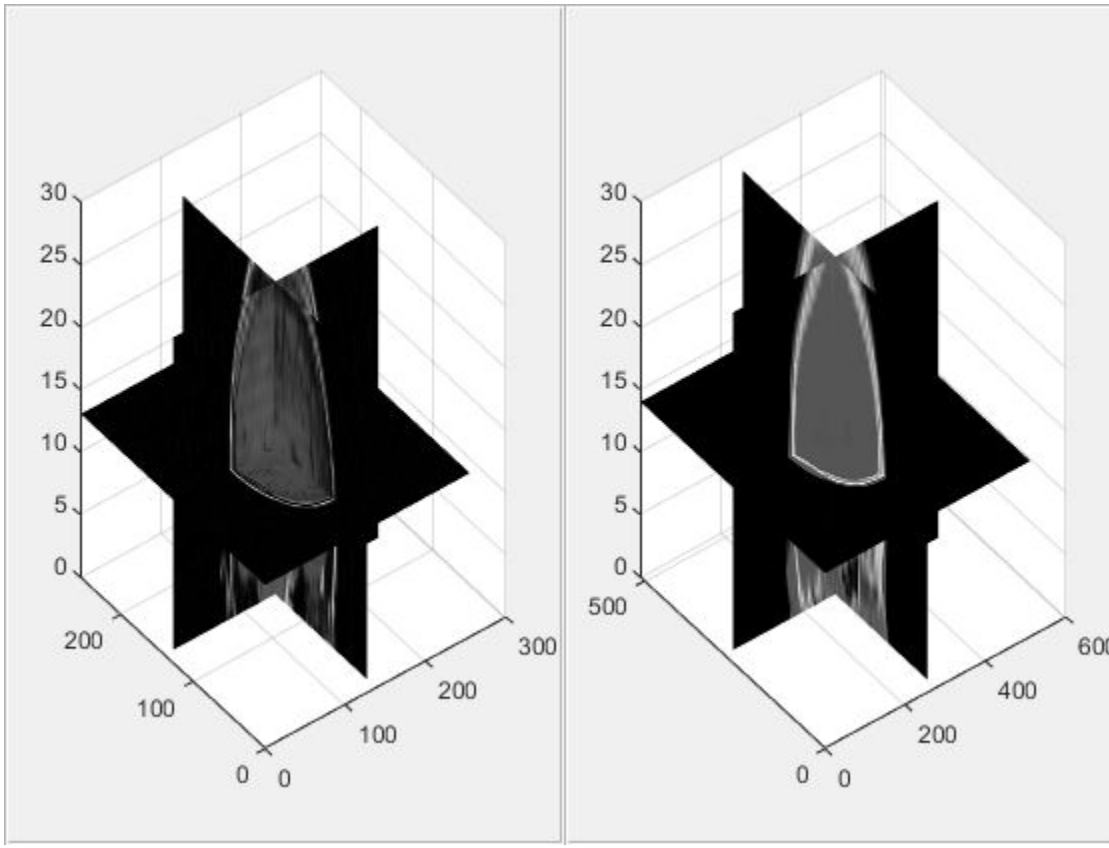
Step 1: Load Images

This example uses two 3-D images of the same patient's head. In registration problems, we consider one image to be the fixed image and the other image to be the moving image. The goal of registration is to align the moving image with the fixed image. In this example, the fixed image is a T1 weighted MRI image. The moving image that we want to register is a CT image. The data is stored in the file format used by the Retrospective Image Registration Evaluation (RIRE) Project. Use `multibandread` to read the binary files that contain image data. Use the `helperReadHeaderRIRE` function to obtain the metadata associated with each image. You can use the following link to find more information about the RIRE file format: RIRE data format

```
fixedHeader  = helperReadHeaderRIRE('rirePatient007MRT1.header');
movingHeader = helperReadHeaderRIRE('rirePatient007CT.header');

fixedVolume  = multibandread('rirePatient007MRT1.bin',...
                             [fixedHeader.Rows, fixedHeader.Columns, fixedHeader.Slices]
                             'int16=>single', 0, 'bsq', 'ieee-be' );

movingVolume = multibandread('rirePatient007CT.bin',...
                             [movingHeader.Rows, movingHeader.Columns, movingHeader.Slic
                             'int16=>single', 0, 'bsq', 'ieee-be' );
```

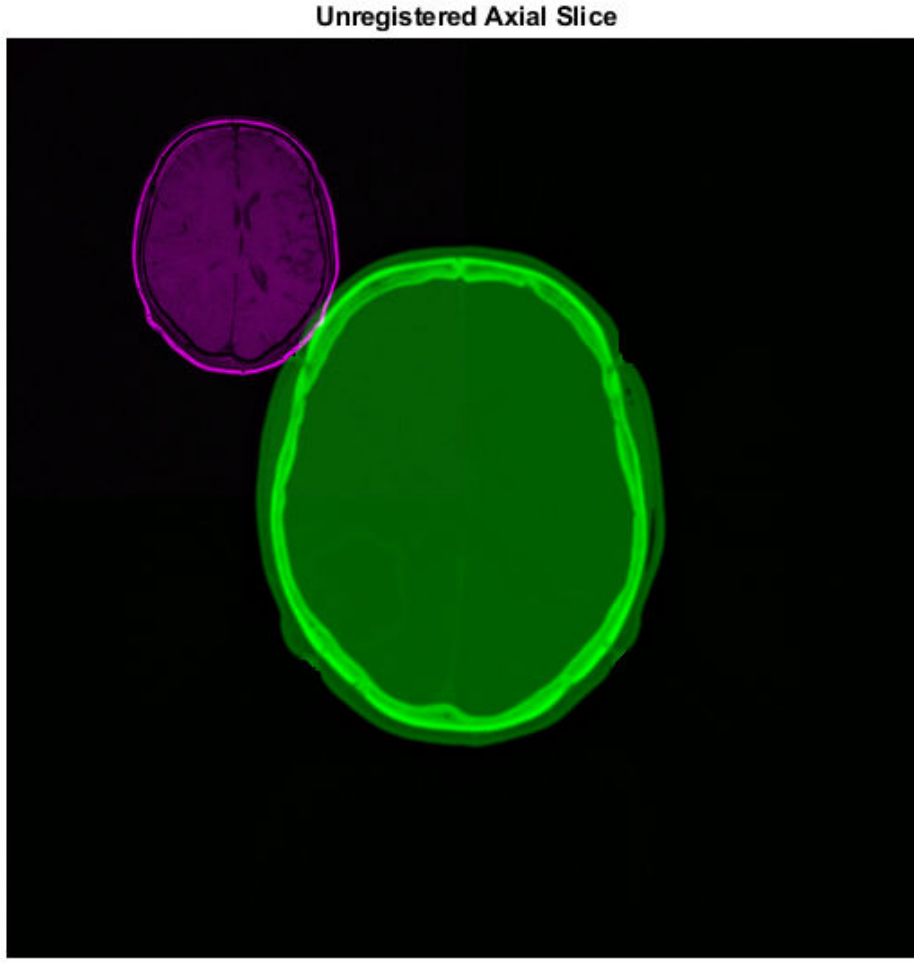
The `helperVolumeRegistration` function is a helper function that is provided to help judge the quality of 3-D registration results. You can interactively rotate the view and both axes will remain in sync.

```
helperVolumeRegistration(fixedVolume,movingVolume);
```



You can also use `imshowpair` to look at single planes from the fixed and moving volumes to get a sense of the overall alignment of the volumes. 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. Use `imshowpair` to observe the misregistration of the image volumes along an axial slice taken through the center of each volume.

```
centerFixed = size(fixedVolume)/2;
centerMoving = size(movingVolume)/2;
figure
imshowpair(movingVolume(:,:,centerMoving(3)), fixedVolume(:,:,centerFixed(3)));
title('Unregistered Axial Slice')
```



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 are from two different modalities, MRI and CT, so the 'multimodal' option is appropriate.

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

The algorithm used by `imregister` will converge to better results more quickly when spatial referencing information about the resolution and/or location of the input imagery is specified. In this case, the resolution of the CT and MRI datasets is defined in the image metadata. Use this metadata to construct `imref3d` spatial referencing objects that we will pass as input arguments for registration.

```
Rfixed  = imref3d(size(fixedVolume),fixedHeader.PixelSize(2),fixedHeader.PixelSize(1),
Rmoving = imref3d(size(movingVolume),movingHeader.PixelSize(2),movingHeader.PixelSize(
```

The properties of the spatial referencing objects define where the associated image volumes are in the world coordinate system and what the pixel extent in each dimension is. The XWorldLimits property of Rmoving defines the position of the moving volume in the X dimension. The PixelExtentInWorld property defines the size of each pixel in world units in the X dimension (along columns). The moving volume extends from 0.3269 to 334.97 in the world X coordinate system and each pixel has an extent of 0.6536mm. Units are in millimeters because the header information used to construct the spatial referencing was in millimeters. The ImageExtentInWorldX property determines the full extent in world units of the moving image volume in world units.

```
Rmoving.XWorldLimits
```

```
ans = 1×2

    0.3268  334.9674
```

```
Rmoving.PixelExtentInWorldX
```

```
ans = 0.6536
```

```
Rmoving.ImageExtentInWorldX
```

```
ans = 334.6406
```

The misalignment between the two volumes includes translation, scaling, and rotation. Use a similarity transformation to register the images.

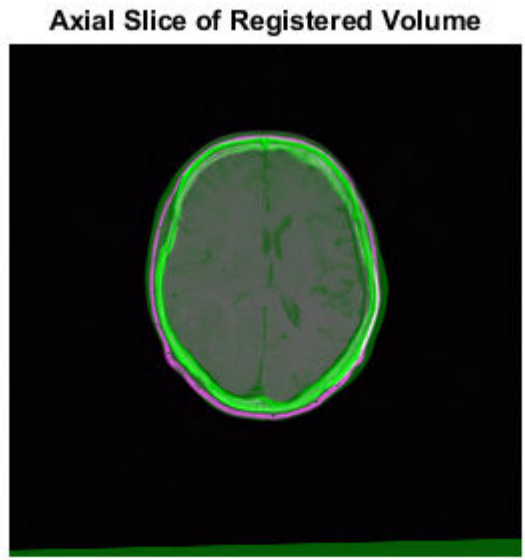
Start by using `imregister` to obtain a registered output image volume that you can view and observe directly to access the quality of registration results.

Specify a non-default setting for the InitialRadius property of the optimizer to achieve better convergence in registration results.

```
optimizer.InitialRadius = 0.004;
movingRegisteredVolume = imregister(movingVolume,Rmoving, fixedVolume,Rfixed, 'rigid',
```

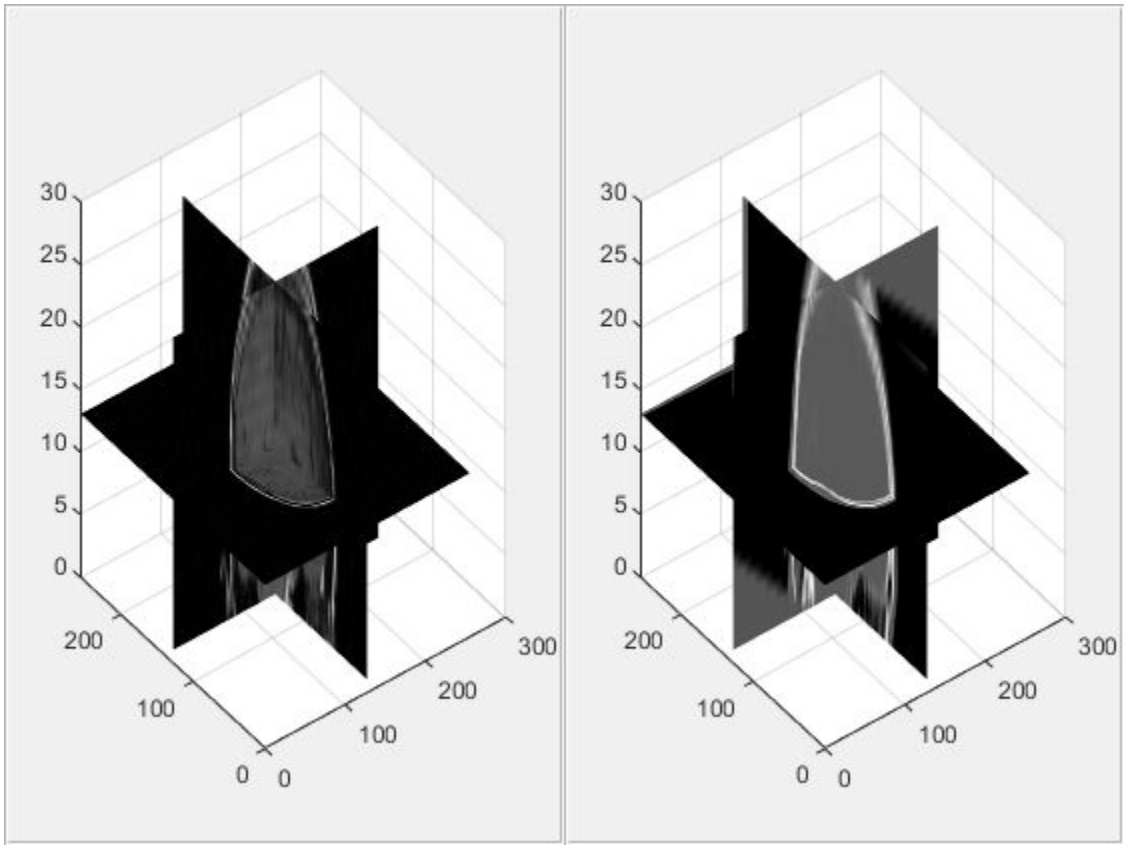
Use `imshowpair` again and repeat the process of examining the alignment of an axial slice taken through the center of the registered volumes to get a sense of how successful the registration is.

```
figure
imshowpair(movingRegisteredVolume(:,:,centerFixed(3)), fixedVolume(:,:,centerFixed(3)))
title('Axial Slice of Registered Volume')
```



From the axial slice above, it looks like the registration was successful. Use `helperVolumeRegistration` again to view the registered volume to continue judging the success of registration.

```
helperVolumeRegistration(fixedVolume,movingRegisteredVolume);
```



Step 3: Get 3-D Geometric Transformation That Aligns Moving With Fixed.

The `imregtform` function can be used when you are interested in the geometric transformation estimate that is used by `imregister` to form the registered output image. `imregtform` uses the same algorithm as `imregister` and takes the same input arguments as `imregister`. Since visual inspection of the resulting volume from `imregister` indicated that the registration was successful, you can call `imregtform` with the same input arguments to get the geometric transformation associated with this registration result.

```
geomtform = imregtform(movingVolume,Rmoving, fixedVolume,Rfixed, 'rigid', optimizer, me
```

```
geomtform =
  affine3d with properties:
```

```
                T: [4x4 double]
    Dimensionality: 3
```

The result of imregtform is a geometric transformation object. This object includes a property, T, that defines the 3-D affine transformation matrix.

```
geomtform.T
```

```
ans = 4×4

    0.9704   -0.0143   -0.2410         0
    0.0228    0.9992    0.0324         0
    0.2404   -0.0369    0.9700         0
  -15.8648  -17.5692   29.1830    1.0000
```

The transformPointsForward method of the geometric transformation can be used to determine where a point [u,v,w] in the moving image maps as a result of the registration. Because spatially referenced inputs were specified to imregtform, the geometric transformation maps points in the world coordinate system from moving to fixed. The transformPointsForward method is used below to determine the transformed location of the center of the moving image in the world coordinate system.

```
centerXWorld = mean(Rmoving.XWorldLimits);
centerYWorld = mean(Rmoving.YWorldLimits);
centerZWorld = mean(Rmoving.ZWorldLimits);
[xWorld,yWorld,zWorld] = transformPointsForward(geomtform,centerXWorld,centerYWorld,cer
```

You can use the worldToSubscript method of Rfixed to determine the element of the fixed volume that aligns with the center of the moving volume.

```
[r,c,p] = worldToSubscript(Rfixed,xWorld,yWorld,zWorld)
```

```
r = 116

c = 132

p = 13
```

Step 4: Apply Geometric Transformation Estimate To Moving Image Volume.

The `imwarp` function can be used to apply the geometric transformation estimate from imregtform to a 3-D volume. The 'OutputView' Name/Value is used to define a spatial

referencing argument that determines the world limits and resolution of the output resampled image. You can produce the same results given by `imregister` by using the spatial referencing object associated with the fixed image. This creates an output volume in which the world limits and resolution of the fixed and moving image are the same. Once the world limits and resolution of both volumes are the same, there is pixel to pixel correspondence between each sample of the moving and fixed volumes.

```
movingRegisteredVolume = imwarp(movingVolume,Rmoving,geomtform,'bicubic','OutputView',R
```

Use imshowpair again to view an axial slice through the center of the registered volume produced by `imwarp`.

```
figure
imshowpair(movingRegisteredVolume(:,:,centerFixed(3)), fixedVolume(:,:,centerFixed(3)))
title('Axial Slice of Registered Volume')
```

See Also

imref3d | imregconfig | imregister | imregtform | imwarp

Registering an Image Using Normalized Cross-Correlation

This example shows how to find a template image within a larger image. Sometimes one image is a subset of another. Normalized cross-correlation can be used to determine how to register or align the images by translating one of them.

Step 1: Read Image

```
onion = imread('onion.png');
peppers = imread('peppers.png');

imshow(onion)
```

```
figure, imshow(peppers)
```

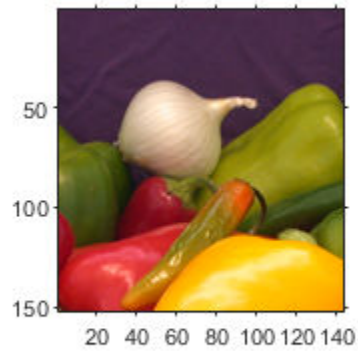
Step 2: Choose Subregions of Each Image

It is important to choose regions that are similar. The image `sub_onion` will be the template, and must be smaller than the image `sub_peppers`. You can get these sub regions using either the non-interactive script below **or** the interactive script.

```
% non-interactively
rect_onion = [111 33 65 58];
rect_peppers = [163 47 143 151];
sub_onion = imcrop(onion,rect_onion);
sub_peppers = imcrop(peppers,rect_peppers);

% OR
```

```
% interactively
%[sub_onion,rect_onion] = imcrop(onion); % choose the pepper below the onion
%[sub_peppers,rect_peppers] = imcrop(peppers); % choose the whole onion

% display sub images
figure, imshow(sub_onion)
```

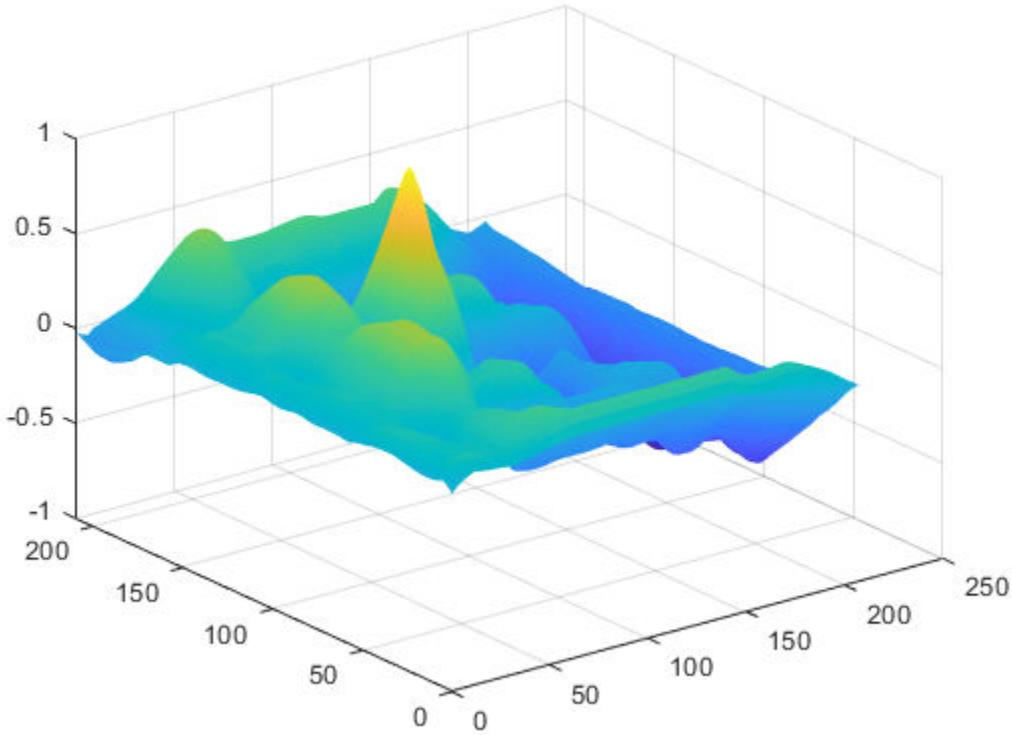
```
figure, imshow(sub_peppers)
```



Step 3: Do Normalized Cross-Correlation and Find Coordinates of Peak

Calculate the normalized cross-correlation and display it as a surface plot. The peak of the cross-correlation matrix occurs where the sub_images are best correlated. `normxcorr2` only works on grayscale images, so we pass it the red plane of each sub image.

```
c = normxcorr2(sub_onion(:,:,1),sub_peppers(:,:,1));
figure, surf(c), shading flat
```



Step 4: Find the Total Offset Between the Images

The total offset or translation between images depends on the location of the peak in the cross-correlation matrix, and on the size and position of the sub images.

```
% offset found by correlation
[max_c, imax] = max(abs(c(:)));
[ypeak, xpeak] = ind2sub(size(c),imax(1));
corr_offset = [(xpeak-size(sub_onion,2))
               (ypeak-size(sub_onion,1))];
```

```
% relative offset of position of subimages
rect_offset = [(rect_peppers(1)-rect_onion(1))
               (rect_peppers(2)-rect_onion(2))];

% total offset
offset = corr_offset + rect_offset;
xoffset = offset(1);
yoffset = offset(2);
```

Step 5: See if the Onion Image was Extracted from the Peppers Image

Figure out where `onion` falls inside of `peppers`.

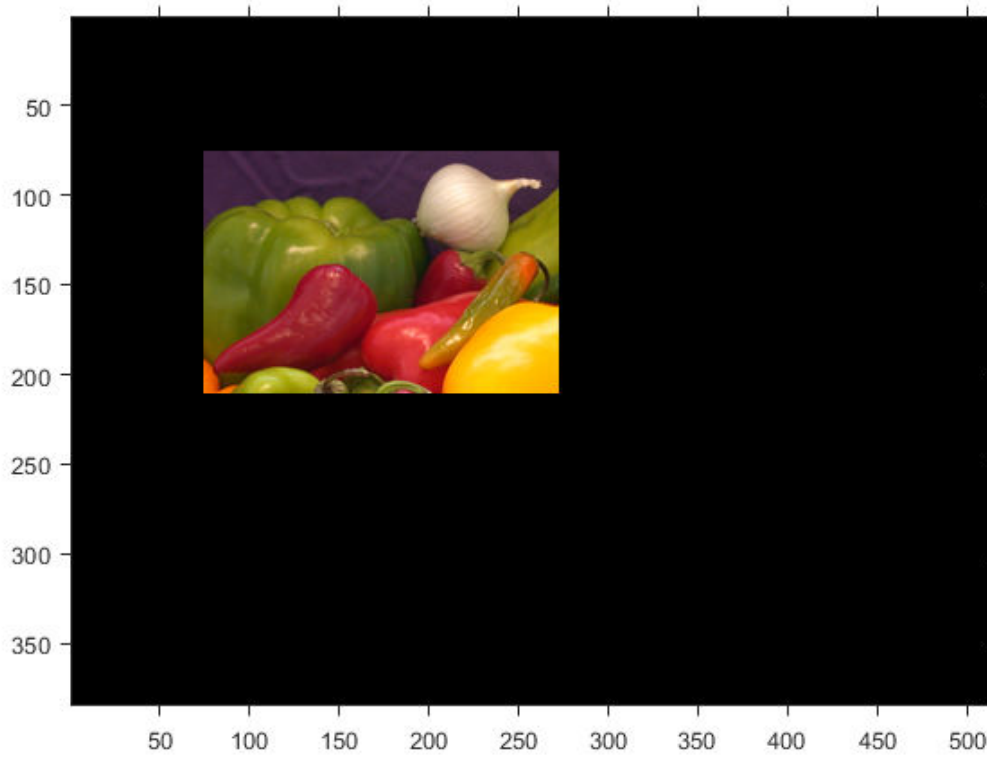
```
xbegin = round(xoffset+1);
xend   = round(xoffset+ size(onion,2));
ybegin = round(yoffset+1);
yend   = round(yoffset+size(onion,1));

% extract region from peppers and compare to onion
extracted_onion = peppers(ybegin:yend,xbegin:xend,:);
if isequal(onion,extracted_onion)
   disp('onion.png was extracted from peppers.png')
end
```

```
onion.png was extracted from peppers.png
```

Step 6: Pad the Onion Image to the Size of the Peppers Image

Pad the `onion` image to overlay on `peppers`, using the offset determined above.

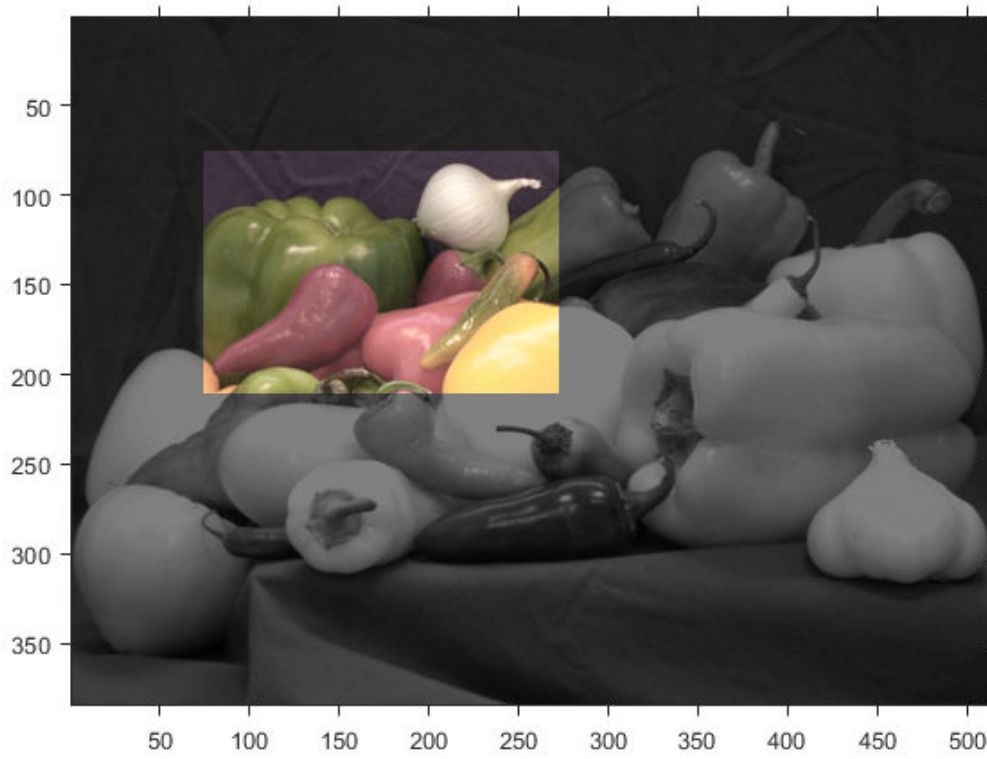
```
recovered_onion = uint8(zeros(size(peppers)));
recovered_onion(ybegin:yend,xbegin:xend,:) = onion;
figure, imshow(recovered_onion)
```



Step 7: Use Alpha Blending To Show Images Together

Display one plane of the `peppers` image with the `recovered_onion` image using alpha blending.

```
figure, imshowpair(peppers(:,:,1),recovered_onion,'blend')
```

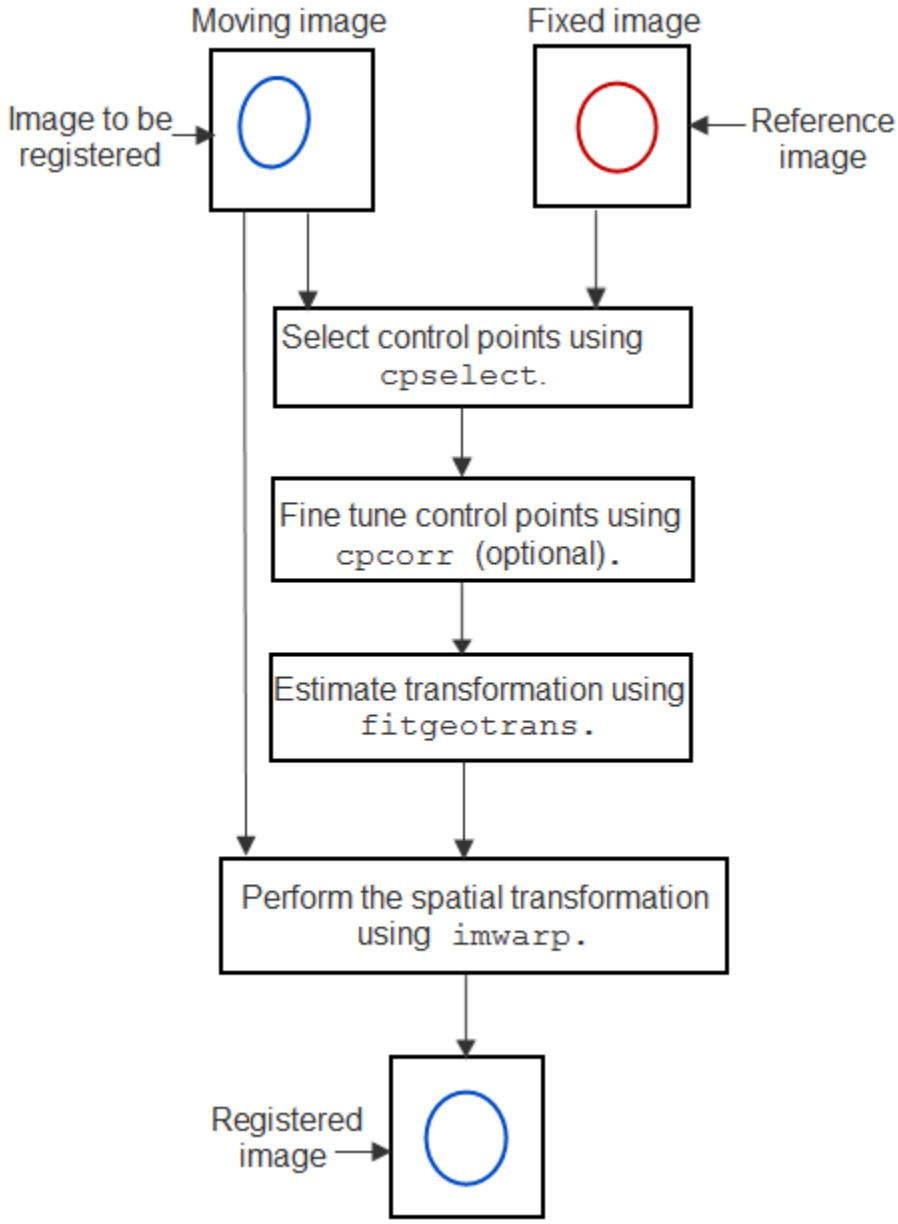

50
100
150
200
250
300
350
50
100
150
200
250
300
350
400
450
500

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 may need to perform several iterations of this process, experimenting with different types of transformations, before you achieve a satisfactory result. Sometimes, you can 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 7-111 for an extended example.


Moving image
Fixed image
Image to be registered
Reference image
Select control points using cpselect.
Fine tune control points using cpcorr (optional).
Estimate transformation using fitgeotrans.
Perform the spatial transformation using imwarp.
Registered image

See Also

`cpcorr` | `cpselect` | `fitgeotrans` | `imwarp`

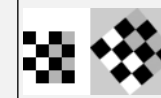
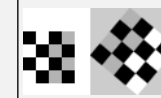
More About

- “Control Point Selection Procedure” on page 7-77
- “Approaches to Registering Images” on page 7-3

Geometric Transformation Types for Control Point Registration

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.

Transformation Type	Description	Minimum Number of Control Point Pairs	Example
`'nonreflective similarity'`	Use this transformation when shapes in the moving image are unchanged, but the image is distorted by some combination of translation, rotation, and scaling. Straight lines remain straight, and parallel lines are still parallel.	2	
`'similarity'`	Same as `'nonreflective similarity'` with the addition of optional reflection.	3	
`'affine'`	Use this transformation when shapes in the moving image exhibit shearing. Straight lines remain straight, and parallel lines remain parallel, but rectangles become parallelograms.	3	
`'projective'`	Use this transformation when the scene appears tilted. Straight lines remain straight, but parallel lines converge toward a vanishing point.	4	

Transformation Type	Description	Minimum Number of Control Point Pairs	Example
`'polynomial'`	Use this transformation when objects in the image are curved. The higher the order of the polynomial, the better the fit, but the result can contain more curves than the fixed image.	6 (order 2) 10 (order 3) 15 (order 4)	
`'piecewise linear'`	Use this transformation when parts of the image appear distorted differently.	4	
`'lwm'`	Use this transformation (local weighted mean), when the distortion varies locally and piecewise linear is not sufficient.	6 (12 recommended)	

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 must select. For example, a nonreflective similarity transformation requires at least two control point pairs. A fourth order polynomial transformation requires 15 control point pairs. For more information about these transformation types, and the special syntaxes they require, see `cpselect`.

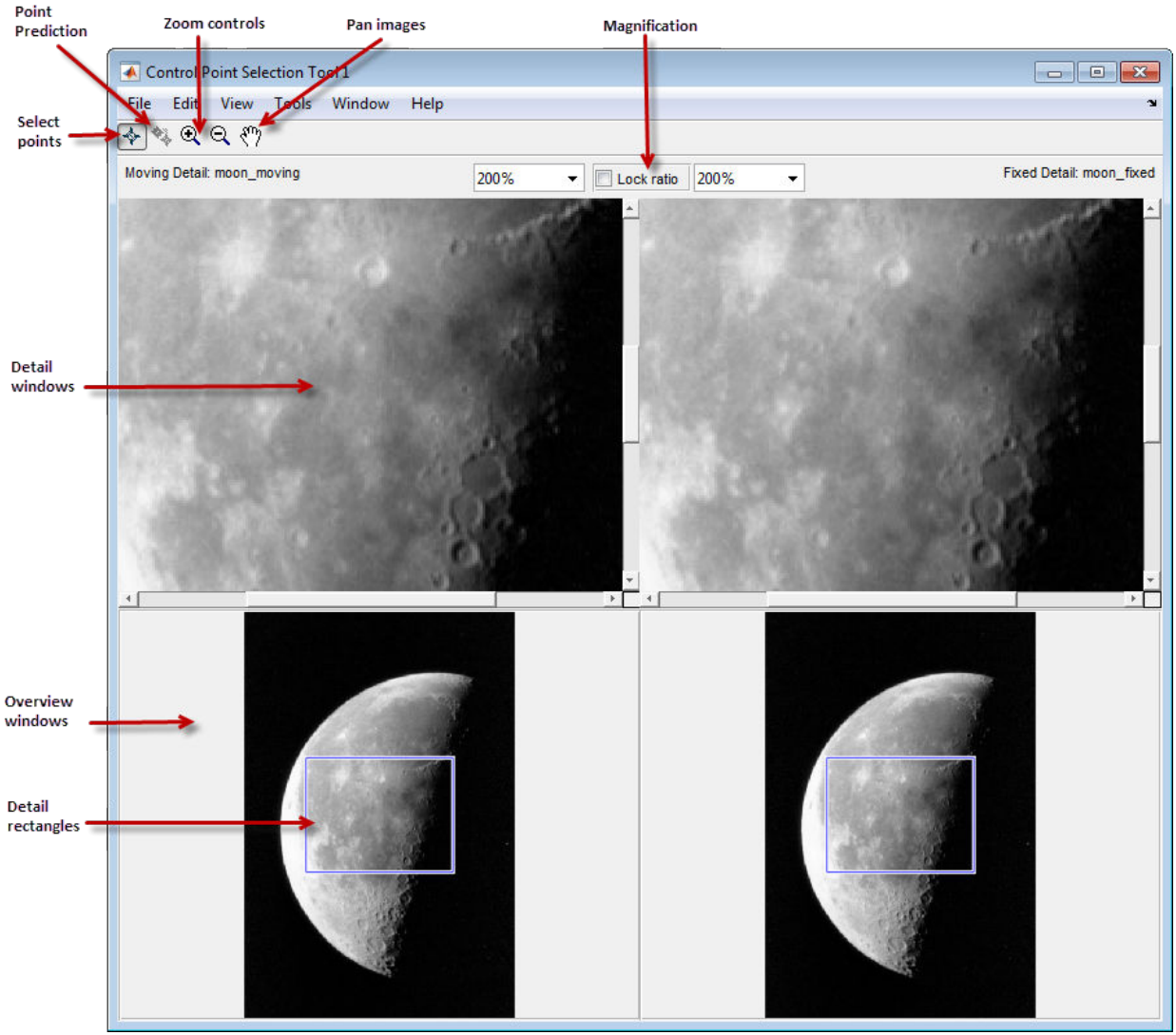
Control Point Selection Procedure

To specify control points in a pair of images interactively, use the Control Point Selection Tool, `cpselect`. The tool displays the image you want to register, called the *moving* image, next to the reference image, called the *fixed* image.

Specifying control points is a four-step process:

1. Start the tool on page 7-80, specifying the moving image and the fixed image.
2. Use navigation aids to explore the image on page 7-83, looking for visual elements that you can identify in both images. `cpselect` provides many ways to navigate around the image. You can pan and zoom to view areas of the image in more detail.
3. Specify matching control point pairs on page 7-88 in the moving image and the fixed image.
4. Save the control points on page 7-95 in the workspace.

The following figure shows the default appearance of the tool when you first start it.



See Also

`cpselect`

More About

- “Start the Control Point Selection Tool” on page 7-80
- “Control Point Registration” on page 7-72

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).

The `cpselect` command has other optional arguments. You can import existing control points, so that you can use the Control Point Selection Tool to modify, delete, or add to existing control points. 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 7-95.

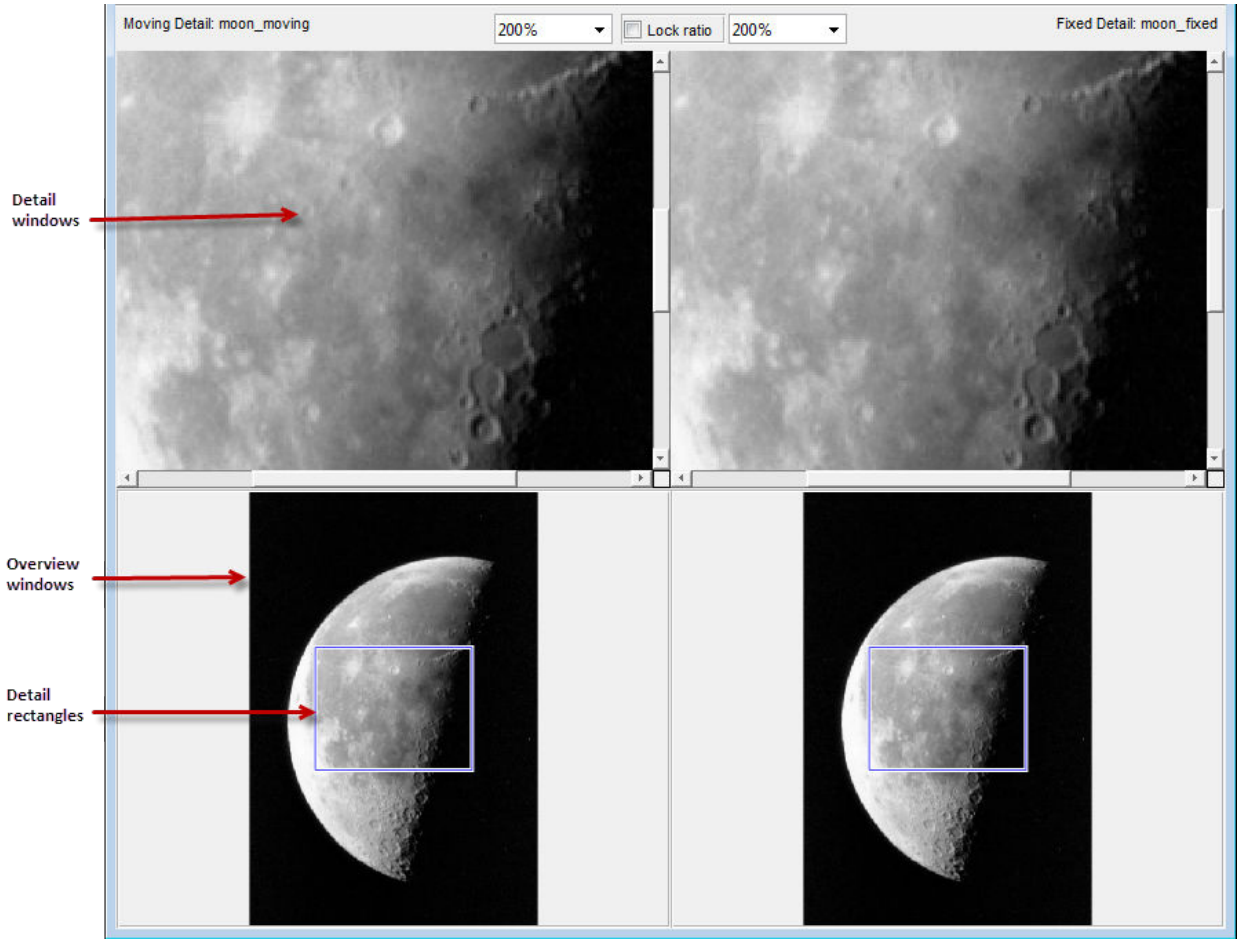
For simplicity, this example uses the same image as the moving and the fixed image, and no prior control points are imported. To walk through an example of an actual registration, see “Register an Aerial Photograph to a Digital Orthophoto” on page 7-111.

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

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.



The next step is to use navigation aids to explore the image, looking for visual elements shared by both images. For more information, see “Find Visual Elements Common to Both Images” on page 7-83.

See Also

`cpselect`

More About

- “Find Visual Elements Common to Both Images” on page 7-83
- “Export Control Points to the Workspace” on page 7-95
- “Control Point Selection Procedure” on page 7-77

Find Visual Elements Common to Both Images

To find visual elements that are common to both images, you can change the section of the image displayed in the Detail view. You can also 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 the Control Point Selection Tool:

In this section...
“Use Scroll Bars to View Other Parts of an Image” on page 7-83 “Use the Detail Rectangle to Change the View” on page 7-83 “Pan the Image Displayed in the Detail Window” on page 7-84 “Zoom In and Out on an Image” on page 7-84 “Specify the Magnification of the Images” on page 7-85 “Lock the Relative Magnification of the Moving and Fixed Images” on page 7-86

Use 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.

Use 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. The Control Point Selection Tool 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.

Pan 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.

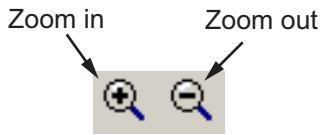
Note As you move the image in the Detail window, the Detail rectangle in the Overview window moves.

Zoom 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 “Specify the Magnification of the Images” on page 7-85 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, the Control Point Selection Tool changes the magnification of the image by a preset amount. (See “Specify the Magnification of the Images” on page 7-85 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. While pressing and holding the mouse button, draw a rectangle defining the area you want to zoom in on. The Control Point Selection Tool magnifies the image so that the chosen section fills the Detail window. The tool 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.

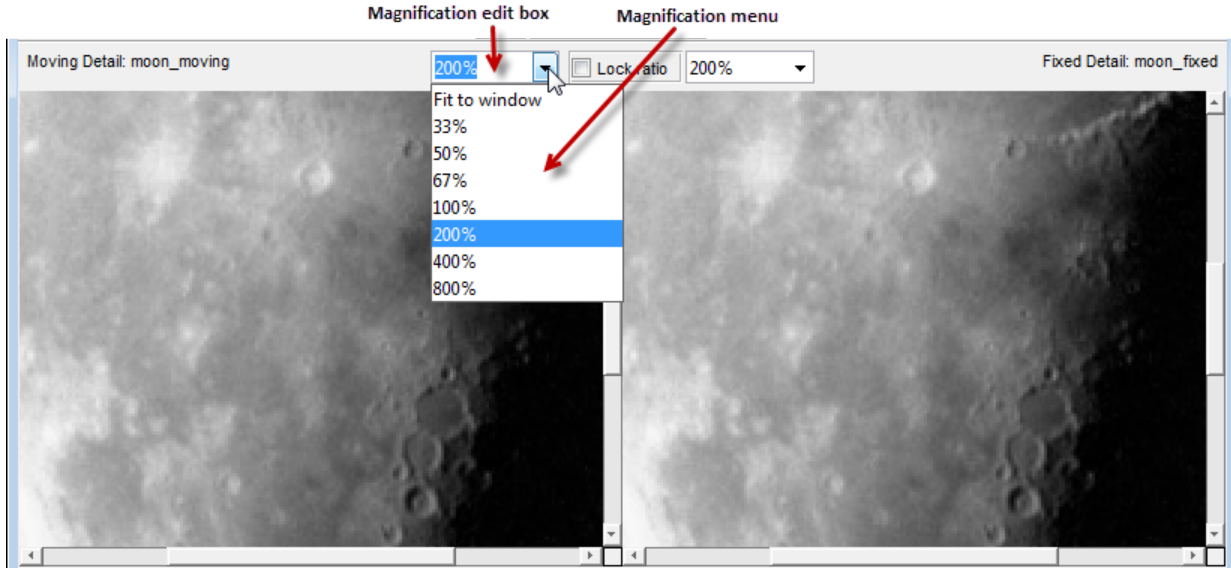
   To keep the relative magnifications of the fixed and moving images synchronized as you zoom in or out, click the **Lock ratio** check box. See “Lock the Relative Magnification of the Moving and Fixed Images” on page 7-86 for more information.

Specify 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 “Zoom In and Out on an Image” on page 7-84 for more information.)

To change the magnification of an image,

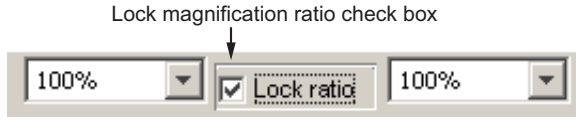
1. Move the cursor into the magnification edit box of the window you want to change. The cursor changes to the text entry cursor.
2. Type a new value in the magnification edit box and press **Enter**, or click the menu associated with the edit box and choose from a list of preset magnifications. The Control Point Selection Tool 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 “Lock the Relative Magnification of the Moving and Fixed Images” on page 7-86 for more information.



Lock 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, the Control Point Selection Tool changes the magnification of *both* the moving and fixed images when you zoom in or out on either one of the images on page 7-84 or specify a magnification value on page 7-85 for either of the images.



The next step is to specify matching control point pairs. For more information, see “Select Matching Control Point Pairs” on page 7-88.

See Also

More About

- “Start the Control Point Selection Tool” on page 7-80
- “Select Matching Control Point Pairs” on page 7-88
- “Control Point Selection Procedure” on page 7-77

Select Matching Control Point Pairs

The Control Point Selection Tool enables you to pick control points in the image to be registered (the moving image) and the reference image (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...
"Pick Control Point Pairs Manually" on page 7-88 "Use Control Point Prediction" on page 7-90 "Move Control Points" on page 7-93 "Delete Control Points" on page 7-94

Pick Control Point Pairs Manually

To specify a pair of control points in your images,

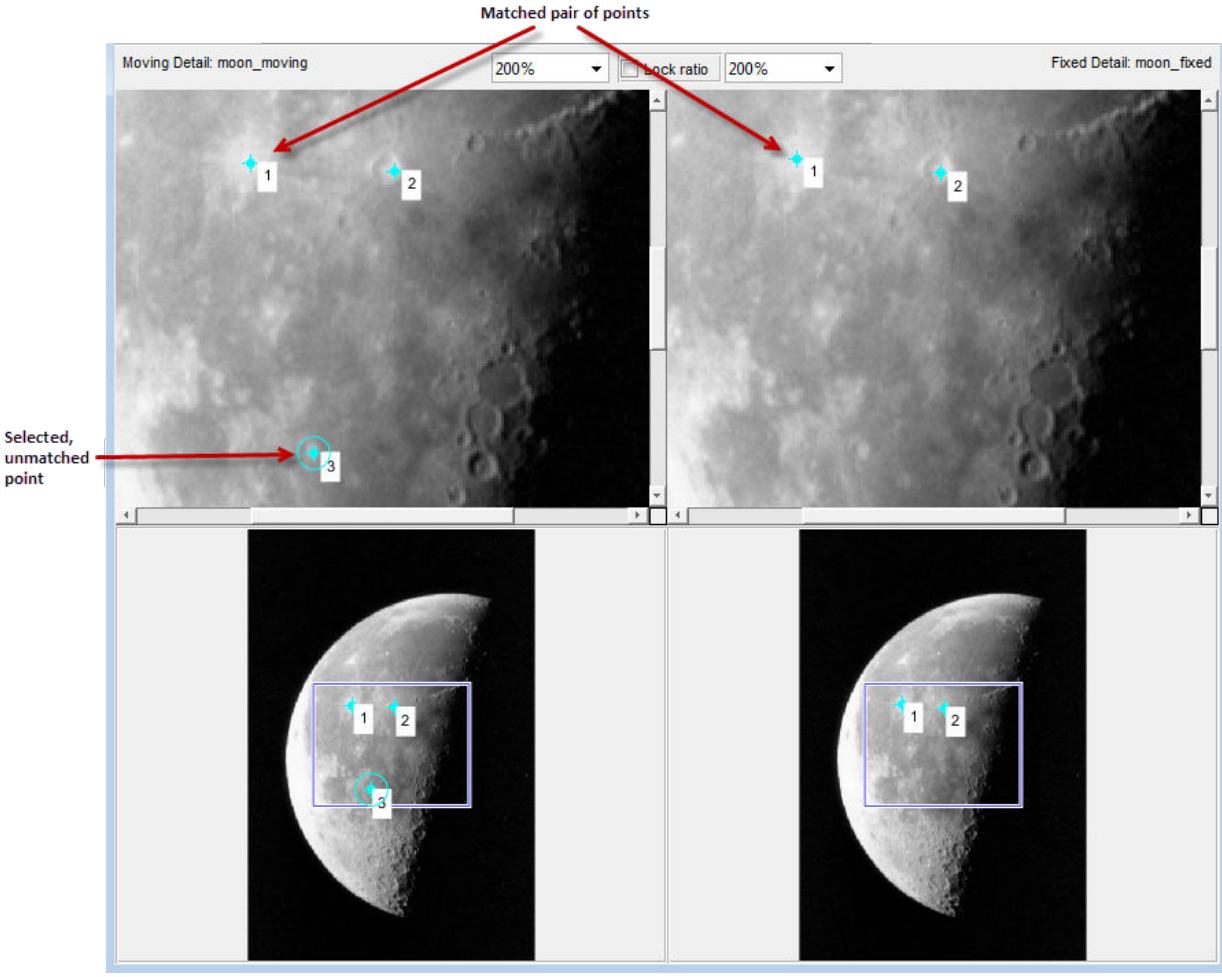
1. Click the **Control Point Selection** button in the Control Point Selection Tool toolbar or select **Add Points** from the Tools menu. Control point selection mode is active by default. The cursor changes to a cross-hairs shape
2. Position the cursor over a feature you have visually selected in any of the images displayed and click the mouse button. `cpselect` places a control point symbol, 1, 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 select control points, the symbol for the point may 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 on page 7-93 or delete on page 7-94 control points after you create them.

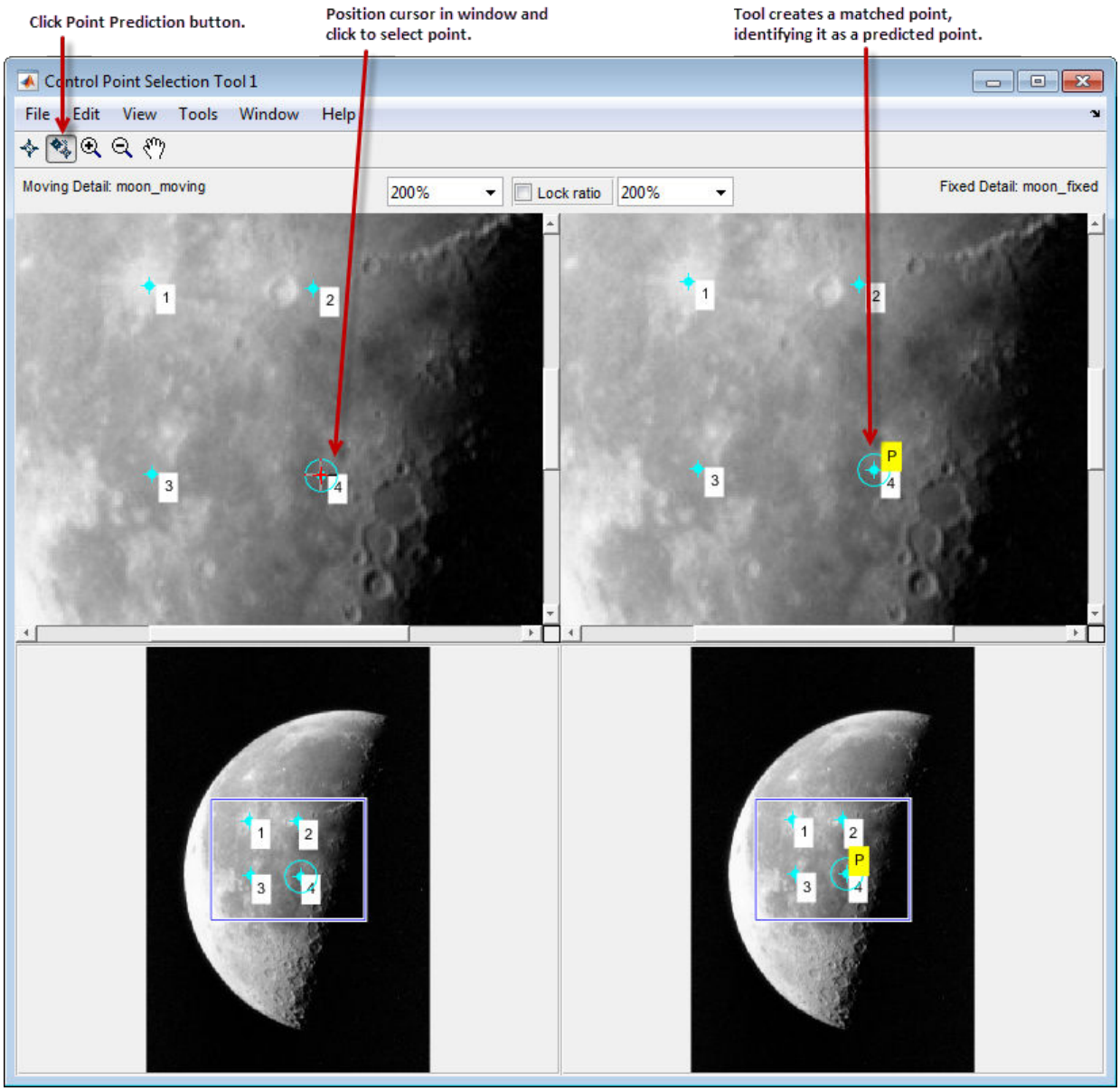
The following figure illustrates control points in several states.



Use 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.


Click Point Prediction button.
Position cursor in window and click to select point.
Tool creates a matched point, identifying it as a predicted point.
Control Point Selection Tool 1
File Edit View Tools Window Help
Moving Detail: moon_moving
200%
Lock ratio
200%
Fixed Detail: moon_fixed
1
2
3
4
P

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 “Move Control Points” on page 7-93.

To use control point prediction,

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

   Note The Control Point Selection Tool predicts control point locations based on the locations of 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 cross-hairs 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 is 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.

Move 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.

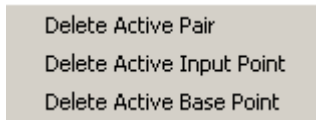
Delete Control Points

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.

See Also

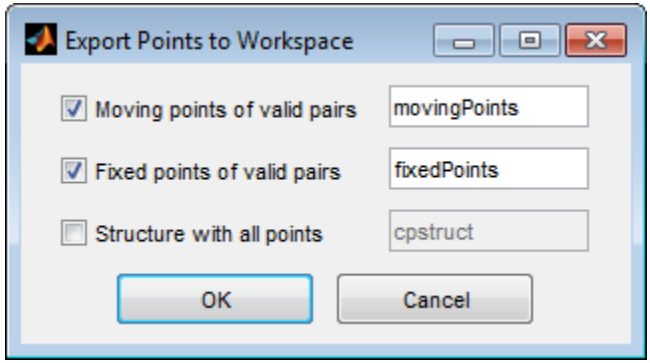
More About

- “Find Visual Elements Common to Both Images” on page 7-83
- “Export Control Points to the Workspace” on page 7-95
- “Control Point Selection Procedure” on page 7-77

Export Control Points to the Workspace

After you specify control points, you must save them in the workspace to make them available for the next step in image registration, processing by `fitgeotrans`.

To save control points to the workspace, select **File** on the Control Point Selection Tool menu bar, then choose the **Export Points to Workspace** option. The Control Point Selection Tool displays this dialog box:

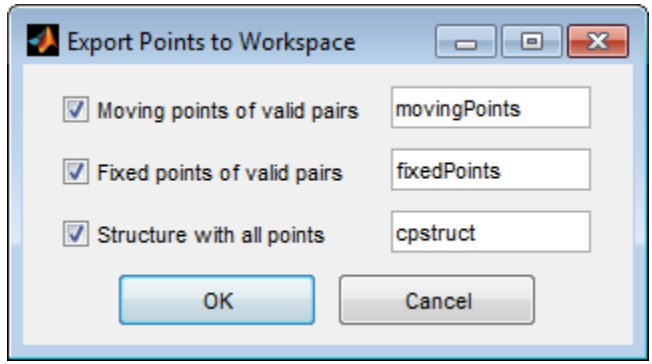


By default, the Control Point Selection Tool saves the coordinates of valid control points. The Control Point Selection Tool does not include unmatched and predicted points in the `movingPoints` and `fixedPoints` arrays. The arrays are *n*-by-2 arrays, where *n* is the number of valid control point pairs you selected. The values in the left column represent the *x*-coordinates and the values in the right column represent the *y*-coordinates.

This example shows the `movingPoints` array containing four valid pairs of control points.

```
movingPoints =

  215.6667  262.3333
  225.7778  311.3333
  156.5556  340.1111
  270.8889  368.8889
```

To save the current state of the Control Point Selection Tool, including unpaired and predicted control points, select the **Structure with all points** check box.



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

```
cpstruct =

         inputPoints: [4x2 double]
          basePoints: [4x2 double]
      inputBasePairs: [4x2 double]
                 ids: [4x1 double]
        inputIdPairs: [4x2 double]
         baseIdPairs: [4x2 double]
    isInputPredicted: [4x1 double]
     isBasePredicted: [4x1 double]
```

You can use the `cpstruct` to restart a control point selection session at the point where you left off. This option is useful if you are picking many points over a long time and want to preserve unmatched and predicted points when you resume work.

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

The Control Point Selection Tool also asks if you want to save your control points when you exit the tool.

See Also

`cpselect` | `cpstruct2pairs` | `fitgeotrans`

More About

- “Select Matching Control Point Pairs” on page 7-88
- “Control Point Selection Procedure” on page 7-77

Find Image Rotation and Scale

This example shows how to align or register two images that differ by a rotation and a scale change. You can use `fitgeotrans` to find the rotation angle and scale factor after manually picking corresponding points. You can then transform the distorted image to recover the original image.

Step 1: Read Image

Read an image into the workspace.

```
original = imread('cameraman.tif');
imshow(original);
text(size(original,2),size(original,1)+15, ...
    'Image courtesy of Massachusetts Institute of Technology', ...
    'FontSize',7,'HorizontalAlignment','right');
```

Step 2: Resize and Rotate the Image

```
scale = 0.7;
distorted = imresize(original,scale); % Try varying the scale factor.

theta = 30;
distorted = imrotate(distorted,theta); % Try varying the angle, theta.
figure, imshow(distorted)
```

Step 3: Select Control Points

Use the Control Point Selection Tool to pick at least two pairs of control points.

```
movingPoints = [151.52  164.79; 131.40 79.04];
fixedPoints = [135.26  200.15; 170.30 79.30];
```

You can run the rest of the example with these pre-picked points, but try picking your own points to see how the results vary.

```
cpselect(distorted,original,movingPoints,fixedPoints);
```

Save control points by choosing the **File** menu, then the **Save Points to Workspace** option. Save the points, overwriting variables `movingPoints` and `fixedPoints`.

Step 4: Estimate Transformation

Fit a nonreflective similarity transformation to your control points.

```
tform = fitgeotrans(movingPoints,fixedPoints,'nonreflectivesimilarity');
```

After you have done Steps 5 and 6, repeat Steps 4 through 6 but try using 'affine' instead of 'NonreflectiveSimilarity'. What happens? Are the results as good as they were with 'NonreflectiveSimilarity'?

Step 5: Solve for Scale and Angle

The geometric transformation, `tform`, contains a transformation matrix in `tform.T`. Since you know that the transformation includes only rotation and scaling, the math is relatively simple to recover the scale and angle.

```
Let sc = s*cos(theta)

Let ss = s*sin(theta)

Then, Tinv = invert(tform), and Tinv.T = [sc -ss 0;

ss sc 0;

tx ty 1]

where tx and ty are x and y translations, respectively.
```

```
tformInv = invert(tform);
Tinv = tformInv.T;
ss = Tinv(2,1);
sc = Tinv(1,1);
scale_recovered = sqrt(ss*ss + sc*sc)
```

```
scale_recovered = 0.7000
```

```
theta_recovered = atan2(ss,sc)*180/pi
```

```
theta_recovered = 29.3741
```

The recovered values of `scale_recovered` and `theta_recovered` should match the values you set in **Step 2: Resize and Rotate the Image**.

Step 6: Recover Original Image

Recover the original image by transforming `distorted`, the rotated-and-scaled image, using the geometric transformation `tform` and what you know about the spatial referencing of `original`. The 'OutputView' Name/Value pair is used to specify the resolution and grid size of the resampled output image.

```
Roriginal = imref2d(size(original));
recovered = imwarp(distorted,tform,'OutputView',Roriginal);
```

Compare `recovered` to `original` by looking at them side-by-side in a montage.

```
figure, imshowpair(original,recovered,'montage')
```

The `recovered` (right) image quality does not match the `original` (left) image because of the distortion and recovery process. In particular, the image shrinking causes loss of information. The artifacts around the edges are due to the limited accuracy of the

transformation. If you were to pick more points in **Step 3: Select Control Points**, the transformation would be more accurate.

See Also

`cpselect` | `fitgeotrans` | `imref2d` | `imresize` | `imrotate` | `imwarp`

More About

- “Select Matching Control Point Pairs” on page 7-88
- “Control Point Selection Procedure” on page 7-77

Find Image Rotation and Scale Using Automated Feature Matching

This example shows how to automatically align two images that differ by a rotation and a scale change. It closely parallels another example titled Find Image Rotation and Scale. Instead of using a manual approach to register the two images, it utilizes feature-based techniques found in the Computer Vision System Toolbox™ to automate the registration process.

In this example, you will use `detectSURFFeatures` and `vision.GeometricTransformEstimator` System object to recover rotation angle and scale factor of a distorted image. You will then transform the distorted image to recover the original image.

Step 1: Read Image

Bring an image into the workspace.

```
original = imread('cameraman.tif');
imshow(original);
text(size(original,2),size(original,1)+15, ...
    'Image courtesy of Massachusetts Institute of Technology', ...
    'FontSize',7,'HorizontalAlignment','right');
```

Image courtesy of Massachusetts Institute of Technology

Step 2: Resize and Rotate the Image

```
scale = 0.7;
J = imresize(original, scale); % Try varying the scale factor.

theta = 30;
distorted = imrotate(J,theta); % Try varying the angle, theta.
figure, imshow(distorted)
```

You can experiment by varying the scale and rotation of the input image. However, note that there is a limit to the amount you can vary the scale before the feature detector fails to find enough features.

Step 3: Find Matching Features Between Images

Detect features in both images.

```
ptsOriginal  = detectSURFFeatures(original);
ptsDistorted = detectSURFFeatures(distorted);
```

Extract feature descriptors.

```
[featuresOriginal,   validPtsOriginal]  = extractFeatures(original,  ptsOriginal);
[featuresDistorted, validPtsDistorted]  = extractFeatures(distorted, ptsDistorted);
```

Match features by using their descriptors.

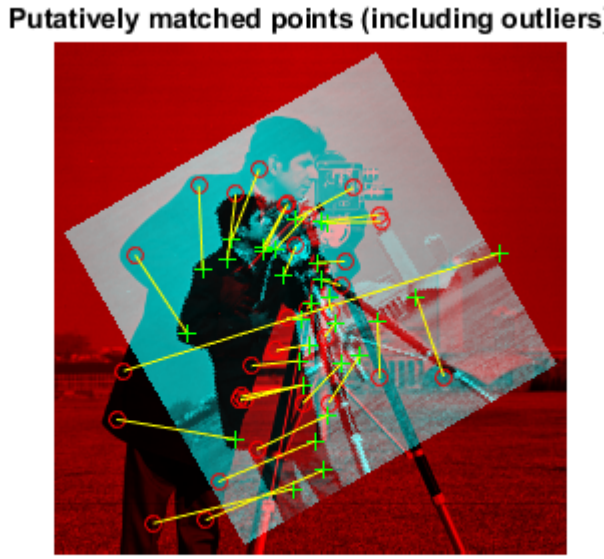
```
indexPairs = matchFeatures(featuresOriginal, featuresDistorted);
```

Retrieve locations of corresponding points for each image.

```
matchedOriginal  = validPtsOriginal(indexPairs(:,1));
matchedDistorted = validPtsDistorted(indexPairs(:,2));
```

Show point matches. Notice the presence of outliers.

```
figure;
showMatchedFeatures(original,distorted,matchedOriginal,matchedDistorted);
title('Putatively matched points (including outliers)');
```



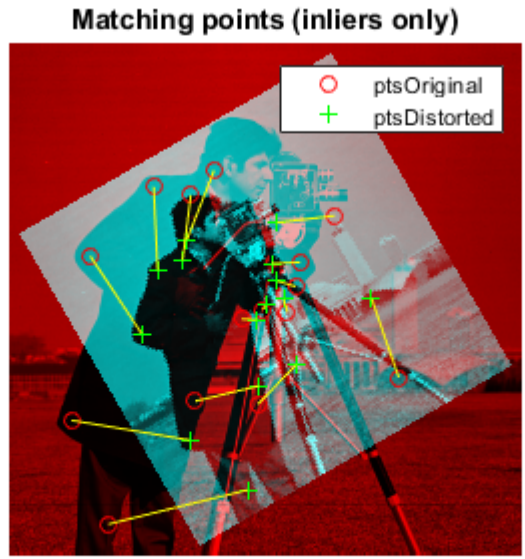
Step 4: Estimate Transformation

Find a transformation corresponding to the matching point pairs using the statistically robust M-estimator SAmple Consensus (MSAC) algorithm, which is a variant of the RANSAC algorithm. It removes outliers while computing the transformation matrix. You may see varying results of the transformation computation because of the random sampling employed by the MSAC algorithm.

```
[tform, inlierDistorted, inlierOriginal] = estimateGeometricTransform(...
    matchedDistorted, matchedOriginal, 'similarity');
```

Display matching point pairs used in the computation of the transformation matrix.

```
figure;
showMatchedFeatures(original,distorted, inlierOriginal, inlierDistorted);
title('Matching points (inliers only)');
legend('ptsOriginal','ptsDistorted');
```



Step 5: Solve for Scale and Angle

Use the geometric transform, TFORM, to recover the scale and angle. Since we computed the transformation from the distorted to the original image, we need to compute its inverse to recover the distortion.

```
Let sc = scale*cos(theta)
Let ss = scale*sin(theta)
```

```
Then, Tinv = [sc -ss  0;
              ss  sc  0;
              tx  ty  1]

where tx and ty are x and y translations, respectively.
```

Compute the inverse transformation matrix.

```
Tinv  = tform.invert.T;

ss = Tinv(2,1);
sc = Tinv(1,1);
scale_recovered = sqrt(ss*ss + sc*sc)
theta_recovered = atan2(ss,sc)*180/pi
```

```
scale_recovered =

  single

    0.7010


theta_recovered =

  single

   30.2351
```

The recovered values should match your scale and angle values selected in **Step 2: Resize and Rotate the Image**.

Step 6: Recover the Original Image

Recover the original image by transforming the distorted image.

```
outputView = imref2d(size(original));
recovered  = imwarp(distorted,tform,'OutputView',outputView);
```

Compare `recovered` to `original` by looking at them side-by-side in a montage.

```
figure, imshowpair(original,recovered,'montage')
```

The `recovered` (right) image quality does not match the `original` (left) image because of the distortion and recovery process. In particular, the image shrinking causes loss of information. The artifacts around the edges are due to the limited accuracy of the transformation. If you were to detect more points in **Step 4: Find Matching Features Between Images**, the transformation would be more accurate. For example, we could have used a corner detector, `detectFASTFeatures`, to complement the SURF feature detector which finds blobs. Image content and image size also impact the number of detected features.

See Also

`detectSURFFeatures` | `estimateGeometricTransform` | `extractFeatures` | `imref2d` | `imresize` | `imrotate` | `imwarp` | `matchFeatures` | `showMatchedFeatures`

Use Cross-Correlation to Improve Control Point Placement

You can 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 pixel regions around each control point in the moving image and around the matching control point in the fixed image. The function then calculates the correlation between the values at each pixel in the region. Next, the `cpcorr` function finds the position with the highest correlation value and uses it as the optimal position of the control point. The function only moves control points up to four 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.

See Also

`cpcorr` | `cpselect` | `cpstruct2pairs`

More About

- “Select Matching Control Point Pairs” on page 7-88
- “Control Point Selection Procedure” on page 7-77

Register an Aerial Photograph to a Digital Orthophoto

This example shows how to use control 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)
```

Image Courtesy of mPower3/Emerge

Aerial Photo Image

Image Courtesy of MassGIS

Orthophoto Image

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). The image has been orthorectified to remove camera, perspective, and relief distortions via a specialized image transformation process. The image is also georegistered (and geocoded): the columns and rows of the digital orthophoto image are aligned to the axes of the Massachusetts State Plane coordinate system. Each pixel center corresponds to a definite geographic location, and every pixel is 1 meter square in map units.

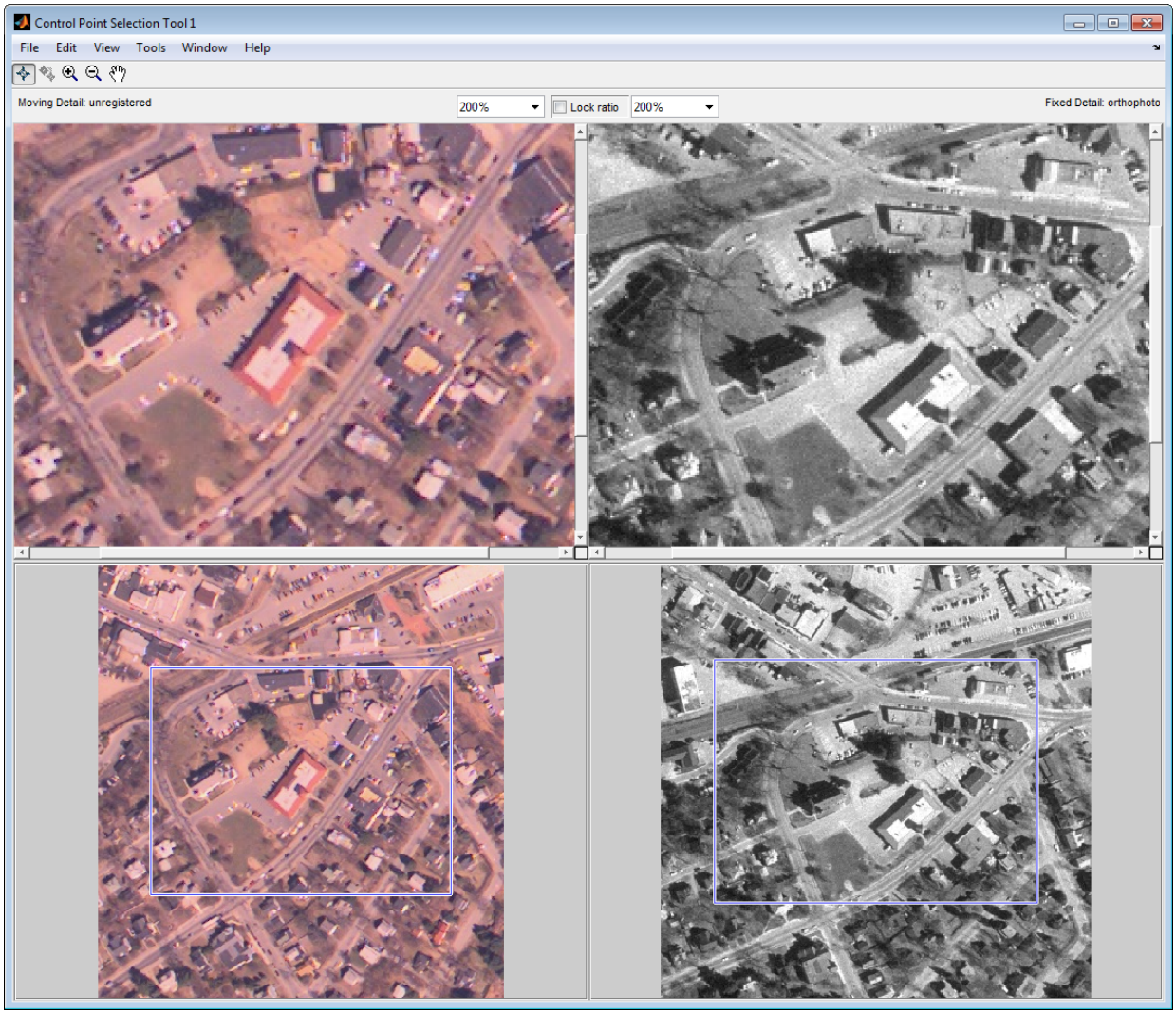
The moving image is `westconcordaerial.png`, a digital aerial photograph supplied by mPower3/Emerge. It is a visible-color RGB image. The image is geometrically

uncorrected: it includes camera perspective, terrain and building relief, internal (lens) distortions, and it does not have any particular alignment with the earth.

The example reads both images into the workspace. The `cpselect` function accepts images from the workspace or character vectors specifying the file path for the images.

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 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 and 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. To use cross-correlation, the images must be in the workspace, and 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 that brings 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. Choose which transformation is suitable for 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.

See Also

`cpcorr` | `cpselect` | `cpstruct2pairs` | `fitgeotrans`

More About

- “Select Matching Control Point Pairs” on page 7-88
- “Control Point Selection Procedure” on page 7-77

Registering an Aerial Photo to an Orthophoto

This example shows how to register an aerial photo to an orthophoto. Two images of the same scene can only be compared directly if they are in the same coordinate system. Image registration is the process of transforming one image into the coordinate system of another image.

Step 1: Read Images

The image `westconcordorthophoto.png` is an orthophoto that has already been registered to the ground. The image `westconcordaerial.png` is unregistered as it was taken from an airplane and is distorted relative to the orthophoto.

```
unregistered = imread('westconcordaerial.png');
imshow(unregistered)
text(size(unregistered,2),size(unregistered,1)+15, ...
    'Image courtesy of mPower3/Emerge', ...
    'FontSize',7,'HorizontalAlignment','right');
```

```
ortho = imread('westconcordorthophoto.png');
imshow(ortho)
text(size(ortho,2),size(ortho,1)+15, ...
    'Image courtesy of Massachusetts Executive Office of Environmental Affairs', ...
    'FontSize',7,'HorizontalAlignment','right');
```

Image courtesy of Massachusetts Executive Office of Environmental Affairs

Step 2: Load and Add Control Points

Four pairs of control points have already been picked. Load these points from a MAT-file. If you want to proceed with these points, go to Step 3: Infer Geometric Transformation.

```
load westconcordpoints
```

Optionally, edit or add to the pre-picked points using the Control Point Selection Tool (`cpselect`). `cpselect` helps you pick pairs of corresponding control points. Control points are landmarks that you can find in both images, like a road intersection, or a

natural feature. The unregistered image is an RGB image but `cpselect` only takes grayscale images, so you will pass it one plane of the RGB image.

```
cpselect(unregistered(:,:,1),'westconcordorthophoto.png',...

movingPoints,fixedPoints)
```

Save control points by choosing the **File** menu, then the **Save Points to Workspace** option. Save the points, overwriting variables `movingPoints` and `fixedPoints`.

Step 3: Infer Geometric Transformation

Because we know that the unregistered image was taken from an airplane, and the topography is relatively flat, it is likely that most of the distortion is projective. `fitgeotrans` will find the parameters of the projective distortion that best fits the stray movingPoints and fixedPoints you picked.

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

Step 4: Transform Unregistered Image

Even though the points were picked on one plane of the unregistered image, you can transform the entire RGB image. `imwarp` will apply the same transformation to each plane. Note that the specification of the 'OutputView' ensures the registered image will be aligned for elementwise comparison with the orthophoto.

```
Rfixed = imref2d(size(ortho));
registered = imwarp(unregistered,t_concord,'OutputView',Rfixed);
```

Step 5: View Registered Image in Context of Orthophoto

```
imshowpair(ortho,registered,'blend')
```

Compare visually how well the registered image overlays on the orthophoto. Try going back to **Step 2: Choose Control Points and using more than four pairs of points**. Are the results better? What if the points are clumped together?

If you want to experiment with larger images, follow the steps above to register `concordaerial.png` to `concordorthophoto.png`.

See Also

`cpcorr` | `cpselect` | `cpstruct2pairs` | `fitgeotrans`

More About

- “Select Matching Control Point Pairs” on page 7-88
- “Control Point Selection Procedure” on page 7-77

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
- “Integral Image” on page 8-5
- “Filter Grayscale and Truecolor (RGB) Images using imfilter Function” on page 8-7
- “imfilter Boundary Padding Options” on page 8-12
- “Filter Images Using Predefined Filters” on page 8-16
- “Generate HDL Code for Image Sharpening” on page 8-21
- “What is Guided Image Filtering?” on page 8-29
- “Perform Flash/No-flash Denoising with Guided Filter” on page 8-31
- “Segment Thermographic Image after Edge-Preserving Filtering” on page 8-36
- “Apply Multiple Filters to Integral Image” on page 8-41
- “Reducing Noise in Image Gradients” on page 8-47
- “Design Linear Filters in the Frequency Domain” on page 8-56

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” on page 16-2 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 = [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]
```

and the correlation kernel is

```
h = [8   1   6
     3   5   7
     4   9   2]
```

You would use the following steps to compute the output pixel at position (2,4):

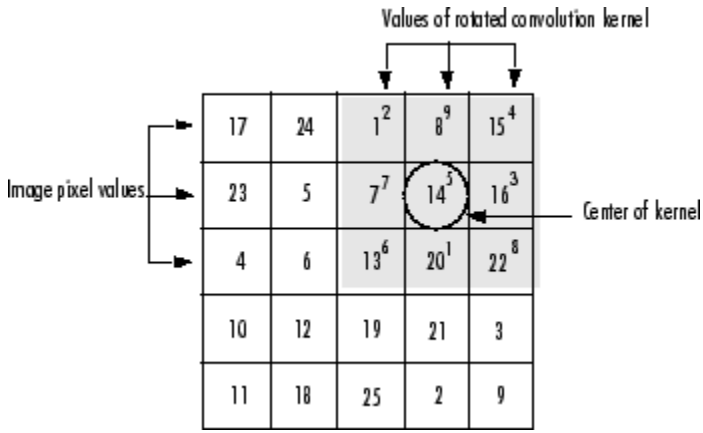
1. Rotate the correlation kernel 180 degrees about its center element to create a convolution kernel.

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

$$1 \cdot 2 + 8 \cdot 9 + 15 \cdot 4 + 7 \cdot 7 + 14 \cdot 5 + 16 \cdot 3 + 13 \cdot 6 + 20 \cdot 1 + 22 \cdot 8 = 575$$

Shown in the following figure.



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.

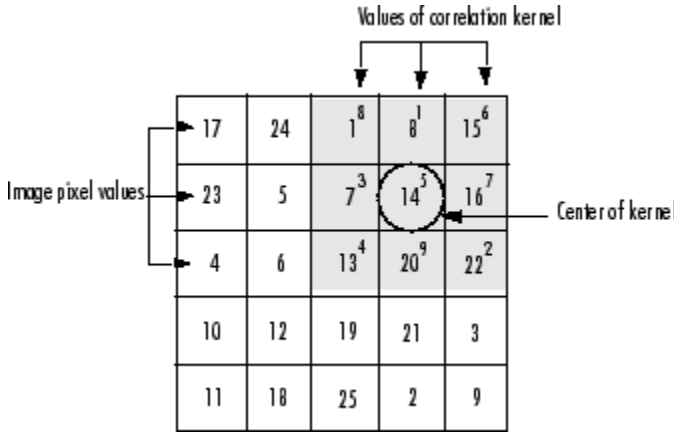
The following figure shows how to compute the (2,4) output pixel of the correlation of A, assuming h is a correlation kernel instead of a convolution kernel, using these steps:

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.

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

See Also

`conv2` | `convn` | `imfilter`

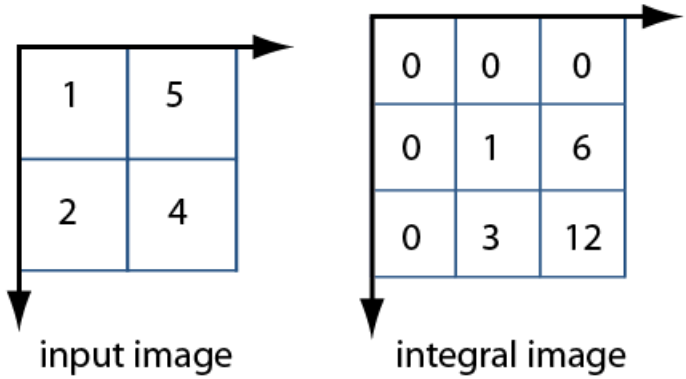
Related Examples

- “Design Linear Filters in the Frequency Domain” on page 8-56
- “Noise Removal” on page 11-151

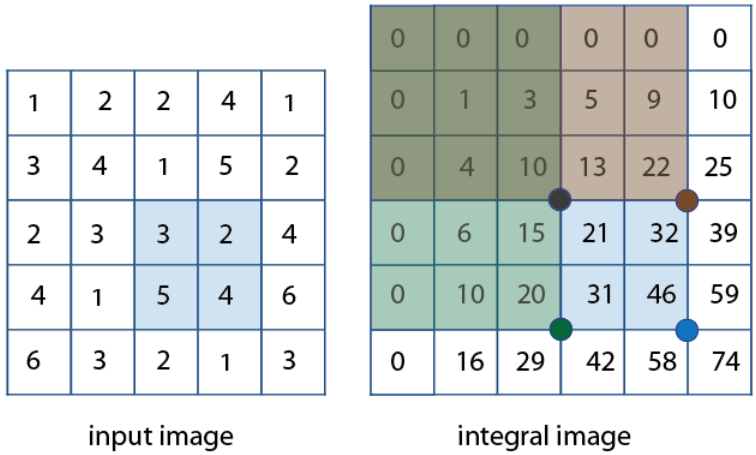
Integral Image

In an integral image, every pixel is the summation of the pixels above and to the left of it.

To illustrate, the following shows and image and its corresponding integral image. The integral image is padded to the left and the top to allow for the calculation. The pixel value at (2,1) in the original image becomes the pixel value (3,2) in the integral image after adding the pixel value above it (2+1) and to the left (3+0). Similarly, the pixel at (2,2) in the original image with the value 4 becomes the pixel at (3,3) in the integral image with the value 12 after adding the pixel value above it (4+5) and adding the pixel to the left of it ((9+3).



Using an integral image, you can rapidly calculate summations over image subregions. Integral images facilitate summation of pixels and can be performed in constant time, regardless of the neighborhood size. The following figure illustrates the summation of a subregion of an image, you can use the corresponding region of its integral image. For example, in the input image below, the summation of the shaded region becomes a simple calculation using four reference values of the rectangular region in the corresponding integral image. The calculation becomes, 46 – 22 – 20 + 10 = 14. The calculation subtracts the regions above and to the left of the shaded region. The area of overlap is added back to compensate for the double subtraction.



In this way, you can calculate summations in rectangular regions rapidly, irrespective of the filter size. Use of integral images was popularized by the Viola-Jones algorithm. To see the full citation for this algorithm and learn how to create an integral image, see `integralImage`.

See Also

`integralBoxFilter` | `integralBoxFilter3` | `integralImage` | `integralImage3`

Related Examples

- "Apply Multiple Filters to Integral Image" on page 8-41

Filter Grayscale and Truecolor (RGB) Images using imfilter Function

This example shows how to filter a 2-D grayscale image with a 5-by-5 filter containing equal weights (often called an averaging filter) using `imfilter`. The example also shows how to filter an truecolor (RGB) image with the same filter. A truecolor image is a 3-D array of size *m*-by-*n*-by-3, where the last dimension represents the three color channels. Filtering a truecolor image with a 2-D filter is equivalent to filtering each plane of the image individually with the same 2-D filter.

There are several MATLAB® functions that perform 2-D and multidimensional filtering that can be compared to `imfilter`. The function `filter2` performs two-dimensional correlation, `conv2` performs two-dimensional convolution, and `convn` performs multidimensional convolution. However, each of these filtering functions always converts the input to `double`, and the output is always `double`. Also, these MATLAB® filtering functions always assume the input is zero padded, and they do not support other padding options. In contrast, `imfilter` does not convert input images to `double`. The `imfilter` function also offers a flexible set of boundary padding options.

Filter 2-D Grayscale Image with an Averaging Filter

Read a grayscale image into the workspace.

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

Display the original image.

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

Create a normalized, 5-by-5, averaging filter.

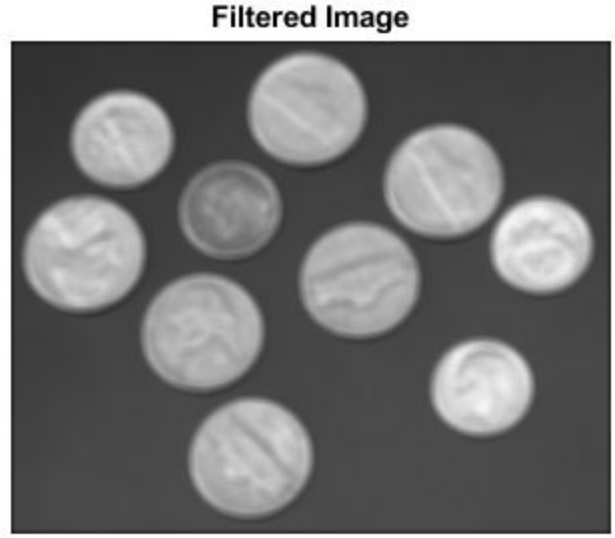
```
h = ones(5,5)/25;
```

Apply the averaging filter to the grayscale image using `imfilter`.

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

Display the filtered image.

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



Filter Multidimensional Truecolor (RGB) Image Using imfilter

Read a truecolor image into the workspace.

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

Create a filter. This averaging filter contains equal weights, and causes the filtered image to look more blurry than the original.

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

Filter the image using `imfilter` and display it.

```
rgb2 = imfilter(rgb,h);
figure
imshow(rgb2)
```

See Also

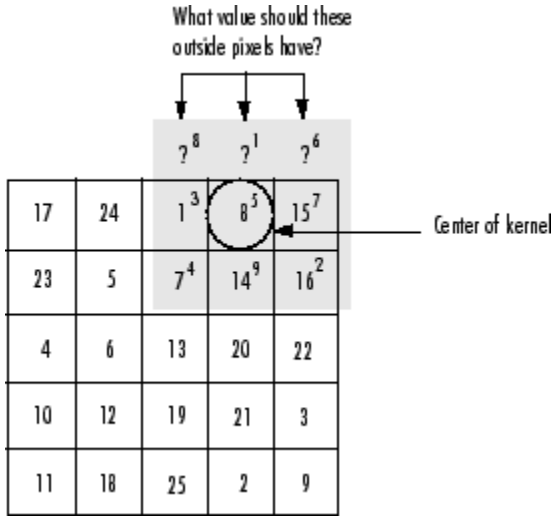
`imfilter`

More About

- “What Is Image Filtering in the Spatial Domain?” on page 8-2

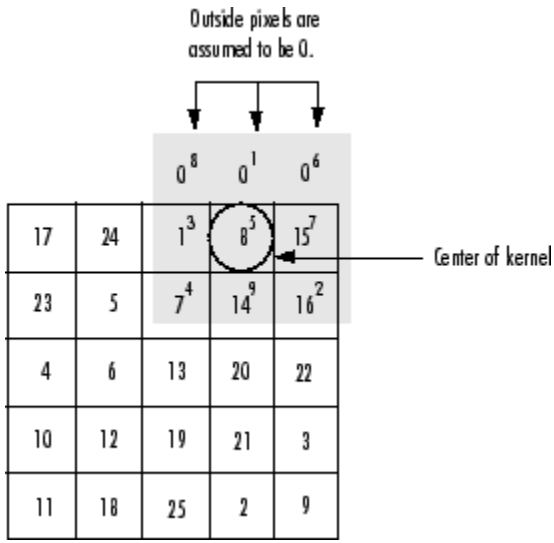
imfilter Boundary Padding Options

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

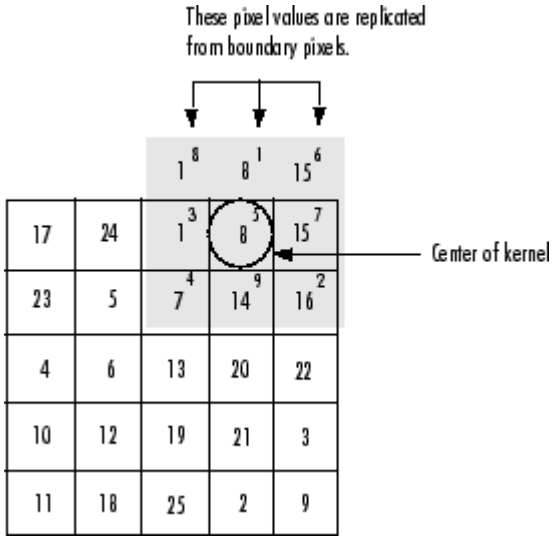
When you filter an image, zero padding can result in a dark band around the edge of the image, as shown in this example.

```
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.

See Also

`imfilter`

Related Examples

- “Design Linear Filters in the Frequency Domain” on page 8-56

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 = 3×3

   -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')
```

Original Image

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

Filtered Image

See Also

`fspecial` | `imfilter`

Related Examples

- "Design Linear Filters in the Frequency Domain" on page 8-56

Generate HDL Code for Image Sharpening

This example shows how to use Vision HDL Toolbox™ to implement an FPGA-based module for image sharpening.

Vision HDL Toolbox provides image and video processing algorithms designed to generate readable, synthesizable code in VHDL and Verilog (with HDL Coder™). The generated HDL code when run on an FPGA (for example, Xilinx XC7Z045) can process 1920x1080 full-resolution images at 60 frames per second.

This example shows how to use Vision HDL Toolbox to generate HDL code that sharpens a blurred image. Since Vision HDL Toolbox algorithms are available as MATLAB® System objects™ and Simulink® blocks, HDL code can be generated from MATLAB or Simulink. This example shows both workflows.

The workflow for an FPGA-targeted design is:

1. Create a behavioral model to represent design goals;

2. Replicate the design using algorithms, interfaces, and data types appropriate for FPGAs and supported for HDL code generation.

3. Simulate the two designs and compare the results to confirm that the HDL-optimized design meets the goals.

4. Generate HDL code from the design created in Step 2.

For Steps 2 and 3 in MATLAB, you must have MATLAB, Vision HDL Toolbox, and Fixed-Point Designer™. In Simulink, you need Simulink, Vision HDL Toolbox, and Fixed-Point Designer. In both cases, you must have HDL Coder to generate HDL code.

Behavioral Model

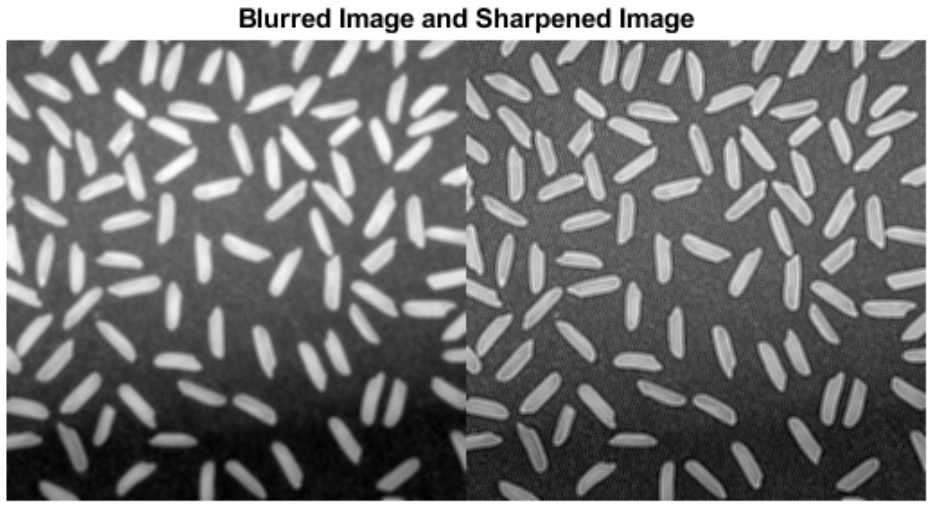
The input image imgBlur is shown on the left in the diagram below. On the right, the image is sharpened using the Image Processing Toolbox™ function `imfilter`.

Simulation time is printed as a bench mark for future comparison.

```
imgBlur = imread('riceblurred.png');
sharpCoeff = [0 0 0;0 1 0;0 0 0]-fspecial('laplacian',0.2);

f = @() imfilter(imgBlur,sharpCoeff,'symmetric');
```

```
fprintf('Elapsed time is %.6f seconds.\n',timeit(f));

imgSharp = imfilter(imgBlur,sharpCoeff,'symmetric');
figure
imshowpair(imgBlur,imgSharp,'montage')
title('Blurred Image and Sharpened Image')
```

```
Elapsed time is 0.000399 seconds.
```



HDL-Optimized Design Considerations

Three key changes need to be made to enable HDL code generation.

- **Use HDL-friendly algorithms:** The functions in Image Processing Toolbox do not support HDL Code generation. Vision HDL Toolbox provides image and video processing algorithms designed for efficient HDL implementations. You can generate HDL code from these algorithms using System objects in MATLAB and blocks in Simulink. Both workflows are provided in this example. To design an FPGA-based module, replace the functions from Image Processing Toolbox with their HDL-friendly counterparts from Vision HDL Toolbox. This example replaces `imfilter` in the

behavioral model with the `visionhdl.ImageFilter` System object in MATLAB, or the `Image Filter` block in Simulink.

- **Use streaming pixel interface:** The functions from Image Processing Toolbox model at a high level of abstraction. They perform full-frame processing, operating on one image frame at a time. FPGA and ASIC implementations, however, perform pixel-stream processing, operating on one image pixel at a time. Vision HDL Toolbox blocks and System objects use a streaming pixel interface. Use `visionhdl.FrameToPixels` System object in MATLAB or `Frame To Pixels` block in Simulink to convert a full frame image or video to a pixel stream. The streaming pixel interface includes control signals that indicate each pixel's position in the frame. Algorithms that operate on a pixel neighborhood use internal memory to store a minimum number of lines. Vision HDL Toolbox provides the streaming pixel interface and automatic memory implementation to address common design issues when targeting FPGAs and ASICs. For more information on the streaming pixel protocol used by System objects from the Vision HDL Toolbox, see streaming pixel interface.

- **Use fixed-point data representation:** Functions from Image Processing Toolbox perform video processing algorithms in the floating-point or integer domain. The System objects and blocks from Vision HDL Toolbox require fixed-point data to generate HDL code to target FPGAs and ASICs. Converting a design to fixed-point can introduce quantization error. Therefore, the HDL-friendly model might generate an output slightly different from that obtained from the behavioral model. For most applications, small quantization errors within a tolerance are acceptable. You can tune the fixed-point settings to suit your requirements.

In this example, we use a static image as the source. This model is also able to process continuous video input.

Generate HDL Code from MATLAB

To generate HDL from MATLAB, your code needs to be divided into two files: test bench and design. The design file is used for implementing the algorithm in the FPGA or ASIC. The test bench file provides the input data to the design file and receives the design output.

Step 1: Create Design File

The function ImageSharpeningHDLDesign.m accepts a pixel stream and a control structure consisting of five control signals, and returns a modified pixel stream and control structure.

In this example, the design contains a System object `visionhdl.ImageFilter`. It is the HDL-friendly counterpart of the `imfilter` function. Configure it with the same coefficients and padding method as `imfilter`.

```
function [pixOut,ctrlOut] = ImageSharpeningHDLDesign(pixIn,ctrlIn)
% ImageSharpeningHDLDesign  Implement algorithms using pixel-stream
%       System objects from the Vision HDL Toolbox

%   Copyright 2015 The MathWorks, Inc.

%#codegen
persistent sharpeningFilter;
if isempty(sharpeningFilter)
    sharpCoeff = [0 0 0;0 1 0;0 0 0]-fspecial('laplacian',0.2);
    sharpeningFilter = visionhdl.ImageFilter(...
    'Coefficients',sharpCoeff,...
    'PaddingMethod','Symmetric',...
    'CoefficientsDataType','Custom',...
    'CustomCoefficientsDataType',numerictype(1,16,12));
end

[pixOut,ctrlOut] = step(sharpeningFilter,pixIn,ctrlIn);
```

Step 2: Create Test Bench File

The test bench ImageSharpeningHDLTestBench.m reads in the blurred image. The `frm2pix` object converts the full image frame to a stream of pixels and control structures. The test bench calls the design function `ImageSharpeningHDLDesign` to process one pixel at a time. After the entire pixel-stream is processed, `pix2frm` converts the output pixel stream to a full-frame image. The test bench compares the output image to the reference output `imgSharp`.

```
...
[pixInVec,ctrlInVec] = step(frm2pix,imgBlur);
for p = 1:numPixPerFrm
    [pixOutVec(p),ctrlOutVec(p)] = ImageSharpeningHDLDesign(pixInVec(p),ctrlInVec(p));
end
imgOut = step(pix2frm,pixOutVec,ctrlOutVec);

% Compare the result
imgDiff = imabsdiff(imgSharp,imgOut);
```

```
fprintf('The maximum difference between corresponding pixels is %d.\n',max(imgDiff(:)))
fprintf('A total of %d pixels are different.\n',nnz(imgDiff));
...
```

Step 3: Simulate Design and Verify Result

Simulate the design with the test bench prior to HDL code generation to make sure there are no runtime errors.

```
ImageSharpeningHDLTestBench
```

```
The maximum difference between corresponding pixels is 1.
A total of 41248 pixels are different.

Simulation took 368.091661 seconds to finish.
```

The test bench displays the comparison result and the time spent on simulation. Due to quantization error and rounding error, out of a total of 256*256=65536 pixels, 38554 of imgOut are different from `imgSharp`. However, the maximum difference in intensity is 1. On a 0 to 255 scale, this difference is visually unnoticeable.

As we can see by comparing the simulation time in MATLAB with that of the behavioral model, the pixel-streaming protocol introduces significant overhead. You can use MATLAB Coder™ to speed up the pixel-streaming simulation in MATLAB. See Accelerate a Pixel-Streaming Design Using MATLAB Coder.

Step 4: Generate HDL Code

Once you are satisfied with the results of the FPGA-targeted model, you can use HDL Coder to generate HDL code from the design. You can run the generated HDL code in HDL simulators or load it into an FPGA and run it in a physical system.

Make sure that the design and test bench files are located in the same writable directory. To generate the HDL code, use the following command:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'ImageSharpeningHDLTestBench';
hdlcfg.TargetLanguage = 'Verilog';
hdlcfg.GenerateHDLTestBench = false;
codegen -config hdlcfg ImageSharpeningHDLDesign
```

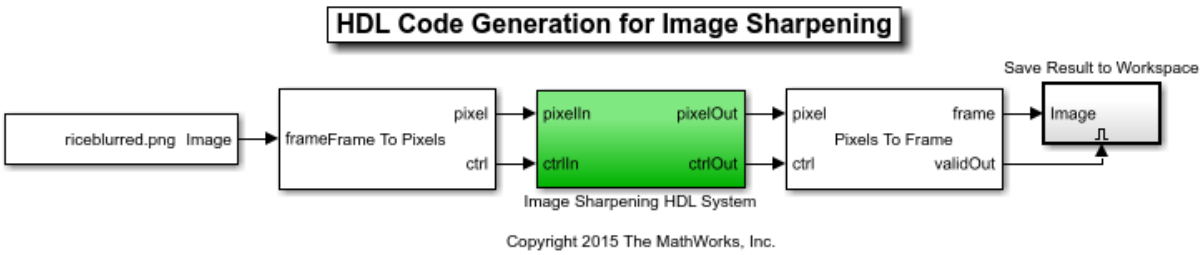
Refer to Getting Started with MATLAB to HDL Workflow for a tutorial on creating and configuring MATLAB to HDL projects.

Generate HDL Code from Simulink

Step 1: Create HDL-Optimized Model

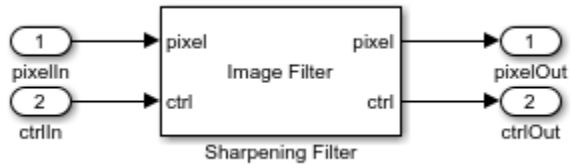
The ImageSharpeningHDLModel model is shown below.

```
modelname = 'ImageSharpeningHDLModel';
open_system(modelname);
set_param(modelname,'Open','on');
```



The model reads in the blurred image. The Frame To Pixels block converts a full-frame image to a pixel stream, and the Pixels To Frame block converts the pixel stream back to a full-frame image. The Image Sharpening HDL System contains an Image Filter block, which is the HDL-friendly counterpart in Vision HDL Toolbox of the `imfilter` function presented in the behavioral model.

```
set_param(modelname,'Open','off');
set_param([modelname '/Image Sharpening HDL System'],'Open','on');
```



Configure the Image Filter block with the same sharpening coefficients and padding method as in the behavioral model, as shown on the masks below.

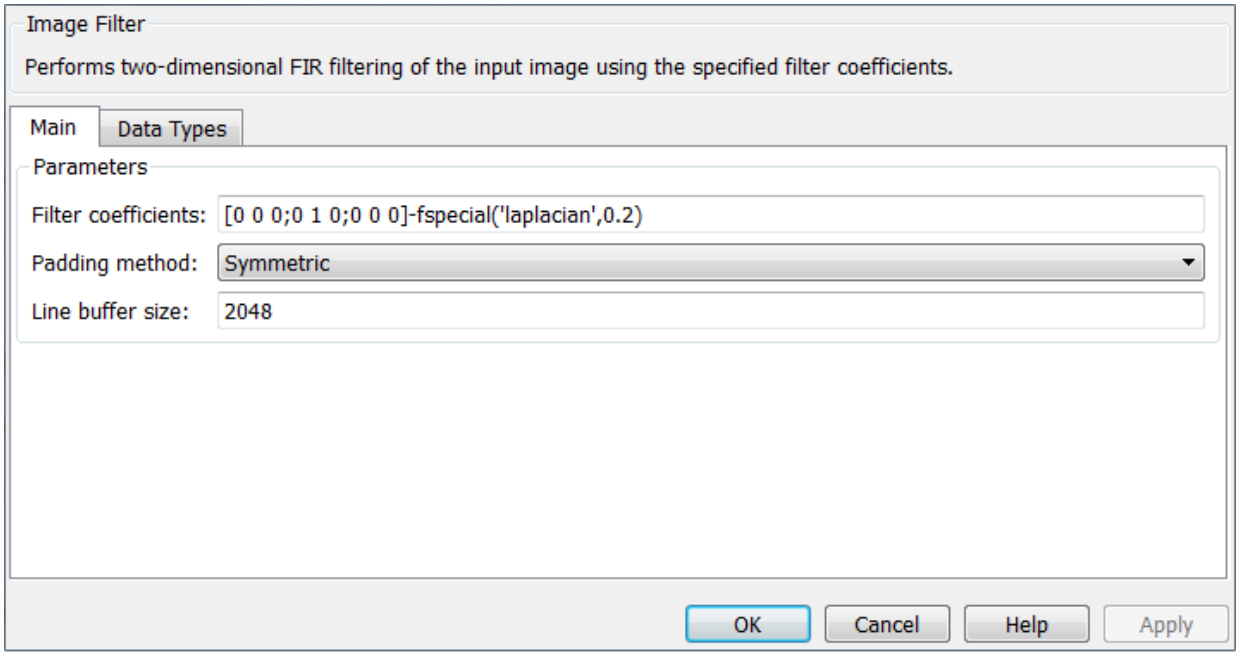


Image Filter
Performs two-dimensional FIR filtering of the input image using the specified filter coefficients.
Main
Data Types
Parameters
Filter coefficients:
[0 0 0;0 1 0;0 0 0]-fspecial('laplacian',0.2)
Padding method:
Symmetric
Line buffer size:
2048
OK
Cancel
Help
Apply

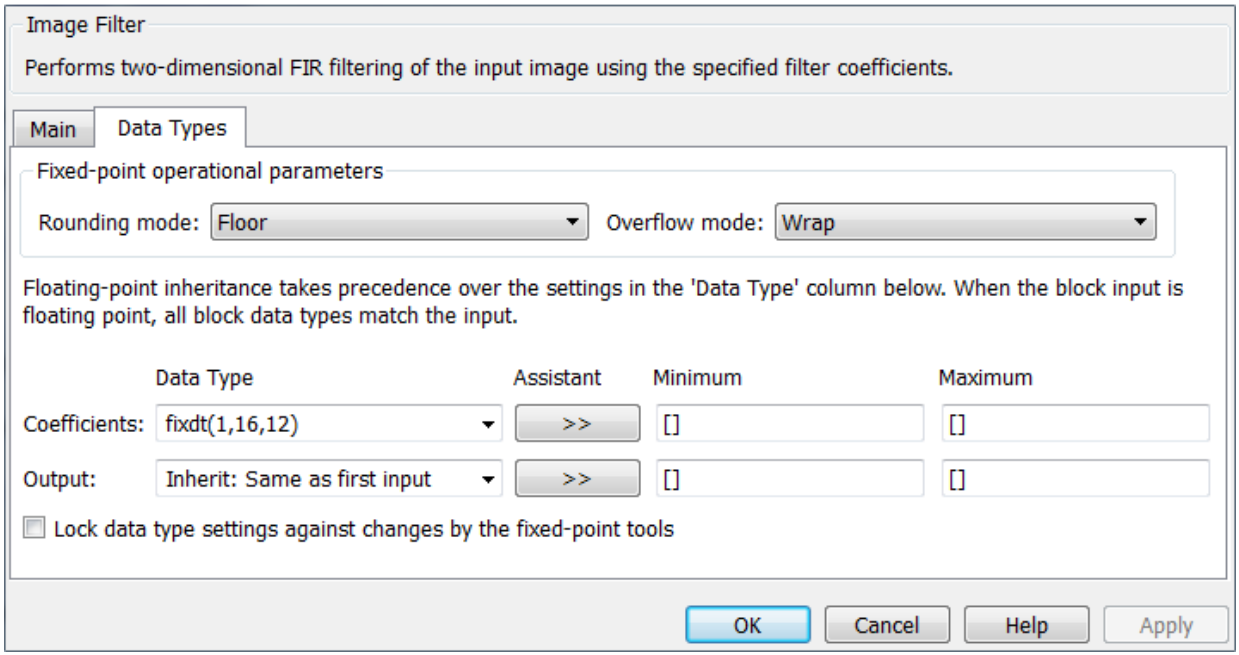


Image Filter
Performs two-dimensional FIR filtering of the input image using the specified filter coefficients.
Main
Data Types
Fixed-point operational parameters
Rounding mode:
Floor
Overflow mode:
Wrap
Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input.
Data Type
Assistant
Minimum
Maximum
Coefficients:
fixdt(1,16,12)
>>
[]
[]
Output:
Inherit: Same as first input
>>
[]
[]
Lock data type settings against changes by the fixed-point tools
OK
Cancel
Help
Apply

Step 2: Simulate Design and Verify Result

```
tic
sim(modelname);
toc
```

```
Elapsed time is 20.761860 seconds.
```

Simulink takes advantage of C code generation to speed up the simulation. Therefore, it is much faster than MATLAB simulation, although still slower than the behavioral model.

The simulation creates a new variable called imgOut in the workspace. Use the following commands to compare `imgOut` with `imSharp` generated from the behavioral model.

```
imgDiff = imabsdiff(imgSharp,imgOut);
fprintf('The maximum difference between corresponding pixels is %d.\n',max(imgDiff(:)))
fprintf('A total of %d pixels are different.\n',nnz(imgDiff));
```

```
The maximum difference between corresponding pixels is 1.
A total of 41248 pixels are different.
```

Due to quantization error and rounding error, out of a total of 256*256=65536 pixels, 38554 of `imgOut` are different from `imgSharp`. However, the maximum difference in intensity is 1. On a 0 to 255 scale, this difference is visually unnoticeable. (This is the same explanation as that presented in Step 3 in the "Generate HDL Code from MATLAB" Section.)

Step 3: Generate HDL Code

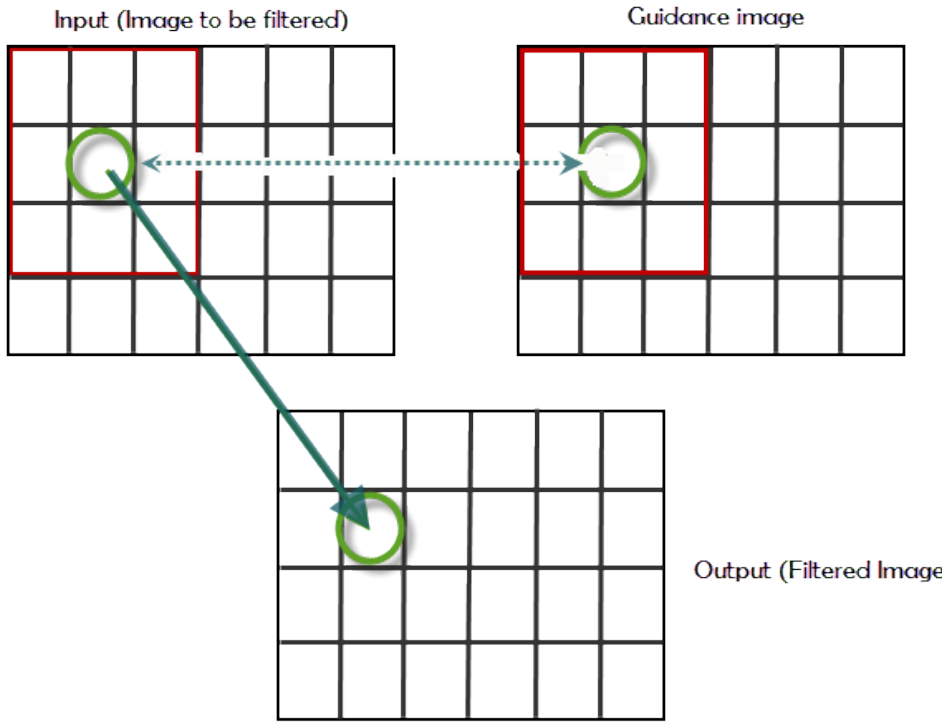
Once you are satisfied with the results of the FPGA-targeted model, you can use HDL Coder to generate HDL code from the design. You can run the generated HDL code in HDL simulators or load it into an FPGA and run it in a physical system.

Generate HDL code from the Image Sharpening HDL System using the following command:

```
makehdl('ImageSharpeningHDLModel/Image Sharpening HDL System')

set_param([modelname '/Image Sharpening HDL System'],'Open','off');
close_system(modelname,0);
close all;
```

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.

See Also

`imguidedfilter`

Related Examples

- “Perform Flash/No-flash Denoising with Guided Filter” on page 8-31

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 into the workspace. This example uses an image of some toys taken without a flash. 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')
```

Read the image that you want to use as the guidance image into the workspace. 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')
```

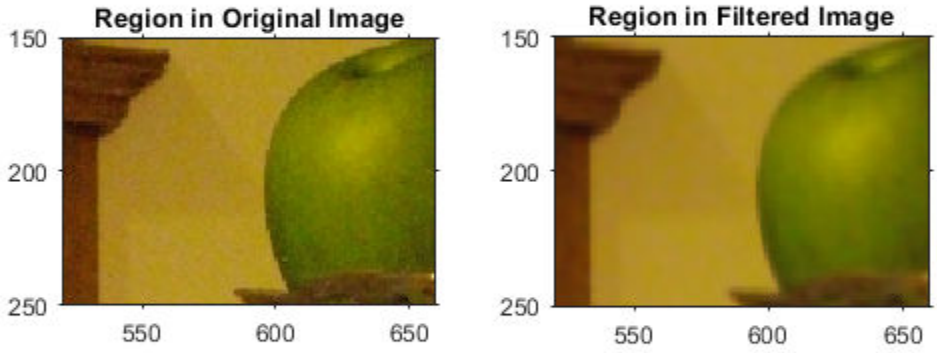
Guidance Image - Camera Flash On

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])
```



See Also

`imguidedfilter`

More About

- “What is Guided Image Filtering?” on page 8-29

Segment Thermographic Image after Edge-Preserving Filtering

This example shows how to work with thermal images, demonstrating a simple segmentation. Thermal images are obtained from thermographic cameras, which detect radiation in the infrared range of the electromagnetic spectrum. Thermographic images capture infrared radiation emitted by all objects above absolute zero.

Read a thermal image into the workspace and use `whos` to understand more about the image data.

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

```
whos I
```

```
  Name        Size              Bytes  Class     Attributes

  I         240x320            307200  single
```

Compute the dynamic range occupied by the data to see the range of temperatures occupied by the image. The pixel values in this image correspond to actual temperatures on the Celsius scale.

```
range = [min(I(:)) max(I(:))]
```

```
range = 1x2 single row vector

   22.4729   77.3727
```

Display the thermal image. Because the thermal image is a single-precision image with a dynamic range outside 0 to 1, you must use the `imshow` auto-scaling capability to display the image.

```
figure
imshow(I,[])
colormap(gca,hot)
title('Original image')
```

Apply edge-preserving smoothing to the image to remove noise while still retaining image details. This is a preprocessing step before segmentation. Use the `imguidedfilter` function to perform smoothing under self-guidance. The `'DegreeOfSmoothing'` parameter controls the amount of smoothing and is dependent on the range of the image. Adjust the `'DegreeOfSmoothing'` to accommodate the range of the thermographic image. Display the filtered image.

```
smoothValue = 0.01*diff(range).^2;
J = imguidedfilter(I,'DegreeOfSmoothing',smoothValue);

figure
imshow(J,[])
colormap(gca,hot)
title('Guided filtered image')
```

Determine threshold values to use in segmentation. The image has 3 distinct regions - the person, the hot object and the background - that appear well separated in intensity (temperature). Use `multithresh` to compute a 2-level threshold for the image. This partitions the image into 3 regions using Otsu's method.

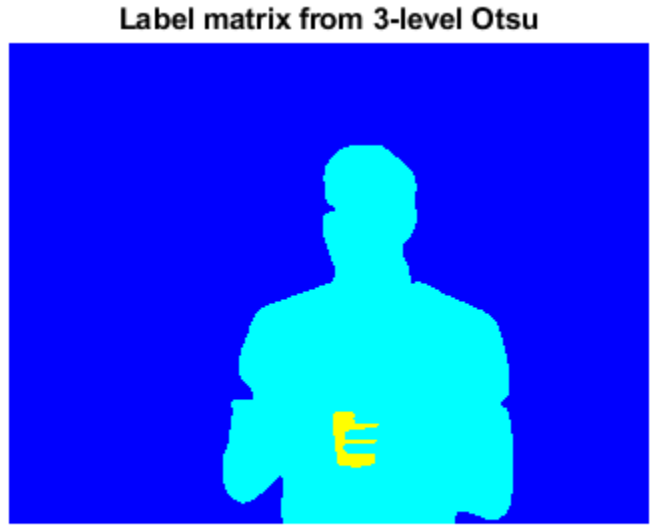
```
thresh = multithresh(J,2)
```

```
thresh = 1x2 single row vector

   27.0018   47.8220
```

Threshold the image using the values returned by `multithresh`. The threshold values are at 27 and 48 Celsius. The first threshold separates the background intensity from the person and the second threshold separates the person from the hot object. Segment the image and fill holes.

```
L = imquantize(J,thresh);
L = imfill(L);
```

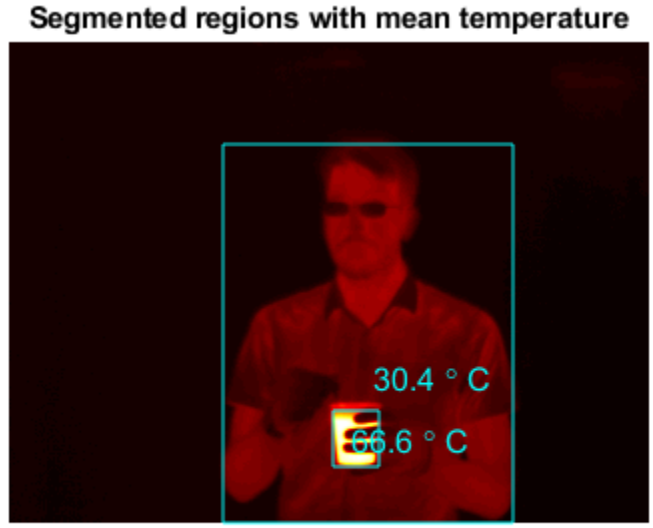
```
figure
imshow(label2rgb(L))
title('Label matrix from 3-level Otsu')
```



Draw a bounding box around the foreground regions in the image and put the mean temperature value of the region in the box. The example assumes that the largest region is the background. Use the `regionprops` function to get information about the regions in the segmented image.

```
props = regionprops(L,I,{'Area','BoundingBox','MeanIntensity','Centroid'});

% Find the index of the background region.
[~,idx] = max([props.Area]);

figure
imshow(I,[])
colormap(gca,hot)
title('Segmented regions with mean temperature')
for n = 1:numel(props)
    % If the region is not background
```

```
    if n ~= idx
        % Draw bounding box around region
       rectangle('Position',props(n).BoundingBox,'EdgeColor','c')

       % Draw text displaying mean temperature in Celsius
       T = [num2str(props(n).MeanIntensity,3) ' \circ C'];
       text(props(n).Centroid(1),props(n).Centroid(2),T,...
           'Color','c','FontSize',12)
    end
end
```



See Also

imfill | imguidedfilter | imquantize | multithresh

More About

- “What is Guided Image Filtering?” on page 8-29

Apply Multiple Filters to Integral Image

This example shows how to apply multiple box filters of varying sizes to an image using integral image filtering. Integral image is a useful image representation from which local image sums can be computed rapidly. A box filter can be thought of as a local weighted sum at each pixel.

Read an image into the workspace and display it.

```
originalImage = imread('cameraman.tif');

figure
imshow(originalImage)
title('Original Image')
```

Define the sizes of the three box filters.

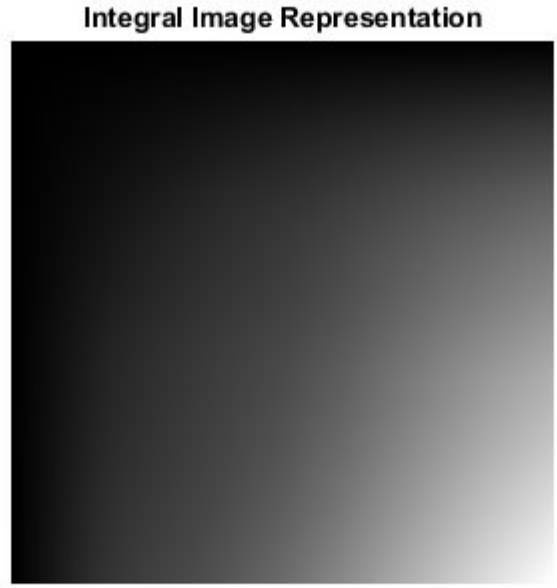
```
filterSizes = [7 7;11 11;15 15];
```

Pad the image to accommodate the size of the largest box filter. Pad each dimension by an amount equal to half the size of the largest filter. Note the use of replicate-style padding to help reduce boundary artifacts.

```
maxFilterSize = max(filterSizes);
padSize = (maxFilterSize - 1)/2;

paddedImage = padarray(originalImage,padSize,'replicate','both');
```

Compute the integral image representation of the padded image using the `integralImage` function and display it. The integral image is monotonically non-decreasing from left to right and top to bottom. Each pixel represents the sum of all pixel intensities to the top and left of the current pixel in the image.

```
intImage = integralImage(paddedImage);

figure
imshow(intImage,[])
title('Integral Image Representation')
```



Apply three box filters of varying sizes to the integral image. The `integralBoxFilter` function can be used to apply a 2-D box filter to the integral image representation of an image.

```
filteredImage1 = integralBoxFilter(intImage, filterSizes(1,:));
filteredImage2 = integralBoxFilter(intImage, filterSizes(2,:));
filteredImage3 = integralBoxFilter(intImage, filterSizes(3,:));
```

The `integralBoxFilter` function returns only parts of the filtering that are computed without padding. Filtering the same integral image with different sized box filters results in different sized outputs. This is similar to the `'valid'` option in the `conv2` function.

```
whos filteredImage*
```

```
  Name                  Size              Bytes  Class     Attributes

  filteredImage1      264x264            557568  double
```

```
  filteredImage2      260x260            540800  double
  filteredImage3      256x256            524288  double
```

Because the image was padded to accommodate the largest box filter prior to computing the integral image, no image content is lost. `filteredImage1` and `filteredImage2` have additional padding that can be cropped.

```
extraPadding1 = (maxFilterSize - filterSizes(1,:))/2;
filteredImage1 = filteredImage1(1+extraPadding1(1):end-extraPadding1(1),...
    1+extraPadding1(2):end-extraPadding1(2) );

extraPadding2 = (maxFilterSize - filterSizes(2,:))/2;
filteredImage2 = filteredImage2(1+extraPadding2(1):end-extraPadding2(1),...
    1+extraPadding2(2):end-extraPadding2(2) );

figure
imshow(filteredImage1,[])
title('Image filtered with [7 7] box filter')
```

```
figure
imshow(filteredImage2,[])
title('Image filtered with [11 11] box filter')
```

```
figure
imshow(filteredImage3,[])
title('Image filtered with [15 15] box filter')
```

See Also

`integralBoxFilter` | `integralBoxFilter3` | `integralImage` | `integralImage3`

More About

- "Integral Image" on page 8-5

Reducing Noise in Image Gradients

This example demonstrates how to reduce noise associated with computing image gradients. Image gradients are used to highlight interesting features in images and are used in many feature detection algorithms like edge/corner detection. Reducing noise in gradient computations is crucial to detecting accurate features.

Read an image into the workspace and convert it to grayscale.

```
originalImage = imread('yellowlily.jpg');
originalImage = rgb2gray(originalImage);

warning off images:initSize:adjustingMag

figure
imshow(originalImage)
```

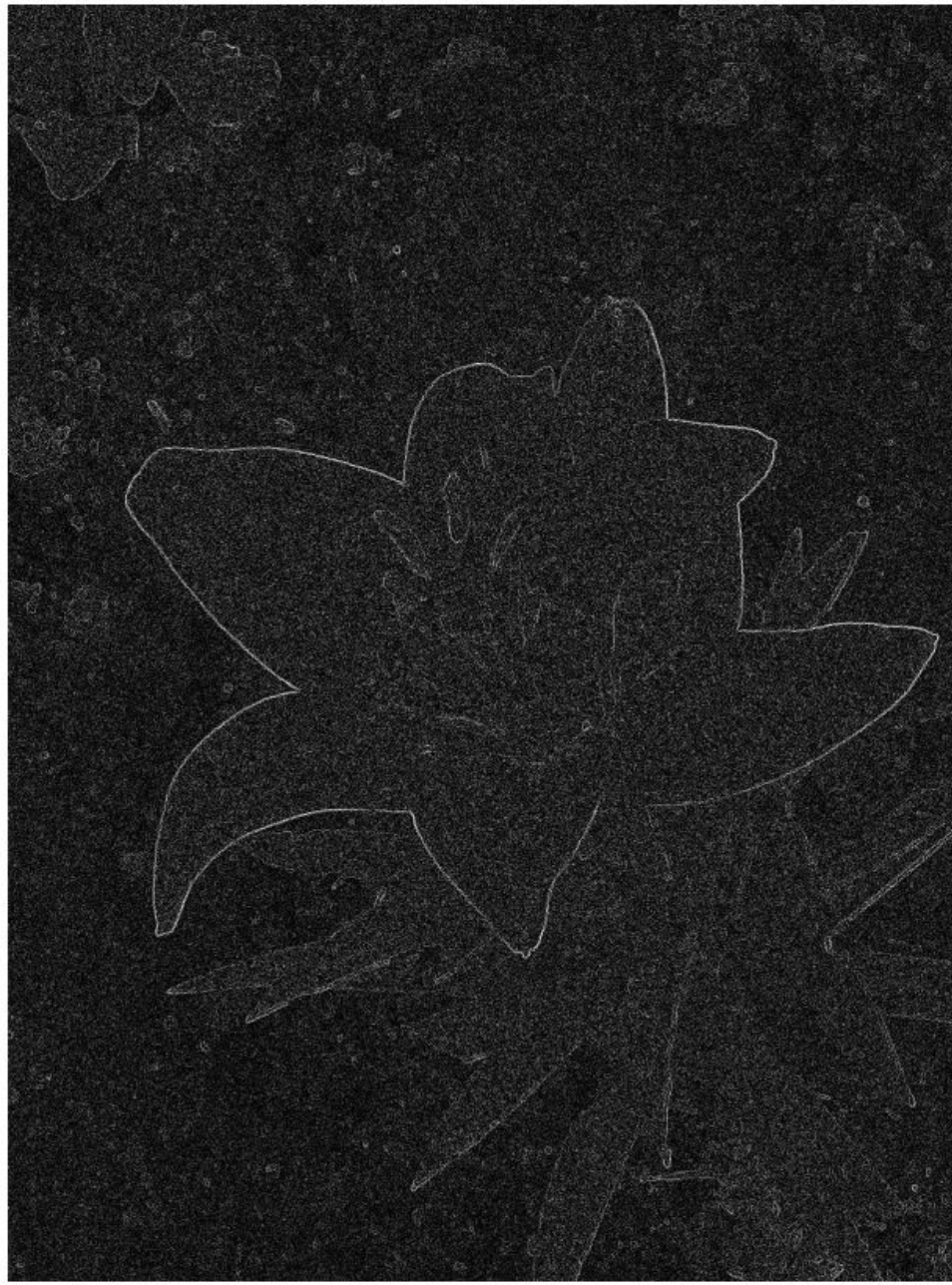
To simulate noise for this example, add some Gaussian noise to the image.

```
noisyImage = imnoise(originalImage,'gaussian');

figure
imshow(noisyImage)
```

Compute the magnitude of the gradient, using the `imgradient` and `imgradientxy` functions. `imgradient` finds the gradient magnitude and direction, and `imgradientxy` finds directional image gradients.

```
sobelGradient = imgradient(noisyImage);

figure
imshow(sobelGradient,[])
title('Sobel Gradient Magnitude')
```

Sobel Gradient Magnitude



Looking at the gradient magnitude image, it is clear that the image gradient is very noisy. The effect of noise can be minimized by smoothing before gradient computation. `imgradient` already offers this capability for small amounts of noise by using the Sobel gradient operator. The Sobel gradient operators are 3x3 filters as shown below. They can be generated using the `fspecial` function.

```
hy = -fspecial('sobel')
```

```
hy = 3×3

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

```
hx = hy'
```

```
hx = 3×3

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

The `hy` filter computes a gradient along the vertical direction while smoothing in the horizontal direction. `hx` smooths in the vertical direction and computes a gradient along the horizontal direction. The `'Prewitt'` and `'Roberts'` method options also provide this capability.

Even with the use of Sobel, Roberts or Prewitts gradient operators, the image gradient may be too noisy. To overcome this, smooth the image using a Gaussian smoothing filter before computing image gradients. Use the `imgaussfilt` function to smooth the image. The standard deviation of the Gaussian filter varies the extent of smoothing. Since smoothing is taken care of by Gaussian filtering, the central or intermediate differencing gradient operators can be used.

```
sigma = 2;
smoothImage = imgaussfilt(noisyImage,sigma);
smoothGradient = imgradient(smoothImage,'CentralDifference');

figure
imshow(smoothGradient,[])
title('Smoothed Gradient Magnitude')
```

Smoothed Gradient Magnitude

See Also

`fspecial` | `imgaussfilt` | `imgradient` | `imnoise`

More About

- “What Is Image Filtering in the Spatial Domain?” on page 8-2

Design Linear Filters in the Frequency Domain

In this section...

"Two-Dimensional Finite Impulse Response (FIR) Filters" on page 8-56
"Transform 1-D FIR Filter to 2-D FIR Filter" on page 8-57
"Frequency Sampling Method" on page 8-60
"Windowing Method" on page 8-61
"Creating the Desired Frequency Response Matrix" on page 8-62
"Computing the Frequency Response of a Filter" on page 8-63

This topic describes functions that perform filtering in the frequency domain. For information about designing filters in the spatial domain, see "What Is Image Filtering in the Spatial Domain?" on page 8-2.

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.

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 a *transformation 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 using the `firpm` function from the Signal Processing Toolbox.

```
b = firpm(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.

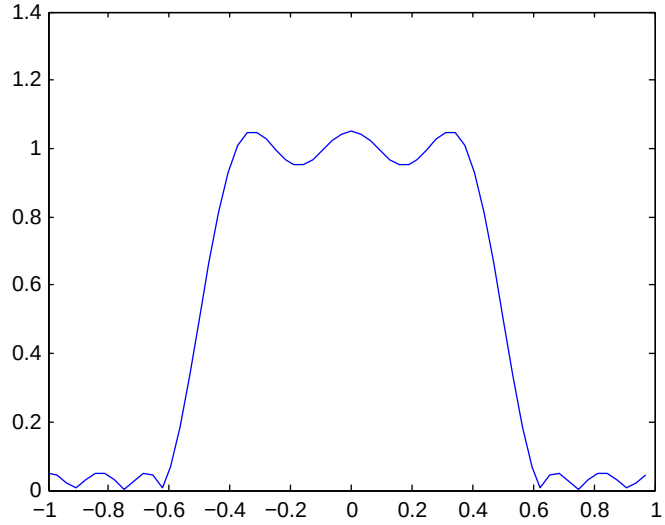
```
h = ftrans2(b);
```

```
h =

  Columns 1 through 9

    0.0001    0.0005    0.0024    0.0063    0.0110    0.0132    0.0110    0.0063    0.0
    0.0005    0.0031    0.0068    0.0042   -0.0074   -0.0147   -0.0074    0.0042    0.0
```

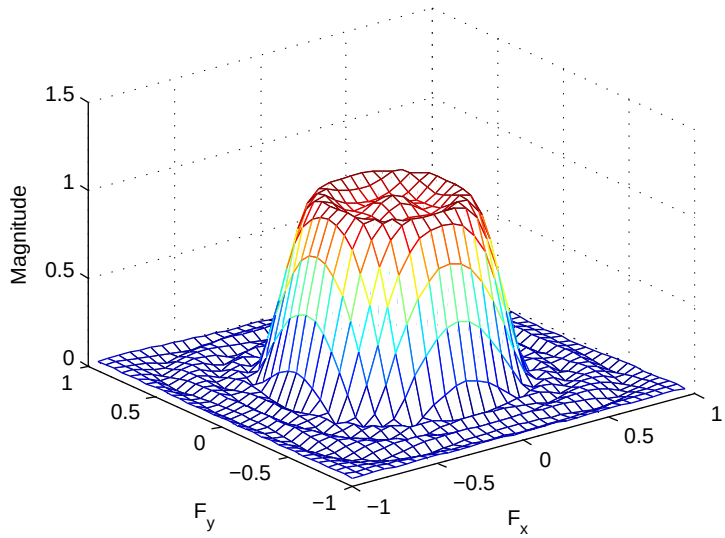
```
    0.0024    0.0068   -0.0001   -0.0191   -0.0251   -0.0213   -0.0251   -0.0191   -0.0
    0.0063    0.0042   -0.0191   -0.0172    0.0128    0.0259    0.0128   -0.0172   -0.0
    0.0110   -0.0074   -0.0251    0.0128    0.0924    0.1457    0.0924    0.0128   -0.0
    0.0132   -0.0147   -0.0213    0.0259    0.1457    0.2021    0.1457    0.0259   -0.0
    0.0110   -0.0074   -0.0251    0.0128    0.0924    0.1457    0.0924    0.0128   -0.0
    0.0063    0.0042   -0.0191   -0.0172    0.0128    0.0259    0.0128   -0.0172   -0.0
    0.0024    0.0068   -0.0001   -0.0191   -0.0251   -0.0213   -0.0251   -0.0191   -0.0
    0.0005    0.0031    0.0068    0.0042   -0.0074   -0.0147   -0.0074    0.0042    0.0
    0.0001    0.0005    0.0024    0.0063    0.0110    0.0132    0.0110    0.0063    0.0

  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

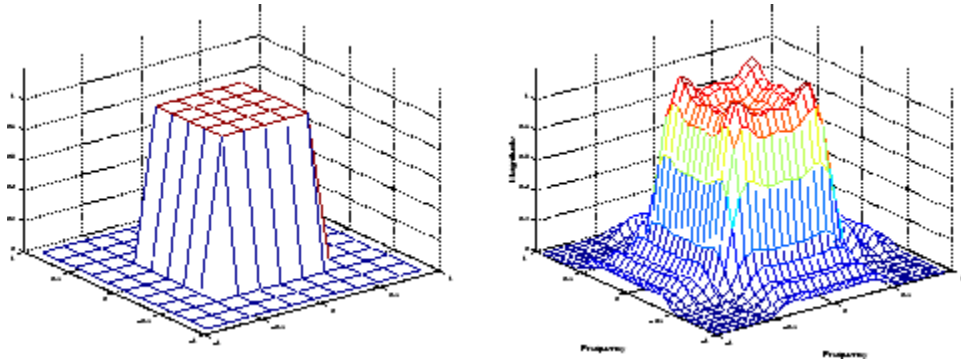
Frequency Sampling Method

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 8-63 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.

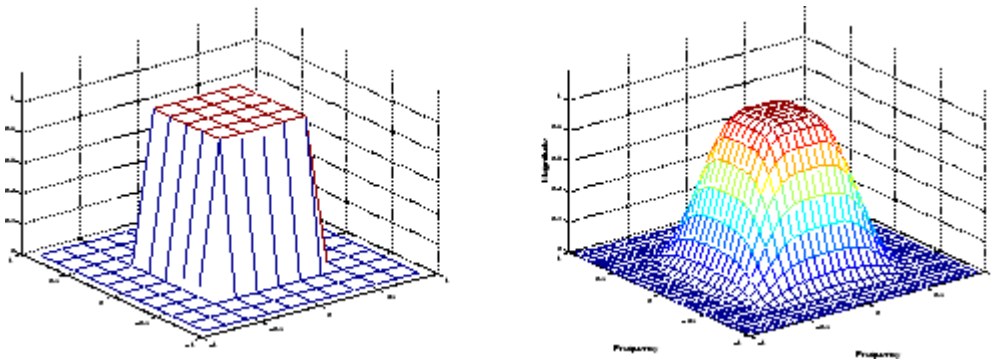
The toolbox provides two functions for window-based filter design, `fwind1` and `fwind2`. `fwind1` designs a two-dimensional filter by using a two-dimensional window that it creates from one or two one-dimensional windows that you specify. `fwind2` designs a two-dimensional filter by using a specified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

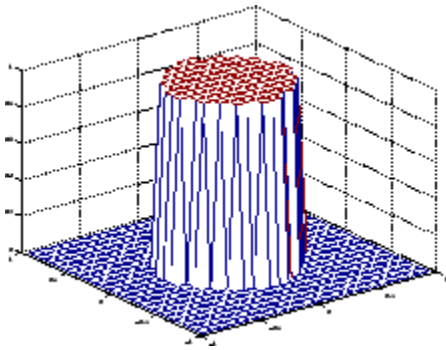
The filter design functions `fsamp2`, `fwind1`, and `fwind2` all create filters based on a desired frequency response magnitude matrix. Frequency response is a mathematical function describing the gain of a filter in response to different input frequencies.

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 low-pass 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 Low-pass Frequency Response

Note that for this frequency response, the filters produced by `fsamp2`, `fwind1`, and `fwind2` are real. This result is desirable for most image processing applications. To achieve this in general, the desired frequency response should be symmetric about the frequency origin (`f1 = 0`, `f2 = 0`).

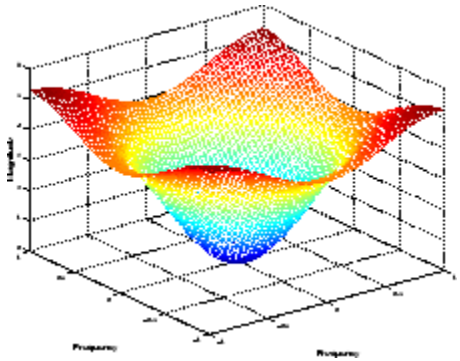
Computing the Frequency Response of a Filter

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

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

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

```
freqz2(h)
```



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.

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
- “Hough Transform” on page 9-21
- “Radon Transform” on page 9-27
- “Detect Lines Using the Radon Transform” on page 9-34
- “The Inverse Radon Transformation” on page 9-39
- “Fan-Beam Projection” on page 9-45
- “Reconstructing an Image from Projection Data” on page 9-55

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 $-\pi \le \omega_1,\omega_2 \le \pi$ is displayed. Note that $F(0,0)$ is the sum of all the values of $f(m,n)$. For this reason, $F(0,0)$ is often called the *constant component* or *DC component* of the Fourier transform. (DC stands for direct current; it is an electrical engineering term that refers to a constant-voltage power source, as opposed to a power source whose voltage varies sinusoidally.)

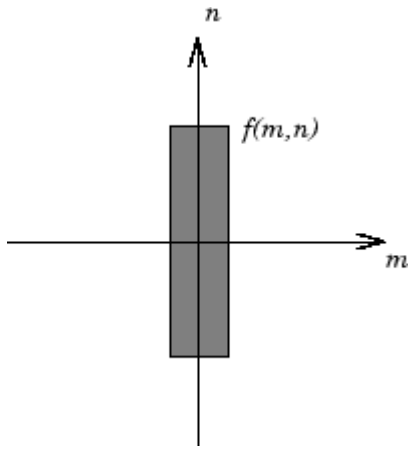
The inverse of a transform is an operation that when performed on a transformed image produces the original image. The inverse two-dimensional Fourier transform is given by

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

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.

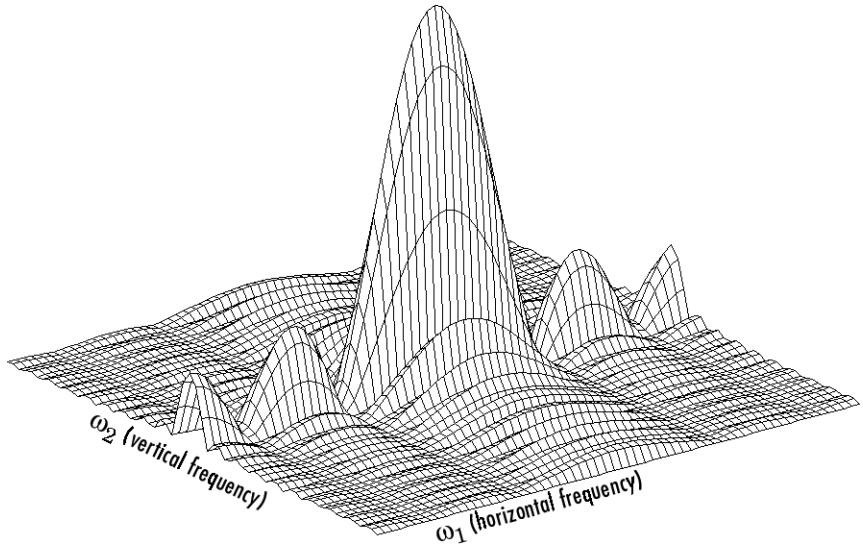


Rectangular Function

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.



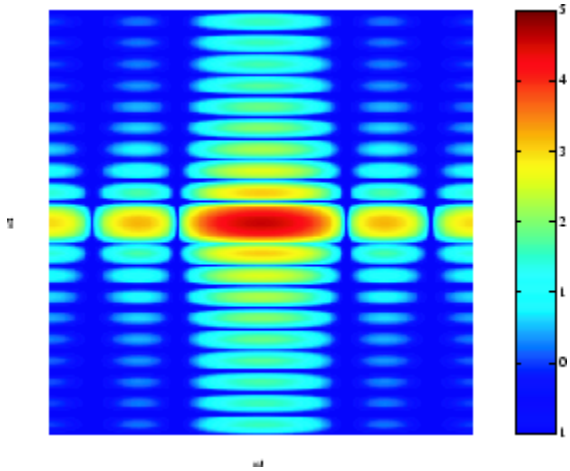
Magnitude Image of a Rectangular Function

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

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

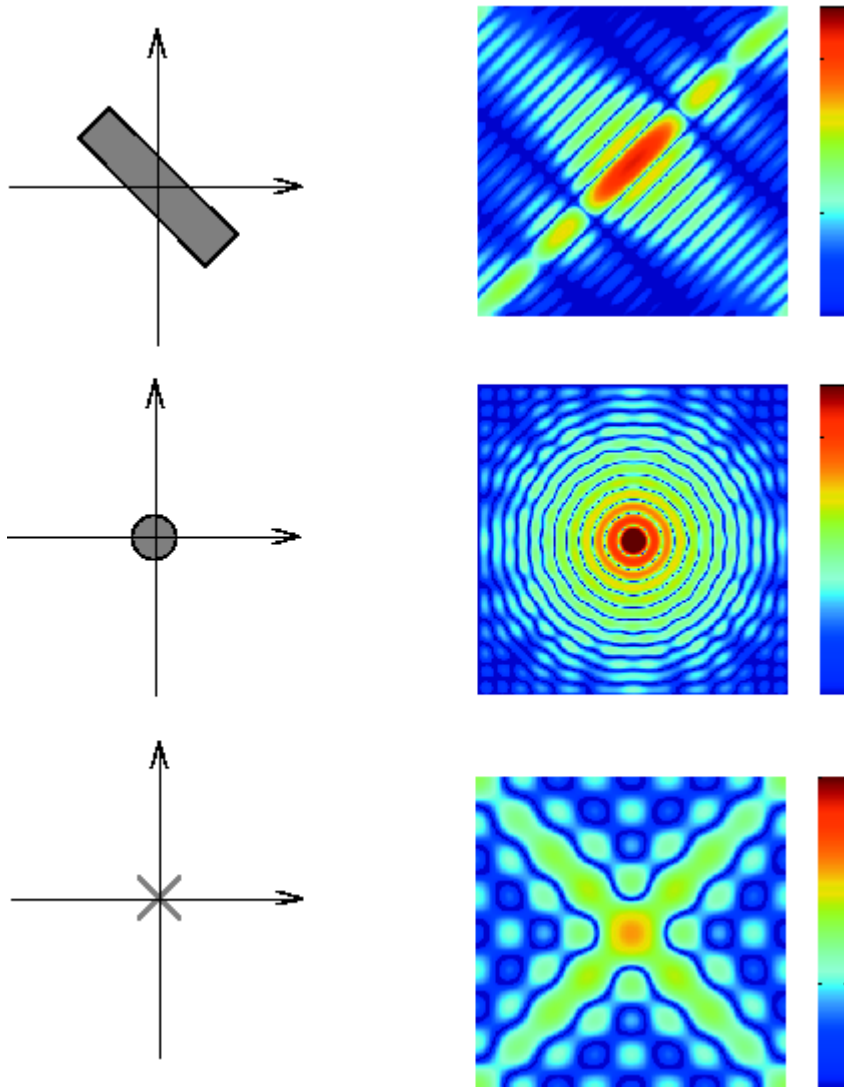
as an image, as shown.



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

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

and

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

The values $F(p,q)$ are the DFT coefficients of $f(m,n)$. 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

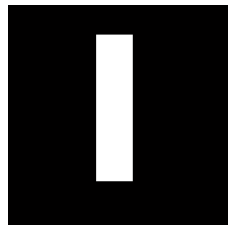
The DFT coefficients $F(p,q)$ are samples of the Fourier transform $F(\omega_1,\omega_2)$.

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

Visualizing the Discrete Fourier Transform

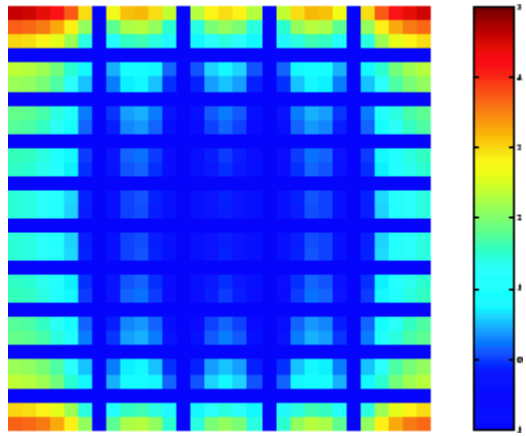
1. Construct a matrix `f` that is similar to the function *f*(*m*,*n*) in the example in "Definition of Fourier Transform" on page 9-2. Remember that *f*(*m*,*n*) is equal to 1 within the rectangular region and 0 elsewhere. Use a binary image to represent *f*(*m*,*n*).

```
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

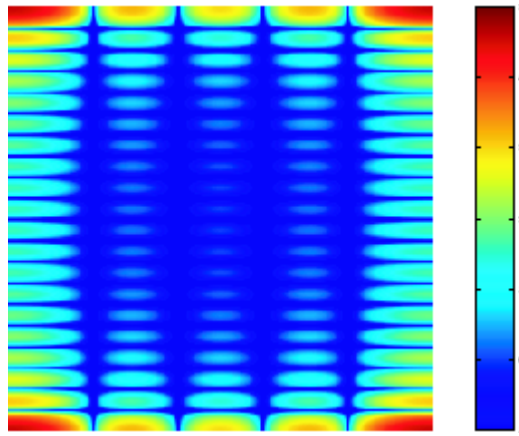
This plot differs from the Fourier transform displayed in “Visualizing the Fourier Transform” on page 9-3. First, the sampling of the Fourier transform is much coarser. Second, the zero-frequency coefficient is displayed in the upper left corner instead of the traditional location in the center.

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.

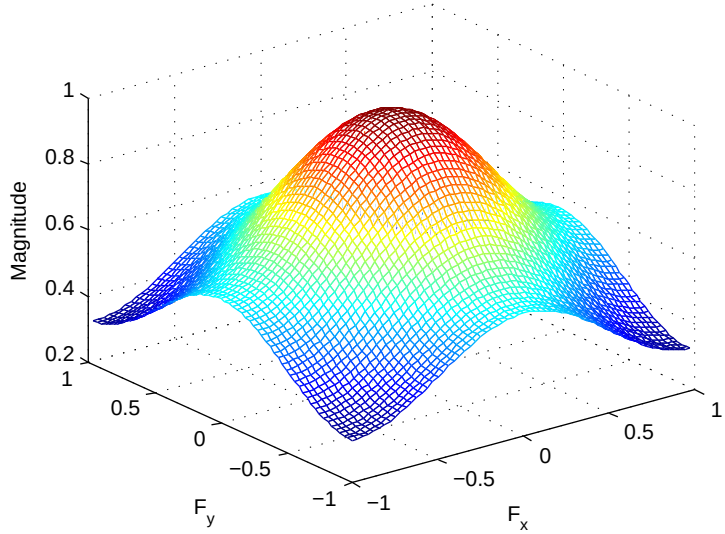
Applications of the Fourier Transform

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 “Design Linear Filters in the Frequency Domain” on page 8-56 for more information about linear filtering, filter design, and frequency responses.

Perform Fast Convolution Using the Fourier Transform

This example shows how to perform fast convolution of two matrices using the Fourier transform. 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 the `imfilter` function.

Create two simple matrices, `A` and `B`. `A` is an M-by-N matrix and `B` is a P-by-Q matrix.

```
A = magic(3);
B = ones(3);
```

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;
```

Compute the two-dimensional DFT of A and B using the `fft2` function. Multiply the two DFTs together and compute the inverse two-dimensional DFT of the result using the `ifft2` function.

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

Extract the nonzero portion of the result and remove the imaginary part caused by roundoff error.

```
C = C(1:5,1:5);
C = real(C)
```

```
C = 5×5

    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
```

Perform FFT-Based Correlation to Locate Image Features

This example shows how to use the Fourier transform 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*.

Read a sample image into the workspace.

```
bw = imread('text.png');
```

Create a template for matching by extracting the letter "a" from the image. Note that you can also create the template by using the interactive syntax of the `imcrop` function.

```
a = bw(32:45,88:98);
```

Compute the correlation of the template image with the original image by rotating the template image by 180 degrees and then using the FFT-based convolution technique.

(Convolution is equivalent to correlation if you rotate the convolution kernel by 180 degrees.) To match the template to the image, use the `fft2` and `ifft2` functions. In the resulting image, bright peaks correspond to occurrences of the letter.

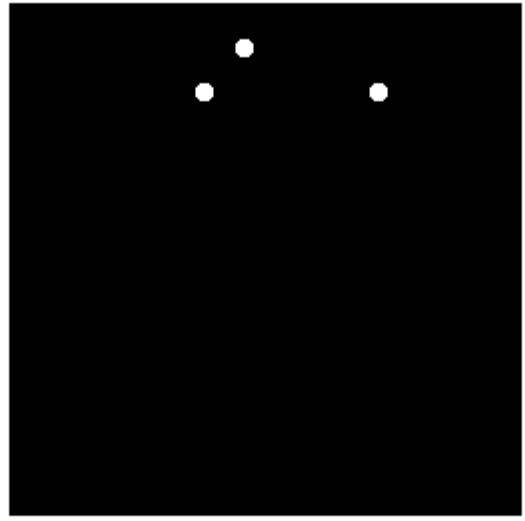
```
C = real(ifft2(fft2(bw) .* fft2(rot90(a,2),256,256)));
figure
imshow(C,[]) % Scale image to appropriate display range.
```

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 thresholded image shows the locations of these peaks as white spots in the thresholded correlation image. (To make the locations easier to see in this figure, the example dilates the thresholded image to enlarge the size of the points.)

```
max(C(:))
```

```
ans = 68.0000
```

```
thresh = 60; % Use a threshold that's a little less than max.
D = C > thresh;
se = strel('disk',5);
E = imdilate(D,se);
figure
imshow(E) % Display pixels with values over the threshold.
```



Discrete Cosine Transform

In this section...

"DCT Definition" on page 9-15
"The DCT Transform Matrix" on page 9-17
"Image Compression with the Discrete Cosine Transform" 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.

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

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

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

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

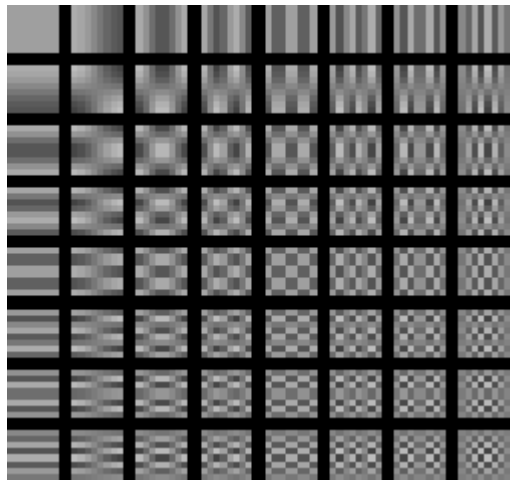
$$A_{mn} = \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} \alpha_p \alpha_q B_{pq} \cos\frac{\pi(2m+1)p}{2M} \cos\frac{\pi(2n+1)q}{2N}, \quad \begin{matrix} 0 \le m \le M-1 \\ 0 \le n \le N-1 \end{matrix}$$

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

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

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

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

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

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

Image Compression with the Discrete Cosine Transform

This example shows how to compress an image using the Discrete Cosine Transform (DCT). The example computes the two-dimensional DCT of 8-by-8 blocks in an 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 example uses the transform matrix computation method.

DCT is used 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.

Read an image into the workspace and convert it to class `double`.

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

Compute the two-dimensional DCT of 8-by-8 blocks in the image. The function `dctmtx` returns the N-by-N DCT transform matrix.

```
T = dctmtx(8);
dct = @(block_struct) T * block_struct.data * T';
B = blockproc(I,[8 8],dct);
```

Discard all but 10 of the 64 DCT coefficients in each block.

```
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);
```

Reconstruct the image using the two-dimensional inverse DCT of each block.

```
invdct = @(block_struct) T' * block_struct.data * T;
I2 = blockproc(B2,[8 8],invdct);
```

Display the original image and the reconstructed image, side-by-side. 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.

```
imshow(I)
```

```
figure
imshow(I2)
```

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*cos(theta) + y*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.

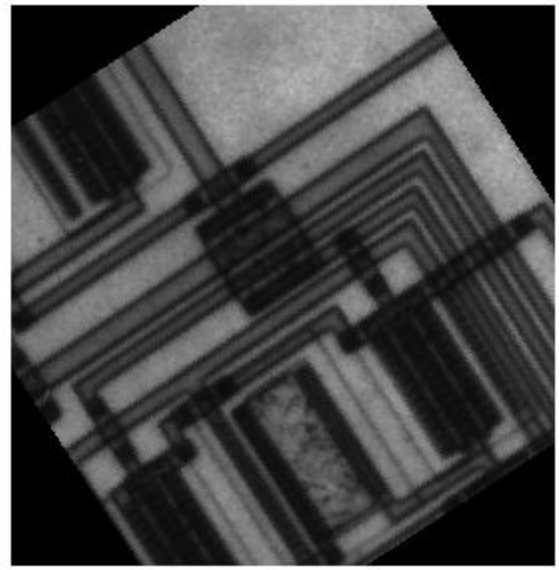
After you identify the peaks in the Hough transform, you can use the `houghlines` function to find the endpoints of the line segments corresponding to peaks in the Hough transform. This function automatically fills in small gaps in the line segments.

Detect Lines in Images Using Hough

This example shows how to detect lines in an image using the `Hough` transform.

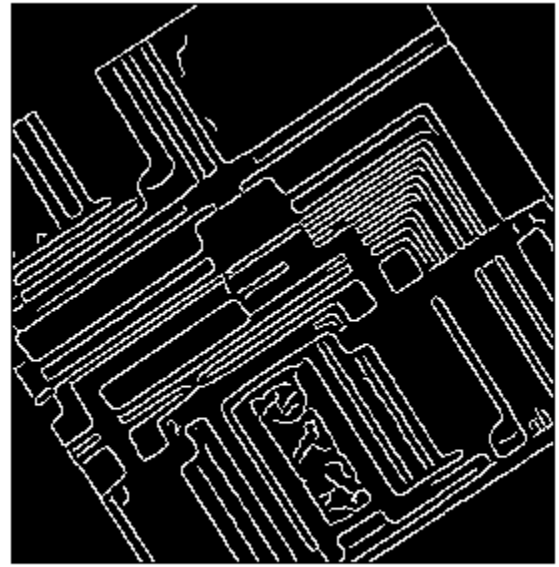
Read an image into the workspace and, to make this example more illustrative, rotate the image. Display the image.

```
I = imread('circuit.tif');
rotI = imrotate(I,33,'crop');
imshow(rotI)
```



Find the edges in the image using the `edge` function.

```
BW = edge(rotI,'canny');
imshow(BW);
```

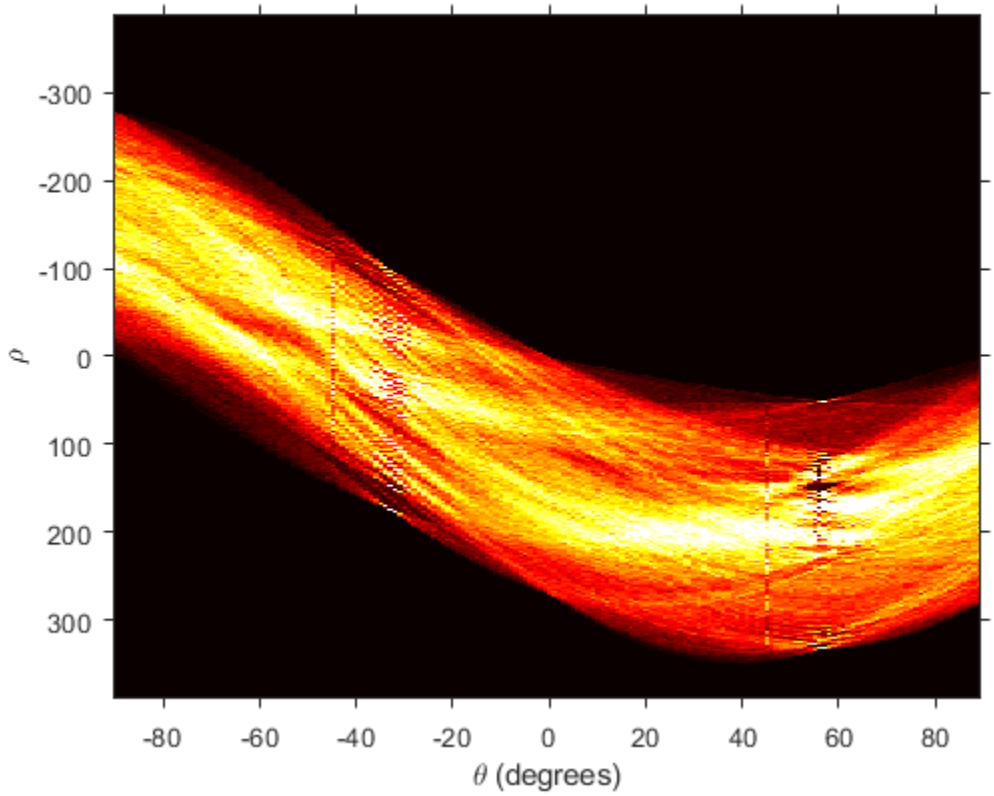


Compute the Hough transform of the binary image returned by `edge`.

```
[H,theta,rho] = hough(BW);
```

Display the transform, `H`, returned by the `hough` function.

```
figure
imshow(imadjust(rescale(H)),[],...
       'XData',theta,...
       'YData',rho,...
       'InitialMagnification','fit');
xlabel('\theta (degrees)')
ylabel('\rho')
axis on
axis normal
hold on
colormap(gca,hot)
```

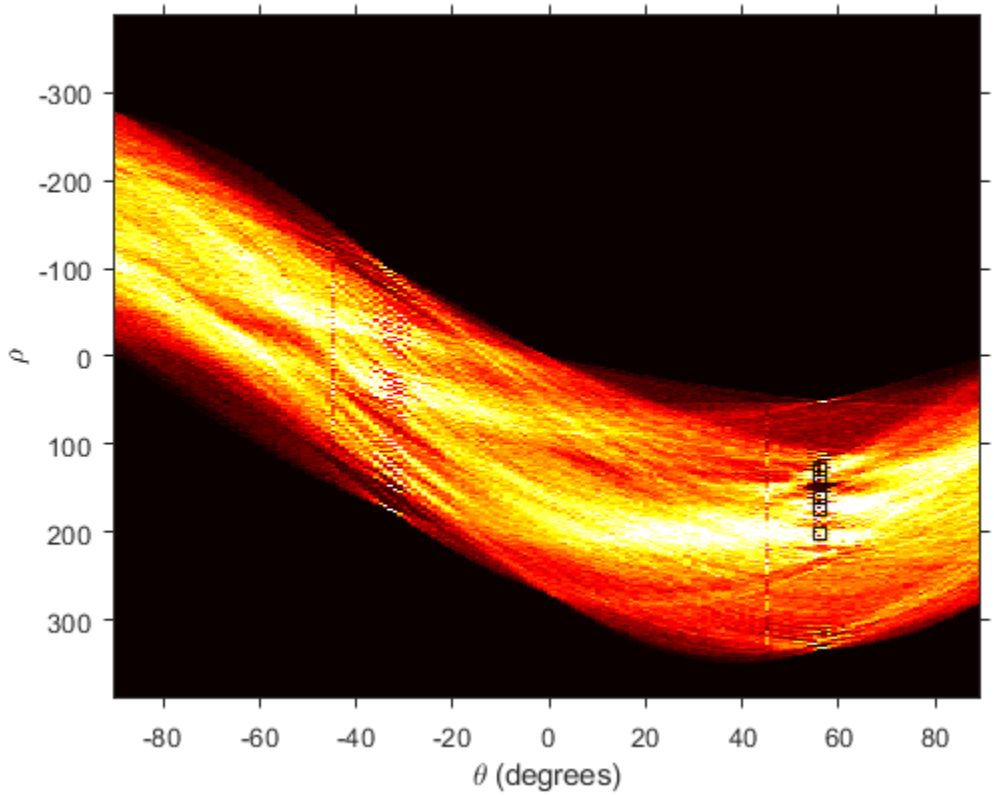


Find the peaks in the Hough transform matrix, H, using the houghpeaks function.

```
P = houghpeaks(H,5,'threshold',ceil(0.3*max(H(:))));
```

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');
```



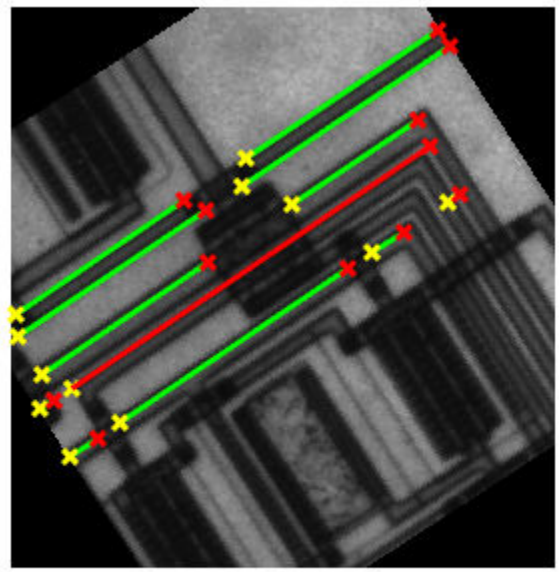
Find lines in the image using the `houghlines` function.

```
lines = houghlines(BW,theta,rho,P,'FillGap',5,'MinLength',7);
```

Create a plot that displays the original image with the lines superimposed on it.

```
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');
```



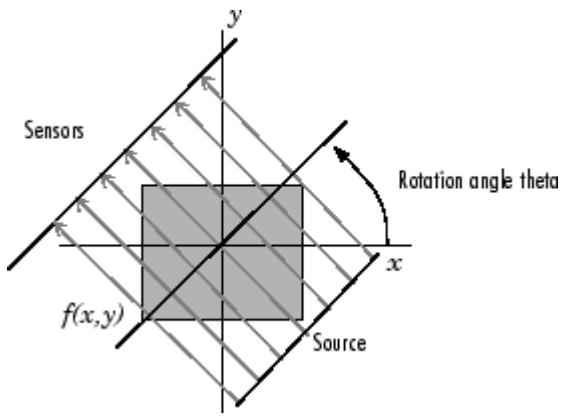
Radon Transform

In this section...
“Plot the Radon Transform of an Image” on page 9-29 “Viewing the Radon Transform as an Image” on page 9-32

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” on page 9-45. To convert parallel-beam projection data to fan-beam projection data, use the `para2fan` function.

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

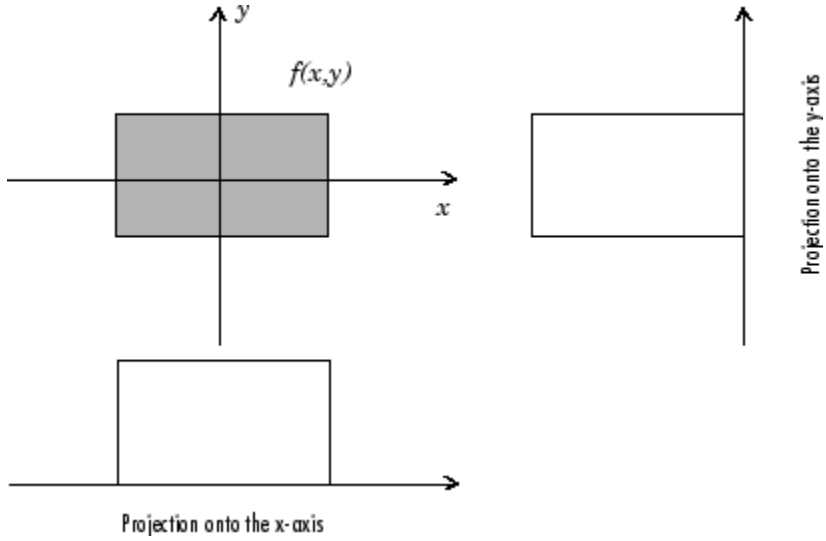
A projection of a two-dimensional function *f(x,y)* is a set of line integrals. The `radon` function computes the line integrals from multiple sources along parallel paths, or *beams*, in a certain direction. The beams are spaced 1 pixel unit apart. To represent an image, the `radon` function takes multiple, 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

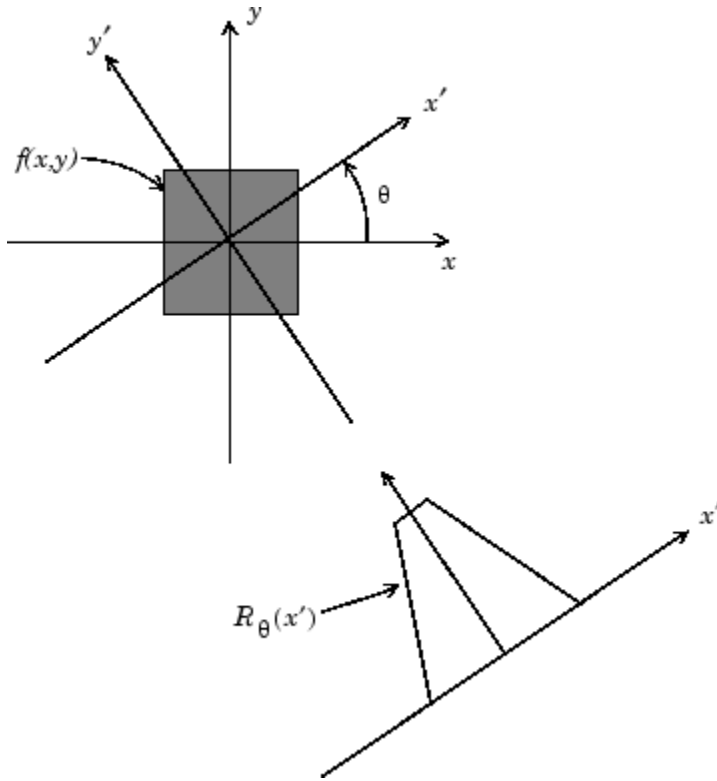
Projections can be computed along any angle theta (θ). In general, the Radon transform of *f(x,y)* is the line integral of *f* parallel to the *y´*-axis

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

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

Plot the Radon Transform of an Image

This example shows how to compute the Radon transform of an image, `I`, for a specific set of angles, `theta`, using the `radon` function. The function returns, `R`, in which the columns contain the Radon transform for each angle in `theta`. The function also returns the vector, `xp` , which contains the corresponding coordinates along the x-axis. The center pixel of `I` is defined to be `floor((size(I)+1)/2)`, which is the pixel on the x-axis corresponding to $x' = 0$.

Create a small sample image for this example that consists of a single square object and display it.

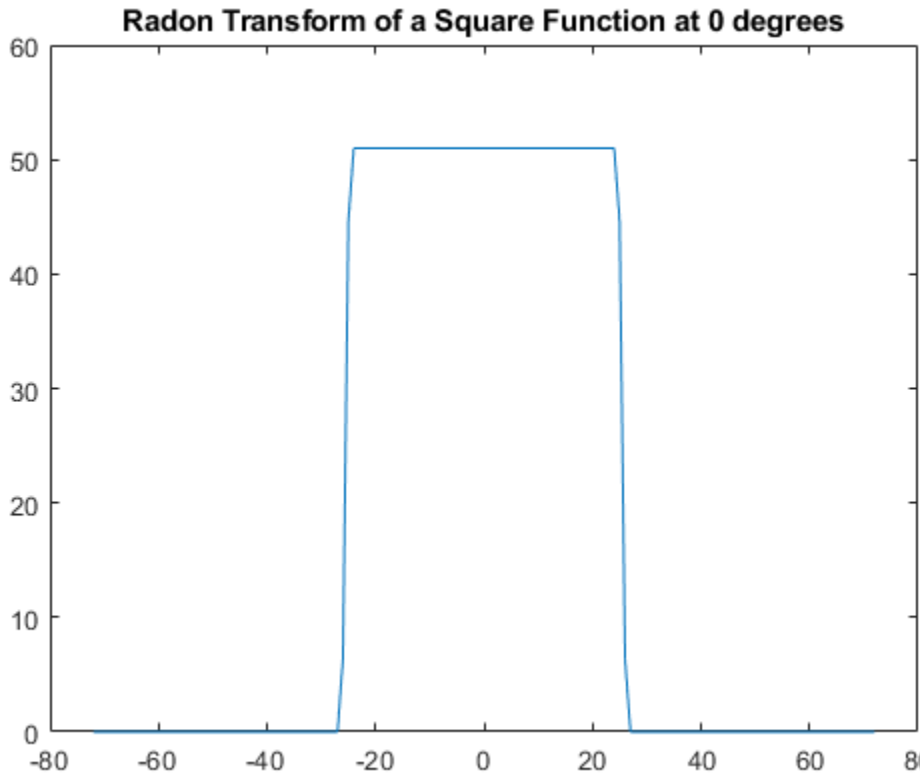
```
I = zeros(100,100);
I(25:75,25:75) = 1;
imshow(I)
```

Calculate the Radon transform of the image for the angles 0 degrees and 45 degrees.

```
[R,xp] = radon(I,[0 45]);
```

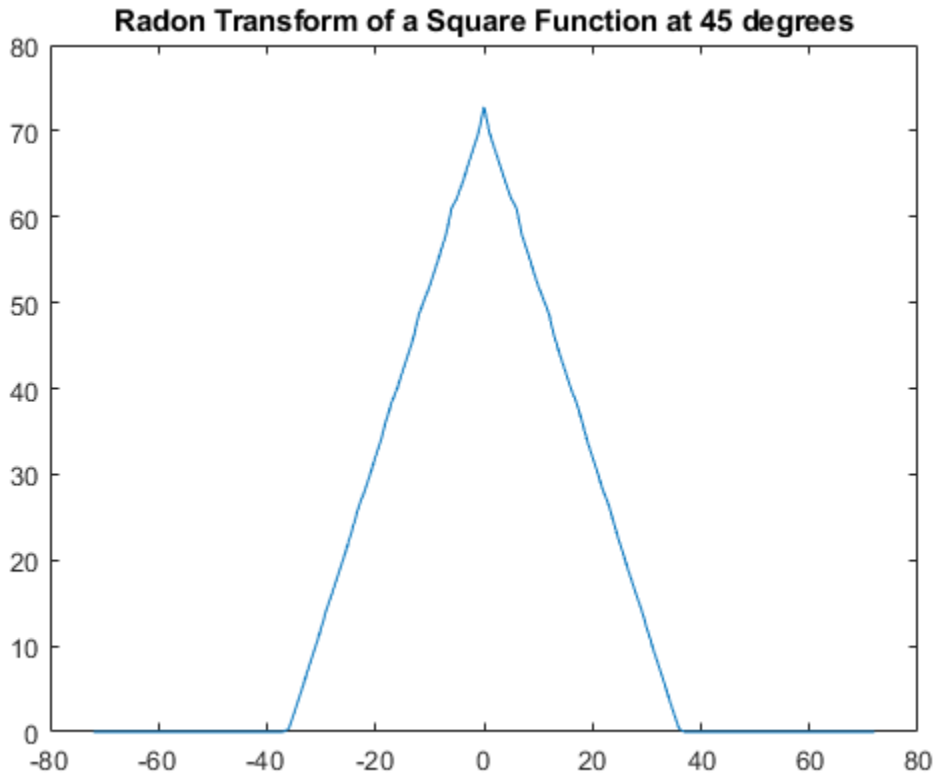
Plot the transform for 0 degrees.

```
figure
plot(xp,R(:,1));
title('Radon Transform of a Square Function at 0 degrees')
```



Plot the transform for 45 degrees.

```
figure
plot(xp,R(:,2));
title('Radon Transform of a Square Function at 45 degrees')
```

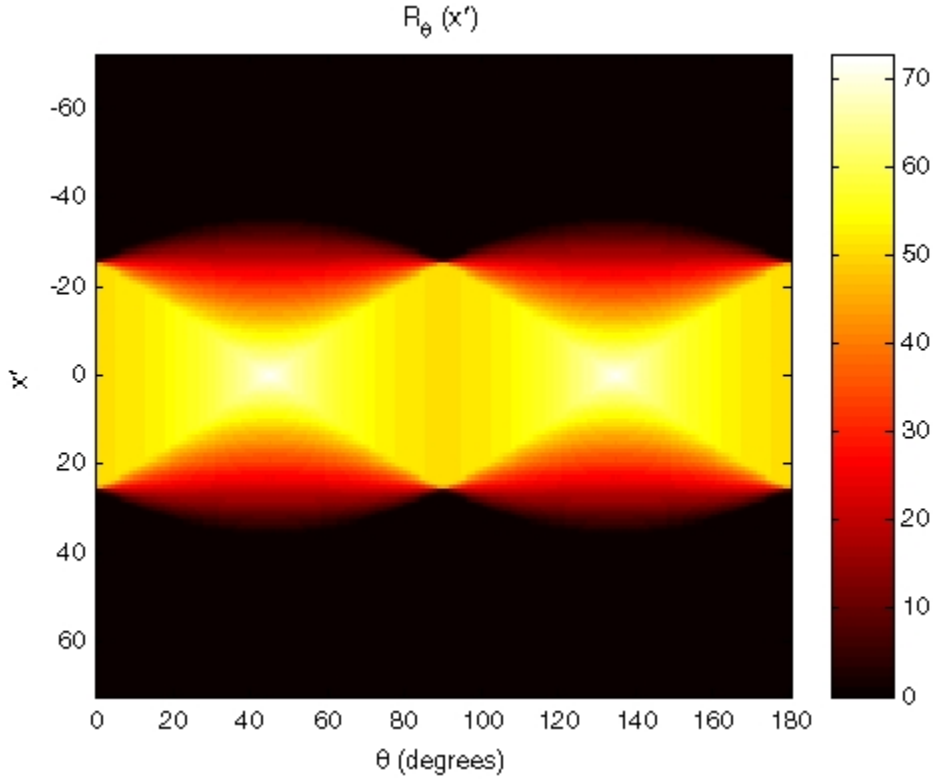


Viewing the Radon Transform as an Image

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

Detect Lines Using the Radon Transform

This example shows how to use the Radon transform to detect lines in an image. 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.

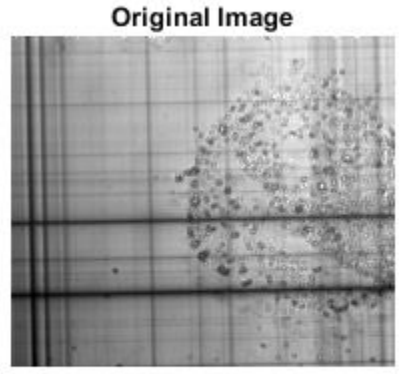
Compute the Radon Transform of an Image

Read an image into the workspace. Convert it into a grayscale image.

```
I = fitsread('solarspectra.fts');
I = rescale(I);
```

Display the original image.

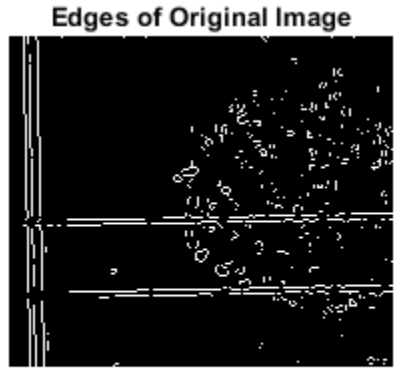
```
figure
imshow(I)
title('Original Image')
```



Compute a binary edge image using the `edge` function. Display the binary image returned by the `edge` function.

```
BW = edge(I);
figure
```

```
imshow(BW)
title('Edges of Original Image')
```

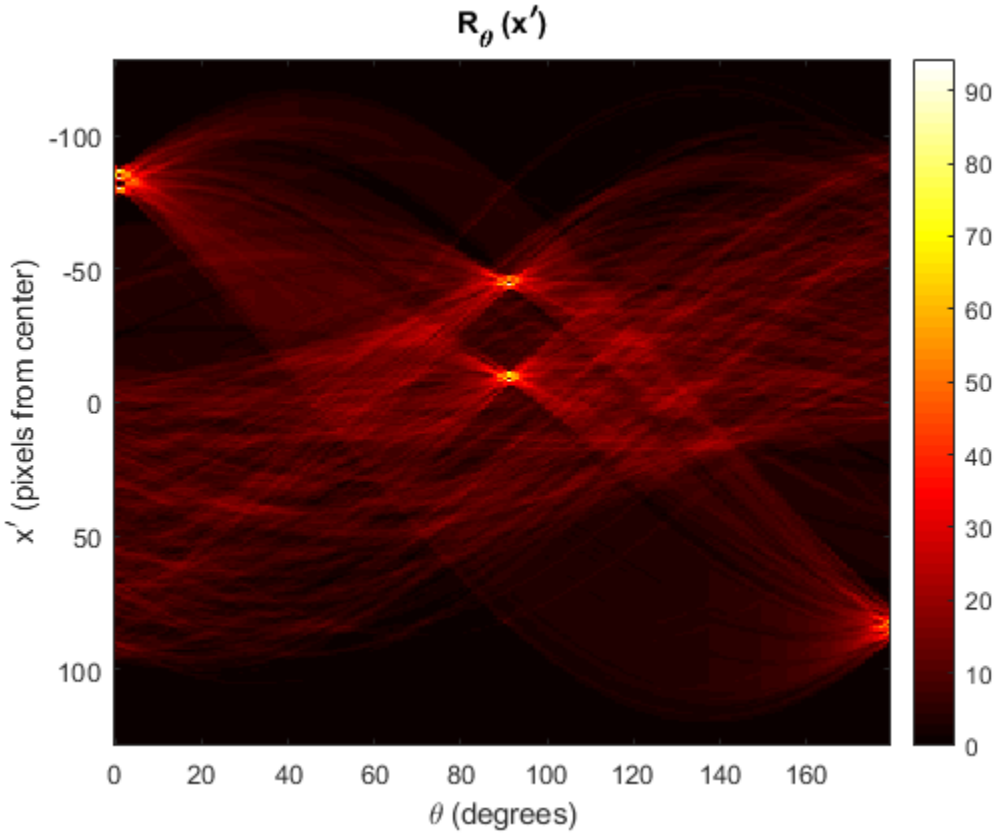


Calculate the radon transform of the image, using the `radon` function, and display the transform. The locations of peaks in the transform correspond to the locations of straight lines in the original image.

```
theta = 0:179;
[R,xp] = radon(BW,theta);
```

Display the result of the radon transform.

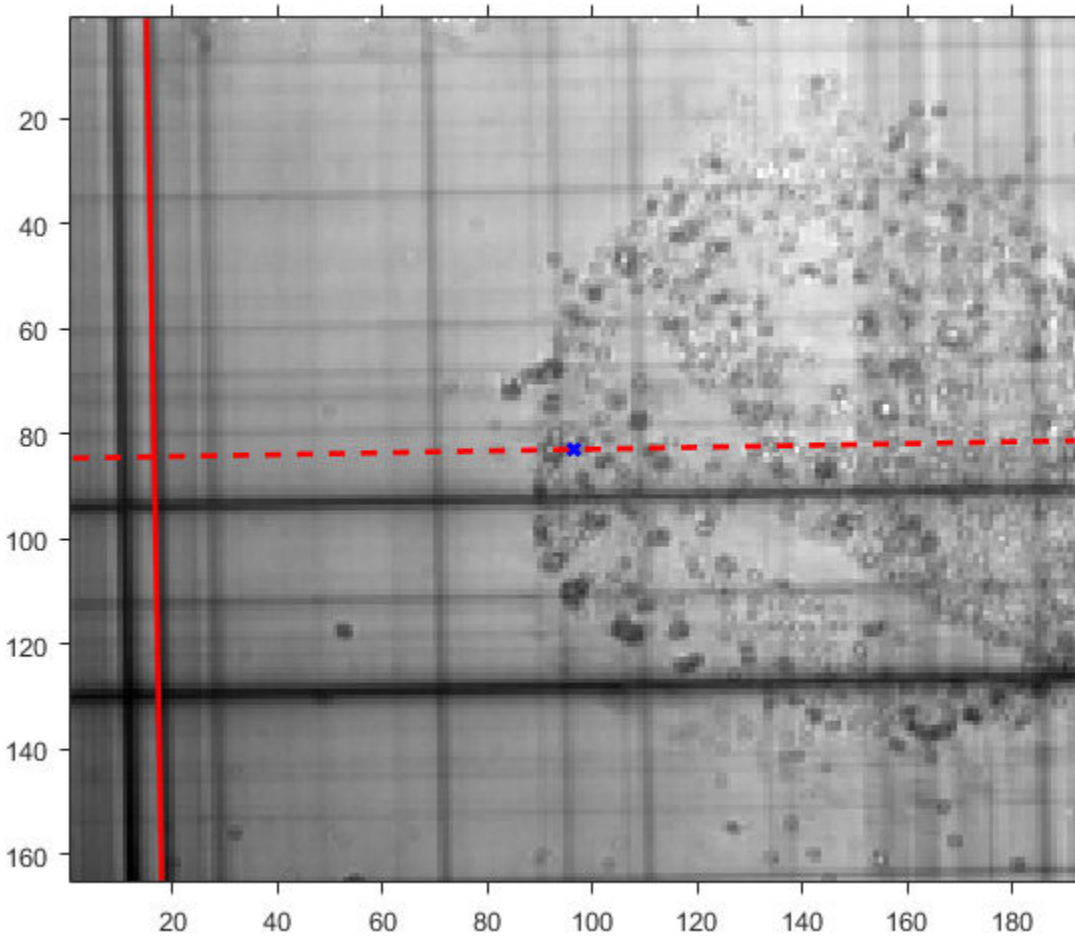
```
figure
imagesc(theta, xp, R); colormap(hot);
xlabel('\theta (degrees)');
ylabel('x^{\prime} (pixels from center)');
title('R_{\theta} (x^{\prime})');
colorbar
```



The strongest peak in R corresponds to $\theta = 1$ degree and x' = -80 pixels from center.

Interpreting the Peaks of the Radon Transform

To visualize this peak in the original figure, find the center of the image, indicated by the blue cross overlaid on the image below. The red dashed line is the radial line that passes through the center at an angle $\theta = 1$ degree. If you travel along this line -80 pixels from center (towards the left), the radial line perpendicularly intersects the solid red line. This solid red line is the straight line with the strongest signal in the Radon transform.

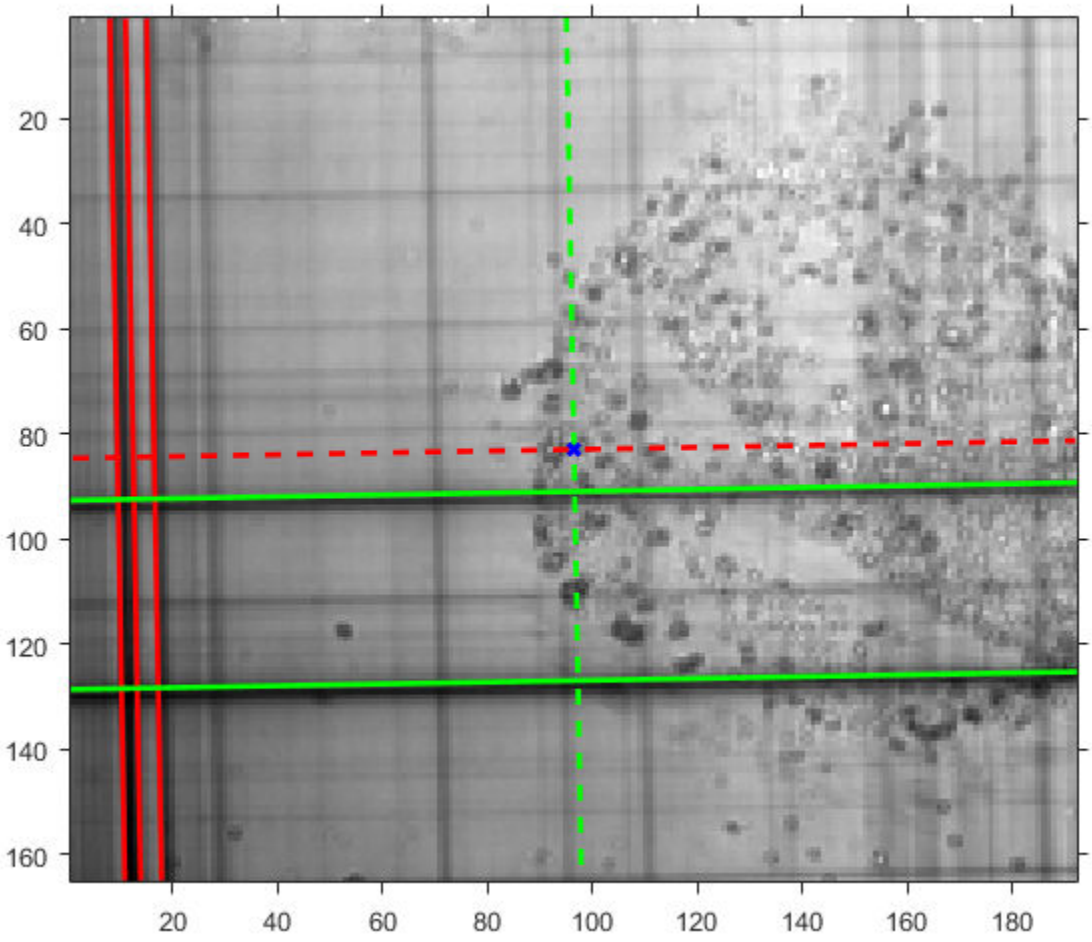


To interpret the Radon transform further, examine the next four strongest peaks in R.

Two strong peaks in R are found at $\theta = 1$ degree, at offsets of -84 and -87 pixels from center. These peaks correspond to the two red lines to the left of the strongest line, overlaid on the image below.

Two other strong peaks are found near the center of R. These peaks are located at $\theta = 91$ degrees, with offsets of -8 and -44 pixels from center. The green dashed line in the image

below is the radial line passing through the center at an angle of 91 degrees. If you travel along the radial line a distance of -8 and -44 pixels from center, then the radial line perpendicularly intersects the solid green lines. These solid green lines correspond to the strong peaks in R.



The fainter lines in the image relate to the weaker peaks in R.

The Inverse Radon Transformation

In this section...

"Inverse Radon Transform Definition" on page 9-39
"Reconstructing an Image from Parallel Projection Data" on page 9-41

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 9-27, 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`.

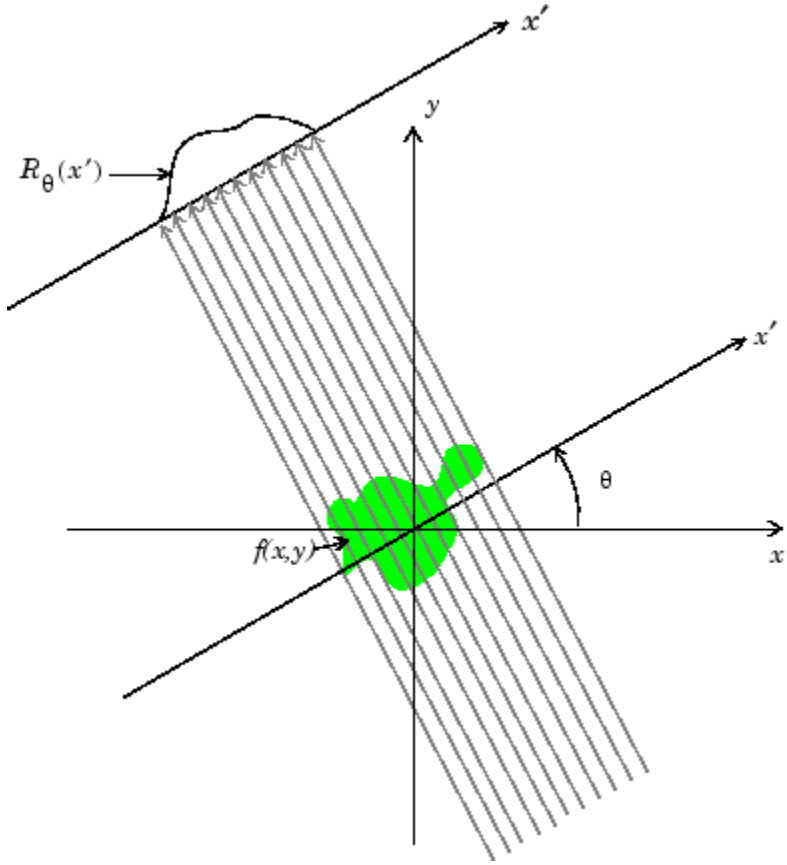
Note, however, that in most application areas, there is no original image from which projections are formed. For example, the inverse Radon transform is commonly used in tomography applications. In X-ray absorption tomography, projections are formed by measuring the attenuation of radiation that passes 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 9-27. *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” on page 9-45. To

convert parallel-beam projection data into fan-beam projection data, use the `para2fan` function.

Improving the Results

`iradon` uses the *filtered back projection* 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 back projection 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 back projection 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.

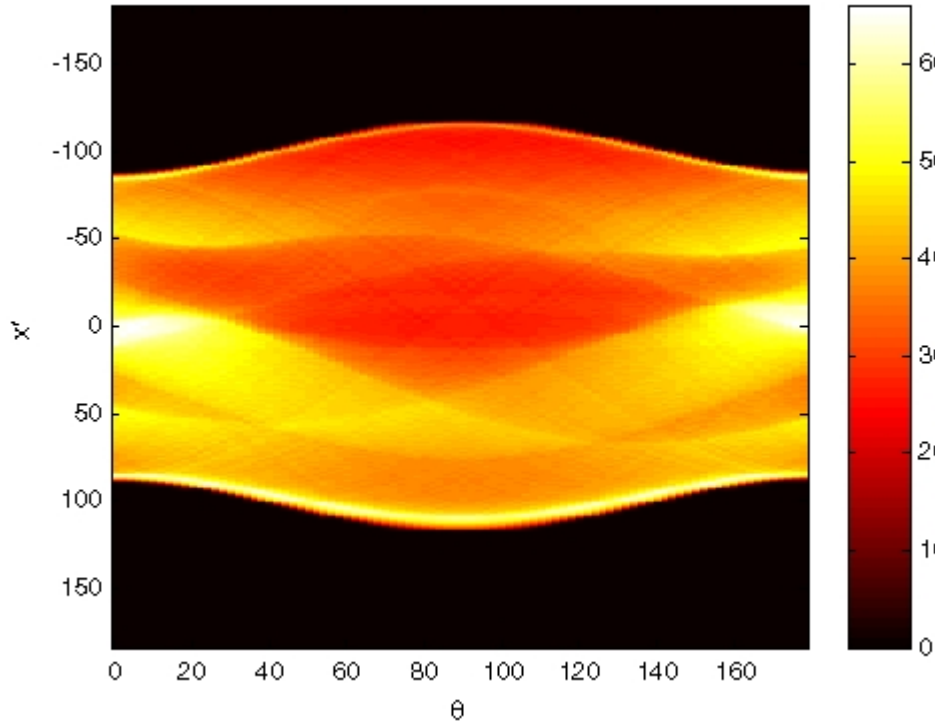
```
P = phantom(256);
imshow(P)
```

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

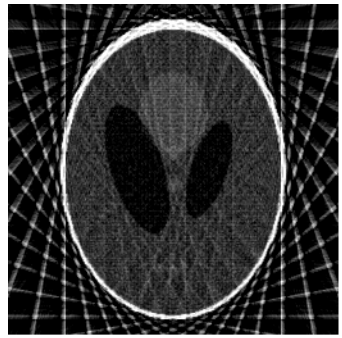
Note how some of the features of the input image appear in this image of the transform. The first column in the Radon transform corresponds to a projection at 0º that is integrating in the vertical direction. The centermost column corresponds to a projection at 90º, which is integrating in the horizontal direction. The projection at 90º has a wider profile than the projection at 0º 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.

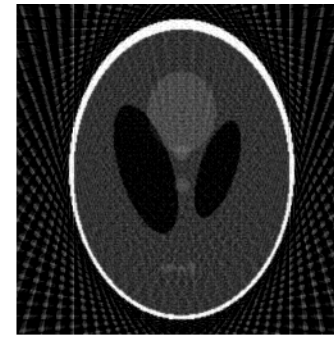
```
I1 = iradon(R1,10);
I2 = iradon(R2,5);
I3 = iradon(R3,2);
imshow(I1)
```

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

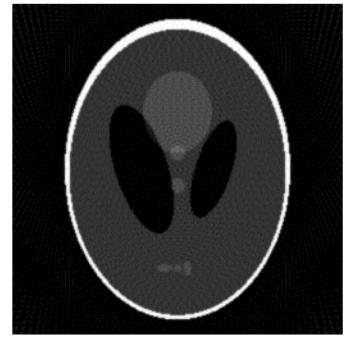
The following figure shows the results of all three reconstructions. Notice how image `I1`, which was reconstructed from only 18 projections, is the least accurate reconstruction. Image `I2`, which was reconstructed from 36 projections, is better, but it is still not clear enough to discern clearly the 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

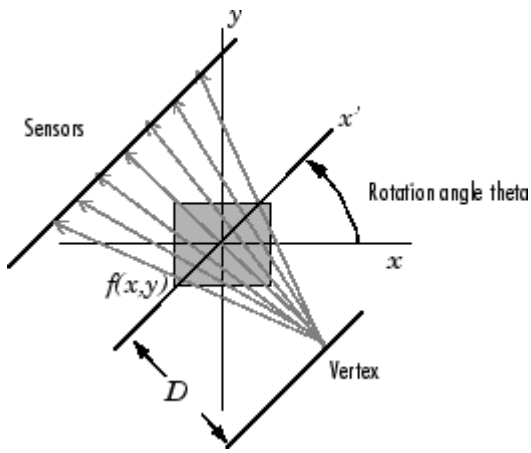
Inverse Radon Transforms of the Shepp-Logan Head Phantom

Fan-Beam Projection

In this section...
“Image Reconstruction from Fan-Beam Projection Data” on page 9-48 “Reconstruct Image using Inverse Fanbeam Projection” on page 9-49

Note For information about creating projection data from line integrals along parallel paths, see “Radon Transform” on page 9-27. To convert fan-beam projection data to parallel-beam projection data, use the `fan2para` function.

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

When you compute fan-beam projection data using 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

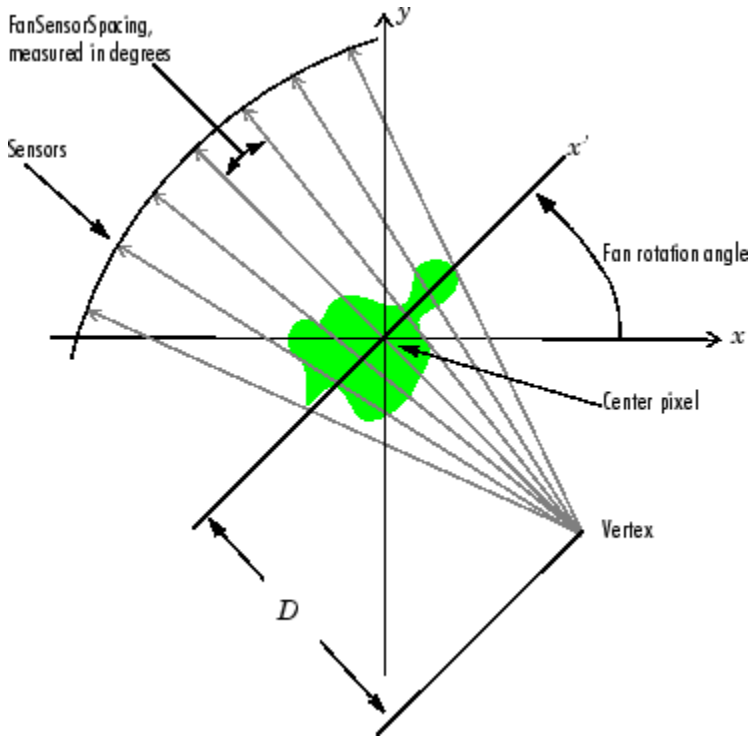
the number of beams, based on the size of the image and the settings of `fanbeam` parameters.

The `FanSensorGeometry` parameter specifies how sensors are aligned: `'arc'` or `'line'`.

Fan Sensor Geometry	Description
`'arc'`	`fanbeam` positions the sensors along an arc, spacing the sensors at 1 degree intervals. Use the `FanSensorSpacing` parameter to control the distance between sensors by specifying the angle between each beam. This is the default fan sensor geometry.
`'line'`	`fanbeam` positions sensors along a straight line, rather than an arc. Use the `FanSensorSpacing` parameter to specify the distance between the sensors, in pixels, along the x' axis.

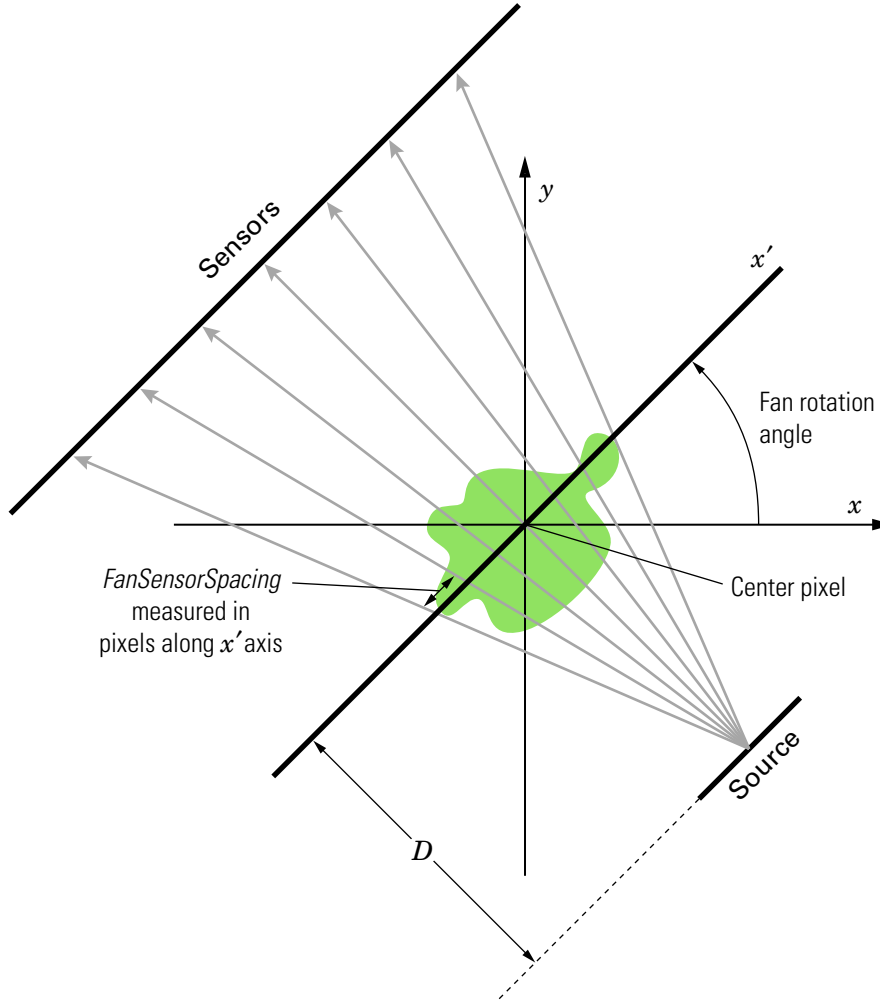
The `FanRotationIncrement` parameter specifies the rotation angle increment. By default, `fanbeam` takes projections at different angles by rotating the source around the center pixel at 1 degree intervals.

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

Image Reconstruction 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 `ifanbeam`.

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 `ifanfeam` function supports certain `iradon` parameters, which it passes to the `iradon` function. See “The Inverse Radon Transformation” on page 9-39 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)
```

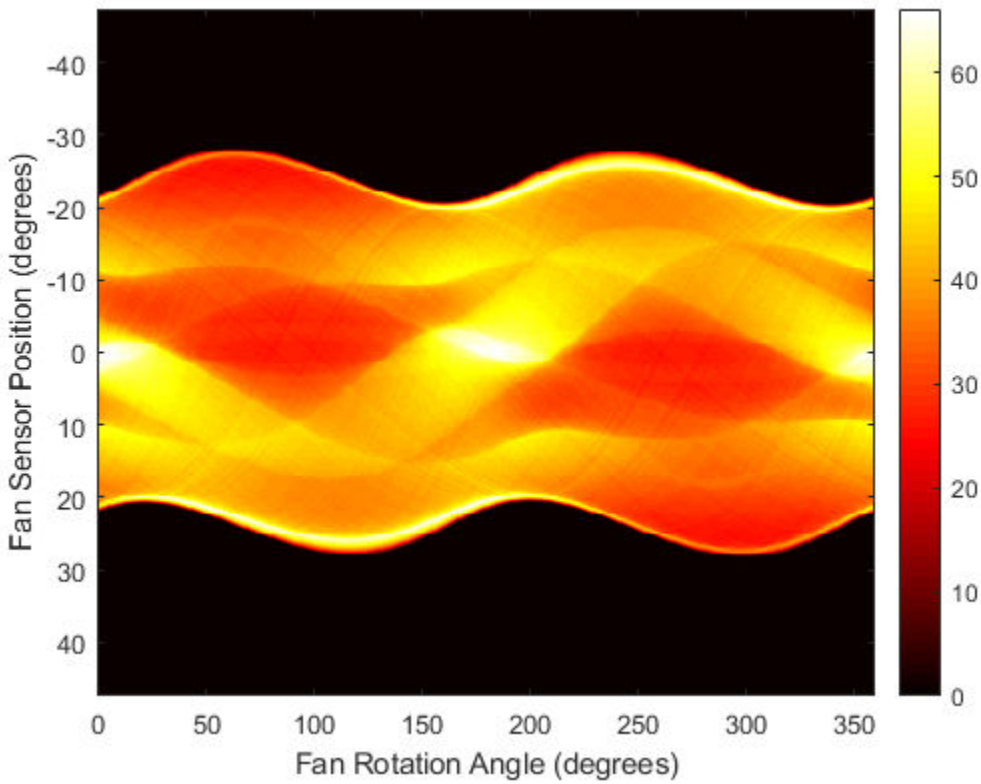
Compute fan-beam projection data of the test image, using the `FanSensorSpacing` parameter to vary the sensor spacing. The example uses the fanbeam arc geometry, so you specify the spacing between sensors by specifying the angular spacing of the beams. The first call spaces the beams at 2 degrees; the 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.

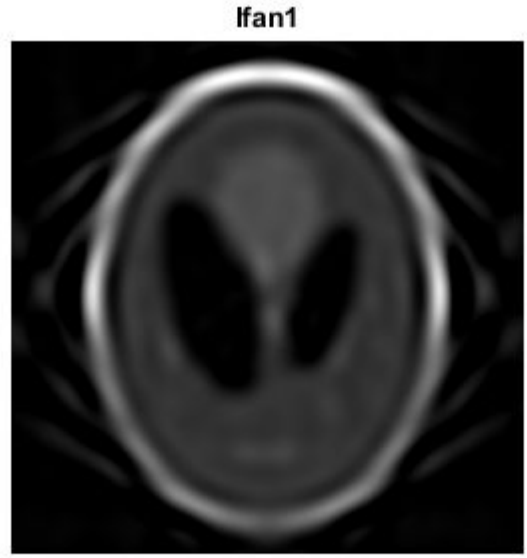
```
figure, imagesc(fan_rot_angles3, sensor_pos3, F3)
colormap(hot); colorbar
xlabel('Fan Rotation Angle (degrees)')
ylabel('Fan Sensor Position (degrees)')
```



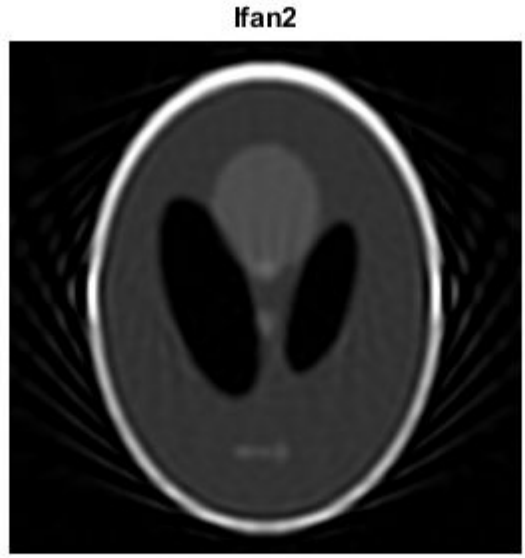
Reconstruct the image from the fan-beam projection data using `ifanbeam` . In each reconstruction, match the fan sensor spacing with the spacing used when the projection

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.

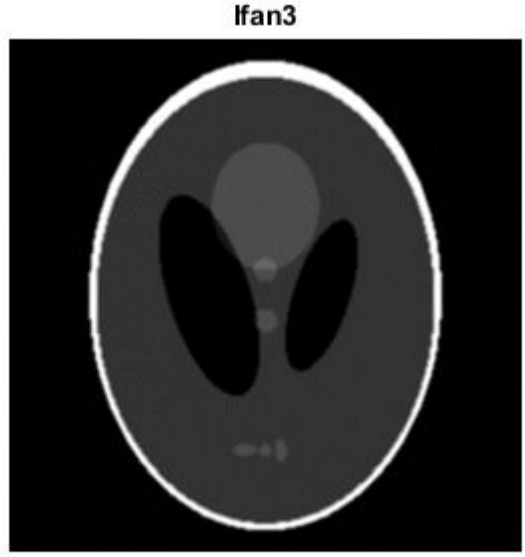
```
output_size = max(size(P));

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')
```


Ifan3

Reconstructing an Image from Projection Data

This example shows how to use `radon`, `iradon`, `fanbeam`, and `ifanbeam` to form projections from a sample image and then reconstruct the image from the projections. While `radon` and `iradon` use a parallel-beam geometry for the projections, `fanbeam` and `ifanbeam` use a fan-beam geometry. To compare parallel-beam and fan-beam geometries, the examples below create synthetic projections for each geometry and then use those synthetic projections to reconstruct the original image.

A real-world application that requires image reconstruction is X-ray absorption tomography where 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 by special medical imaging devices and then an internal image of the specimen is reconstructed using `iradon` or `ifanbeam`.

The function `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 function `ifanbeam` reconstructs an image from fan-beam projections, which have one emitter and multiple sensors.

See the Image Processing Toolbox™ User's Guide for diagrams that illustrate both geometries.

Create Head Phantom

The test image is the Shepp-Logan head phantom which can be generated using the function `phantom`. The phantom image illustrates many qualities that are found in real-world tomographic imaging of human heads. The bright elliptical shell along the exterior is analogous to a skull and the many ellipses inside are analogous to brain features or tumors.

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

Parallel Beam - Calculate Synthetic Projections

Calculate synthetic projections using parallel-beam geometry and vary the number of projection angles. For each of these calls to `radon`, the output is a matrix in which each column is the Radon transform for one of the angles in the corresponding `theta`.

```
theta1 = 0:10:170;
[R1,~] = radon(P,theta1);
num_angles_R1 = size(R1,2)
```

```
num_angles_R1 = 18
```

```
theta2 = 0:5:175;
[R2,~] = radon(P,theta2);
num_angles_R2 = size(R2,2)
```

```
num_angles_R2 = 36
```

```
theta3 = 0:2:178;
[R3,xp] = radon(P,theta3);
num_angles_R3 = size(R3,2)
```

```
num_angles_R3 = 90
```

Note that for each angle, the projection is computed at **N** points along the xp-axis, where **N** is a constant that depends on the diagonal distance of the image such that every pixel will be projected for all possible projection angles.

```
N_R1 = size(R1,1)
```

```
N_R1 = 367
```

```
N_R2 = size(R2,1)
```

```
N_R2 = 367
```

```
N_R3 = size(R3,1)
```

```
N_R3 = 367
```

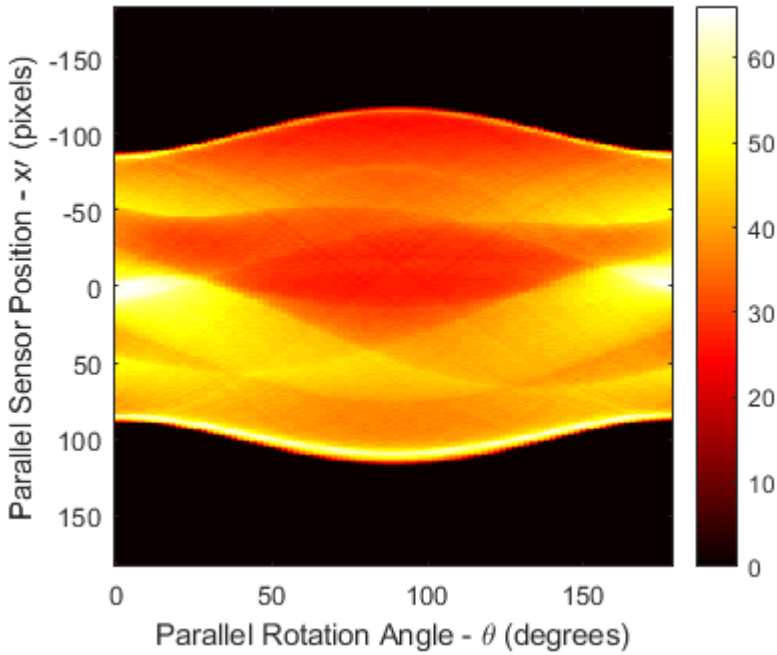
So, if you use a smaller head phantom, the projection needs to be computed at fewer points along the xp-axis.

```
P_128 = phantom(128);
[R_128,xp_128] = radon(P_128,theta1);
N_128 = size(R_128,1)
```

```
N_128 = 185
```

Display the projection data R3. Some of the features of the original phantom image are visible in the image of R3. The first column of R3 corresponds to a projection at 0 degrees, which is integrating in the vertical direction. The centermost column corresponds to a projection at 90 degrees, which is integrating in the horizontal directions. The projection at 90 degrees has a wider profile than the projection at 0 degrees due to the large vertical semi-axis of the outermost ellipse of the phantom.

```
imagesc(theta3,xp,R3)
colormap(hot)
colorbar
xlabel('Parallel Rotation Angle - \theta (degrees)');
ylabel('Parallel Sensor Position - x\prime (pixels)');
```



Parallel Beam - Reconstruct Head Phantom from Projection Data

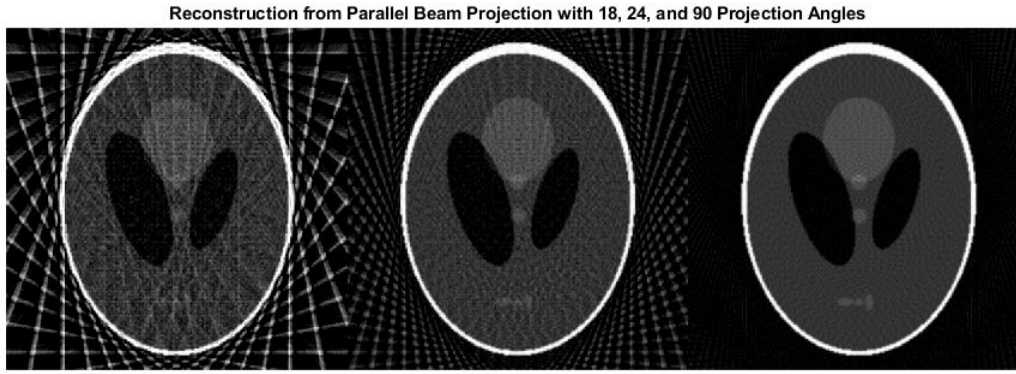
Match the parallel rotation-increment, `dtheta`, in each reconstruction with that used above to create the corresponding synthetic projections. In a real-world case, you would know the geometry of your transmitters and sensors, but not the source image, `P`.

The following three reconstructions (`I1`, `I2`, and `I3`) show the effect of varying the number of angles at which projections are made. For `I1` and `I2` some features that were visible in the original phantom are not clear. Specifically, look at the three ellipses at the bottom of each image. The result in `I3` closely resembles the original image, `P`.

Notice the significant artifacts present in `I1` and `I2`. To avoid these artifacts, use a larger number of angles.

```
% Constrain the output size of each reconstruction to be the same as the
% size of the original image, |P|.
output_size = max(size(P));
```

```
dtheta1 = theta1(2) - theta1(1);
I1 = iradon(R1,dtheta1,output_size);

dtheta2 = theta2(2) - theta2(1);
I2 = iradon(R2,dtheta2,output_size);

dtheta3 = theta3(2) - theta3(1);
I3 = iradon(R3,dtheta3,output_size);

figure
montage({I1,I2,I3},'Size',[1 3])
title('Reconstruction from Parallel Beam Projection with 18, 24, and 90 Projection Angl
```



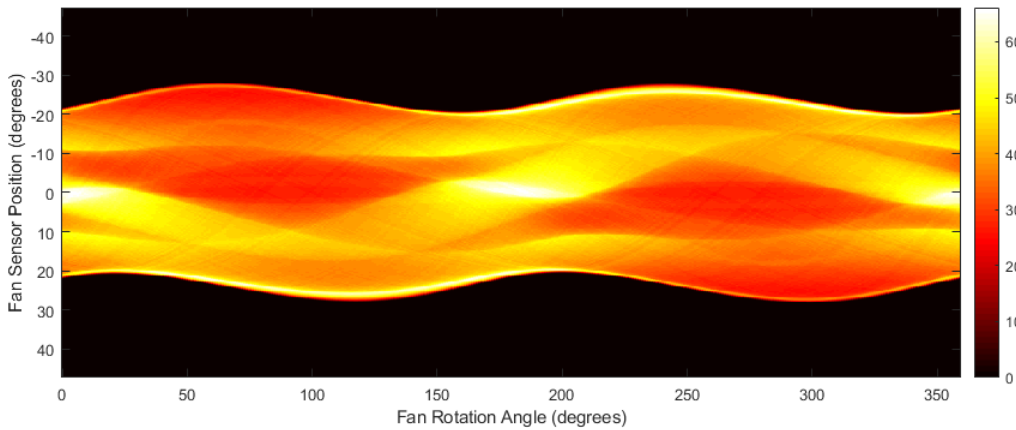
Fan Beam - Calculate Synthetic Projections

Calculate synthetic projections using fan-beam geometry and vary the 'FanSensorSpacing'.

```
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);
```

Display the projection data F3. Notice that the fan rotation angles range from 0 to 360 degrees and the same patterns occur at an offset of 180 degrees because the same features are being sampled from both sides. You can correlate features in this image of fan-beam projections with the same features in the image of parallel-beam projections, above.

```
imagesc(fan_rot_angles3, sensor_pos3, F3)
colormap(hot)
colorbar
xlabel('Fan Rotation Angle (degrees)')
ylabel('Fan Sensor Position (degrees)')
```



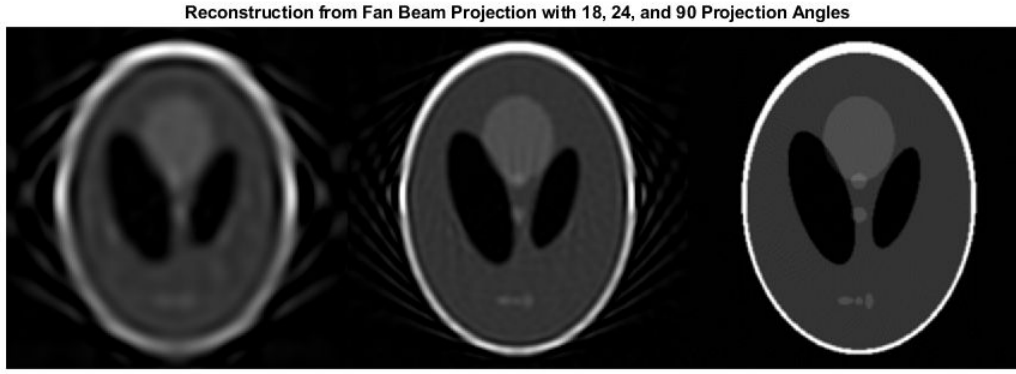
Fan Beam - Reconstruct Head Phantom from Projection Data

Match the fan-sensor-spacing in each reconstruction with that used to create each of the synthetic projections. In a real-world case, you would know the geometry of your transmitters and sensors, but not the source image, P.

Changing the value of the 'FanSensorSpacing' effectively changes the number of sensors used at each rotation angle. For each of these fan-beam reconstructions, the same rotation angles are used. This is in contrast to the parallel-beam reconstructions which each used different rotation angles.

Note that 'FanSensorSpacing' is only one parameter of several that you can control when using `fanbeam` and `ifanbeam`. You can also convert back and forth between parallel- and fan-beam projection data using the functions `fan2para` and `para2fan`.

```
Ifan1 = ifanbeam(F1,D,'FanSensorSpacing',dsensor1,'OutputSize',output_size);
Ifan2 = ifanbeam(F2,D,'FanSensorSpacing',dsensor2,'OutputSize',output_size);
Ifan3 = ifanbeam(F3,D,'FanSensorSpacing',dsensor3,'OutputSize',output_size);

figure
montage({Ifan1,Ifan2,Ifan3},'Size',[1 3])
title('Reconstruction from Fan Beam Projection with 18, 24, and 90 Projection Angles')
```



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.

- “Morphological Dilation and Erosion” on page 10-2
- “Structuring Elements” on page 10-5
- “Dilate an Image to Enlarge a Shape” on page 10-10
- “Erode an Image To Remove Thin Lines” on page 10-15
- “Operations That Combine Dilation and Erosion” on page 10-17
- “Skeletonization” on page 10-23
- “Find Object Perimeters” on page 10-25
- “Pixel Connectivity” on page 10-27
- “Understanding Morphological Reconstruction” on page 10-31
- “Flood-Fill Operations” on page 10-43
- “Detecting a Cell Using Image Segmentation” on page 10-46
- “Granulometry of Snowflakes” on page 10-53
- “Distance Transform of a Binary Image” on page 10-58
- “Label and Measure Objects in a Binary Image” on page 10-60
- “Lookup Table Operations” on page 10-65

Morphological Dilation and Erosion

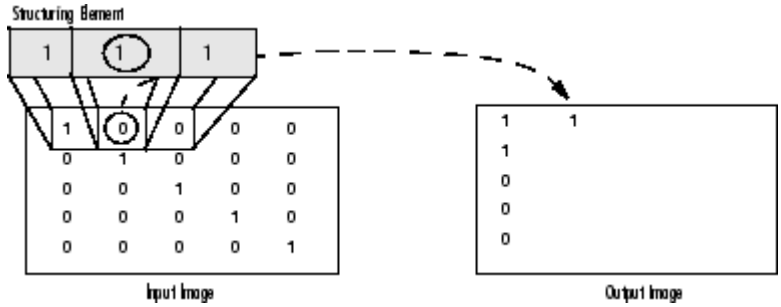
Morphology is a broad set of image processing operations that process images based on shapes. Morphological operations apply a structuring element to an input image, creating an output image of the same size. In a morphological operation, the value of each pixel in the output image is based on a comparison 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

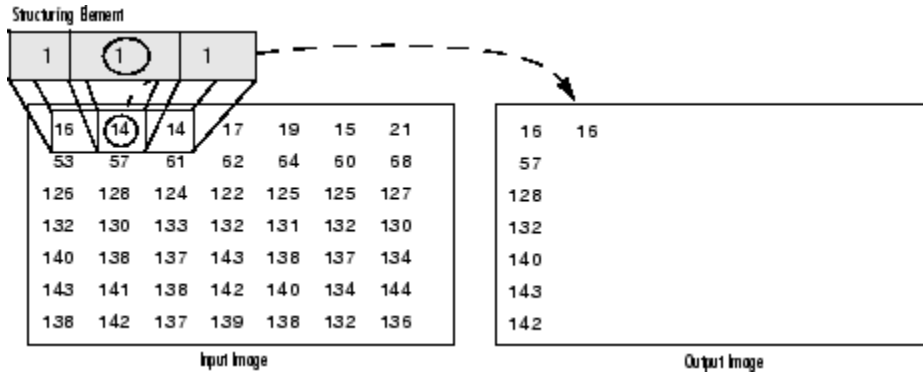
Operation	Rule
Dilation	The value of the output pixel is the *maximum* value of all the pixels in the input pixel's neighborhood. In a binary image, if any of the pixels is set to the value `1`, the output pixel is set to 1.
Erosion	The value of the output pixel is the *minimum* value of all the pixels in the input pixel's neighborhood. In a binary image, if any of the pixels is set to `0`, the output pixel is set to 0.

The following figure illustrates the dilation of a binary image. Note how the structuring element defines the neighborhood of the pixel of interest, which is circled. (See “Structuring Elements” on page 10-5 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

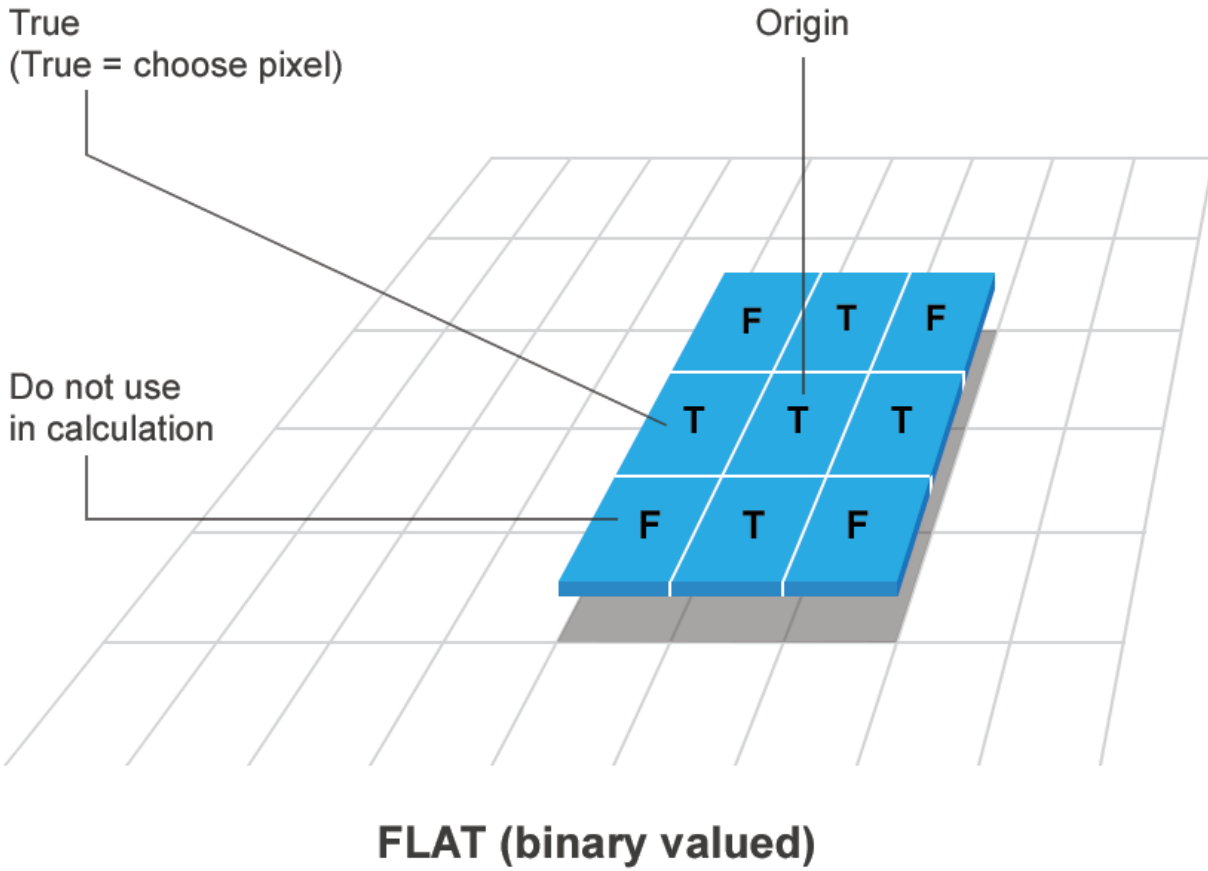
Operation	Rule
Dilation	Pixels beyond the image border are assigned the minimum value afforded by the data type. For binary images, these pixels are assumed to be set to 0. For grayscale images, the minimum value for `uint8` images is `0`.
Erosion	Pixels beyond the image border are assigned the *maximum* value afforded by the data type. For binary images, these pixels are assumed to be set to `1`. For grayscale images, the maximum value for `uint8` images is `255`.

Note By using the minimum value for dilation operations and the maximum value for erosion operations, the toolbox avoids *border effects*, where regions near the borders of the output image do not appear to be homogeneous with the rest of the image. For example, if erosion padded with a minimum value, eroding an image would result in a black border around the edge of the output image.

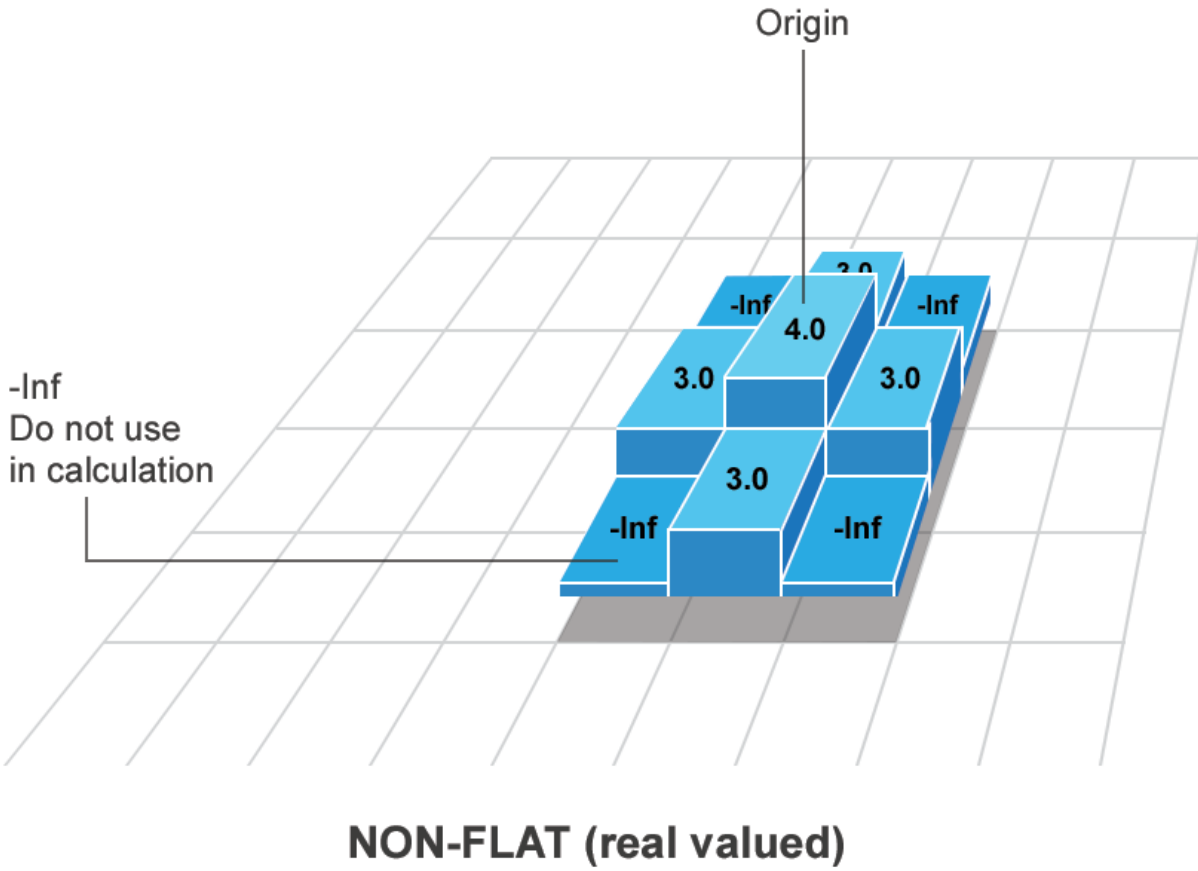
Structuring Elements

An essential part of the morphological dilation and erosion operations is the structuring element used to probe the input image. A structuring element is a matrix that identifies the pixel in the image being processed and defines the neighborhood used in the processing of each pixel. 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.

There are two types of structuring elements: flat and nonflat. A flat structuring element is a binary valued neighborhood, either 2-D or multidimensional, in which the true pixels are included in the morphological computation, and the false pixels are not. The center pixel of the structuring element, called the *origin*, identifies the pixel in the image being processed. Use the `strel` function to create a flat structuring element. You can use flat structuring elements with both binary and grayscale images. The following figure illustrates a flat structuring element.



A nonflat structuring element is a matrix of type `double` that identifies the pixel in the image being processed and defines the neighborhood used in the processing of that pixel. A nonflat structuring element contains finite values used as additive offsets in the morphological computation. The center pixel of the matrix, called the origin, identifies the pixel in the image that is being processed. Pixels in the neighborhood with the value `-Inf` are not used in the computation. Use the `offsetstrel` function to create a nonflat structuring element. You can use nonflat structuring elements only with grayscale images.



Determine the Origin of a Structuring Element

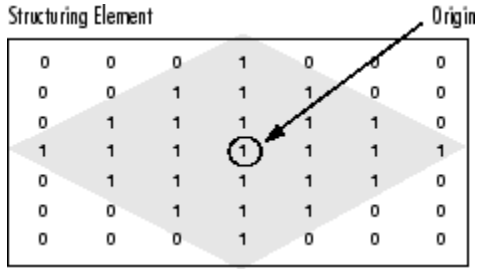
The morphological functions use this code to get the coordinates of the origin of structuring elements of any size and dimension:

```
origin = floor((size(nhood)+1)/2)
```

where `nhood` is the neighborhood defining the structuring element. To see the neighborhood of a flat structuring element, view the `Neighborhood` property of the

`strel` object. To see the neighborhood of a nonflat structuring element, view the `Offset` property of the `offsetstrel` object.

For example, the following illustrates the origin of a flat, diamond-shaped structuring element.



Structuring Element Decomposition

To enhance performance, the `strel` and `offsetstrel` functions 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 see the sequence of structuring elements used in a decomposition, use the `decompose` method. Both `strel` objects and `offsetstrel` objects support decompose methods. The `decompose` method 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.

```
SE = strel('diamond',4)
```

```
SE =

strel is a diamond shaped structuring element with properties:
```

```
      Neighborhood: [9x9 logical]
    Dimensionality: 2
```

Call the decompose method. The method returns an array of structuring elements.

```
decompose(SE)
```

```
ans =

  3x1 strel array with properties:

    Neighborhood
    Dimensionality
```

Dilate an Image to Enlarge a Shape

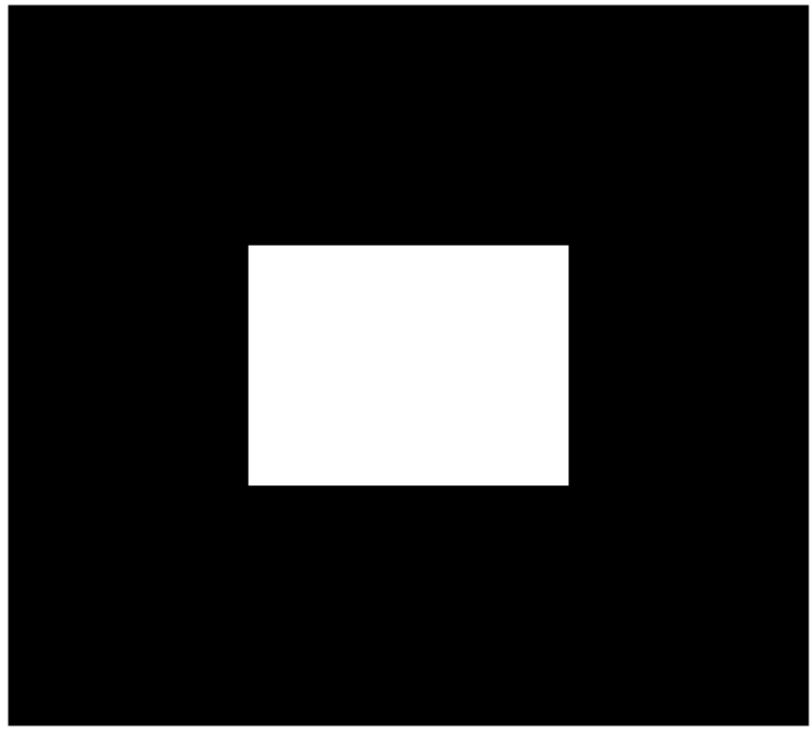
This example shows how to dilate an image using the `imdilate` function. The morphological dilation operation expands or thickens foreground objects in an image.

Create a simple sample binary image containing one foreground object: the square region of 1's in the middle of the image.

```
BW = zeros(9,10);
BW(4:6,4:7) = 1
```

```
BW = 9×10

     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
```

```
imshow(imresize(BW,40,'nearest'))
```



Create a structuring element to use with `imdilate`. To dilate a geometric object, you typically create a structuring element that is the same shape as the object.

```
SE = strel('square',3)
```

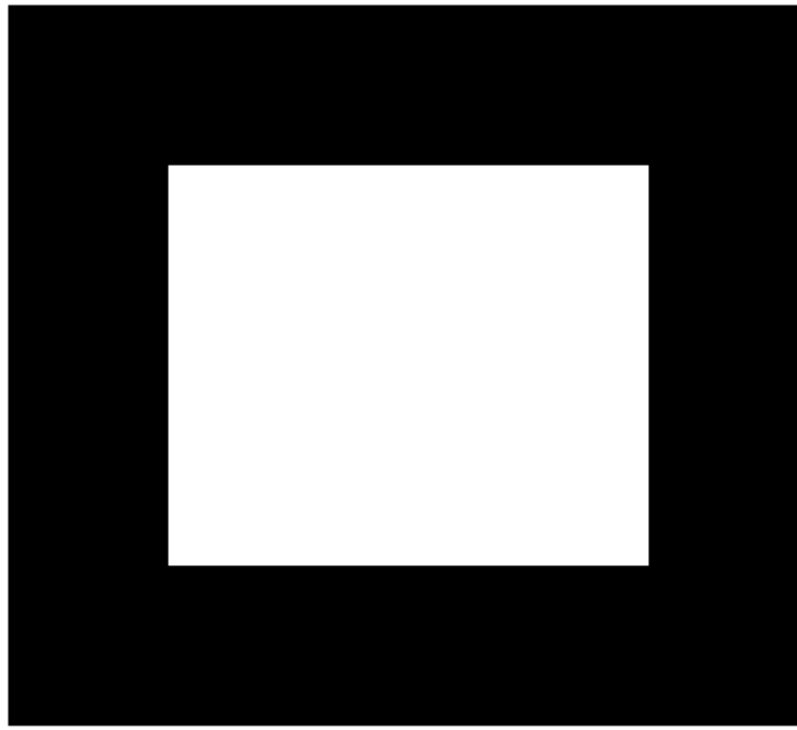
```
SE = 
strel is a square shaped structuring element with properties:

      Neighborhood: [3x3 logical]
    Dimensionality: 2
```

Dilate the image, passing the input image and the structuring element to `imdilate`. Note how dilation adds a rank of 1's to all sides of the foreground object.

```
BW2 = imdilate(BW,SE)
```

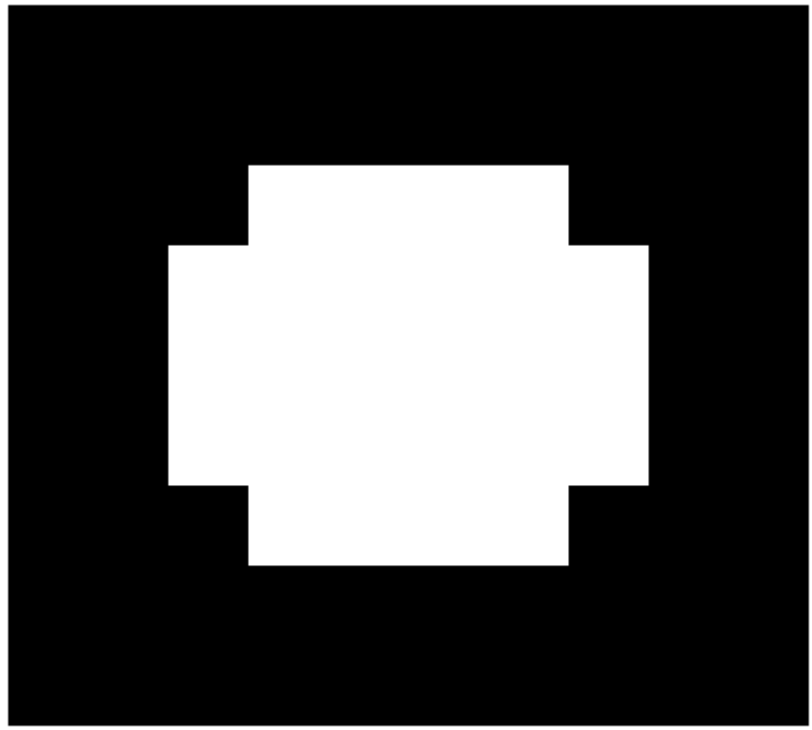
```
BW2 = 9×10

     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
```

```
imshow(imresize(BW2,40,'nearest'))
```



For comparison, create a structuring element that is a different shape. Dilate the original image using the new structuring element.

```
SE2 = strel('diamond',1);
BW3 = imdilate(BW,SE2);
imshow(imresize(BW3,40,'nearest'))
```

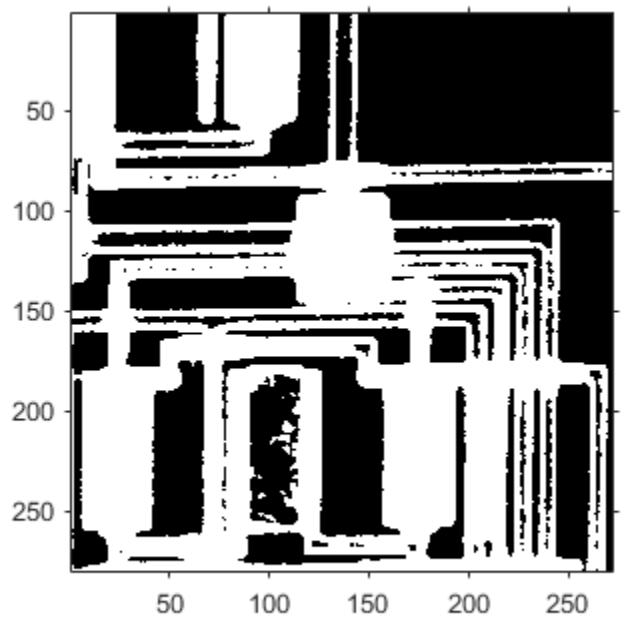


Erode an Image To Remove Thin Lines

This example shows how to erode a binary image using the `imerode` function.

Read a binary image into the workspace. Display the image.

```
BW1 = imread('circbw.tif');
figure
imshow(BW1)
```



Create a structuring element. The following code creates a diagonal structuring element object.

```
SE = strel('arbitrary',eye(7))
```

```
SE =
strel is a arbitrary shaped structuring element with properties:
```

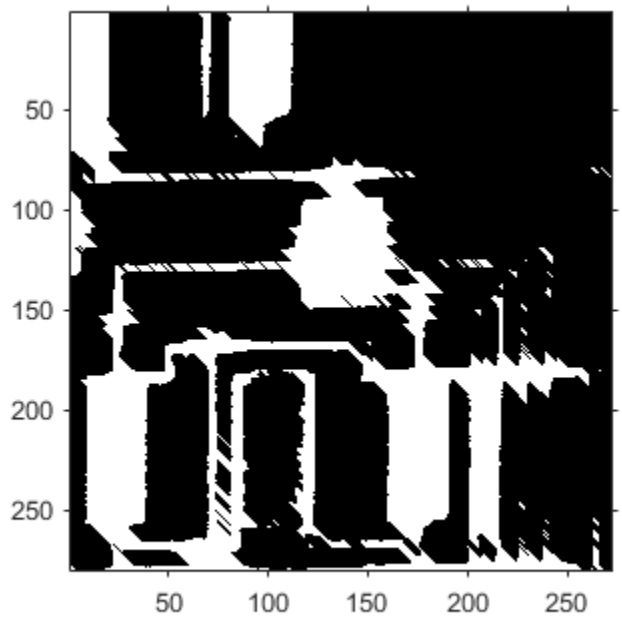
```
  Neighborhood: [7x7 logical]
Dimensionality: 2
```

Erode the image, specifying the input image and the structuring element as arguments to the `imerode` function.

```
BW2 = imerode(BW1,SE);
```

Display the original image and the eroded image. Notice the diagonal streaks on the right side of the output image. These are due to the shape of the structuring element.

```
figure
imshow(BW2)
```



Operations That Combine 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. This table lists 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
`bwhitmiss`	Logical `AND` of an image, eroded with one structuring element, and the image's complement, eroded with a second structuring element.
`imbothat`	Subtracts the original image from a morphologically closed version of the image. Can be used to find intensity troughs in an image.
`imclose`	Dilates an image and then erodes the dilated image using the same structuring element for both operations.
`imopen`	Erodes an image and then dilates the eroded image using the same structuring element for both operations.
`imtophat`	Subtracts a morphologically opened image from the original image. Can be used to enhance contrast in an image.

The following example uses `imdilate` and `imerode` to illustrate how to implement a morphological opening. The toolbox also includes the `imopen` function, which performs this processing in a single step.

Use Morphological Opening to Extract Large Image Features

You can use morphological opening to remove small objects from an image while preserving the shape and size of larger objects in the image.

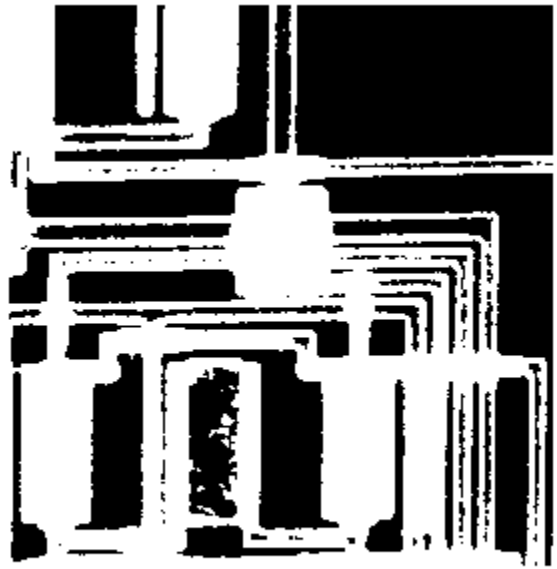
In this example, you use morphological opening on an image of a circuitboard to remove all the circuit lines from the image. The output image contains only the rectangular shapes of the microchips.

Open an Image In One Step

You can use the `imopen` function to perform erosion and dilation in one step.

Read the image into the workspace, and display it.

```
BW1 = imread('circbw.tif');
figure
imshow(BW1)
```

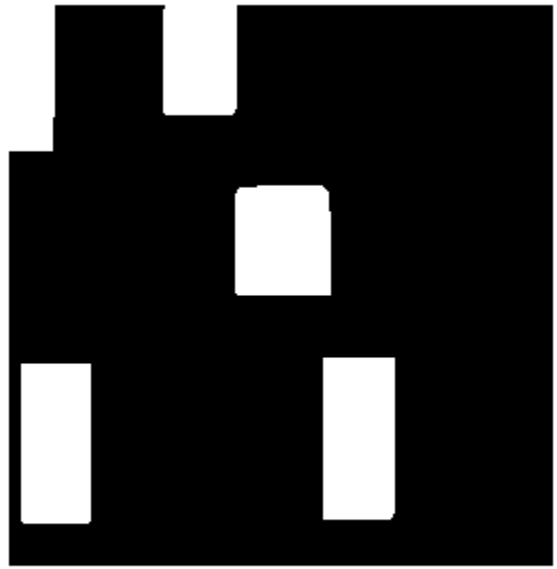


Create a structuring element. The structuring element should be large enough to remove the lines when you erode the image, but not large enough to remove the rectangles. It should consist of all 1's, so it removes everything but large contiguous patches of foreground pixels.

```
SE = strel('rectangle',[40 30]);
```

Open the image.

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

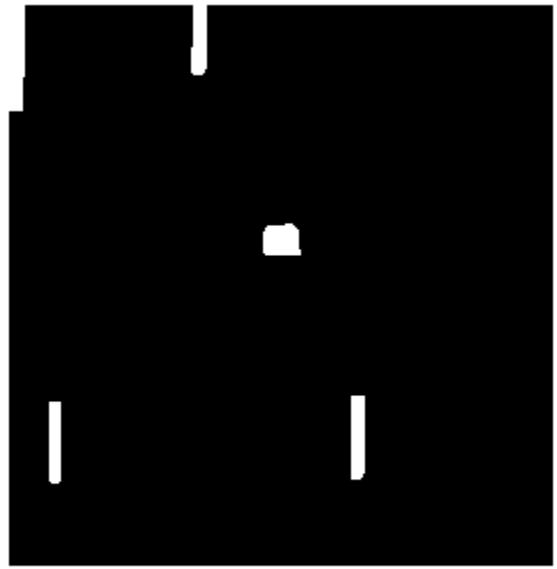


Open an Image By Performing Erosion Then Dilation

You can also perform erosion and dilation sequentially.

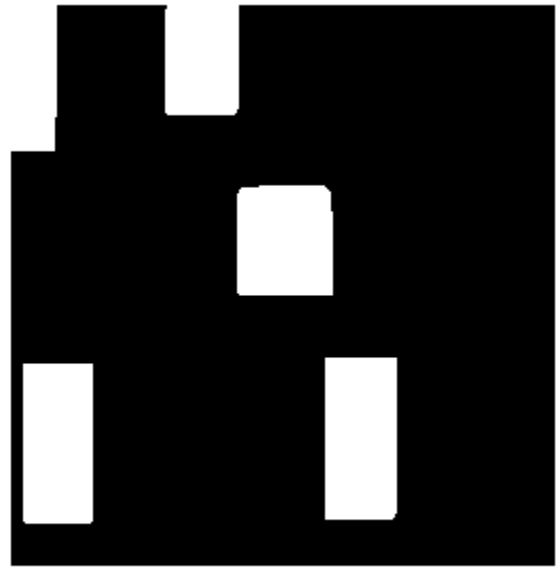
Erode the image with the structuring element. This removes all the lines, but also shrinks the rectangles.

```
BW3 = imerode(BW1,SE);
imshow(BW3)
```



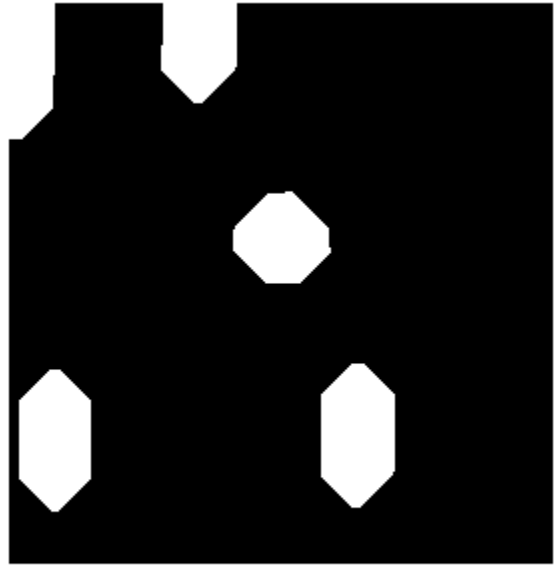
To restore the rectangles to their original sizes, dilate the eroded image using the same structuring element, SE.

```
BW4 = imdilate(BW3,SE);
imshow(BW4)
```



By performing the operations sequentially, you have the flexibility to change the structuring element. Create a different structuring element, and dilate the eroded image using the new structuring element.

```
SE = strel('diamond',15);
BW5 = imdilate(BW3,SE);
imshow(BW5)
```



See Also

`imclose` | `imdilate` | `imerode` | `imopen`

More About

- “Morphological Dilation and Erosion” on page 10-2

Skeletonization

To reduce all objects in an image to lines, without changing the essential structure of the image, use the `bwskel` function. This process is known as *skeletonization*. The following example illustrates how to skeletonize objects in a binary image.

Skeletonize Binary Image

This example shows how to skeletonize a binary image using the `bwskel` function.

Read a binary image into the workspace.

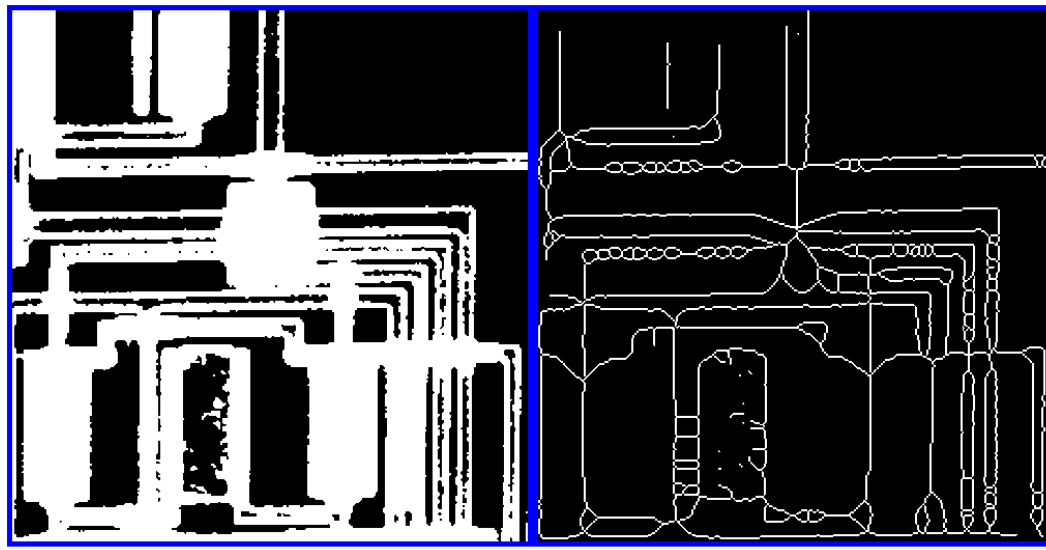
```
BW1 = imread('circbw.tif');
```

Skeletonize objects in the image using the bwskel function.

```
BW2 = bwskel(BW1);
```

View the original image and the skeletonized image side by side.

```
montage({BW1,BW2},'BackgroundColor','blue','BorderSize',5)
```



See Also

`bwskel`

More About

- “Operations That Combine Dilation and Erosion” on page 10-17

Find Object Perimeters

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`.

The following example finds object perimeters in a binary image.

Find Perimeter Pixels in Binary Image

This example shows how to find the perimeter pixels in a binary image using the `bwperim` function.

Read a binary image into the workspace.

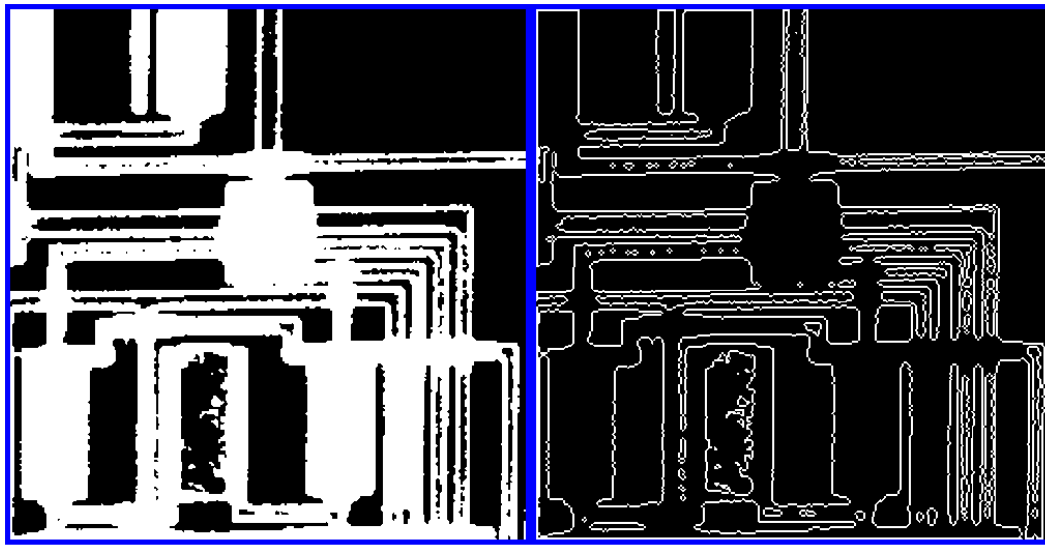
```
BW1 = imread('circbw.tif');
```

Find the perimeters of objects in the image.

```
BW2 = bwperim(BW1);
```

Display the original image and the image showing perimeters side-by-side.

```
montage({BW1,BW2},'BackgroundColor','blue','BorderSize',5)
```



See Also

bwperim

More About

- “Operations That Combine Dilation and Erosion” on page 10-17

Pixel Connectivity

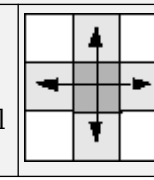
Morphological processing starts at the peaks in the marker image and spreads throughout the rest of the image based on the connectivity of the pixels. Connectivity defines which pixels are connected to other pixels. A set of pixels in a binary image that form a connected group is called an *object* or a *connected component*.

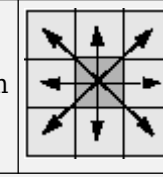
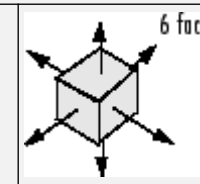
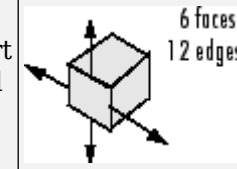
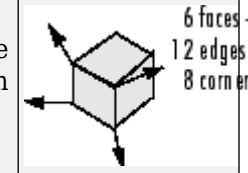
Determining which pixels create a connected component depends on how pixel connectivity is defined. For example, this binary image contains one foreground object or two, depending on the connectivity. If the foreground is 4-connected, the image is all one object — there is no distinction between a foreground 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:

Value	Meaning	
Two-Dimensional Connectivities		
4-connected	Pixels are connected if their edges touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal or vertical direction.	

Value	Meaning	
8-connected	Pixels are connected if their edges or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal, vertical, or diagonal direction.	
Three-Dimensional Connectivities		
6-connected	Pixels are connected if their faces touch. Two adjoining pixels are part of the same object if they are both on and are connected in: • One of these directions: in, out, left, right, up, and down	
18-connected	Pixels are connected if their faces or edges touch. Two adjoining pixels are part of the same object if they are both on and are connected in • One of these directions: in, out, left, right, up, and down • A combination of two directions, such as right-down or in-up	
26-connected	Pixels are connected if their faces, edges, or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected in • One of these directions: in, out, left, right, up, and down • A combination of two directions, such as right-down or in-up • A combination of three directions, such as in-right-up or in-left-down	

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.

```
CONN = [ 0 1 0; 0 1 0; 0 1 0 ]
CONN =
     0     1     0
     0     1     0
     0     1     0
```

Note Connectivity arrays must be symmetric about their center element. Also, you can use a 2-D connectivity array with a 3-D image; the connectivity affects each "page" in the 3-D image.

See Also

`boundarymask` | `bwareaopen` | `bwconncomp` | `bwconncomp` | `conndef` | `imfill` | `iptcheckconn`

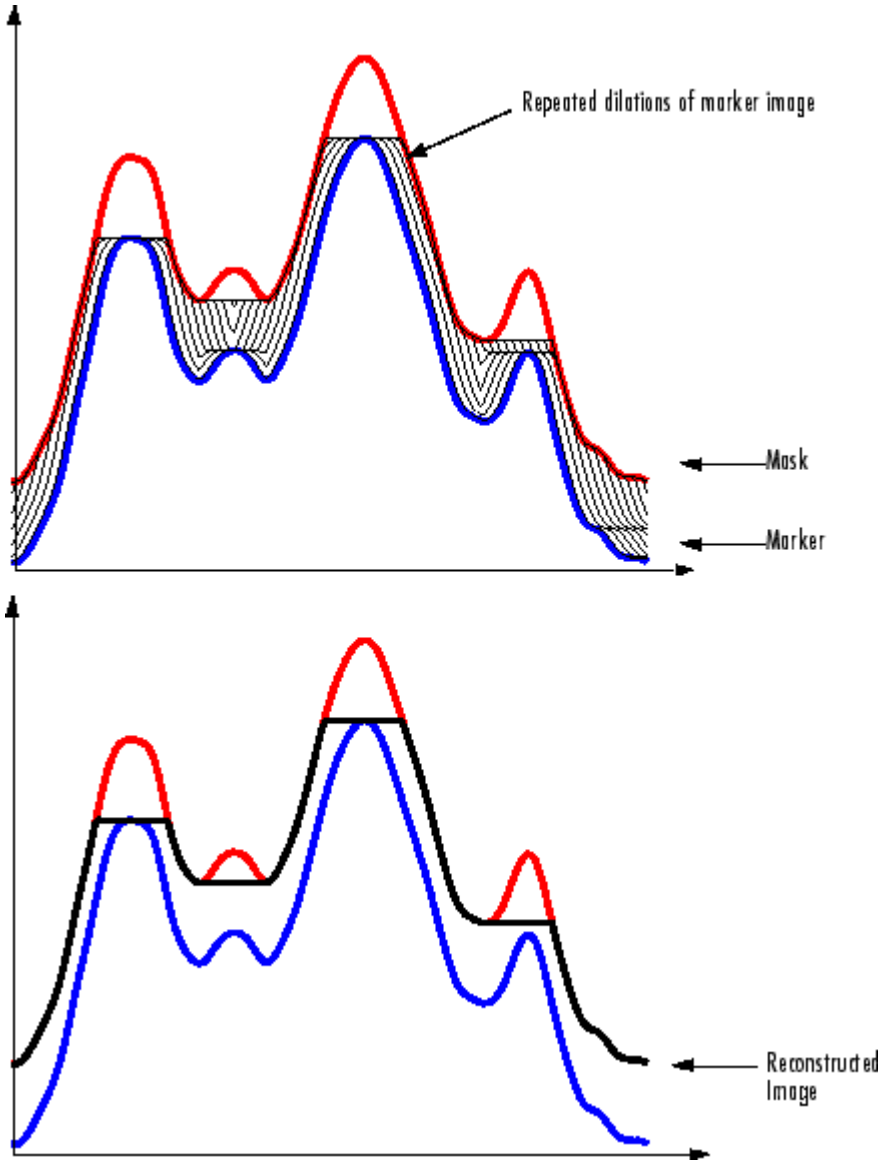
More About

- “Understanding Morphological Reconstruction” on page 10-31

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 pixel connectivity on page 10-27, 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 = imsubtract(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
```

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 10-35
- “Understanding the Maxima and Minima Functions” on page 10-35
- “Finding Areas of High or Low Intensity” on page 10-36
- “Suppressing Minima and Maxima” on page 10-37
- “Imposing a Minimum” on page 10-39

Terminology

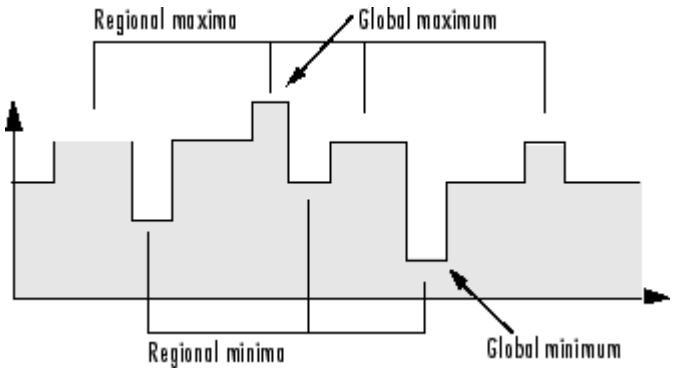
This section uses the following terms.

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 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.

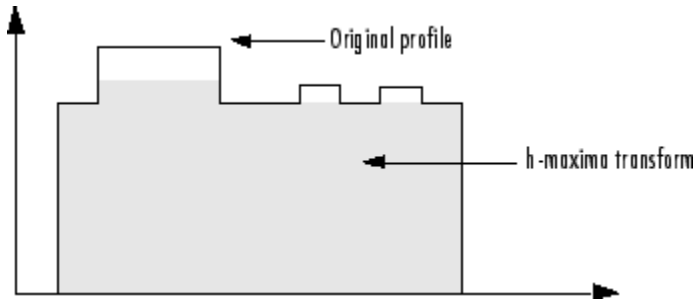
For example, this simple image contains two primary regional maxima, the blocks of pixels containing the value 14 and 18, and several smaller maxima, set to 11.

```
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
       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)
```

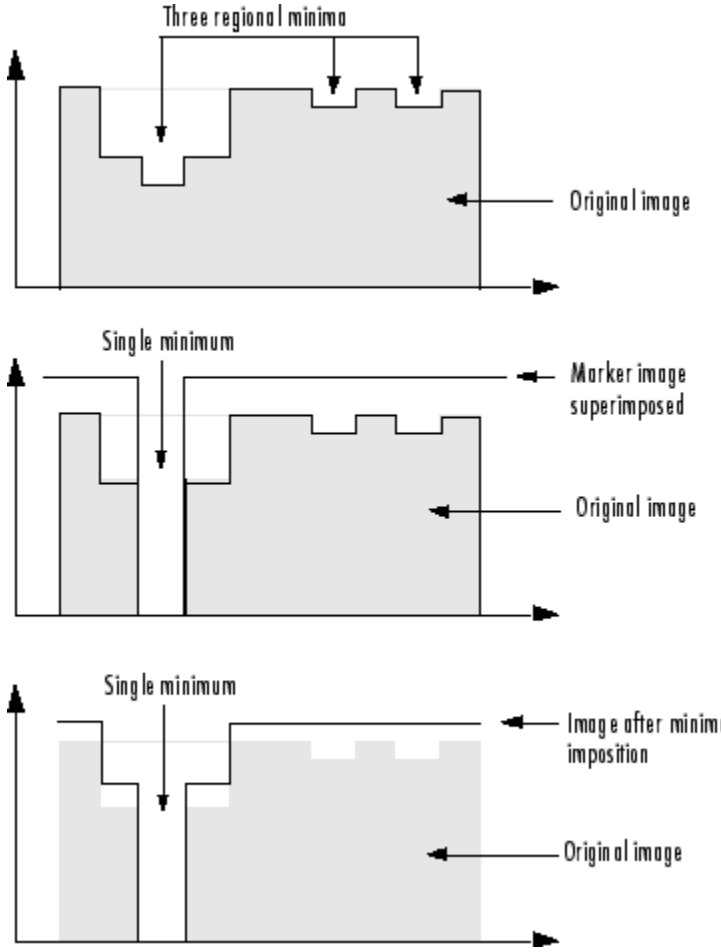
```
marker = 0     0     0     0     0     0     0     0     0     0
         0     0     0     0     0     0     0     0     0     0
         0     0     1     0     0     0     0     0     0     0
         0     0     0     0     0     0     0     0     0     0
         0     0     0     0     0     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 data type (`0` for `uint8` values). `imimposemin` also changes the values of all the other pixels in the image to eliminate the other minima.

```
I = imimposemin(mask,marker)
I =
    11    11    11    11    11    11    11    11    11    11
    11     8     8     8    11    11    11    11    11    11
    11     8     0     8    11    11    11    11    11    11
    11     8     8     8    11    11    11    11    11    11
    11    11    11    11    11    11    11    11    11    11
    11    11    11    11    11     0     0     0    11    11
    11    11    11    11    11     0     0     0    11    11
    11    11    11    11    11     0     0     0    11    11
    11    11    11    11    11    11    11    11    11    11
    11    11    11    11    11    11    11    11    11    11
```

This figure illustrates in 1-D how `imimposemin` changes the profile of row 2 of the image.



Imposing a Minimum

See Also

More About

- “Pixel Connectivity” on page 10-27
- “Flood-Fill Operations” on page 10-43

Flood-Fill Operations

The `imfill` function performs a *flood-fill* operation on binary and grayscale images. For binary images, `imfill` changes connected background pixels (`0`s) to foreground pixels (`1`s), stopping when it reaches object boundaries. For grayscale images, `imfill` brings the intensity values of dark areas that are surrounded by lighter areas up to the same intensity level as surrounding pixels. (In effect, `imfill` removes regional minima that are not connected to the image border. See “Finding Areas of High or Low Intensity” on page 10-36 for more information.) This operation can be useful in removing irrelevant artifacts from images.

Specifying Connectivity

For both binary and grayscale images, the boundary of the fill operation is determined by the pixel connectivity on page 10-27 that you specify.

Note `imfill` differs from the other object-based operations in that it operates on *background* pixels. When you specify connectivity with `imfill`, you are specifying the connectivity of the background, not the foreground.

The implications of connectivity can be illustrated with this matrix.

```
BW = logical([0     0     0     0     0     0     0     0;
              0     1     1     1     1     1     0     0;
              0     1     0     0     0     1     0     0;
              0     1     0     0     0     1     0     0;
              0     1     0     0     0     1     0     0;
              0     1     1     1     1     0     0     0;
              0     0     0     0     0     0     0     0;
              0     0     0     0     0     0     0     0]);
```

If the background is 4-connected, this binary image contains two separate background elements (the part inside the loop and the part outside). If the background is 8-connected, the pixels connect diagonally, and there is only one background element.

Specifying the Starting Point

For binary images, you can specify the starting point of the fill operation by passing in the location subscript or by using `imfill` in interactive mode, selecting starting pixels with a mouse.

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

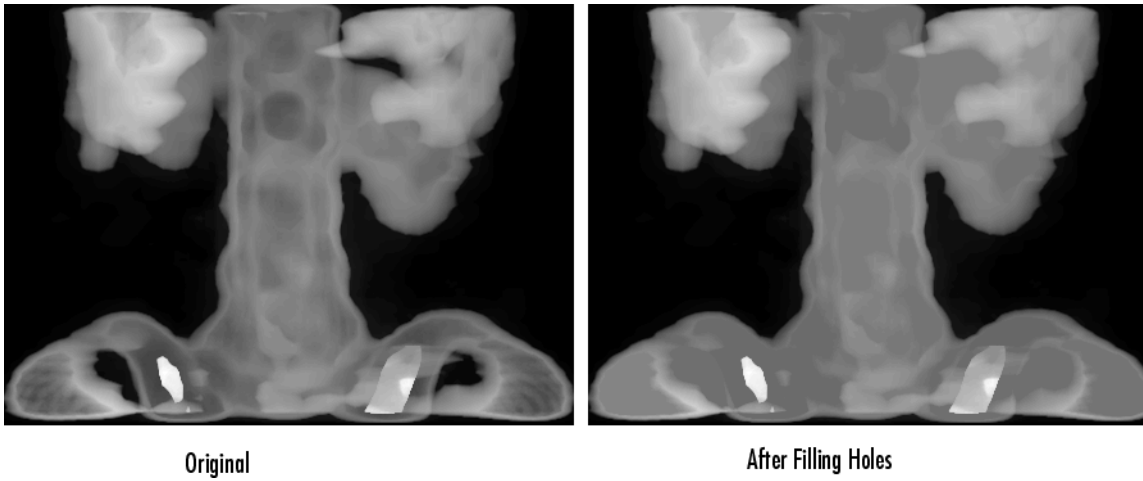
A common use of the flood-fill operation is to fill holes in images. For example, suppose you have an image, binary or grayscale, in which the foreground objects represent spheres. In the image, these objects should appear as disks, but instead are doughnut shaped because of reflections in the original photograph. Before doing any further

processing of the image, you might want to first fill in the “doughnut 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)
```



See Also

`imfill`

More About

- “Understanding Morphological Reconstruction” on page 10-31

Detecting a Cell Using Image Segmentation

This example shows how to detect a cell using edge detection and basic morphology. An object can be easily detected in an image if the object has sufficient contrast from the background. In this example, the cells are prostate cancer cells.

Step 1: Read Image

Read in the `cell.tif` image, which is an image of a prostate cancer cell.

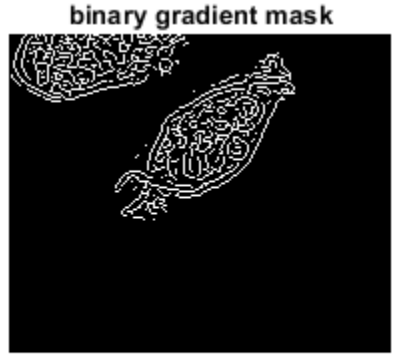
```
I = imread('cell.tif');
figure, imshow(I), title('original image');
text(size(I,2),size(I,1)+15, ...
    'Image courtesy of Alan Partin', ...
    'FontSize',7,'HorizontalAlignment','right');
text(size(I,2),size(I,1)+25, ....
    'Johns Hopkins University', ...
    'FontSize',7,'HorizontalAlignment','right');
```

Step 2: Detect Entire Cell

Two cells are present in this image, but only one cell can be seen in its entirety. We will detect this cell. Another word for object detection is segmentation. The object to be segmented differs greatly in contrast from the background image. Changes in contrast can be detected by operators that calculate the gradient of an image. The gradient image

can be calculated and a threshold can be applied to create a binary mask containing the segmented cell. First, we use `edge` and the Sobel operator to calculate the threshold value. We then tune the threshold value and use `edge` again to obtain a binary mask that contains the segmented cell.

```
[~, threshold] = edge(I, 'sobel');
fudgeFactor = .5;
BWs = edge(I,'sobel', threshold * fudgeFactor);
figure, imshow(BWs), title('binary gradient mask');
```



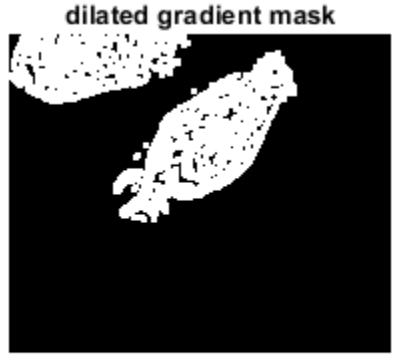
Step 3: Dilate the Image

The binary gradient mask shows lines of high contrast in the image. These lines do not quite delineate the outline of the object of interest. Compared to the original image, you can see gaps in the lines surrounding the object in the gradient mask. These linear gaps will disappear if the Sobel image is dilated using linear structuring elements, which we can create with the `strel` function.

```
se90 = strel('line', 3, 90);
se0 = strel('line', 3, 0);
```

The binary gradient mask is dilated using the vertical structuring element followed by the horizontal structuring element. The `imdilate` function dilates the image.

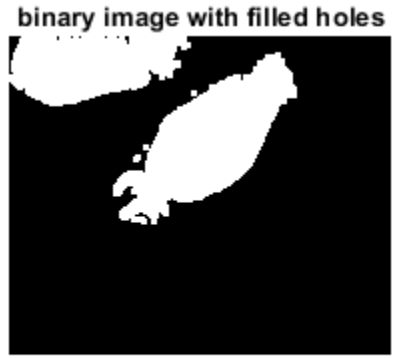
```
BWsdil = imdilate(BWs, [se90 se0]);
figure, imshow(BWsdil), title('dilated gradient mask');
```



Step 4: Fill Interior Gaps

The dilated gradient mask shows the outline of the cell quite nicely, but there are still holes in the interior of the cell. To fill these holes we use the imfill function.

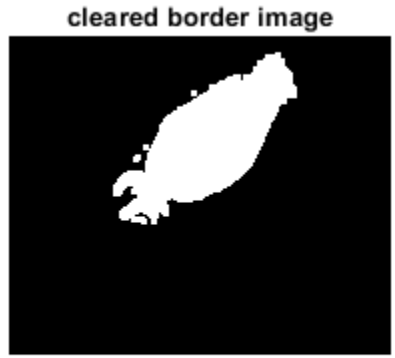
```
BWdfill = imfill(BWsdil, 'holes');
figure, imshow(BWdfill);
title('binary image with filled holes');
```



Step 5: Remove Connected Objects on Border

The cell of interest has been successfully segmented, but it is not the only object that has been found. Any objects that are connected to the border of the image can be removed using the imclearborder function. The connectivity in the imclearborder function was set to 4 to remove diagonal connections.

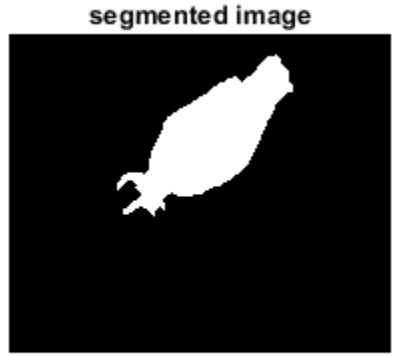
```
BWnobord = imclearborder(BWdfill, 4);
figure, imshow(BWnobord), title('cleared border image');
```



Step 6: Smoothen the Object

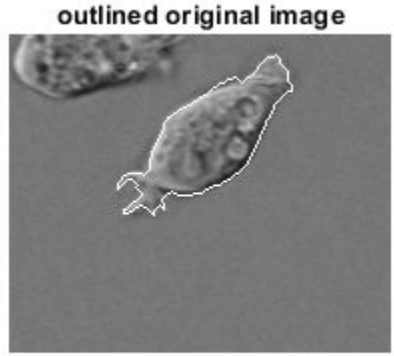
Finally, in order to make the segmented object look natural, we smoothen the object by eroding the image twice with a diamond structuring element. We create the diamond structuring element using the `strel` function.

```
seD = strel('diamond',1);
BWfinal = imerode(BWnobord,seD);
BWfinal = imerode(BWfinal,seD);
figure, imshow(BWfinal), title('segmented image');
```



An alternate method for displaying the segmented object would be to place an outline around the segmented cell. The outline is created by the `bwperim` function.

```
BWoutline = bwperim(BWfinal);
Segout = I;
Segout(BWoutline) = 255;
figure, imshow(Segout), title('outlined original image');
```



See Also

bwperim | edge | imclearborder | imdilate | imerode | imfill | strel

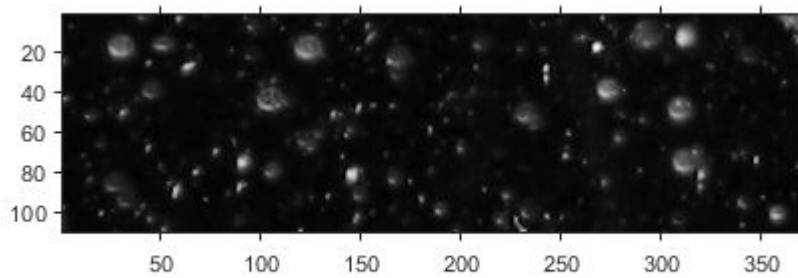
Granulometry of Snowflakes

This example shows how to calculate the size distribution of snowflakes in an image by using granulometry. Granulometry determines the size distribution of objects in an image without explicitly segmenting (detecting) each object first.

Read Image

Read in the 'snowflakes.png' image, which is a photograph of snowflakes.

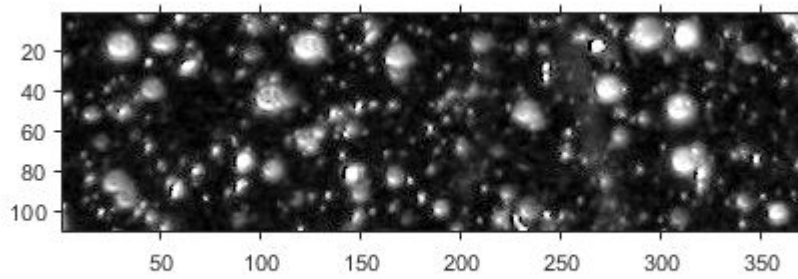
```
I = imread('snowflakes.png');
imshow(I)
```



Enhance Contrast

Your first step is to maximize the intensity contrast in the image. You can do this using the `adapthisteq` function, which performs contrast-limited adaptive histogram equalization. Rescale the image intensity using the `imadjust` function so that it fills the data type's entire dynamic range.

```
claheI = adapthisteq(I,'NumTiles',[10 10]);
claheI = imadjust(claheI);
imshow(claheI)
```

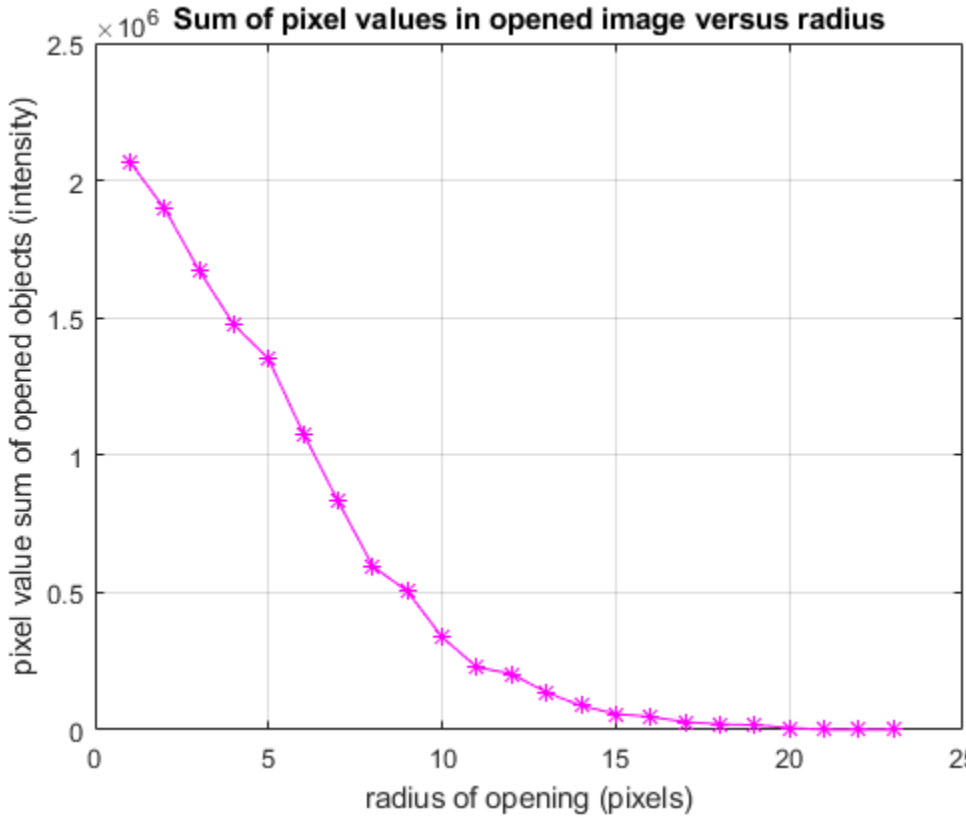


Determine Intensity Surface Area Distribution in Enhanced Image

Granulometry estimates the intensity surface area distribution of snowflakes as a function of size. Granulometry likens image objects to stones whose sizes can be determined by sifting them through screens of increasing size and collecting what remains after each pass. Image objects are sifted by opening the image with a structuring element of increasing size and counting the remaining intensity surface area (summation of pixel values in the image) after each opening.

Choose a counter limit so that the intensity surface area goes to zero as you increase the size of your structuring element. For display purposes, leave the first entry in the surface area array empty.

```
radius_range = 0:22;
intensity_area = zeros(size(radius_range));
for counter = radius_range
    remain = imopen(claheI, strel('disk', counter));
    intensity_area(counter + 1) = sum(remain(:));
end
figure
plot(intensity_area, 'm - *')
grid on
title('Sum of pixel values in opened image versus radius')
xlabel('radius of opening (pixels)')
ylabel('pixel value sum of opened objects (intensity)')
```

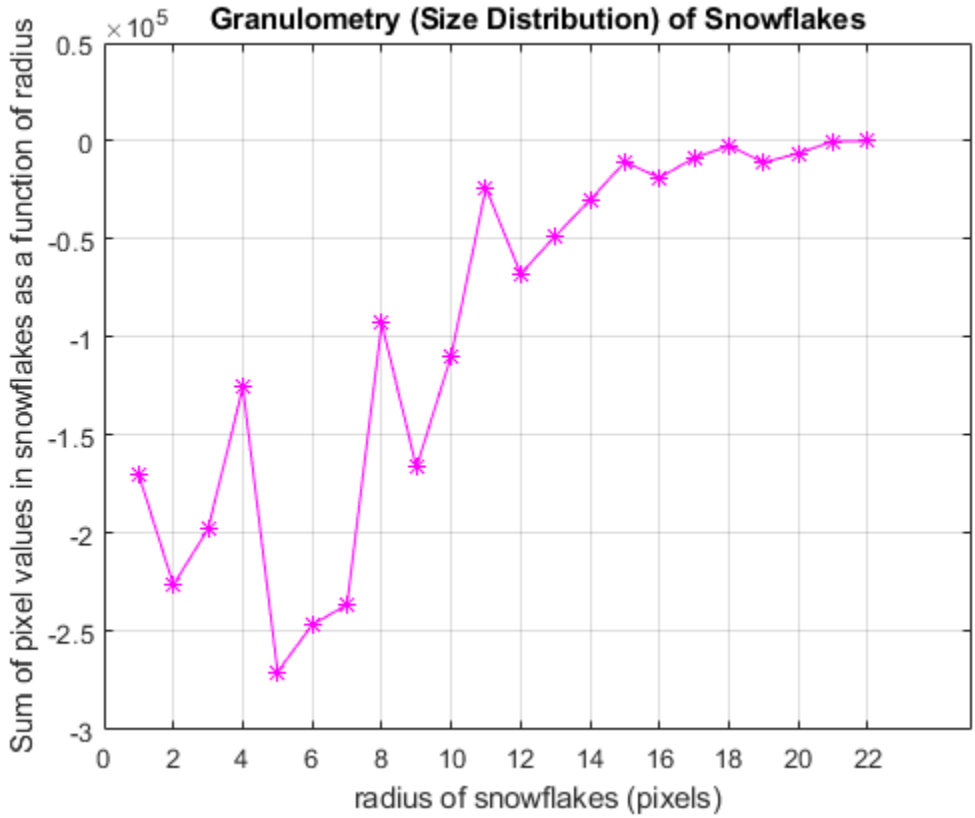


Calculate First Derivative of Distribution

A significant drop in intensity surface area between two consecutive openings indicates that the image contains objects of comparable size to the smaller opening. This is equivalent to the first derivative of the intensity surface area array, which contains the size distribution of the snowflakes in the image. Calculate the first derivative with the `diff` function.

```
intensity_area_prime = diff(intensity_area);
plot(intensity_area_prime, 'm - *')
grid on
title('Granulometry (Size Distribution) of Snowflakes')
ax = gca;
ax.XTick = [0 2 4 6 8 10 12 14 16 18 20 22];
```

```
xlabel('radius of snowflakes (pixels)')
ylabel('Sum of pixel values in snowflakes as a function of radius')
```

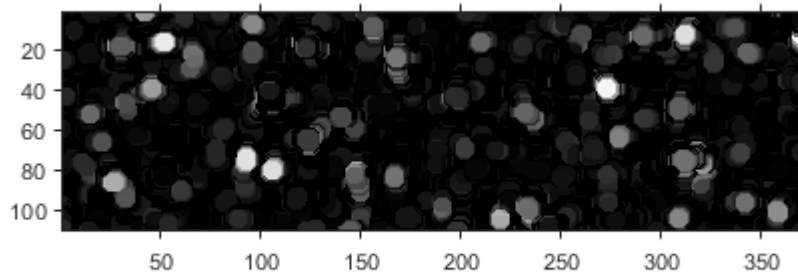


Extract Snowflakes Having a Particular Radius

Notice the minima and the radii where they occur in the graph. The minima tell you that snowflakes in the image have those radii. The more negative the minimum point, the higher the snowflakes' cumulative intensity at that radius. For example, the most negative minimum point occurs at the 5 pixel radius mark. You can extract the snowflakes having a 5 pixel radius with the following steps.

```
open5 = imopen(claheI,strel('disk',5));
open6 = imopen(claheI,strel('disk',6));
```

```
rad5 = imsubtract(open5,open6);
imshow(rad5,[])
```



See Also

adapthisteq | imadjust | imopen | imsubtract | strel

Distance Transform of a Binary Image

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

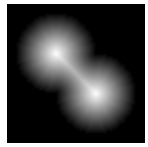
Distance Metric	Description	Illustration
Euclidean	The Euclidean distance is the straight-line distance between two pixels.	Image: 0 0 0 / 0 1 0 / 0 0 0 Distance Transform: 1.41 1.0 1.41 / 1.0 0.0 1.0 / 1.41 1.0 1.41
City Block	The city block distance metric measures the path between the pixels based on a 4-connected neighborhood. Pixels whose edges touch are 1 unit apart; pixels diagonally touching are 2 units apart.	Image: 0 0 0 / 0 1 0 / 0 0 0 Distance Transform: 2 1 2 / 1 0 1 / 2 1 2
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: 0 0 0 / 0 1 0 / 0 0 0 Distance Transform: 1 1 1 / 1 0 1 / 1 1 1
Quasi-Euclidean	The quasi-Euclidean metric measures the total Euclidean distance along a set of horizontal, vertical, and diagonal line segments.	Image: 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 Distance Transform: 2.8 2.4 2.0 2.4 2.8 / 2.4 1.4 1.0 1.4 2.4 / 2.0 1.0 0 1.0 2.0 / 2.4 1.4 1.0 1.4 2.4 / 2.8 2.4 2.0 2.4 2.8

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')
```



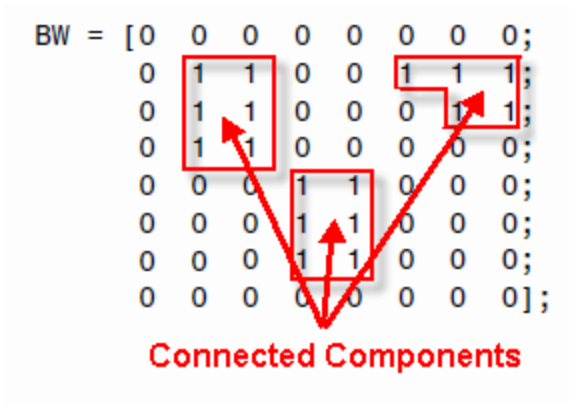
Label and Measure Objects in a Binary Image

In this section...

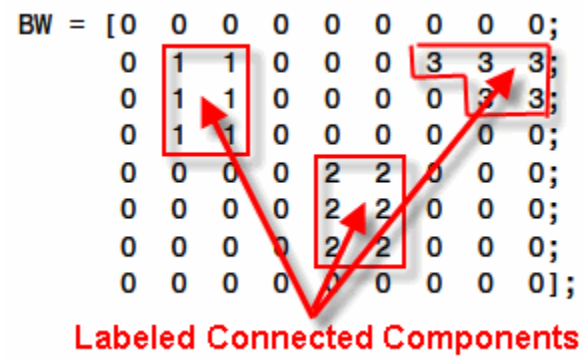
"Understanding Connected-Component Labeling" on page 10-60
"Selecting Objects in a Binary Image" on page 10-62
"Finding the Area of the Foreground of a Binary Image" on page 10-63
"Finding the Euler Number of a Binary Image" on page 10-63

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.

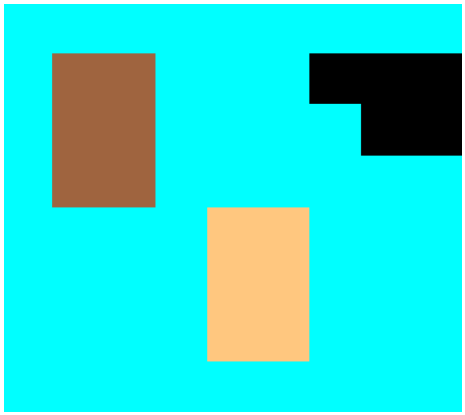
For visualizing connected components, it is useful to construct a label matrix. Use the `labelmatrix` function. To inspect the results, display the label matrix as a pseudo-color image using `label2rgb`.

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
`bwlabel`	2-D	Double-precision label matrix	High	4 or 8
`bwlabeln`	N-D	Double-precision label matrix	High	Any
`bwconncomp`	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 cross-hairs 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.

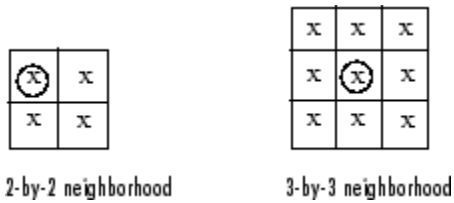
Lookup Table Operations

In this section...

"Creating a Lookup Table" on page 10-65
"Using a Lookup Table" on page 10-65

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.



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)
```

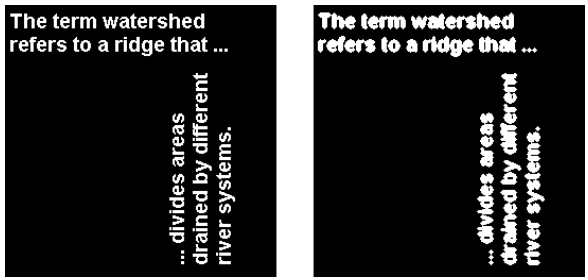


Image Before and After Applying Lookup Table Operation

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 topic describes functions that support a range of standard image processing operations for analyzing and enhancing images.

- “Pixel Values” on page 11-3
- “Intensity Profile of Images” on page 11-5
- “Contour Plot of Image Data” on page 11-9
- “Measuring Regions in Grayscale Images” on page 11-11
- “Finding the Length of a Pendulum in Motion” on page 11-17
- “Create Image Histogram” on page 11-25
- “Image Mean, Standard Deviation, and Correlation Coefficient” on page 11-27
- “Edge Detection” on page 11-28
- “Boundary Tracing in Images” on page 11-31
- “Quadtree Decomposition” on page 11-37
- “Detect and Measure Circular Objects in an Image” on page 11-41
- “Identifying Round Objects” on page 11-58
- “Measuring Angle of Intersection” on page 11-65
- “Measuring the Radius of a Roll of Tape” on page 11-73
- “Texture Analysis” on page 11-77
- “Detect Regions of Texture in Images” on page 11-79
- “Texture Analysis Using the Gray-Level Co-Occurrence Matrix (GLCM)” on page 11-81
- “Create a Gray-Level Co-Occurrence Matrix” on page 11-82
- “Specify Offset Used in GLCM Calculation” on page 11-84
- “Derive Statistics from GLCM and Plot Correlation” on page 11-85
- “Adjust Image Intensity Values to Specified Range” on page 11-88
- “Gamma Correction” on page 11-90
- “Contrast Enhancement Techniques” on page 11-92

- “Specify Contrast Adjustment Limits” on page 11-97
- “Histogram Equalization” on page 11-99
- “Adaptive Histogram Equalization” on page 11-103
- “Enhance Color Separation Using Decorrelation Stretching” on page 11-106
- “Enhancing Multispectral Color Composite Images” on page 11-115
- “Low-Light Image Enhancement” on page 11-128
- “Apply Gaussian Smoothing Filters to Images” on page 11-141
- “Noise Removal” on page 11-151
- “Compute 3-D Superpixels of Input Volumetric Intensity Image” on page 11-159
- “Image Quality Metrics” on page 11-162
- “Train and Use a No-Reference Quality Assessment Model” on page 11-164
- “Obtain Local Structural Similarity Index” on page 11-169
- “Compare Image Quality at Various Compression Levels” on page 11-172
- “Anatomy of an eSFR Chart” on page 11-174
- “Evaluate Quality Metrics on eSFR Test Chart” on page 11-179
- “Correct Colors Using Color Correction Matrix” on page 11-194
- “Install Sample Data Using the Add-Ons Explorer” on page 11-204

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 4-49.

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. `impixel` removes the stars used to indicate selected points.

```
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 multi-line 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 multi line 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 about specifying the interpolation method, see “Resize an Image with imresize Function” on page 6-2.) `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 cross-hairs when you move it over the displayed image. Using the mouse, specify line segments by clicking the endpoints. `improfile` draws a line between the endpoints. When you finish specifying the path, press **Return**. In the following figure, the line is shown in red.

```
improfile
```

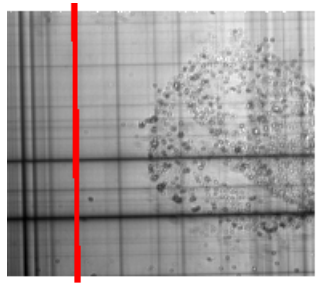
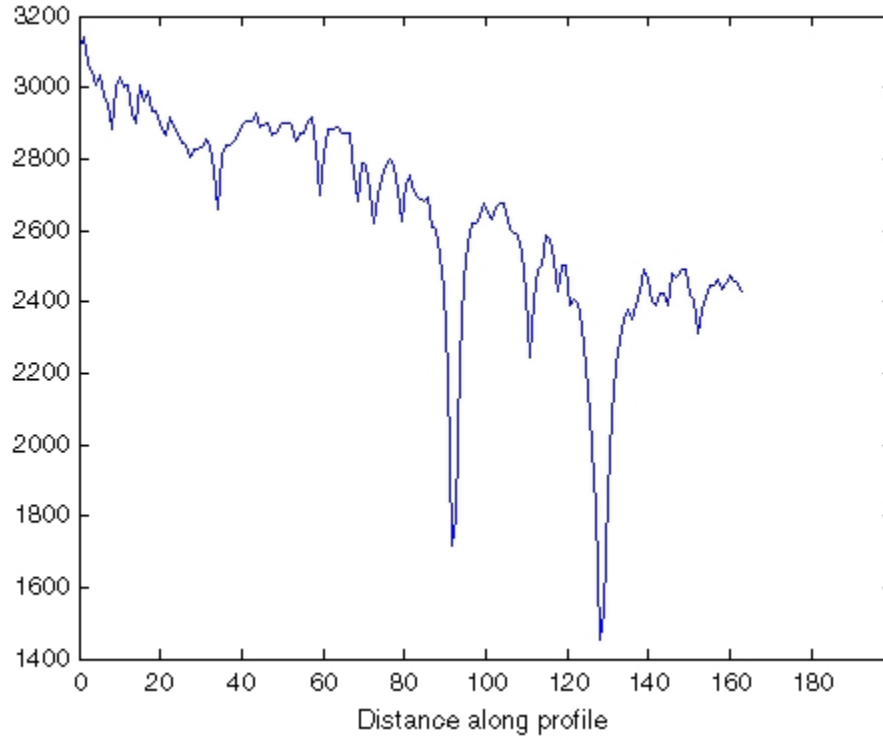


Image Courtesy of Ann Walker

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 multi-line path, `improfile` plots the intensity values in a three-dimensional view.

Display an RGB image using `imshow`.

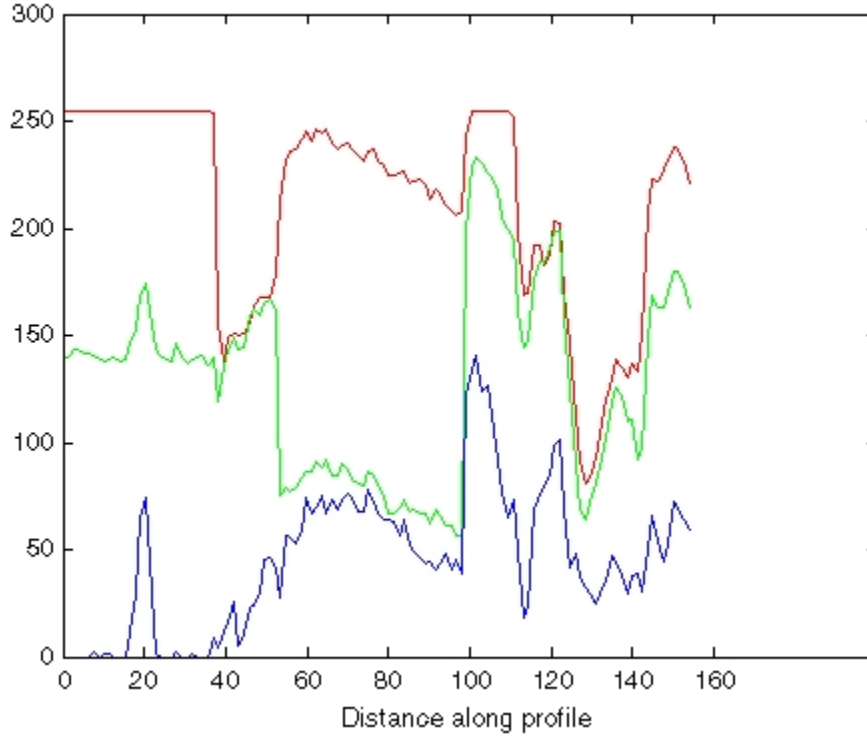
```
imshow peppers.png
```

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

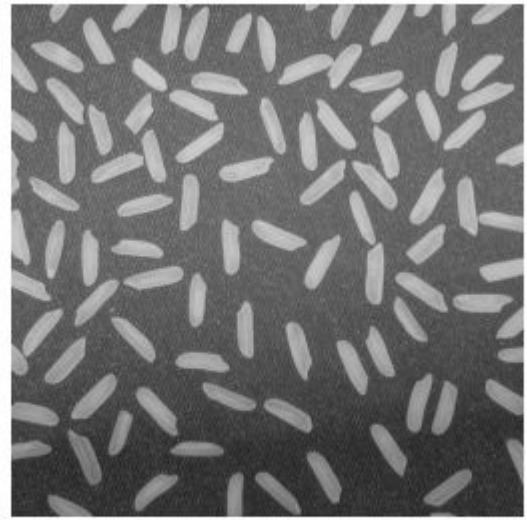
A contour is a path in an image along which the image intensity values are equal to a constant. You can create a contour plot of the data in a grayscale image using `imcontour`. This function is similar to the `contour` function in MATLAB, but it automatically sets up the axes so their orientation and 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.

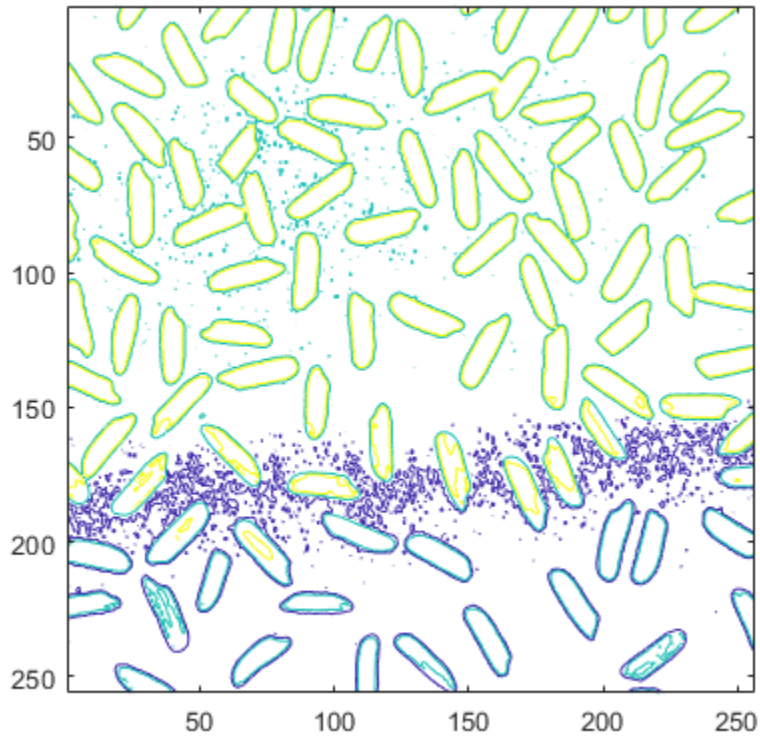
Read grayscale image and display it. The example uses an example image of grains of rice.

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



Create a contour plot of the image using `imcontour` .

```
figure;
imcontour(I,3)
```



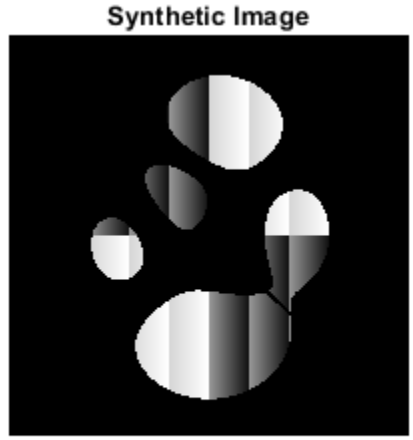
Measuring Regions in Grayscale Images

This example shows how to measure properties of objects in a grayscale image. To accomplish this, first segment the grayscale image to get a binary image of objects. Then, use regionprops to analyze the original grayscale pixel values corresponding to each object in the binary image.

Step 1: Create Synthetic Image

Use a helper function, `propsSynthesizeImage`, to create a grayscale image that contains five distinct regions.

```
I = propsSynthesizeImage;
imshow(I)
title('Synthetic Image')
```



Step 2: Create a Binary Image

Segment the grayscale image by creating a binary image containing the objects in the image.

```
BW = I > 0;
imshow(BW)
title('Binary Image')
```

Step 3: Calculate Object Properties Using Pixel Values of Grayscale Image

The `regionprops` function supports several properties that can be used with grayscale images, including 'WeightedCentroid', 'MeanIntensity', 'MinIntensity', and 'MaxIntensity'. These properties use the original pixel values of the objects for their calculations.

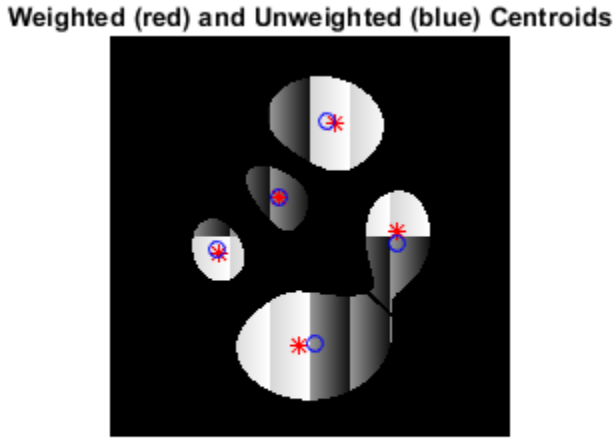
For example, you can use `regionprops` to calculate both the centroid and weighted centroid of objects in the image. Note how you pass in the binary image (`BW`) containing your objects and the original grayscale image (`I`) as arguments into `regionprops`.

```
s = regionprops(BW,I,{'Centroid','WeightedCentroid'});
```

To compare the weighted centroid locations with the unweighted centroid locations, display the original image and then, using the `hold` and `plot` functions, superimpose the centroids on the image.

```
imshow(I)
title('Weighted (red) and Unweighted (blue) Centroids');
hold on
```

```
numObj = numel(s);
for k = 1 : numObj
    plot(s(k).WeightedCentroid(1), s(k).WeightedCentroid(2), 'r*')
    plot(s(k).Centroid(1), s(k).Centroid(2), 'bo')
end
hold off
```



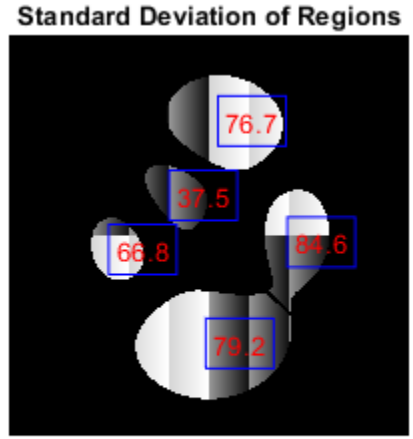
Step 4: Calculate Custom Pixel Value-Based Properties

You can use the 'PixelValues' property to do custom calculations based on the pixel values of the original grayscale image. The 'PixelValues' property returns a vector containing the grayscale values of pixels in a region.

As an example, calculate the standard deviation of each region.

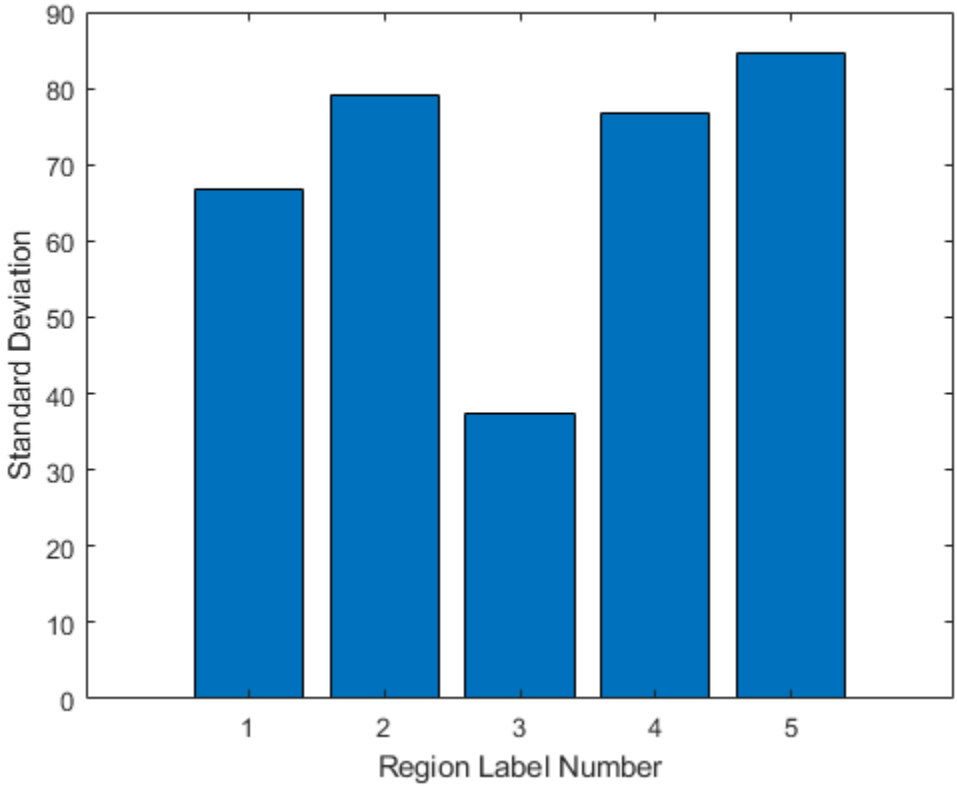
```
s = regionprops(BW,I,{'Centroid','PixelValues','BoundingBox'});
imshow(I)
title('Standard Deviation of Regions')
hold on
for k = 1 : numObj
    s(k).StandardDeviation = std(double(s(k).PixelValues));
    text(s(k).Centroid(1),s(k).Centroid(2), ...
        sprintf('%2.1f', s(k).StandardDeviation), ...
```

```
        'EdgeColor','b','Color','r');
end
hold off
```



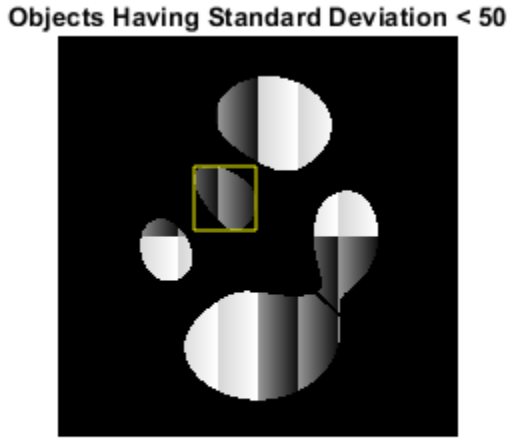
This figure shows the standard deviation measurement superimposed on each object in the image. You can also view the results in other ways, for example as a bar plot showing the standard deviation by label number.

```
figure
bar(1:numObj,[s.StandardDeviation])
xlabel('Region Label Number')
ylabel('Standard Deviation')
```



You can use the plot to determine how to partition the data. For example, the following code identifies objects with a standard deviation lower than 50.

```
sStd = [s.StandardDeviation];
lowStd = find(sStd < 50);

imshow(I)
title('Objects Having Standard Deviation < 50')
hold on
for k = 1 : length(lowStd)
    rectangle('Position',s(lowStd(k)).BoundingBox,'EdgeColor','y');
end
hold off
```



See Also

`regionprops`

Finding the Length of a Pendulum in Motion

This example shows you how to calculate the length of a pendulum in motion. You can capture images in a time series with the Image Acquisition Toolbox™ and analyze them with the Image Processing Toolbox™.

Step 1: Acquire Images

Load the image frames of a pendulum in motion. The frames in the MAT-file `pendulum.mat` were acquired using the following functions in the Image Acquisition Toolbox.

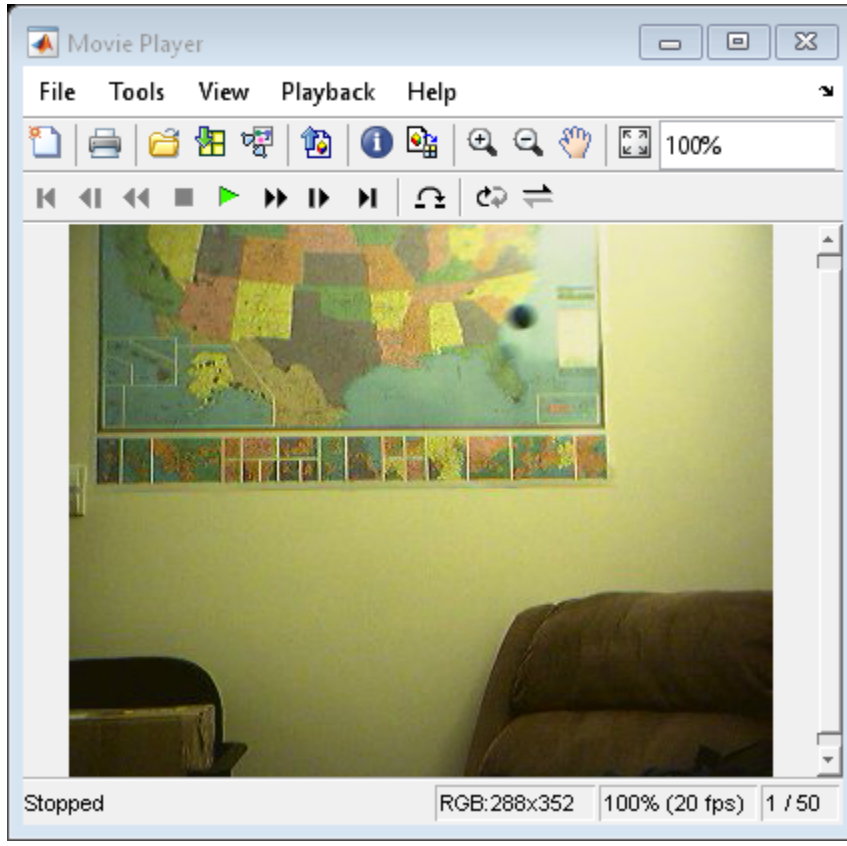
```
% Access an image acquisition device (video object).
% vidimage=videoinput('winvideo',1,'RGB24_352x288');

% Configure object to capture every fifth frame.
% vidimage.FrameGrabInterval = 5;

% Configure the number of frames to be logged.
% nFrames=50;
% vidimage.FramesPerTrigger = nFrames;

% Access the device's video source.
% src=getselectedsource(vidimage);

% Configure device to provide thirty frames per second.
% src.FrameRate = 30;

% Open a live preview window. Focus camera onto a moving pendulum.
% preview(vidimage);

% Initiate the acquisition.
% start(vidimage);

% Wait for data logging to finish before retrieving the data.
% wait(vidimage, 10);

% Extract frames from memory.
% frames = getdata(vidimage);

% Clean up. Delete and clear associated variables.
% delete(vidimage)
% clear vidimage
```

```
% load MAT-file

load pendulum;
```

Step 2: Explore Sequence with IMPLAY

Run the following command to explore the image sequence in `implay`.

```
implay(frames);
```



Step 3: Select Region where Pendulum is Swinging

You can see that the pendulum is swinging in the upper half of each frame in the image series. Create a new series of frames that contains only the region where the pendulum is swinging.

To crop a series of frames using `imcrop`, first perform `imcrop` on one frame and store its output image. Then use the previous output's size to create a series of frame regions. For convenience, use the `rect` that was loaded by `pendulum.mat` in `imcrop`.

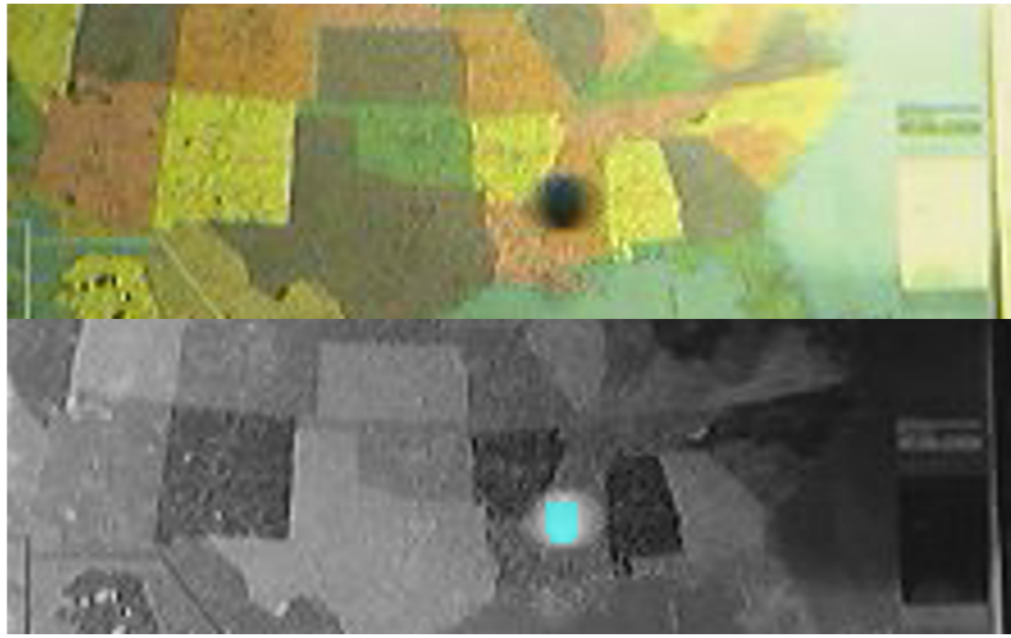
```
nFrames = size(frames,4);
first_frame = frames(:,:,:,1);
first_region = imcrop(first_frame,rect);
frame_regions = repmat(uint8(0), [size(first_region) nFrames]);
for count = 1:nFrames
  frame_regions(:,:,:,count) = imcrop(frames(:,:,:,count),rect);
end
imshow(frames(:,:,:,1))
```

Step 4: Segment the Pendulum in Each Frame

Notice that the pendulum is much darker than the background. You can segment the pendulum in each frame by converting the frame to grayscale, thresholding it using `imbinarize`, and removing background structures using `imopen` and `imclearborder`.

initialize array to contain the segmented pendulum frames.

```
seg_pend = false([size(first_region,1) size(first_region,2) nFrames]);
centroids = zeros(nFrames,2);
se_disk = strel('disk',3);
for count = 1:nFrames
    fr = frame_regions(:,:,:,count);

    gfr = rgb2gray(fr);
    gfr = imcomplement(gfr);

    bw = imbinarize(gfr,.7);  % threshold is determined experimentally
    bw = imopen(bw,se_disk);
    bw = imclearborder(bw);
    seg_pend(:,:,count) = bw;

    montage({fr,labeloverlay(gfr,bw)});
    pause(0.2)

end
```



Step 5: Find the Center of the Segmented Pendulum in Each Frame

You can see that the shape of the pendulum varied in different frames. This is not a serious issue because you just need its center. You will use the pendulum centers to find the length of the pendulum.

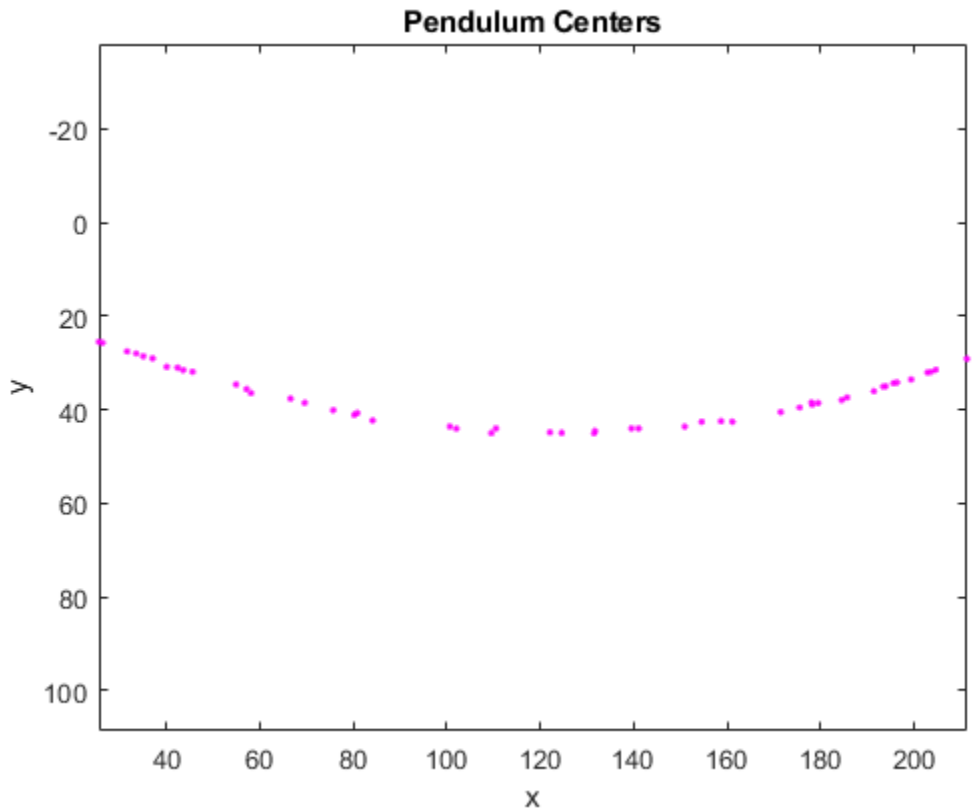
Use `regionprops` to calculate the center of the pendulum.

```
pend_centers = zeros(nFrames,2);
for count = 1:nFrames
    property = regionprops(seg_pend(:,:,count), 'Centroid');
    pend_centers(count,:) = property.Centroid;
end
```

Display pendulum centers using `plot`.

```
x = pend_centers(:,1);
y = pend_centers(:,2);
```

```
figure
plot(x,y,'m.')
axis ij
axis equal
hold on;
xlabel('x');
ylabel('y');
title('Pendulum Centers');
```



Step 6: Calculate Radius by Fitting a Circle Through Pendulum Centers

Rewrite the basic equation of a circle:

```
(x-xc)^2 + (y-yc)^2 = radius^2
```

where `(xc,yc)` is the center, in terms of parameters `a`, `b`, `c` as

`x^2 + y^2 + a*x + b*y + c = 0`

where `a = -2*xc`, `b = -2*yc`, and `c = xc^2 + yc^2 - radius^2`.

You can solve for parameters `a`, `b`, and `c` using the least squares method. Rewrite the above equation as

`a*x + b*y + c = -(x^2 + y^2)`

which can also be rewritten as

`[x y 1] * [a;b;c] = -x^2 - y^2`.

Solve this equation using the backslash(\) operator.

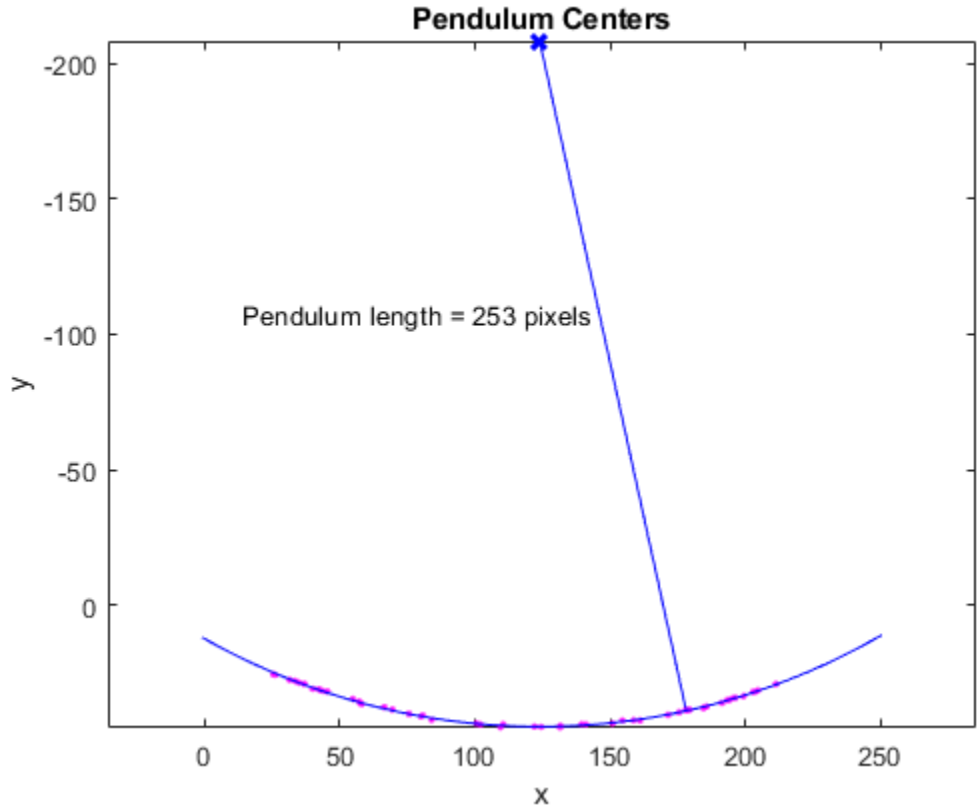
The circle radius is the length of the pendulum in pixels.

```
abc = [x y ones(length(x),1)] \ -(x.^2 + y.^2);
a = abc(1);
b = abc(2);
c = abc(3);
xc = -a/2;
yc = -b/2;
circle_radius = sqrt((xc^2 + yc^2) - c);
pendulum_length = round(circle_radius)
```

```
pendulum_length =

   253
```

Superimpose circle and circle center on the plot of pendulum centers.

```
circle_theta = pi/3:0.01:pi*2/3;
x_fit = circle_radius*cos(circle_theta)+xc;
y_fit = circle_radius*sin(circle_theta)+yc;

plot(x_fit,y_fit,'b-');
plot(xc,yc,'bx','LineWidth',2);
plot([xc x(1)],[yc y(1)],'b-');
text(xc-110,yc+100,sprintf('Pendulum length = %d pixels', pendulum_length));
```



Copyright 1993-2018 The MathWorks, Inc.

See Also

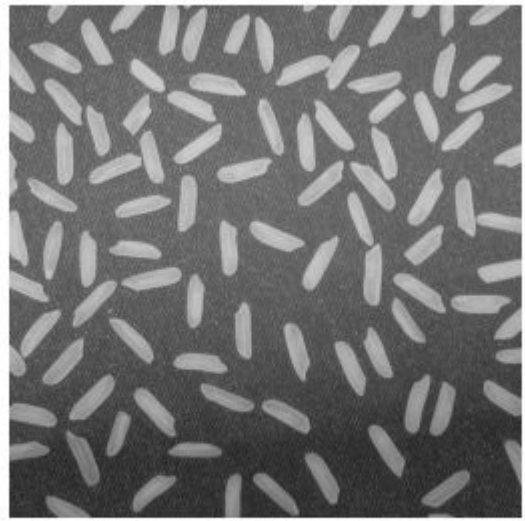
`imbinarize` | `imclearborder` | `imcomplement` | `imopen` | `labeloverlay` | `regionprops`

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);
```

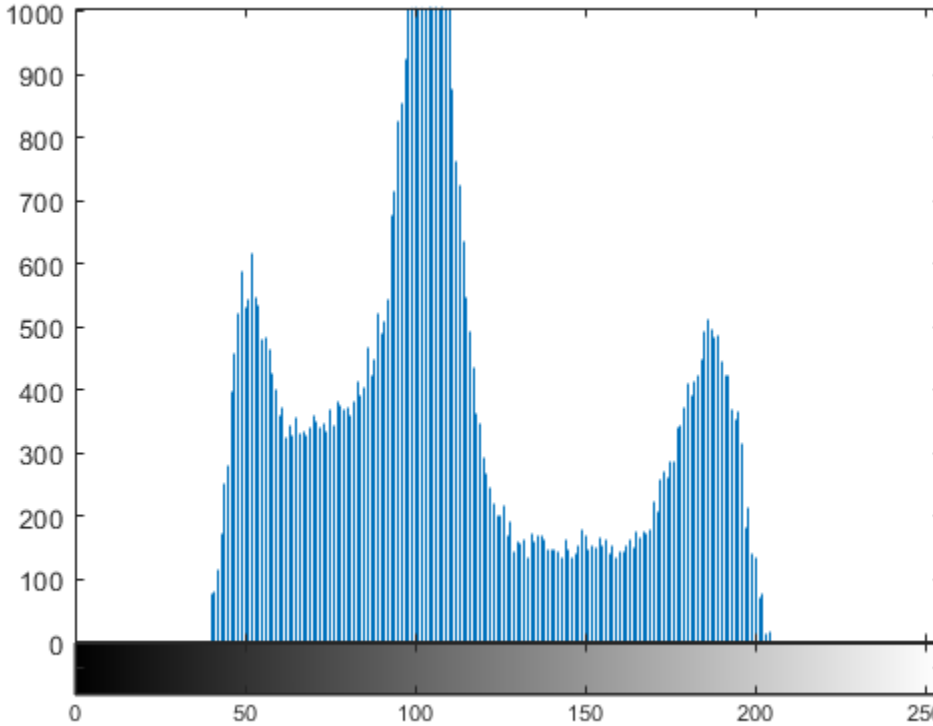


Image Mean, Standard Deviation, and Correlation Coefficient

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 several derivative estimators, each of which implements one of these definitions. 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 affected by noise, and more likely to detect true weak edges.

Detect Edges in Images

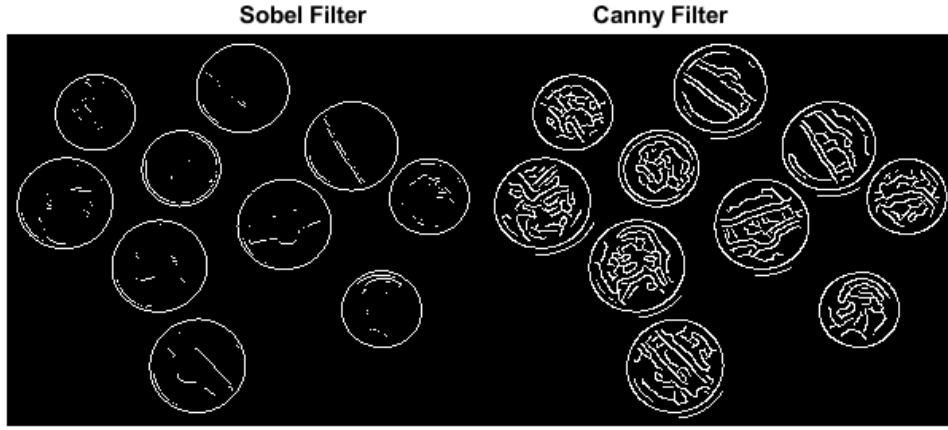
This example shows how to detect edges in an image using both the Canny edge detector and the Sobel edge detector.

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');
```


Sobel Filter
Canny Filter

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.

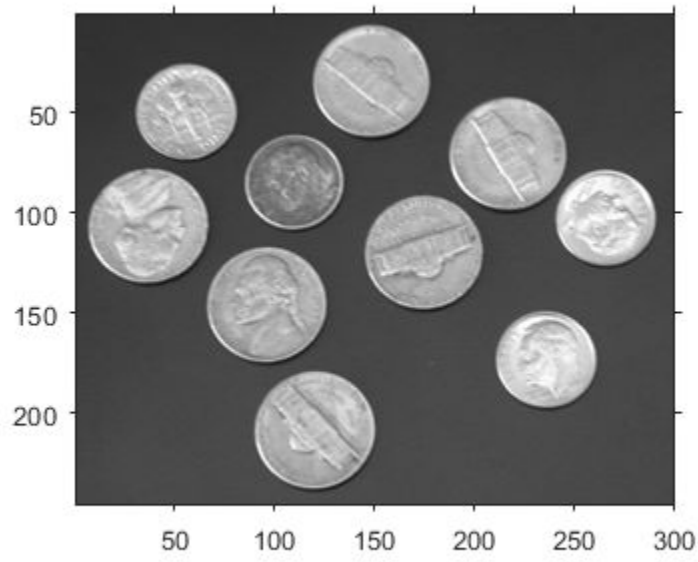
For both functions, the nonzero pixels in the binary image belong to an object, and pixels with the value 0 (zero) constitute the background.

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.

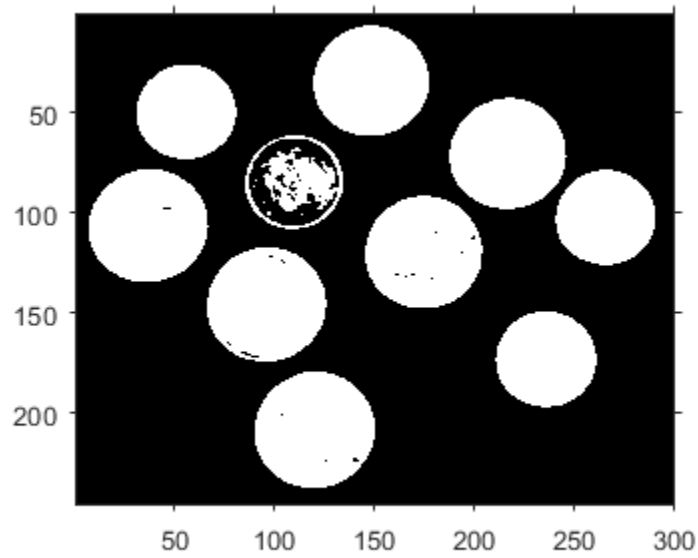
Read image and display it.

```
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)
```

```
dim = 1×2

   246   300
```

```
col = round(dim(2)/2)-90;
row = min(find(BW(:,col)))
```

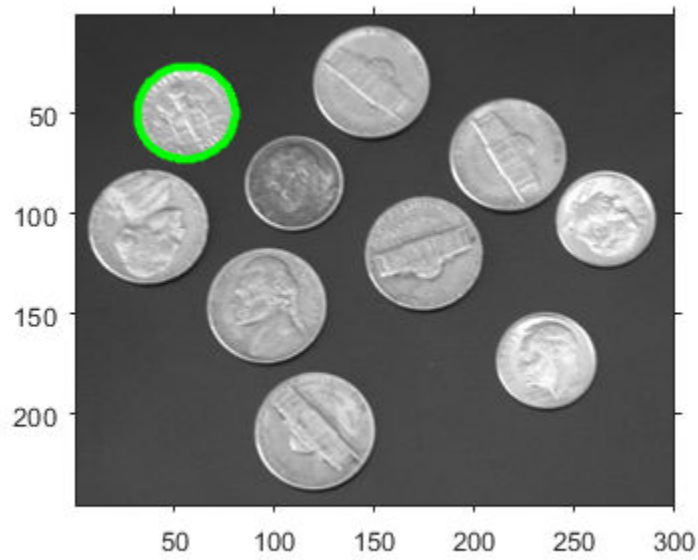
```
row = 27
```

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);
```

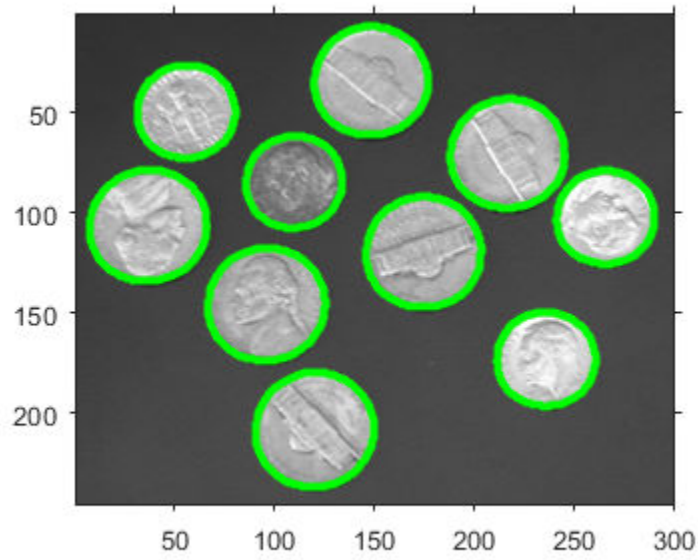


To trace the boundaries of all the coins in the image, use the `bwboundaries` function. By default, `bwboundaries` finds the boundaries of all objects in an image, including objects inside other objects. In the binary image used in this example, some of the coins contain black areas that `bwboundaries` 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
```

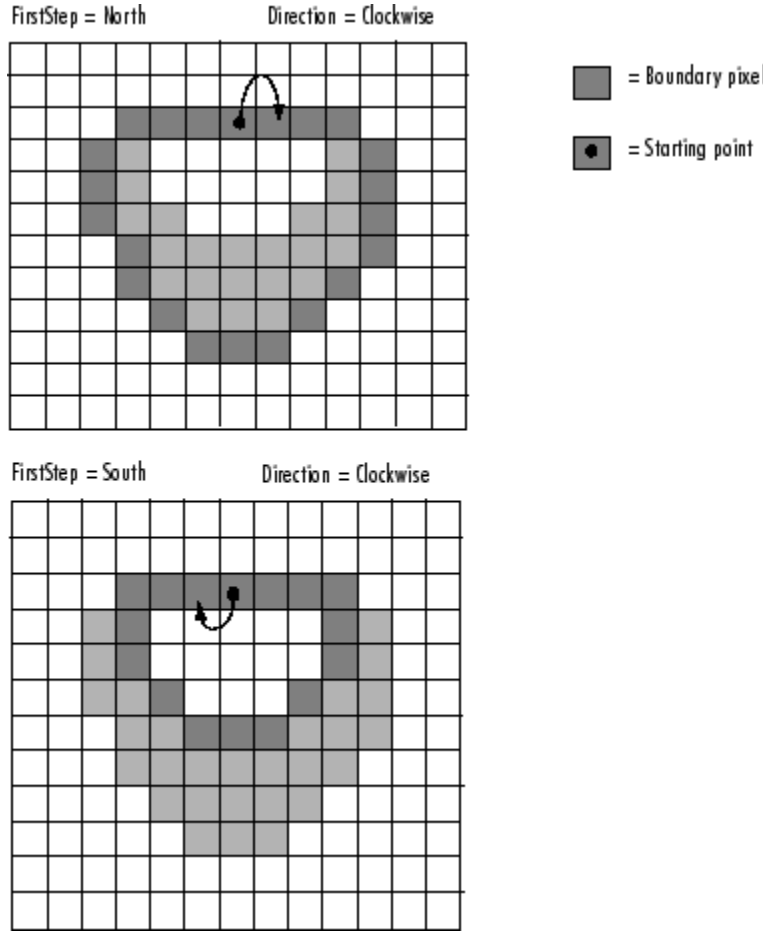


Select First Step and Direction for Tracing

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).



Impact of First Step and Direction Parameters on Boundary Tracing

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. Blocks can be as small as 1-by-1, unless you specify otherwise.

`qtdecomp` returns the quadtree decomposition as a sparse matrix, the same size as `I`. The nonzero elements represent the upper left corners of the blocks. The value of each nonzero element indicates the block size.

Perform Quadtree Decomposition on an Image

This example shows how to perform quadtree decomposition on a 512-by-512 grayscale image.

Read the grayscale image into the workspace.

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

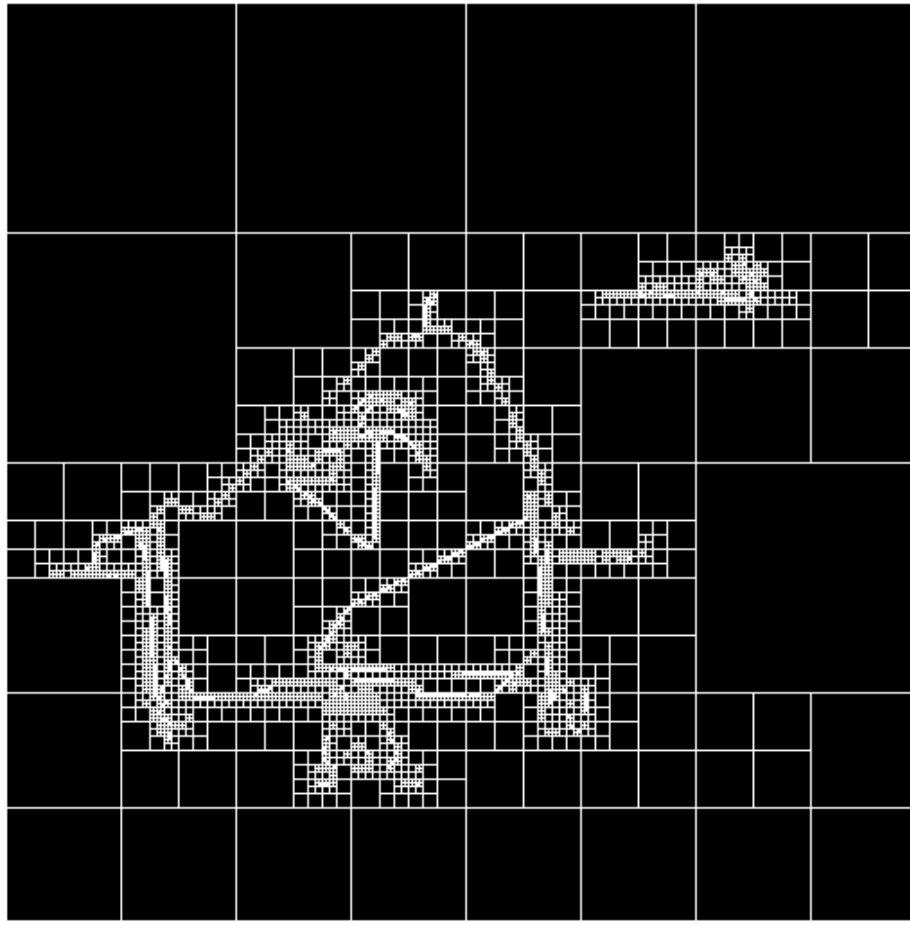
Perform the quadtree decomposition by calling the `qtdecomp` function, specifying as arguments the image and the test criteria used to determine the homogeneity of each block in the decomposition. For example, the criterion might be a threshold calculation such as `max(block(:)) - min(block(:)) >= 0.27`. 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.

```
S = qtdecomp(I,0.27);
```

View a block representation of the quadtree decomposition. Each black square represents a homogeneous block, and the white lines represent the boundaries between blocks.

Notice how the blocks are smaller in areas corresponding to large changes in intensity in the image.

```
blocks = repmat(uint8(0),size(S));

for dim = [512 256 128 64 32 16 8 4 2 1];
  numblocks = length(find(S==dim));
  if (numblocks > 0)
    values = repmat(uint8(1),[dim dim numblocks]);
    values(2:dim,2:dim,:) = 0;
    blocks = qtsetblk(blocks,S,dim,values);
  end
end

blocks(end,1:end) = 1;
blocks(1:end,end) = 1;

imshow(I), figure, imshow(blocks,[])
```



Detect and Measure Circular Objects in an Image

This example shows how to use `imfindcircles` to automatically detect circles or circular objects in an image. It also shows the use of `viscircles` to visualize the detected circles.

Step 1: Load Image

This example uses an image of round plastic chips of various colors.

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

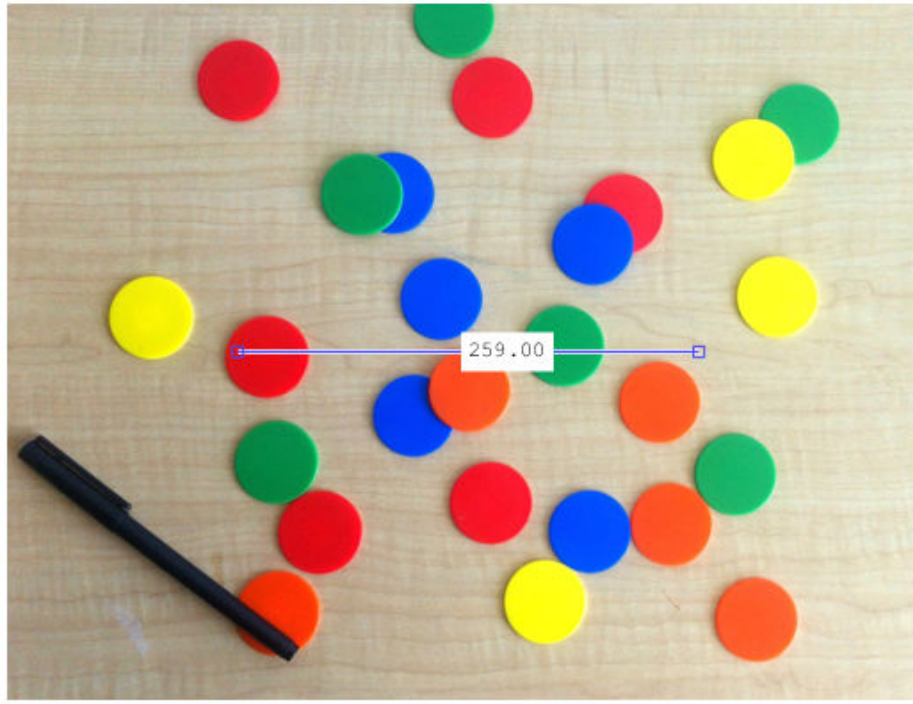
Besides having plenty of circles to detect, there are a few interesting things going on in this image from a circle detection point-of-view:

1. There are chips of different colors, which have different contrasts with respect to the background. On one end, the blue and red ones have strong contrast on this background. On the other end, some of the yellow chips do not contrast well with the background.
2. Notice how some chips are on top of each other and some others that are close together and almost touching each other. Overlapping object boundaries and object occlusion are usually challenging scenarios for object detection.

Step 2: Determine Radius Range for Searching Circles

`imfindcircles` needs a radius range to search for the circles. A quick way to find the appropriate radius range is to use the interactive tool `imdistline` to get an approximate estimate of the radii of various objects.

```
d = imdistline;
```



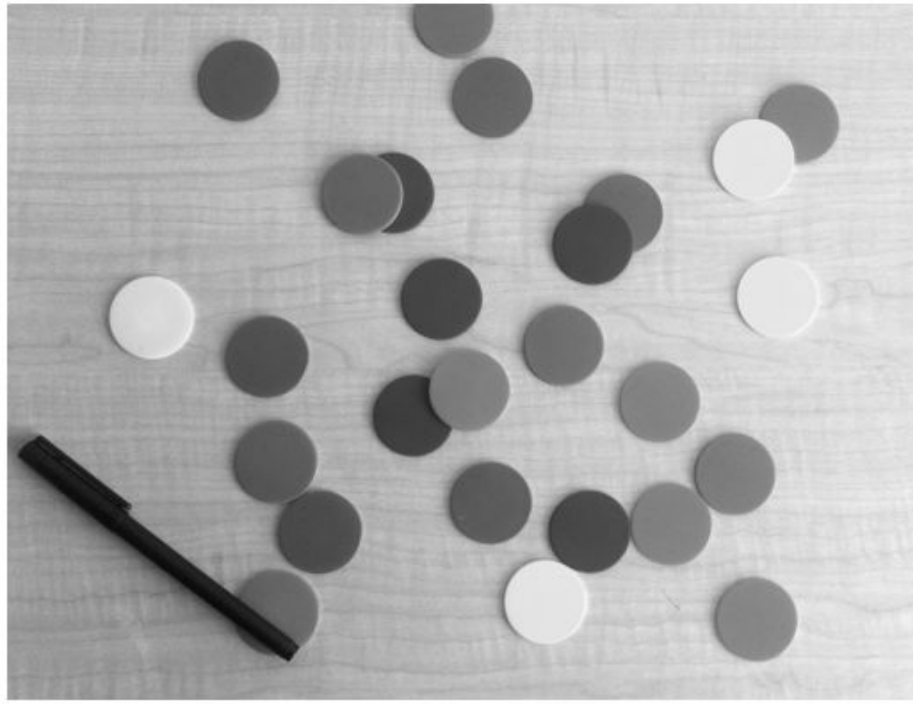
`imdistline` creates a draggable tool that can be moved to fit across a chip and the numbers can be read to get an approximate estimate of its radius. Most chips have radius in the range of 21-23 pixels. Use a slightly larger radius range of 20-25 pixels just to be sure. Before that remove the `imdistline` tool.

```
delete(d)
```

Step 3: Initial Attempt to Find Circles

Call `imfindcircles` on this image with the search radius of [20 25] pixels. Before that, it is a good practice to ask whether the objects are brighter or darker than the background. To answer that question, look at the grayscale version of this image.

```
gray_image = rgb2gray(rgb);
imshow(gray_image)
```



The background is quite bright and most of the chips are darker than the background. But, by default, `imfindcircles` finds circular objects that are brighter than the background. So, set the parameter 'ObjectPolarity' to 'dark' in `imfindcircles` to search for dark circles.

```
[centers,radii] = imfindcircles(rgb,[20 25],'ObjectPolarity','dark')
```

```
centers =

     []


radii =
```

```
     []
```

Note that the outputs `centers` and `radii` are empty, which means that no circles were found. This happens frequently because `imfindcircles` is a circle *detector*, and similar to most detectors, `imfindcircles` has an internal *detection threshold* that determines its sensitivity. In simple terms it means that the detector's confidence in a certain (circle) detection has to be greater than a certain level before it is considered a *valid* detection. `imfindcircles` has a parameter 'Sensitivity' which can be used to control this internal threshold, and consequently, the sensitivity of the algorithm. A higher 'Sensitivity' value sets the detection threshold lower and leads to detecting more circles. This is similar to the sensitivity control on the motion detectors used in home security systems.

Step 4: Increase Detection Sensitivity

Coming back to the chip image, it is possible that at the default sensitivity level all the circles are lower than the internal threshold, which is why no circles were detected. By default, 'Sensitivity', which is a number between 0 and 1, is set to 0.85. Increase 'Sensitivity' to 0.9.

```
[centers,radii] = imfindcircles(rgb,[20 25],'ObjectPolarity','dark', ...
    'Sensitivity',0.9)
```

```
centers = 8×2

  146.1895  198.5824
  328.8132  135.5883
  130.3134   43.8039
  175.2698  297.0583
  312.2831  192.3709
  327.1315  297.0081
  243.9893  166.4538
  271.5873  280.8920


radii = 8×1

   23.1604
   22.5710
   22.9576
   23.7356
   22.9551
   22.9929
   22.9055
```

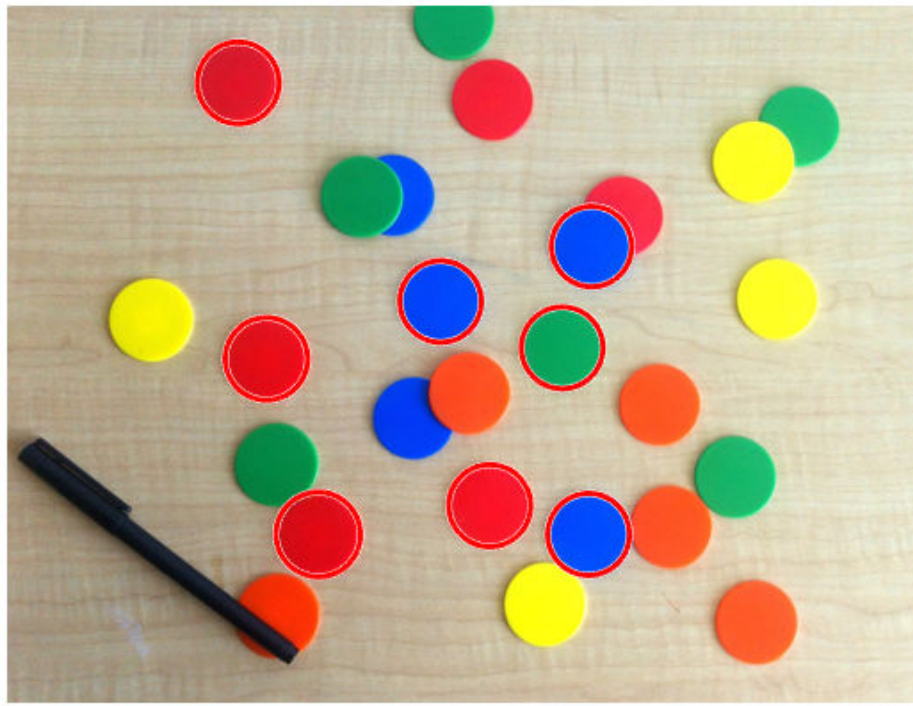
```
23.0298
```

This time `imfindcircles` found some circles - eight to be precise. `centers` contains the locations of circle centers and `radii` contains the estimated radii of those circles.

Step 5: Draw the Circles on the Image

The function `viscircles` can be used to draw circles on the image. Output variables `centers` and `radii` from `imfindcircles` can be passed directly to `viscircles`.

```
imshow(rgb)
h = viscircles(centers,radii);
```



The circle centers seem correctly positioned and their corresponding radii seem to match well to the actual chips. But still quite a few chips were missed. Try increasing the 'Sensitivity' even more, to 0.92.

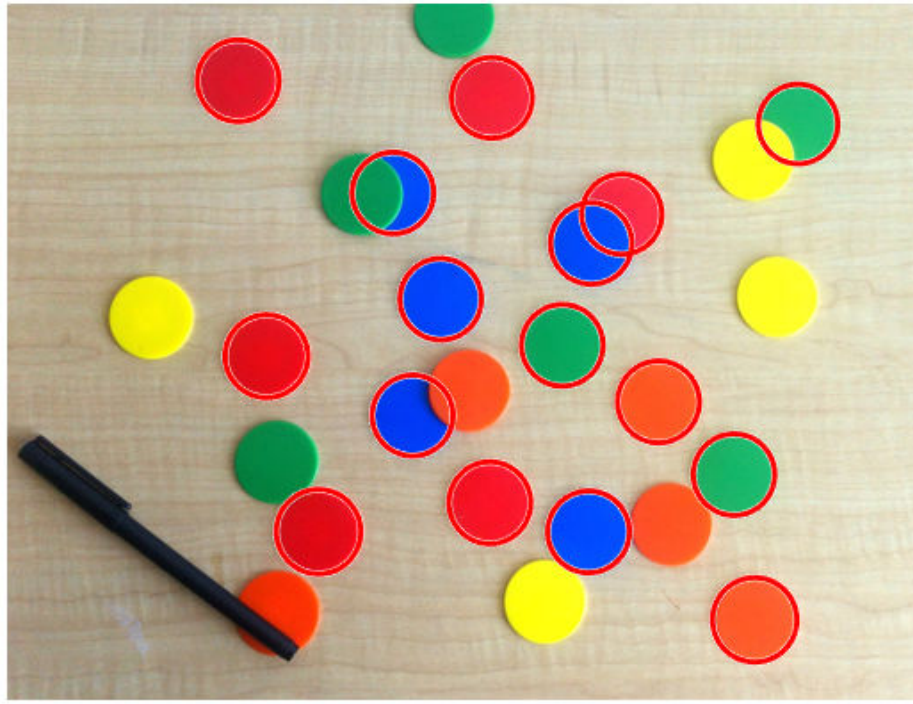
```
[centers,radii] = imfindcircles(rgb,[20 25],'ObjectPolarity','dark', ...
    'Sensitivity',0.92);
```

```
length(centers)
```

```
ans = 16
```

So increasing 'Sensitivity' gets us even more circles. Plot these circles on the image again.

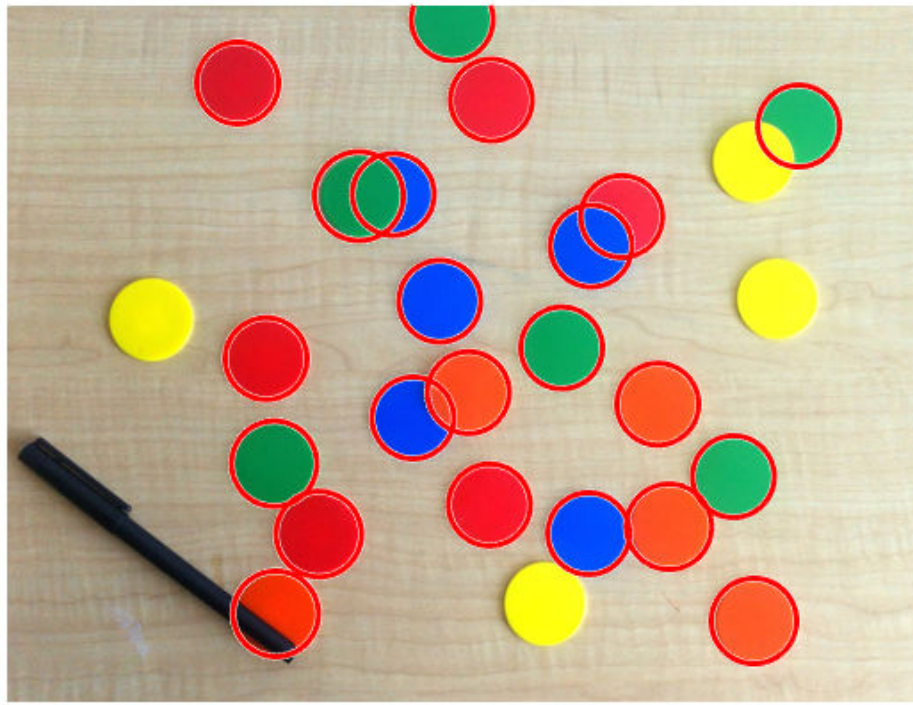
```
delete(h)  % Delete previously drawn circles
h = viscircles(centers,radii);
```



Step 6: Use the Second Method (Two-stage) for Finding Circles

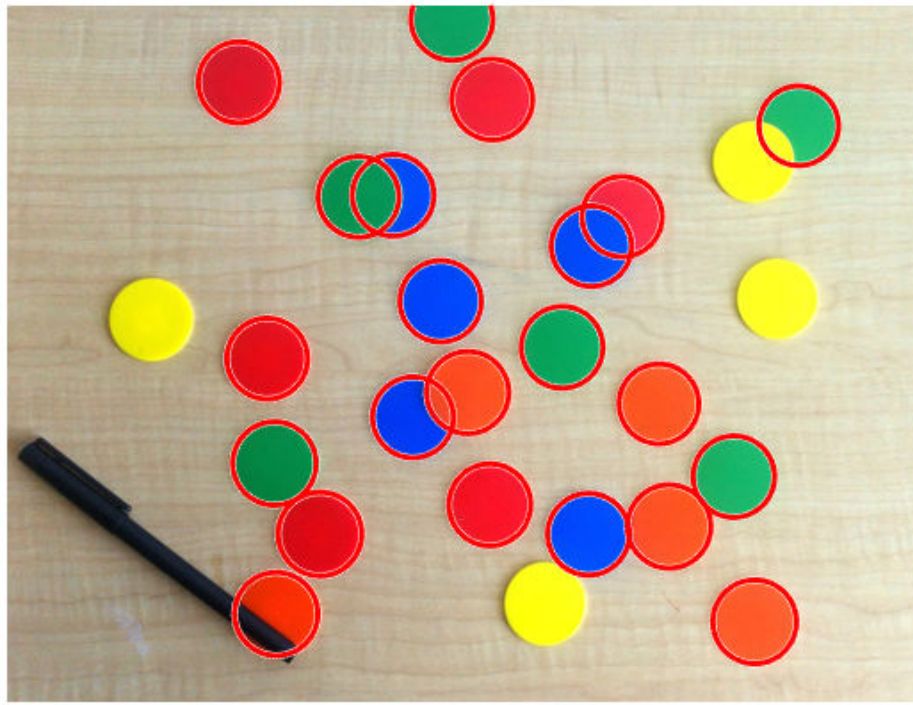
This result looks better. `imfindcircles` has two different methods for finding circles. So far the default method, called the *phase coding* method, was used for detecting circles. There's another method, popularly called the *two-stage* method, that is available in `imfindcircles`. Use the two-stage method and show the results.

```
[centers,radii] = imfindcircles(rgb,[20 25],'ObjectPolarity','dark', ...
          'Sensitivity',0.92,'Method','twostage');

delete(h)
h = viscircles(centers,radii);
```



The two-stage method is detecting more circles, at the Sensitivity of 0.92. In general, these two method are complementary in that have they have different strengths. The Phase coding method is typically faster and slightly more robust to noise than the two-stage method. But it may also need higher 'Sensitivity' levels to get the same number of detections as the two-stage method. For example, the phase coding method also finds the same chips if the 'Sensitivity' level is raised higher, say to 0.95.

```
[centers,radii] = imfindcircles(rgb,[20 25],'ObjectPolarity','dark', ...
          'Sensitivity',0.95);

delete(h)
viscircles(centers,radii);
```

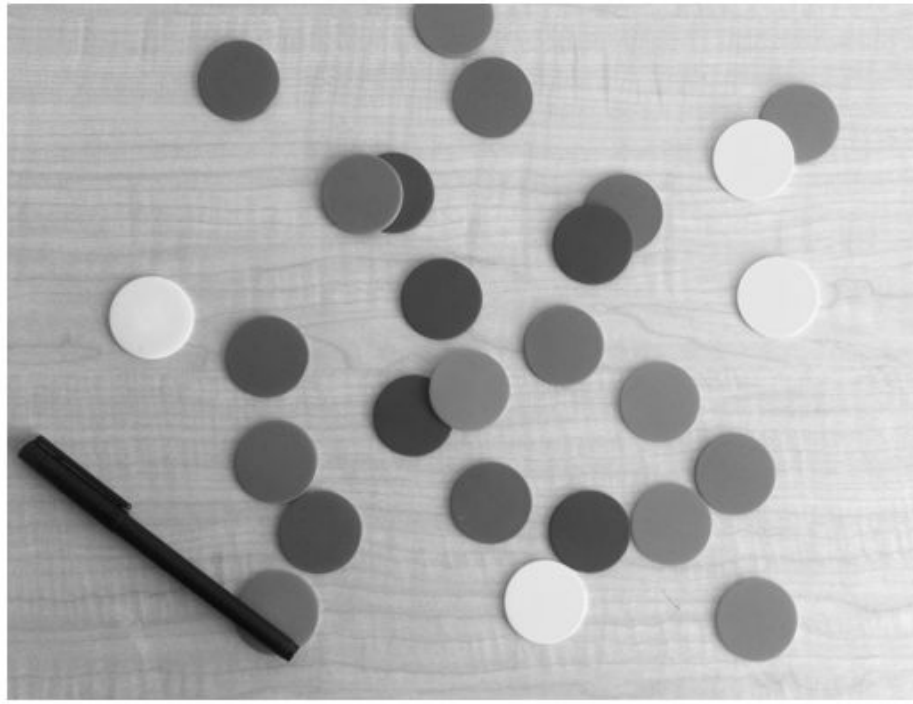


Note that both the methods in `imfindcircles` find the centers and radii of the partially visible (occluded) chips accurately.

Step 7: Why are Some Circles Still Getting Missed?

Looking at the last result, it is curious that `imfindcircles` does not find the yellow chips in the image. The yellow chips do not have strong contrast with the background. In fact they seem to have very similar intensities as the background. Is it possible that the yellow chips are not really 'darker' than the background as was assumed? To confirm, show the grayscale version of this image again.

```
imshow(gray_image)
```



Step 8: Find 'Bright' Circles in the Image

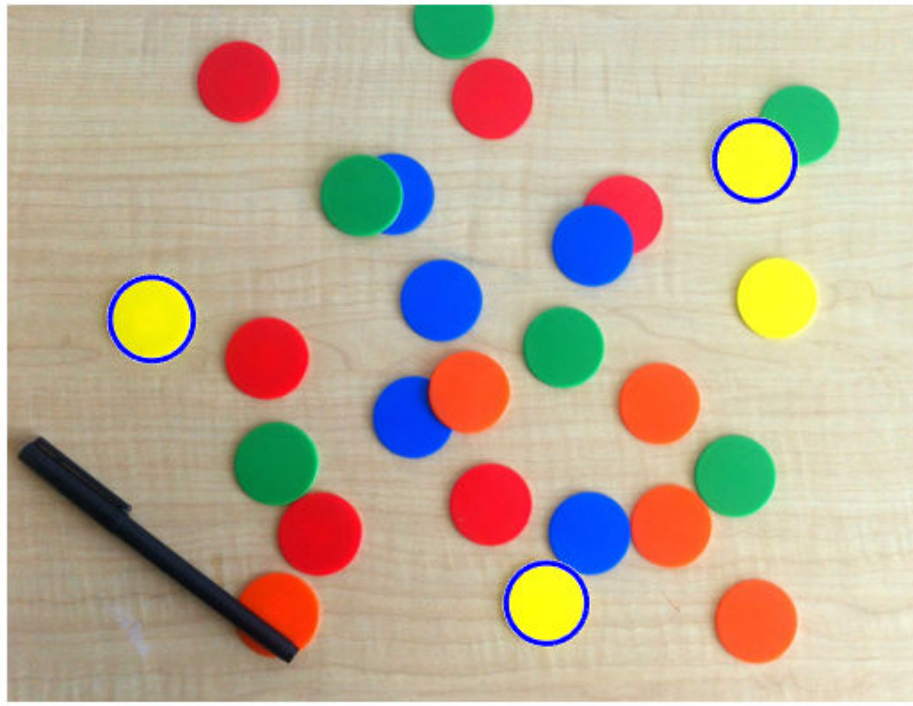
The yellow chips are almost the same intensity, maybe even brighter, as compared to the background. Therefore, to detect the yellow chips, change 'ObjectPolarity' to 'bright'.

```
[centersBright,radiiBright] = imfindcircles(rgb,[20 25], ...
    'ObjectPolarity','bright','Sensitivity',0.92);
```

Step 9: Draw 'Bright' Circles with Different Color

Draw the *bright* circles in a different color, by changing the 'Color' parameter in `viscircles`.

```
imshow(rgb)

hBright = viscircles(centersBright, radiiBright,'Color','b');
```



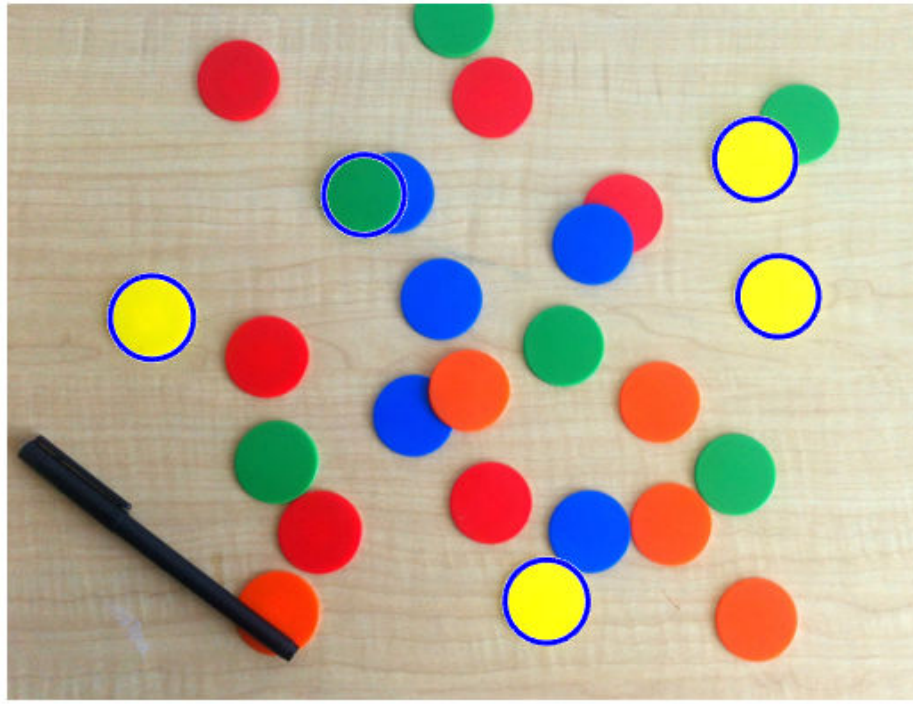
Note that three of the missing yellow chips were found, but one yellow chip is still missing. These yellow chips are hard to find because they don't *stand out* as well as others on this background.

Step 10: Lower the Value of 'EdgeThreshold'

There is another parameter in `imfindcircles` which may be useful here, namely 'EdgeThreshold'. To find circles, `imfindcircles` uses only the edge pixels in the image. These edge pixels are essentially pixels with high gradient value. The 'EdgeThreshold'

parameter controls how *high* the gradient value at a pixel has to be before it is considered an edge pixel and included in computation. A high value (closer to 1) for this parameter will allow only the strong edges (higher gradient values) to be included, whereas a low value (closer to 0) is more permissive and includes even the weaker edges (lower gradient values) in computation. In case of the missing yellow chip, since the contrast is low, some of the boundary pixels (on the circumference of the chip) are expected to have low gradient values. Therefore, lower the 'EdgeThreshold' parameter to ensure that the most of the edge pixels for the yellow chip are included in computation.

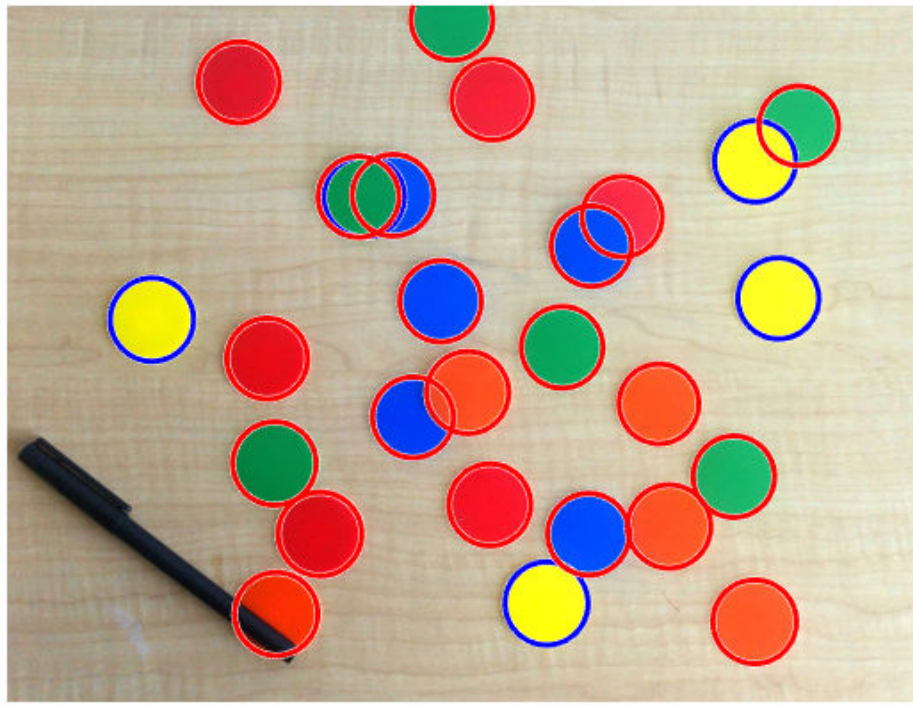
```
[centersBright,radiiBright,metricBright] = imfindcircles(rgb,[20 25], ...
    'ObjectPolarity','bright','Sensitivity',0.92,'EdgeThreshold',0.1);

delete(hBright)
hBright = viscircles(centersBright, radiiBright,'Color','b');
```



Step 11: Draw 'Dark' and 'Bright' Circles Together

Now `imfindcircles` finds all of the yellow ones, and a green one too. Draw these chips in blue, together with the other chips that were found earlier (with 'ObjectPolarity' set to 'dark'), in red.

```
h = viscircles(centers,radii);
```



All the circles are detected. A final word - it should be noted that changing the parameters to be more aggressive in detection may find more circles, but it also increases the likelihood of detecting false circles. There is a trade-off between the number of true circles that can be found (detection rate) and the number of false circles that are found with them (false alarm rate).

Happy circle hunting!

See Also

`imfindcircles` | `viscircles`

Related Examples

- “Identifying Round Objects” on page 11-58
- “Measuring the Radius of a Roll of Tape” on page 11-73

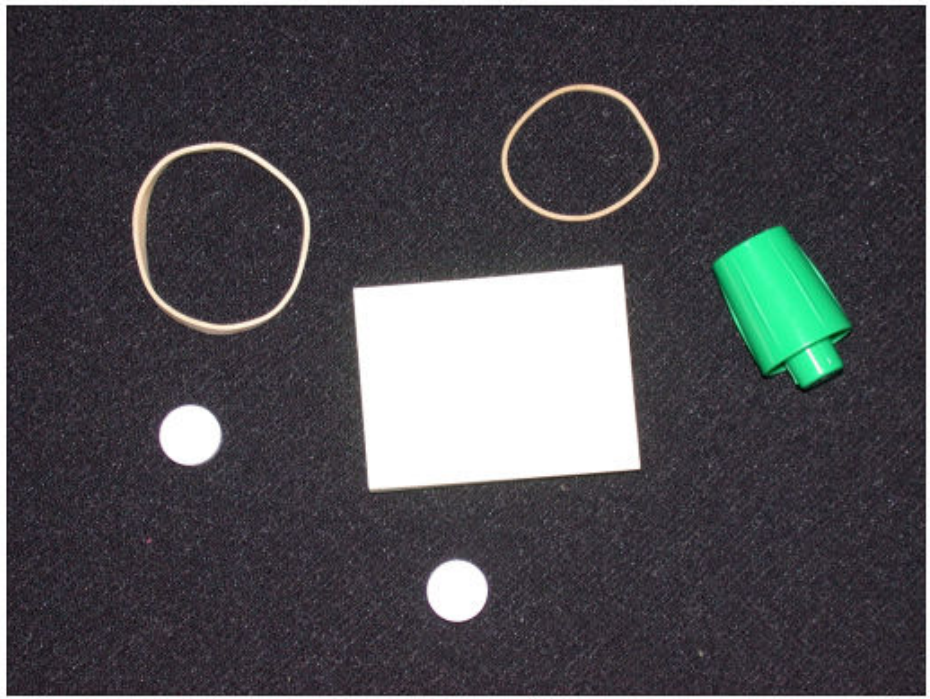
Identifying Round Objects

This example shows how to classify objects based on their roundness using `bwboundaries`, a boundary tracing routine.

Step 1: Read Image

Read in `pills_etc.png`.

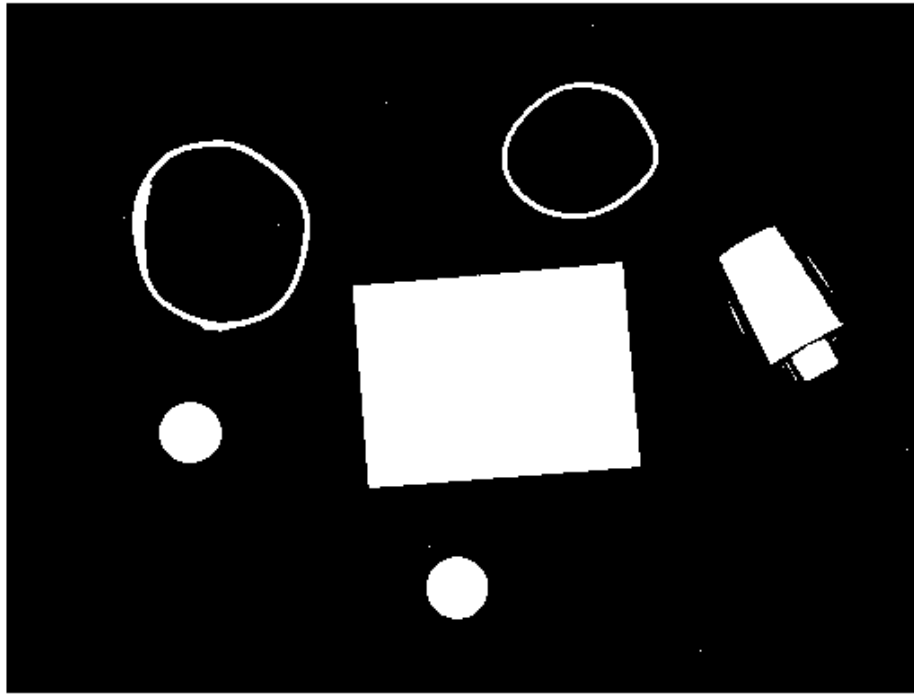
```
RGB = imread('pillsetc.png');
imshow(RGB);
```



Step 2: Threshold the Image

Convert the image to black and white in order to prepare for boundary tracing using `bwboundaries`.

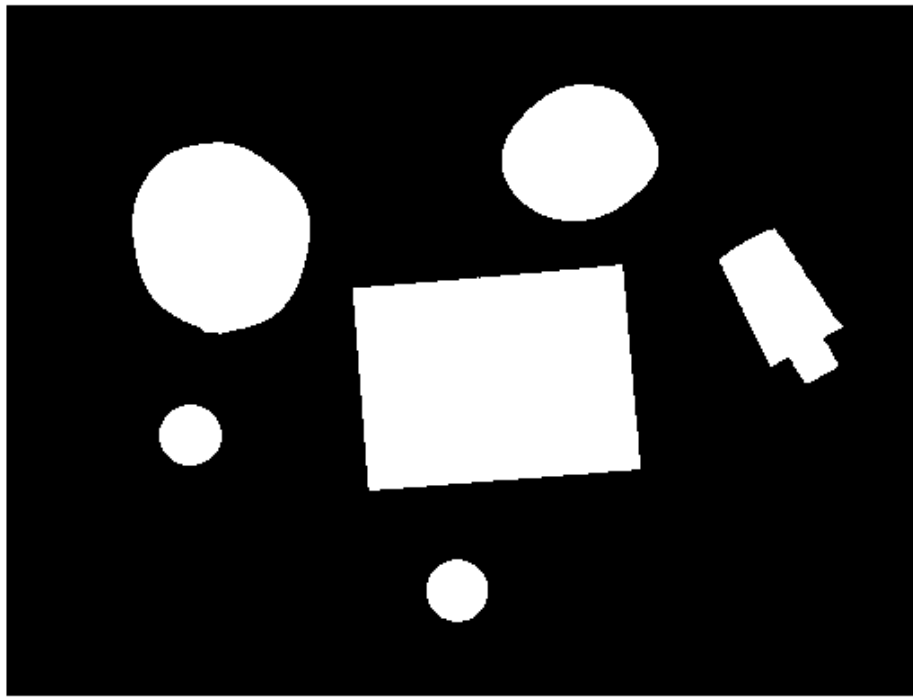
```
I = rgb2gray(RGB);
bw = imbinarize(I);
imshow(bw)
```



Step 3: Remove the Noise

Using morphology functions, remove pixels which do not belong to the objects of interest.

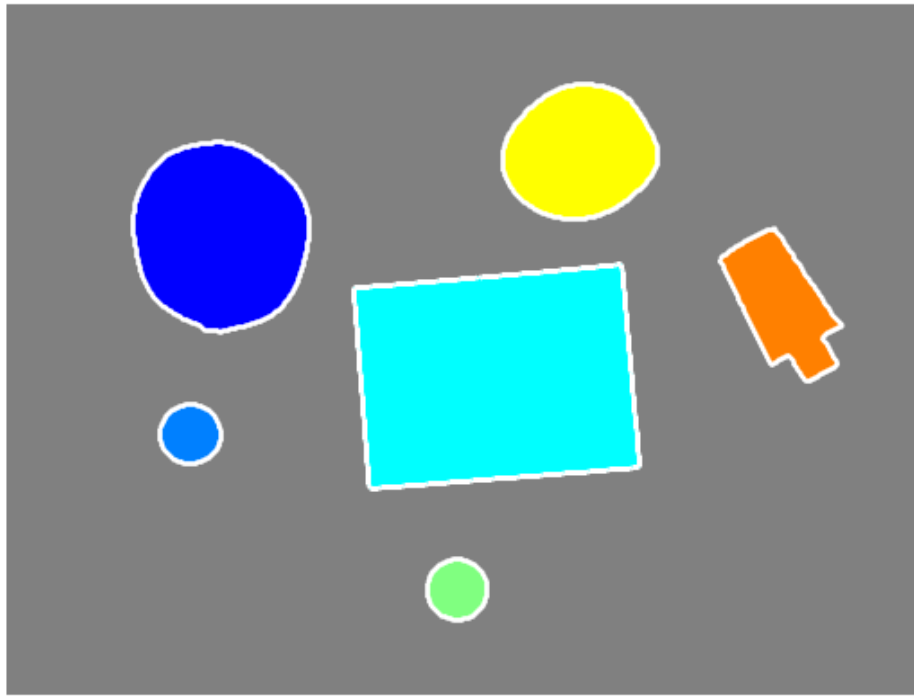
```
% remove all object containing fewer than 30 pixels
bw = bwareaopen(bw,30);

% fill a gap in the pen's cap
se = strel('disk',2);
bw = imclose(bw,se);

% fill any holes, so that regionprops can be used to estimate
% the area enclosed by each of the boundaries
bw = imfill(bw,'holes');

imshow(bw)
```



Step 4: Find the Boundaries

Concentrate only on the exterior boundaries. Option 'noholes' will accelerate the processing by preventing `bwboundaries` from searching for inner contours.

```
[B,L] = bwboundaries(bw,'noholes');

% Display the label matrix and draw each boundary
imshow(label2rgb(L, @jet, [.5 .5 .5]))
hold on
for k = 1:length(B)
  boundary = B{k};
  plot(boundary(:,2), boundary(:,1), 'w', 'LineWidth', 2)
end
```



Step 5: Determine which Objects are Round

Estimate each object's area and perimeter. Use these results to form a simple metric indicating the roundness of an object:

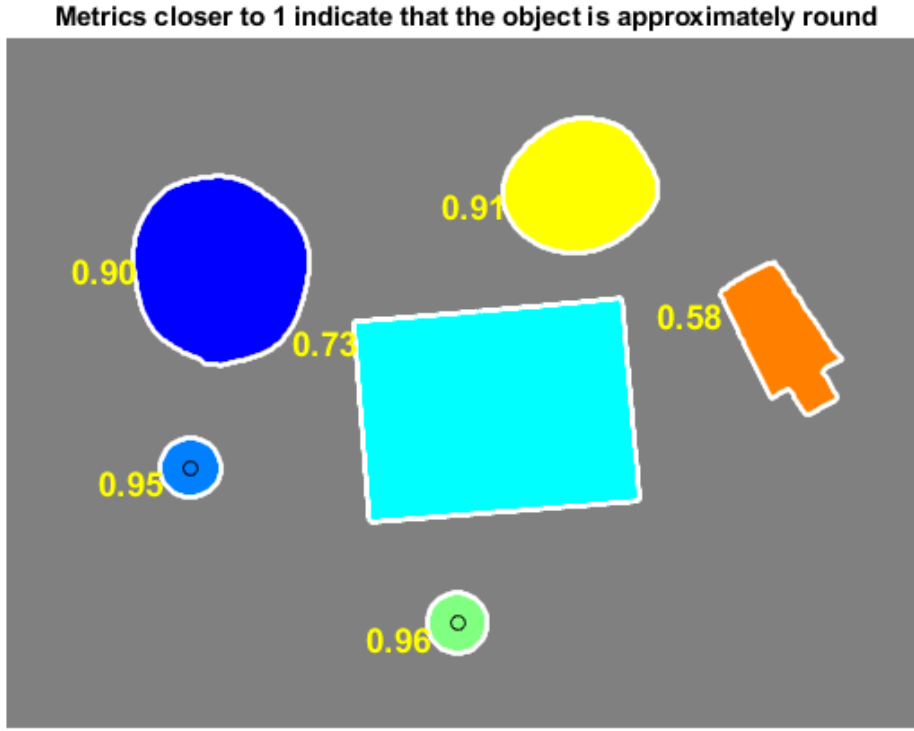
`metric = 4*pi*area/perimeter^2.`

This metric is equal to one only for a circle and it is less than one for any other shape. The discrimination process can be controlled by setting an appropriate threshold. In this example use a threshold of 0.94 so that only the pills will be classified as round.

Use `regionprops` to obtain estimates of the area for all of the objects. Notice that the label matrix returned by `bwboundaries` can be reused by `regionprops`.

```
stats = regionprops(L,'Area','Centroid');

threshold = 0.94;

% loop over the boundaries
for k = 1:length(B)

  % obtain (X,Y) boundary coordinates corresponding to label 'k'
  boundary = B{k};

  % compute a simple estimate of the object's perimeter
  delta_sq = diff(boundary).^2;
  perimeter = sum(sqrt(sum(delta_sq,2)));

  % obtain the area calculation corresponding to label 'k'
  area = stats(k).Area;

  % compute the roundness metric
  metric = 4*pi*area/perimeter^2;

  % display the results
  metric_string = sprintf('%2.2f',metric);

  % mark objects above the threshold with a black circle
  if metric > threshold
    centroid = stats(k).Centroid;
    plot(centroid(1),centroid(2),'ko');
  end

  text(boundary(1,2)-35,boundary(1,1)+13,metric_string,'Color','y',...
```

```
        'FontSize',14,'FontWeight','bold');

end

title(['Metrics closer to 1 indicate that ',...
        'the object is approximately round']);
```



See Also

bwareaopen | bwboundaries | imbinarize | imclose | imfill | label2rgb | regionprops | strel

Related Examples

- “Detect and Measure Circular Objects in an Image” on page 11-41
- “Measuring the Radius of a Roll of Tape” on page 11-73

Measuring Angle of Intersection

This example shows how to measure the angle and point of intersection between two beams using `bwtraceboundary`, which is a boundary tracing routine. A common task in machine vision applications is hands-free measurement using image acquisition and image processing techniques.

Step 1: Load Image

Read in `gantrycrane.png` and draw arrows pointing to two beams of interest. It is an image of a gantry crane used to assemble a bridge.

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

text(size(RGB,2),size(RGB,1)+15,'Image courtesy of Jeff Mather',...
     'FontSize',7,'HorizontalAlignment','right');

line([300 328],[85 103],'color',[1 1 0]);
line([268 255],[85 140],'color',[1 1 0]);

text(150,72,'Measure the angle between these beams','Color','y',...
     'FontWeight', 'bold');
```

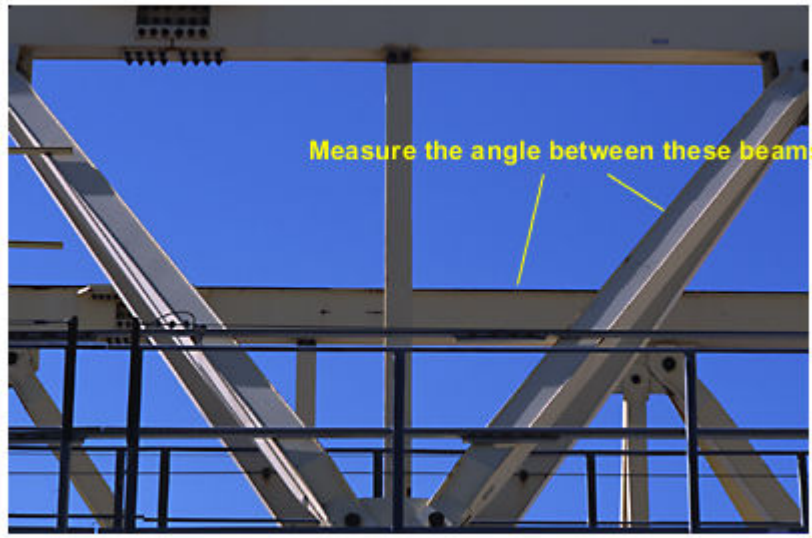


Image courtesy of Jeff Mather

Step 2: Extract the Region of Interest

Crop the image to obtain only the beams of the gantry crane chosen earlier. This step will make it easier to extract the edges of the two metal beams.

```
% you can obtain the coordinates of the rectangular region using
% pixel information displayed by imtool
start_row = 34;
start_col = 208;

cropRGB = RGB(start_row:163, start_col:400, :);

imshow(cropRGB)
```

```
% Store (X,Y) offsets for later use; subtract 1 so that each offset will
% correspond to the last pixel before the region of interest
offsetX = start_col-1;
offsetY = start_row-1;
```

Step 3: Threshold the Image

Convert the image to black and white for subsequent extraction of the edge coordinates using `bwtraceboundary` routine.

```
I = rgb2gray(cropRGB);
BW = imbinarize(I);
BW = ~BW;  % complement the image (objects of interest must be white)
imshow(BW)
```

Step 4: Find Initial Point on Each Boundary

The `bwtraceboundary` routine requires that you specify a single point on a boundary. This point is used as the starting location for the boundary tracing process.

To extract the edge of the lower beam, pick a column in the image and inspect it until a transition from a background pixel to the object pixel occurs. Store this location for later use in `bwtraceboundary` routine. Repeat this procedure for the other beam, but this time tracing horizontally.

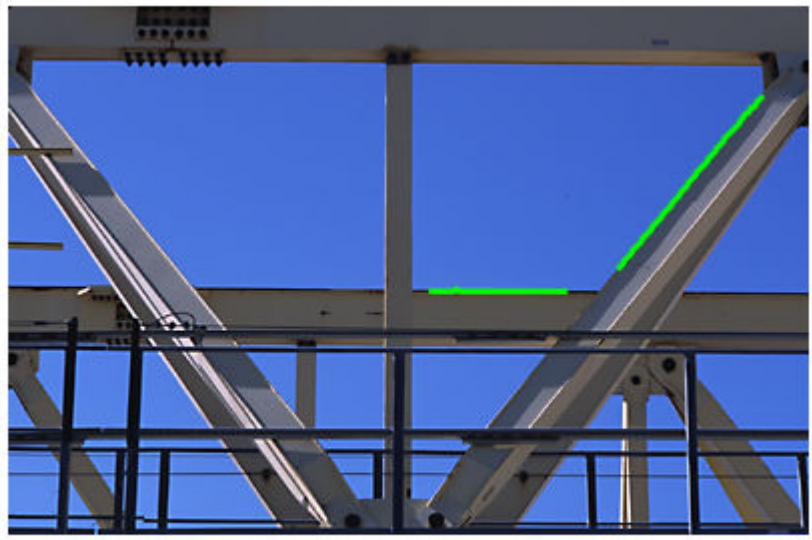
```
dim = size(BW);

% horizontal beam
col1 = 4;
row1 = find(BW(:,col1), 1);

% angled beam
row2 = 12;
col2 = find(BW(row2,:), 1);
```

Step 5: Trace the Boundaries

The `bwtraceboundary` routine is used to extract (X, Y) locations of the boundary points. In order to maximize the accuracy of the angle and point of intersection calculations, it is important to extract as many points belonging to the beam edges as possible. You should determine the number of points experimentally. Since the initial point for the horizontal

bar was obtained by scanning from north to south, it is safest to set the initial search step to point towards the outside of the object, i.e. 'North'.

```
boundary1 = bwtraceboundary(BW, [row1, col1], 'N', 8, 70);

% set the search direction to counterclockwise, in order to trace downward.
boundary2 = bwtraceboundary(BW, [row2, col2], 'E', 8, 90,'counter');

imshow(RGB); hold on;

% apply offsets in order to draw in the original image
plot(offsetX+boundary1(:,2),offsetY+boundary1(:,1),'g','LineWidth',2);
plot(offsetX+boundary2(:,2),offsetY+boundary2(:,1),'g','LineWidth',2);
```



Step 6: Fit Lines to the Boundaries

Although (X,Y) coordinates pairs were obtained in the previous step, not all of the points lie exactly on a line. Which ones should be used to compute the angle and point of

intersection? Assuming that all of the acquired points are equally important, fit lines to the boundary pixel locations.

The equation for a line is y = [x 1]*[a; b]. You can solve for parameters 'a' and 'b' in the least-squares sense by using `polyfit`.

```
ab1 = polyfit(boundary1(:,2), boundary1(:,1), 1);
ab2 = polyfit(boundary2(:,2), boundary2(:,1), 1);
```

Step 7: Find the Angle of Intersection

Use the dot product to find the angle.

```
vect1 = [1 ab1(1)]; % create a vector based on the line equation
vect2 = [1 ab2(1)];
dp = dot(vect1, vect2);

% compute vector lengths
length1 = sqrt(sum(vect1.^2));
length2 = sqrt(sum(vect2.^2));

% obtain the larger angle of intersection in degrees
angle = 180-acos(dp/(length1*length2))*180/pi
```

```
angle = 129.4971
```

Step 8: Find the Point of Intersection

Solve the system of two equations in order to obtain (X,Y) coordinates of the intersection point.

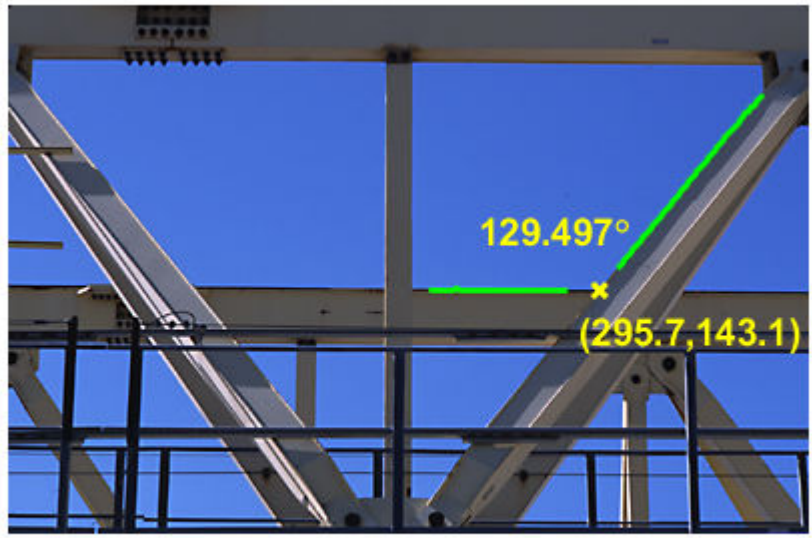
```
intersection = [1 ,-ab1(1); 1, -ab2(1)] \ [ab1(2); ab2(2)];
% apply offsets in order to compute the location in the original,
% i.e. not cropped, image.
intersection = intersection + [offsetY; offsetX]
```

```
intersection = 2×1

  143.0917
  295.7494
```

Step 9: Plot the Results.

```
inter_x = intersection(2);
inter_y = intersection(1);
```

```
% draw an "X" at the point of intersection
plot(inter_x,inter_y,'yx','LineWidth',2);

text(inter_x-60, inter_y-30, [sprintf('%1.3f',angle),'{\circ}'],...
     'Color','y','FontSize',14,'FontWeight','bold');

interString = sprintf('(%2.1f,%2.1f)', inter_x, inter_y);

text(inter_x-10, inter_y+20, interString,...
     'Color','y','FontSize',14,'FontWeight','bold');
```



See Also

bwboundaries | bwtraceboundary | imbinarize | polyfit

Related Examples

- “Detect Lines Using the Radon Transform” on page 9-34

More About

- “Hough Transform” on page 9-21
- “Radon Transform” on page 9-27

Measuring the Radius of a Roll of Tape

This example shows how to measure the radius of a roll of tape, which is partially obscured by the tape dispenser. Utilize `imfindcircles` to accomplish this task.

Step 1: Read Image

Read in `tape.png`.

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

hTxt = text(15,15,'Estimate radius of the roll of tape',...
     'FontWeight','bold','Color','y');
```

Step 2: Find the Circle

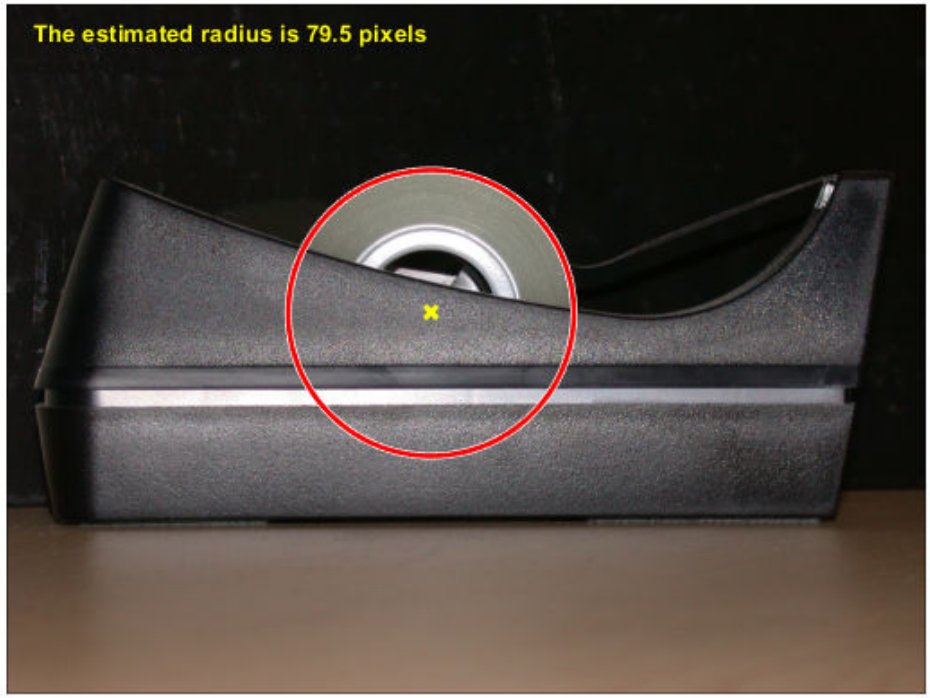
Find the center and the radius of the circle in the image using `imfindcircles`.

```
Rmin = 60;
Rmax = 100;
[center, radius] = imfindcircles(RGB,[Rmin Rmax],'Sensitivity',0.9)
```

```
center = 1×2

  236.9291  172.4747


radius = 79.5305
```

Step 3: Highlight the Circle Outline and Center

```
% Display the circle
viscircles(center,radius);

% Display the calculated center
hold on;
plot(center(:,1),center(:,2),'yx','LineWidth',2);
hold off;

delete(hTxt);
message = sprintf('The estimated radius is %2.1f pixels', radius);
text(15,15,message,'Color','y','FontWeight','bold');
```



See Also

`imfindcircles` | `viscircles`

Related Examples

- “Identifying Round Objects” on page 11-58
- “Detect and Measure Circular Objects in an Image” on page 11-41

Texture Analysis

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 is a small value; in areas of rough texture, the range is larger. Similarly, calculating the standard deviation of pixels in a neighborhood can indicate the degree of variability of pixel values in that region. The table lists these functions.

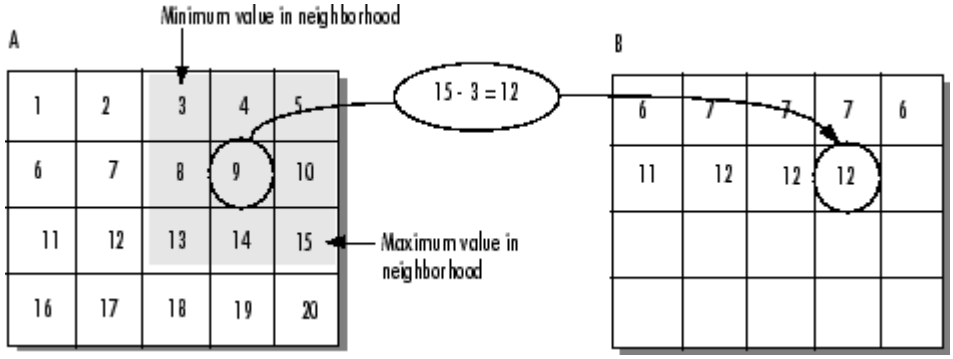
Function	Description
`rangefilt`	Calculates the local range of an image.
`stdfilt`	Calculates the local standard deviation of an image.
`entropyfilt`	Calculates the local entropy of a grayscale image. Entropy is a statistical measure of randomness.

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. 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

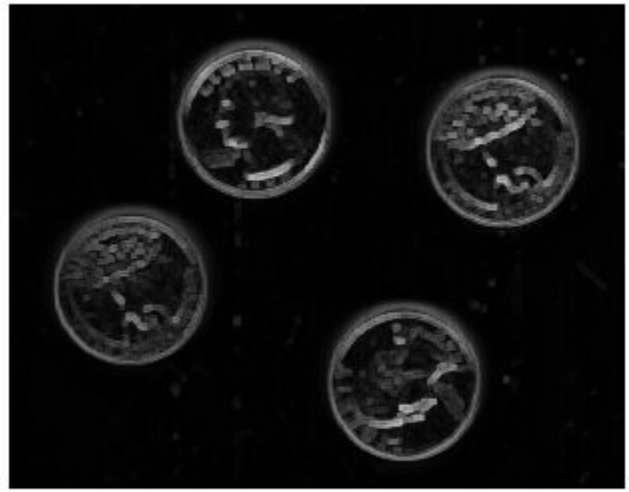
This example shows how to detect regions of texture in an image using the texture filter functions

Read an image into the workspace and display it. 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.

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

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)
```



Texture Analysis Using the 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 11-77 cannot provide information about shape, that is, the spatial relationships of pixels in an image.)

After you create the GLCMs, using `graycomatrix`, you can derive several statistics from them using `graycoprops`. 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.

See Also

Related Examples

- “Derive Statistics from GLCM and Plot Correlation” on page 11-85

More About

- “Create a Gray-Level Co-Occurrence Matrix” on page 11-82

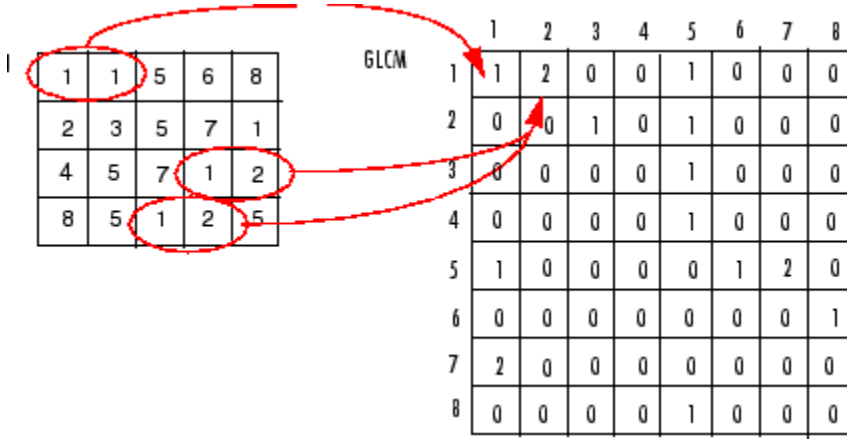
Create a Gray-Level Co-Occurrence Matrix

To create a GLCM, use the `graycomatrix` function. The 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 11-85 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

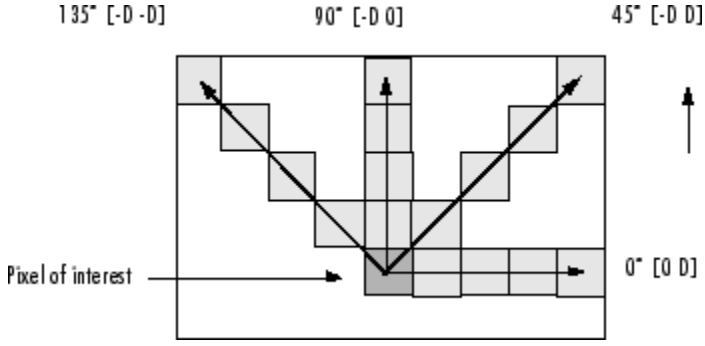
By default, the `graycomatrix` function creates a single GLCM, with the spatial relationship, or *offset*, defined as two horizontally adjacent pixels. However, a single GLCM might not be enough to describe the textural features of the input image. For example, a single horizontal offset might not be 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 four 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.

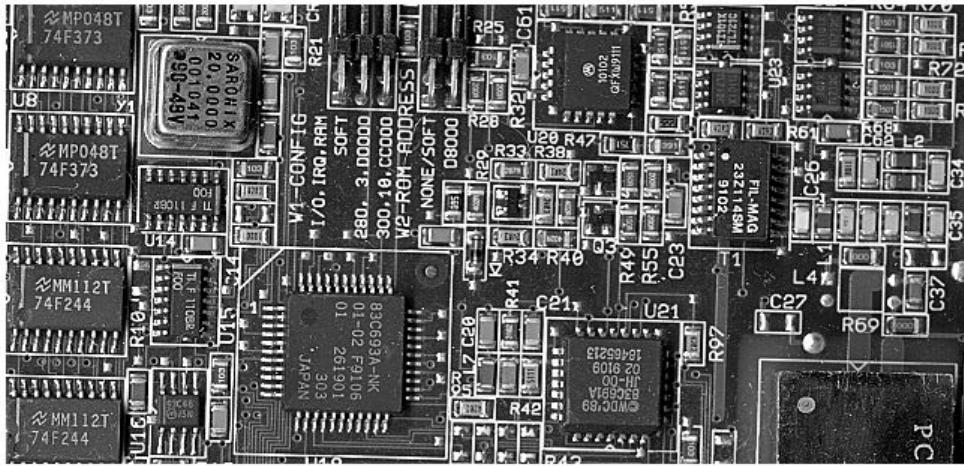


Derive Statistics from GLCM and Plot Correlation

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 an image into the workspace and display it. The example converts the truecolor image to a grayscale image and then, for this example, rotates it 90 degrees.

```
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.

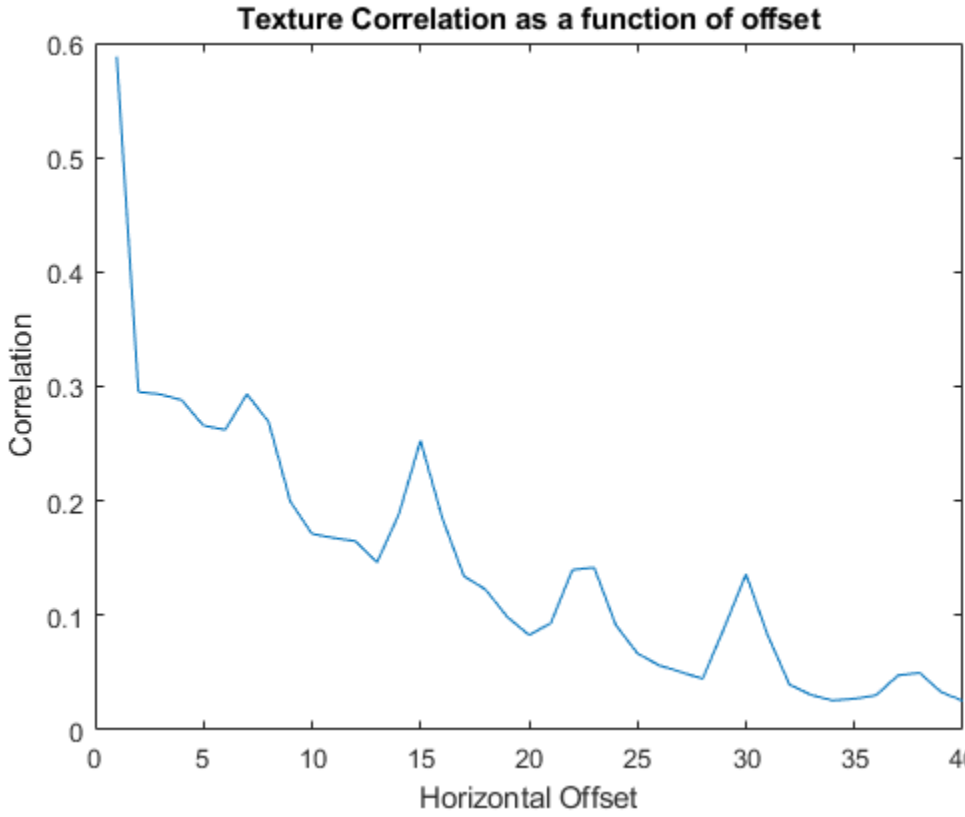
```
glcms = graycomatrix(circuitBoard,'Offset',offsets0);
```

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.

Adjust Image Intensity Values to Specified Range

This example shows how to increase the contrast in a low-contrast grayscale image by remapping the data values to fill the entire available intensity range [0, 255].

Read image into the workspace.

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

Adjust the contrast of the image using `imadjust`.

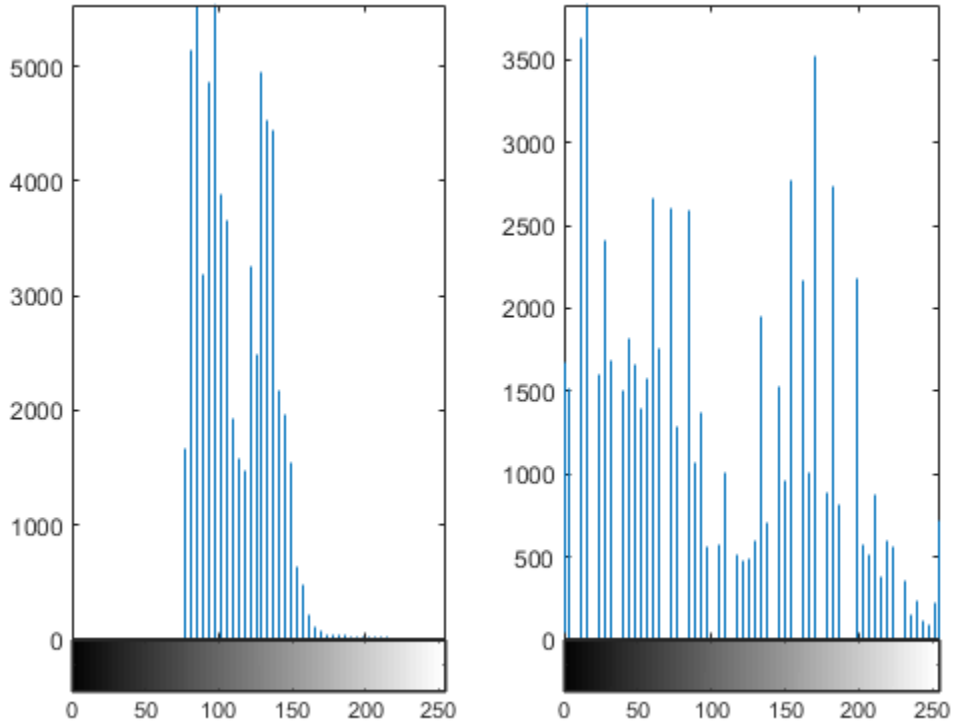
```
J = imadjust(I);
```

Display the original image and the adjusted image, side-by-side. Note the increased contrast in the adjusted image.

```
imshowpair(I,J,'montage')
```

Plot the histogram of the adjust image. Note that the histogram of the adjusted image uses values across the whole range.

```
figure
subplot(1,2,1)
imhist(I,64)
subplot(1,2,2)
imhist(J,64)
```



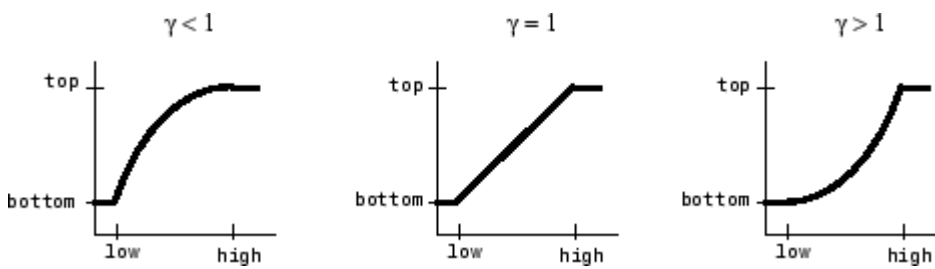
Gamma Correction

`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 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

Specify Gamma when Adjusting Contrast

This example shows how to specify gamma when adjusting contrast with the imadjust function. By default, imadjust uses a gamma value of 1, which means that it uses a linear mapping between intensity values in the original image and the output image. A gamma value less than 1 weights the mapping toward higher (brighter) output values. A gamma value of more than 1 weights output values toward lower (darker) output values.

Read an image into the workspace. This example reads an indexed image and then converts it into a grayscale image.

```
[X,map] = imread('forest.tif');
I = ind2gray(X,map);
```

Adjust the contrast, specifying a gamma value of less than 1 (0.5). Notice that in the call to `imadjust`, the example specifies the data ranges of the input and output images 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.

```
J = imadjust(I,[],[],0.5);
```

Display the original image with the contrast-adjusted image.

```
imshowpair(I,J,'montage')
```

Contrast Enhancement Techniques

This example shows several image enhancement approaches. Three functions are particularly suitable for contrast enhancement: `imadjust`, `histeq`, and `adapthisteq`. This example compares their use for enhancing grayscale and truecolor images.

Enhance Grayscale Images

Using the default settings, compare the effectiveness of the following three techniques:

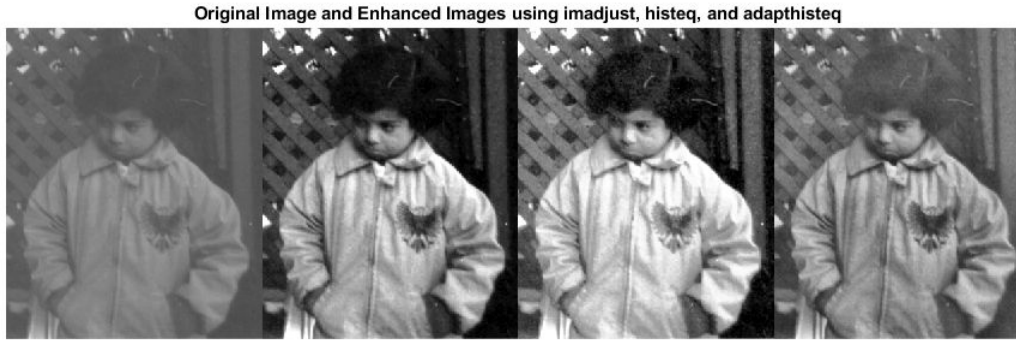
- **`imadjust`** increases the contrast of the image by mapping the values of the input intensity image to new values such that, by default, 1% of the data is saturated at low and high intensities of the input data.
- **`histeq`** performs histogram equalization. It enhances the contrast of images by transforming the values in an intensity image so that the histogram of the output image approximately matches a specified histogram (uniform distribution by default).
- **`adapthisteq`** performs contrast-limited adaptive histogram equalization. Unlike `histeq`, it operates on small data regions (tiles) rather than the entire image. Each tile's contrast is enhanced so that the histogram of each output region approximately matches the specified histogram (uniform distribution by default). The contrast enhancement can be limited in order to avoid amplifying the noise which might be present in the image.

Read a grayscale image into the workspace. Enhance the image using the three contrast adjustment techniques.

```
pout = imread('pout.tif');
pout_imadjust = imadjust(pout);
pout_histeq = histeq(pout);
pout_adapthisteq = adapthisteq(pout);
```

Display the original image and the three contrast adjusted images as a montage.

```
montage({pout,pout_imadjust,pout_histeq,pout_adapthisteq},'Size',[1 4])
title("Original Image and Enhanced Images using imadjust, histeq, and adapthisteq")
```

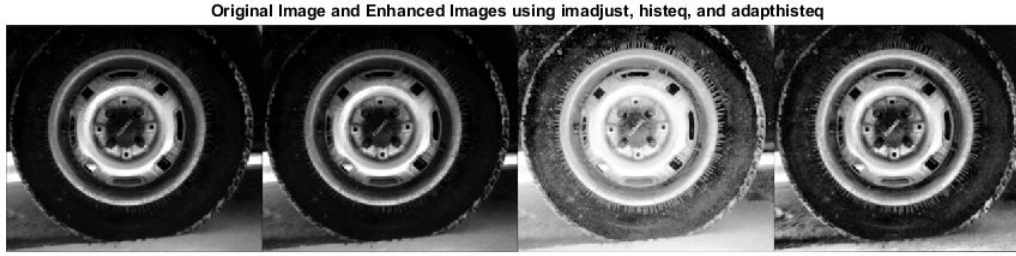


Read a second grayscale image into the workspace and enhance the image using the three contrast adjustment techniques.

```
tire = imread('tire.tif');
tire_imadjust = imadjust(tire);
tire_histeq = histeq(tire);
tire_adapthisteq = adapthisteq(tire);
```

Display the original image and the three contrast adjusted images as a montage.

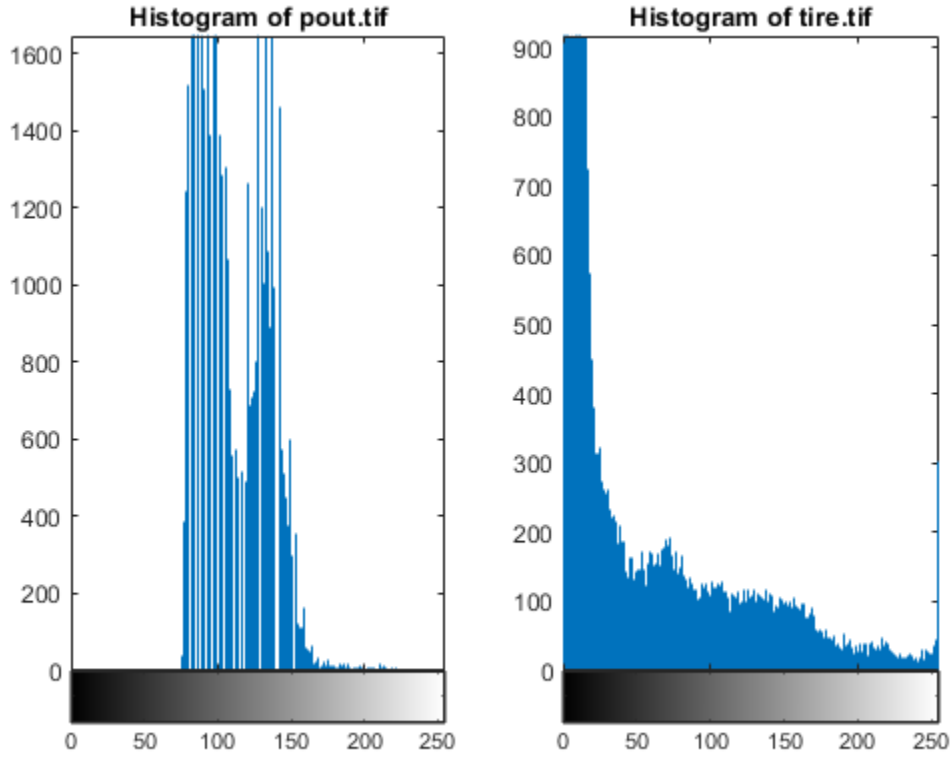
```
montage({tire,tire_imadjust,tire_histeq,tire_adapthisteq},'Size',[1 4])
title("Original Image and Enhanced Images using imadjust, histeq, and adapthisteq")
```



Notice that `imadjust` had little effect on the image of the tire, but it caused a drastic change in the case of pout. Plotting the histograms of `pout.tif` and `tire.tif` reveals that most of the pixels in the first image are concentrated in the center of the histogram,

while in the case of `tire.tif`, the values are already spread out between the minimum of 0 and maximum of 255 thus preventing `imadjust` from being effective in adjusting the contrast of the image.

```
figure
subplot(1,2,1)
imhist(pout)
title('Histogram of pout.tif')
subplot(1,2,2)
imhist(tire)
title('Histogram of tire.tif');
```



Histogram equalization, on the other hand, substantially changes both images. Many of the previously hidden features are exposed, especially the debris particles on the tire.

Unfortunately, at the same time, the enhancement over-saturates several areas of both images. Notice how the center of the tire, part of the child's face, and the jacket became washed out.

Concentrating on the image of the tire, it would be preferable for the center of the wheel to stay at about the same brightness while enhancing the contrast in other areas of the image. In order for that to happen, a different transformation would have to be applied to different portions of the image. The Contrast-Limited Adaptive Histogram Equalization technique, implemented in `adapthisteq`, can accomplish this. The algorithm analyzes portions of the image and computes the appropriate transformations. A limit on the level of contrast enhancement can also be set, thus preventing the over-saturation caused by the basic histogram equalization method of `histeq`. This is the most sophisticated technique in this example.

Enhance Color Images

Contrast enhancement of color images is typically done by converting the image to a color space that has image luminosity as one of its components, such as the L*a*b* color space. Contrast adjustment is performed on the luminosity layer 'L*' only, and then the image is converted back to the RGB color space. Manipulating luminosity affects the intensity of the pixels, while preserving the original colors.

Read an image into the workspace. The `'shadow.tif'` image is an indexed image, so convert the image to a truecolor (RGB) image. Then, convert the image from the RGB color space to the L*a*b* color space.

```
[X,map] = imread('shadow.tif');
shadow = ind2rgb(X,map);
shadow_lab = rgb2lab(shadow);
```

The values of luminosity span a range from 0 to 100. Scale the values to the range [0 1], which is the expected range of images with data type `double`.

```
max_luminosity = 100;
L = shadow_lab(:,:,1)/max_luminosity;
```

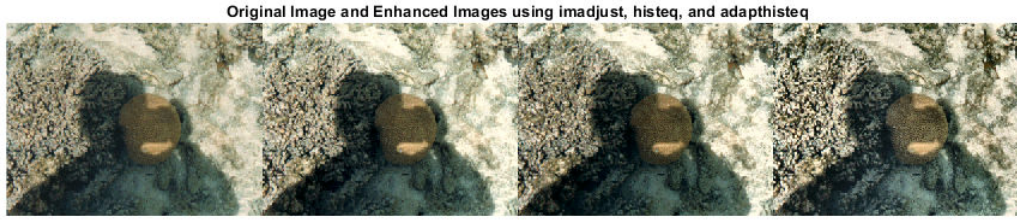
Perform the three types of contrast adjustment on the luminosity channel, and keep the a* and b* channels unchanged. Convert the images back to the RGB color space.

```
shadow_imadjust = shadow_lab;
shadow_imadjust(:,:,1) = imadjust(L)*max_luminosity;
shadow_imadjust = lab2rgb(shadow_imadjust);
```

```
shadow_histeq = shadow_lab;
shadow_histeq(:,:,1) = histeq(L)*max_luminosity;
shadow_histeq = lab2rgb(shadow_histeq);

shadow_adapthisteq = shadow_lab;
shadow_adapthisteq(:,:,1) = adapthisteq(L)*max_luminosity;
shadow_adapthisteq = lab2rgb(shadow_adapthisteq);
```

Display the original image and the three contrast adjusted images as a montage.

```
figure
montage({shadow,shadow_imadjust,shadow_histeq,shadow_adapthisteq},'Size',[1 4])
title("Original Image and Enhanced Images using imadjust, histeq, and adapthisteq")
```



See Also

adapthisteq | histeq | imadjust

More About

- “Adaptive Histogram Equalization” on page 11-103
- “Histogram Equalization” on page 11-99

Specify Contrast Adjustment Limits

You can optionally specify the range of the input values and the output values using `imadjust`. You specify these ranges in two vectors that you pass to `imadjust` as arguments. The first vector specifies the low- and high-intensity values that you want to map. The second vector specifies the scale over which you want to map them.

Note 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 11-98.

Specify Contast Adjustment Limits as Range

This example shows how to specify contast adjustment limits as a range using the `imadjust` function. This example decreases the contrast of an image by narrowing the range of the data.

Read an image into the workspace.

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

Adjust the contrast of the image, specifying the range of values used in the output image. 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.

```
J = imadjust(I,[0 0.2],[0.5 1]);
```

Display the original image and the contrast-adjusted image.

```
imshowpair(I,J,'montage')
```

Set Image Intensity Adjustment Limits Automatically

For a more convenient way to specify 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.

Histogram Equalization

The process of adjusting intensity values can be done automatically using *histogram equalization*. Histogram equalization involves transforming the intensity values so that the histogram of the output image approximately matches a specified histogram. By default, the histogram equalization function, `histeq`, tries to match a flat histogram with 64 bins, but you can specify a different histogram instead.

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 Intensity Values Using Histogram Equalization

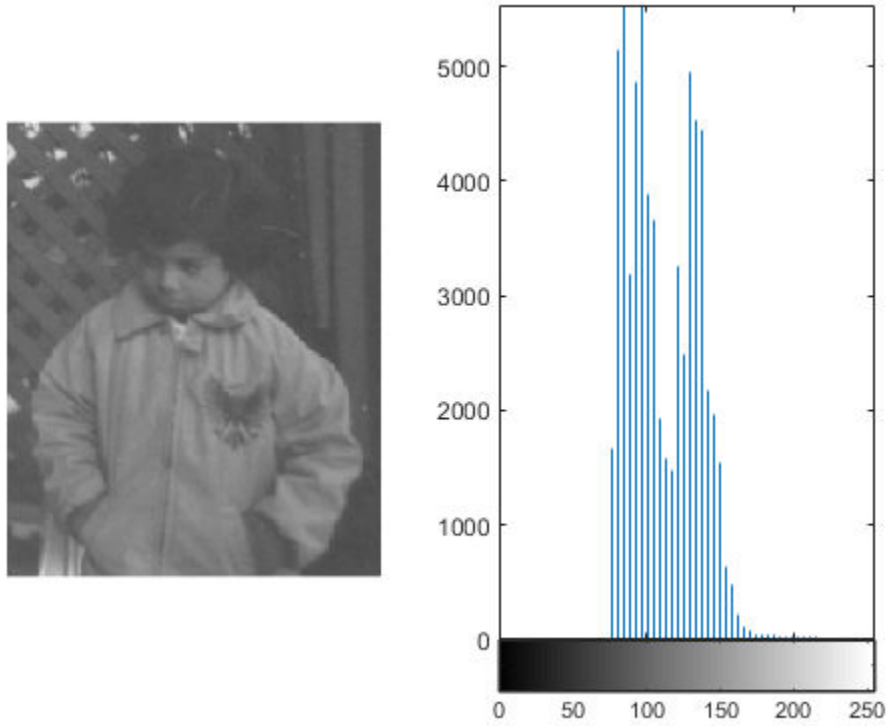
This example shows how to use histogram equalization to adjust the contrast of a grayscale image. The original image has low contrast, with most pixel values in the middle of the intensity range. `histeq` produces an output image with pixel values evenly distributed throughout the range.

Read an image into the workspace.

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

Display the image and its histogram.

```
figure
subplot(1,2,1)
imshow(I)
subplot(1,2,2)
imhist(I,64)
```

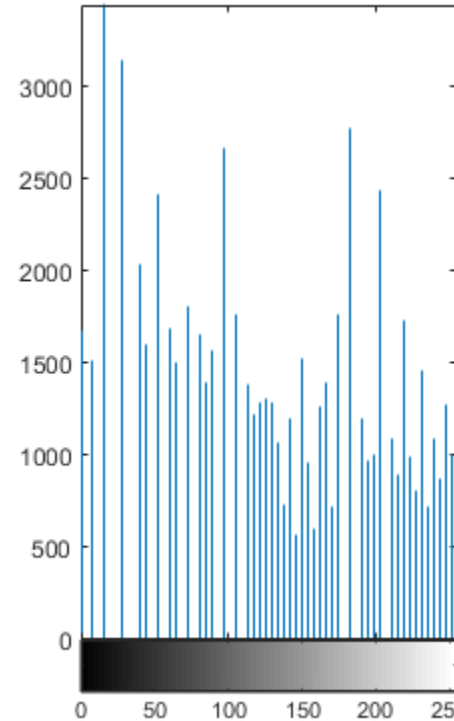


Adjust the contrast using histogram equalization. In this example, the histogram equalization function, `histeq`, tries to match a flat histogram with 64 bins, which is the default behavior. You can specify a different histogram instead.

```
J = histeq(I);
```

Display the contrast-adjusted image and its new histogram.

```
figure
subplot(1,2,1)
imshow(J)
subplot(1,2,2)
imhist(J,64)
```



Plot Transformation Curve for Histogram Equalization

This example shows how to plot the transformation curve for histogram equalization. `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.

Read image into the workspace.

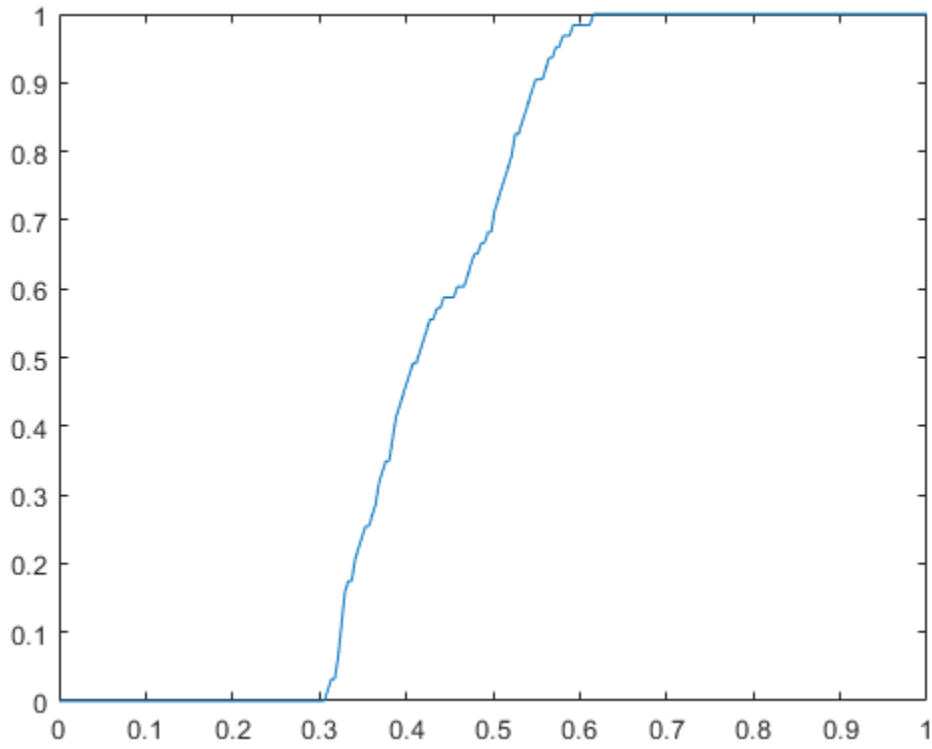
```
I = imread('pout.tif');
```

Adjust the contrast using histogram equalization, using the `histeq` function. Specify the gray scale transformation return value, `T`, which is a vector that maps graylevels in the intensity image `I` to gray levels in `J`.

```
[J,T] = histeq(I);
```

Plot the transformation curve. 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.

```
figure
plot((0:255)/255,T);
```



Adaptive Histogram Equalization

As an alternative to using `histeq`, you can perform contrast-limited adaptive histogram equalization (CLAHE) using the `adapthisteq` function. While `histeq` works on the entire image, `adapthisteq` operates on small regions in the image, called *tiles*. `adapthisteq` enhances the contrast of each tile, 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.

Adjust Contrast using Adaptive Histogram Equalization

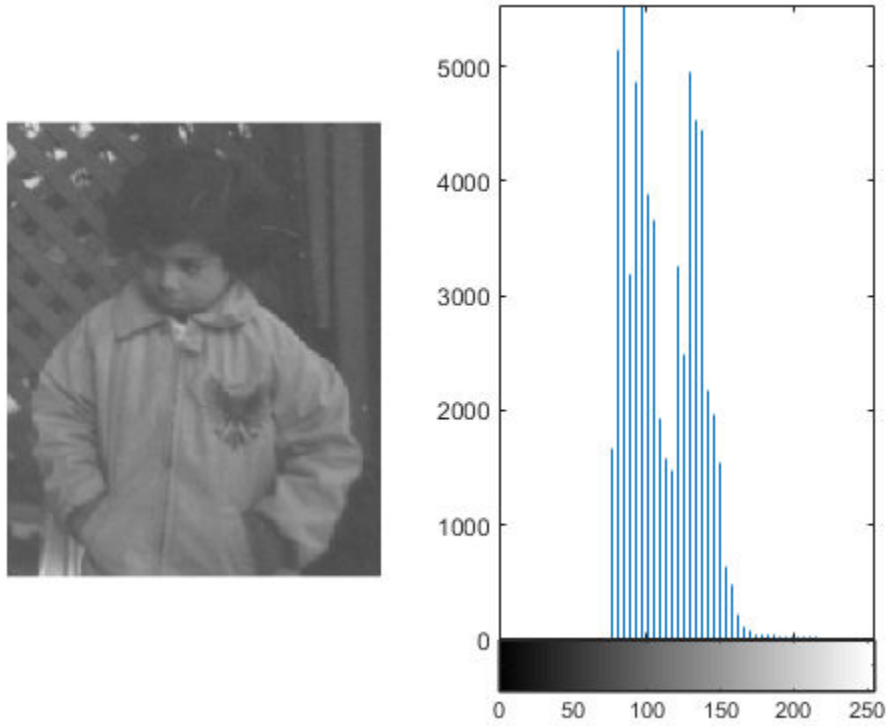
This example shows how to adjust the contrast in an image using CLAHE.

Read image into the workspace.

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

View the original image and its histogram.

```
figure
subplot(1,2,1)
imshow(I)
subplot(1,2,2)
imhist(I,64)
```

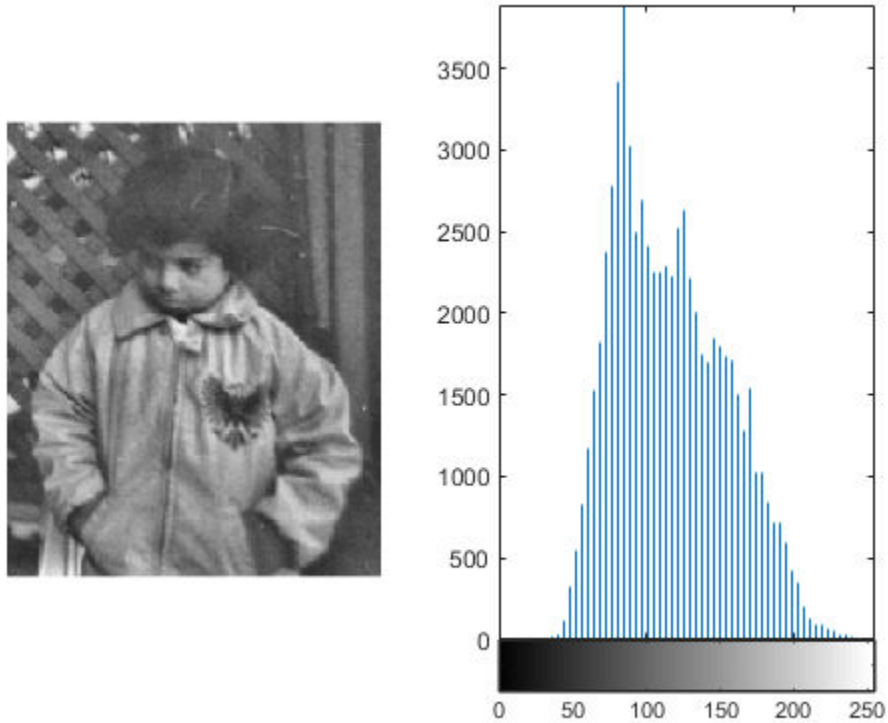


Adjust the contrast of the image using adaptive histogram equalization.

```
J = adapthisteq(I);
```

Display the contrast-adjusted image with its histogram.

```
figure
subplot(1,2,1)
imshow(J)
subplot(1,2,2)
imhist(J,64)
```



See Also

More About

- “Histogram Equalization” on page 11-99

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 “Linear Contrast Stretching” on page 11-112 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 band-wise statistics, you can use the entire original image or, with the `subset` option, any selected subset of it.

Simple Decorrelation Stretching

This example shows how to perform decorrelation stretching to three color bands of an image. A color band scatterplot of the images shows how the bands are decorrelated and equalized.

Perform Decorrelation Stretch

Read an image from the library of images available in the `imdata` folder. This examples uses a LANDSAT image of the Little Colorado River. The image has seven bands, but just read in the three visible colors.

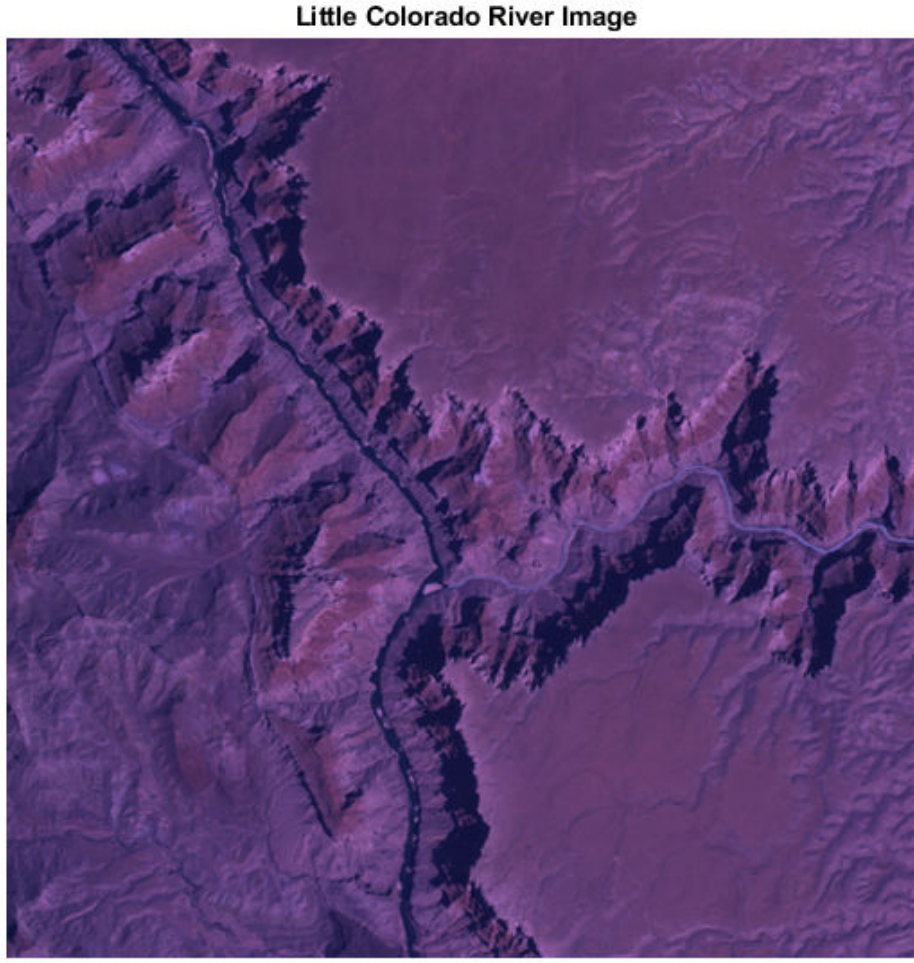
```
A = multibandread('littlecoriver.lan', [512, 512, 7], ...
  'uint8=>uint8', 128, 'bil', 'ieee-le', ...
  {'Band','Direct',[3 2 1]});
```

Perform the decorrelation stretch.

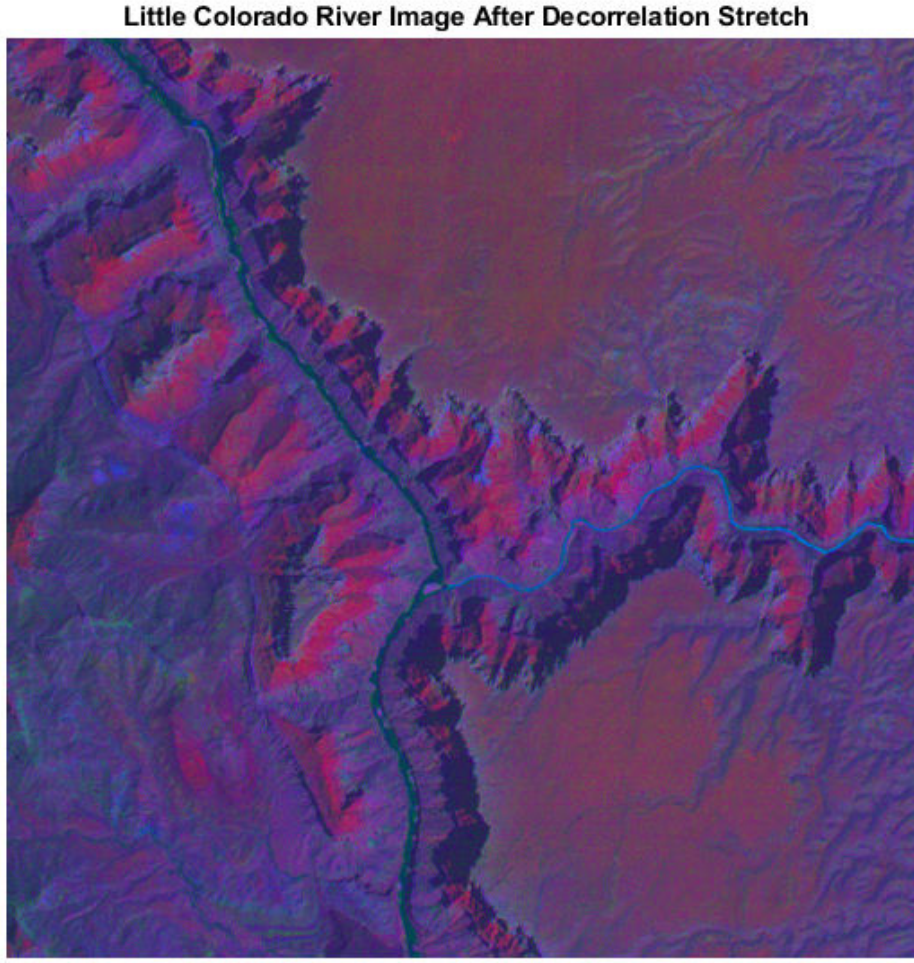
```
B = decorrstretch(A);
```

Display the original image and the processed image. Compare the two images. The original has a strong violet (red-bluish) tint, while the transformed image has a somewhat expanded color range.

```
figure
imshow(A)
title('Little Colorado River Image')
```




```
figure
imshow(B)
title('Little Colorado River Image After Decorrelation Stretch')
```



Create a Color Band Scatterplot

First separate the three color channels of the original image.

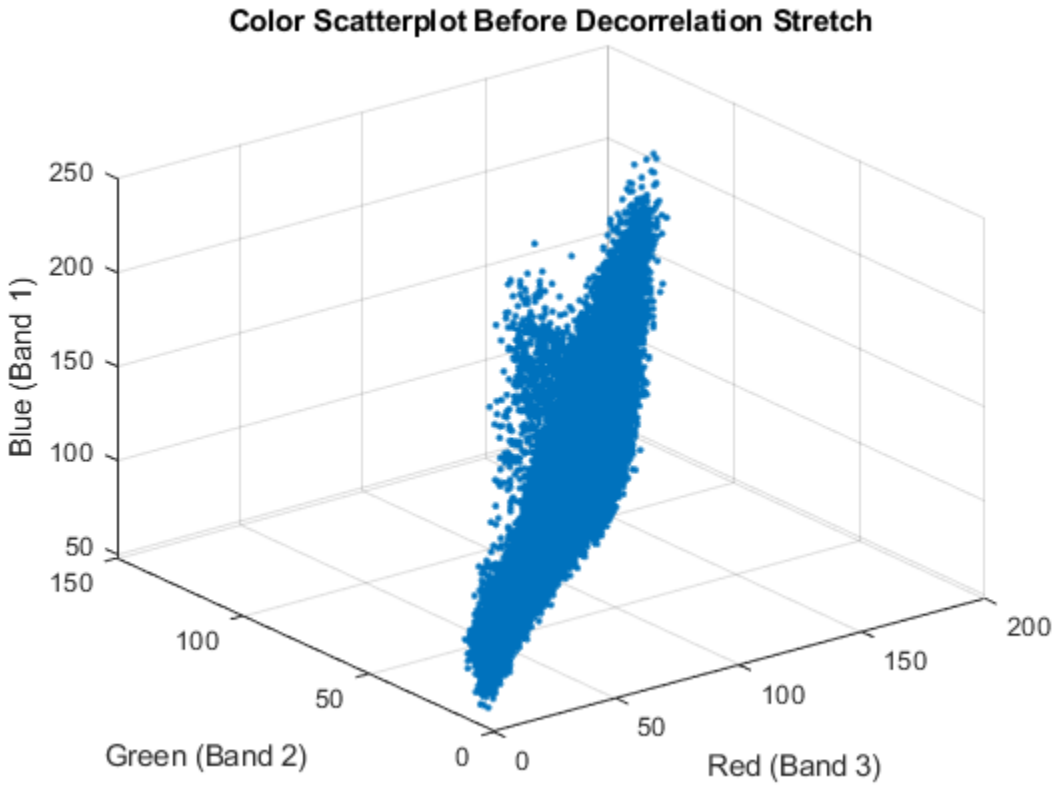
```
rA = A(:,:,1);
gA = A(:,:,2);
bA = A(:,:,3);
```

Separate the three color channels of the image after decorrelation stretching.

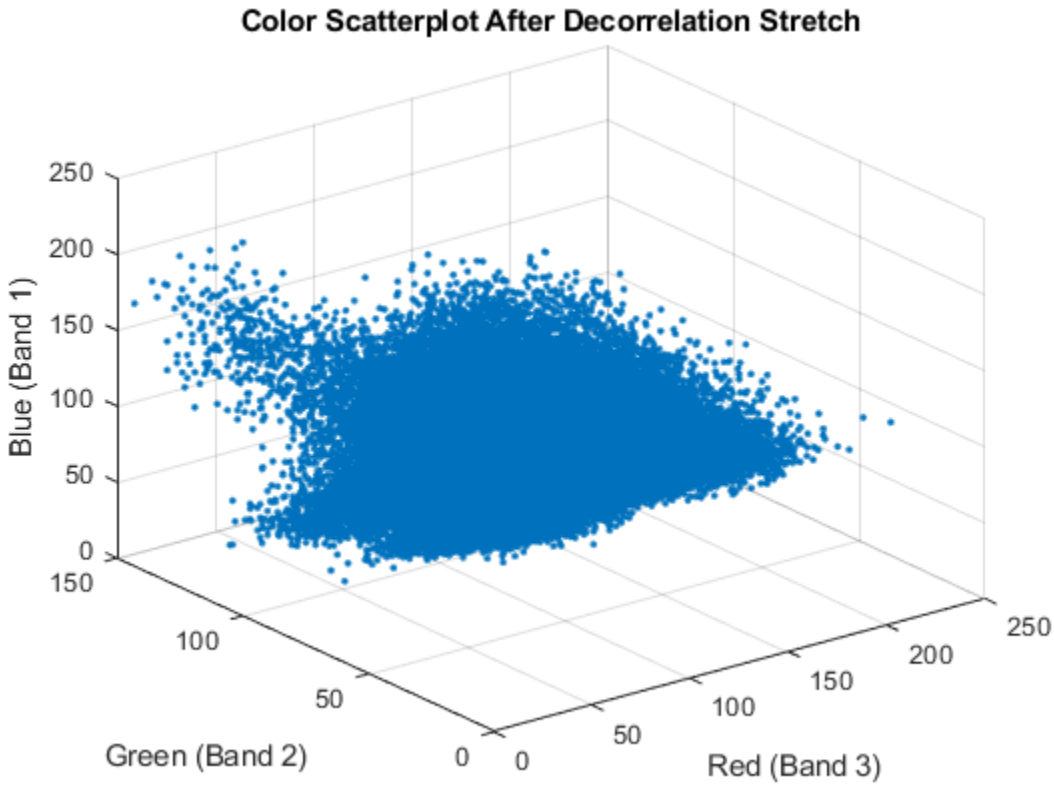
```
rB = B(:,:,1);
gB = B(:,:,2);
bB = B(:,:,3);
```

Display the color scatterplot of the original image. Then display the color scatterplot of the image after decorrelation stretching.

```
figure
plot3(rA(:),gA(:),bA(:),'.')
grid on
xlabel('Red (Band 3)')
ylabel('Green (Band 2)')
zlabel('Blue (Band 1)')
title('Color Scatterplot Before Decorrelation Stretch')
```




```
figure
plot3(rB(:),gB(:),bB(:),'.')
grid on
xlabel('Red (Band 3)')
ylabel('Green (Band 2)')
zlabel('Blue (Band 1)')
title('Color Scatterplot After Decorrelation Stretch')
```



Linear Contrast Stretching

Adding linear contrast stretch enhances the resulting image by further expanding the color range. The following example uses the `Tol` option to saturate equal fractions of the image at high and low intensities. Without the `Tol` option, `decorrstretch` applies no linear contrast stretch.

See the `stretchlim` function reference page for more about calculating saturation limits.

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.

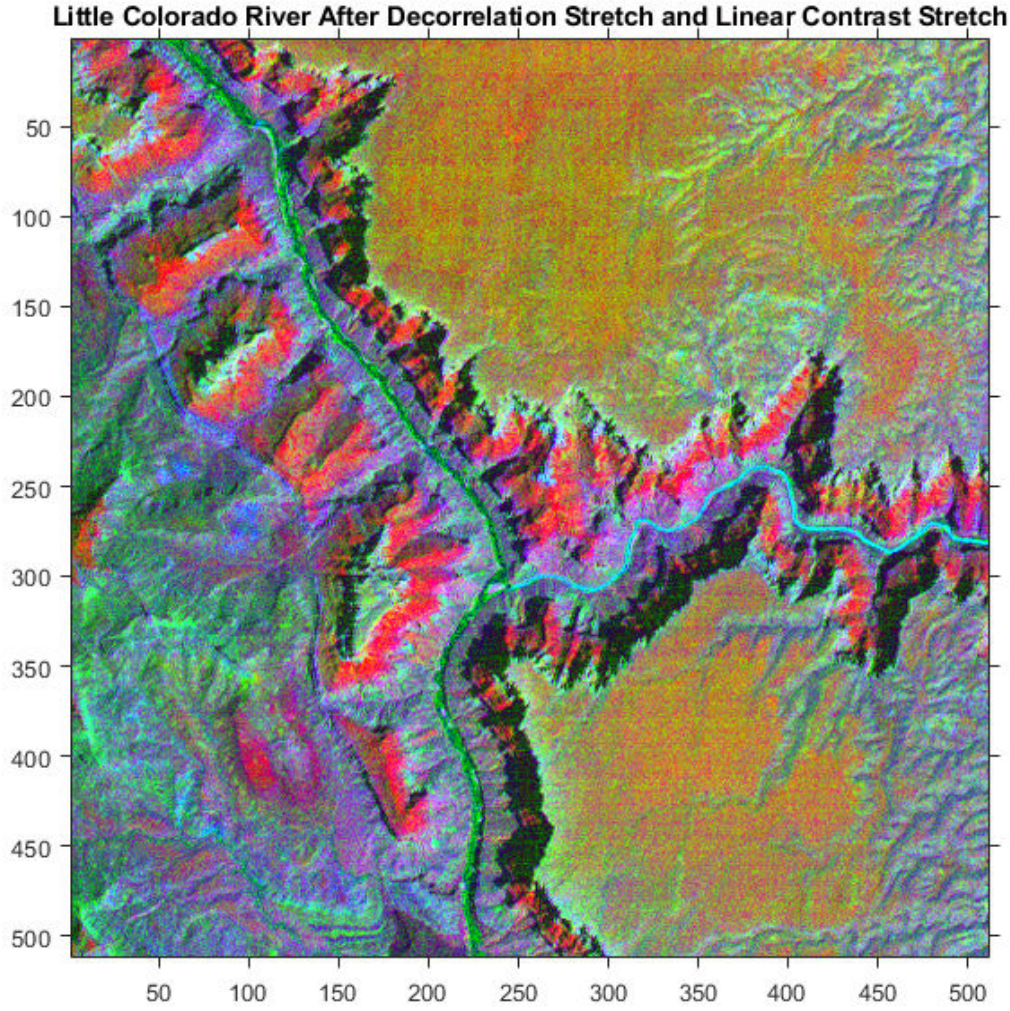
Decorrelation Stretch with Linear Contrast Stretch

Read the three visible color channels of the LANDSAT image of the Little Colorado River.

```
A = multibandread('littlecoriver.lan', [512, 512, 7], ...
  'uint8=>uint8', 128, 'bil', 'ieee-le', ...
  {'Band','Direct',[3 2 1]});
```

Apply decorrelation stretching, specifying the linear contrast stretch. Setting the value `'Tol'` to 0.01 maps the transformed color range within each band to a normalized interval between 0.01 and 0.99, saturating 2%.

```
C = decorrstretch(A,'Tol',0.01);
imshow(C)
title(['Little Colorado River After Decorrelation Stretch and ',...
  'Linear Contrast Stretch'])
```


Little Colorado River After Decorrelation Stretch and Linear Contrast Stretch
50
100
150
200
250
300
350
400
450
500
50
100
150
200
250
300
350
400
450
500

Enhancing Multispectral Color Composite Images

This example shows some basic image composition and enhancement techniques for use with multispectral data. It is often necessary to enhance multispectral radiance or reflectance data to create an image that is suitable for visual interpretation. The example uses Landsat thematic mapper imagery covering part of Paris, France. Seven spectral bands are stored in one file in the Erdas LAN format. Concepts covered include:

- Reading multispectral data from Erdas LAN files
- Constructing color composites from different band combinations
- Enhancing imagery with a contrast stretch
- Enhancing imagery with a decorrelation stretch
- Using scatterplots

Step 1: Construct Truecolor Composite from a Multispectral Image

The LAN file, `paris.lan`, contains a 7-band 512-by-512 Landsat image. A 128-byte header is followed by the pixel values, which are band interleaved by line (BIL) in order of increasing band number. They are stored as unsigned 8-bit integers, in little-endian byte order.

Read bands 3, 2, and 1 from the LAN file using the MATLAB® function `multibandread`. These bands cover the visible part of the spectrum. When they are mapped to the red, green, and blue planes, respectively, of an RGB image, the result is a standard truecolor composite. The final input argument to `multibandread` specifies which bands to read, and in which order, so that you can construct an RGB composite in a single step.

```
truecolor = multibandread('paris.lan', [512, 512, 7], 'uint8=>uint8', ...
                          128,  'bil', 'ieee-le', {'Band','Direct',[3 2 1]});
```

The truecolor composite has very little contrast and the colors are unbalanced.

```
f1 = figure;
imshow(truecolor);
figure(f1);
title('Truecolor Composite (Un-enhanced)')
text(size(truecolor,2), size(truecolor,1) + 15,...
  'Image courtesy of Space Imaging, LLC',...
  'FontSize', 7, 'HorizontalAlignment', 'right')
```

Truecolor Composite (Un-enhanced)

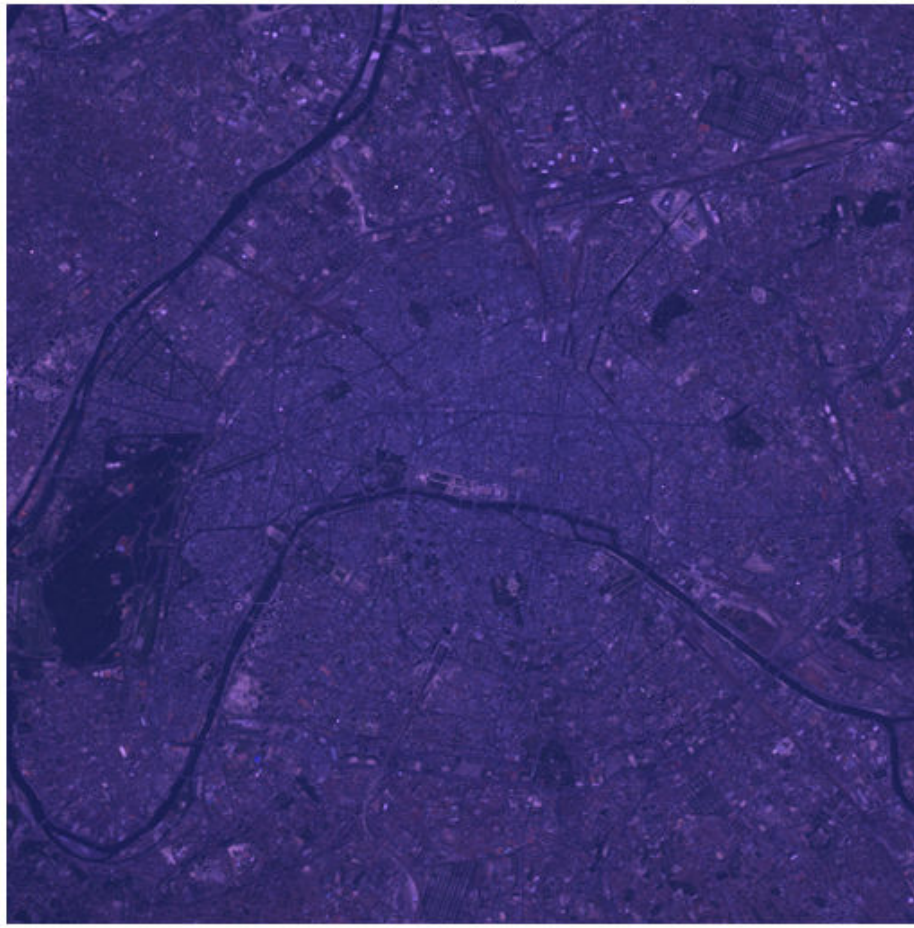
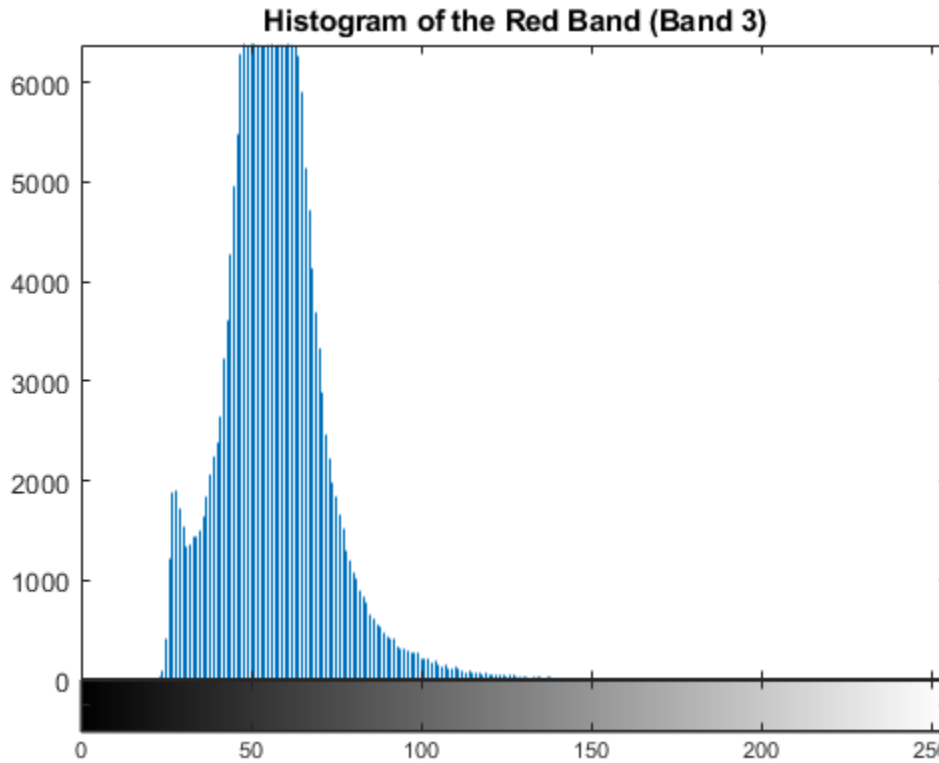


Image courtesy of Space Imaging, LLC

Step 2: Use Histograms to Explore Un-Enhanced Truecolor Composite

By viewing a histogram of the red band, for example, you can see that the data is concentrated within a small part of the available dynamic range. This is one reason why the truecolor composite appears dull.

```
f2 = figure;
imhist(truecolor(:,:,1))
figure(f2);
title('Histogram of the Red Band (Band 3)')
```

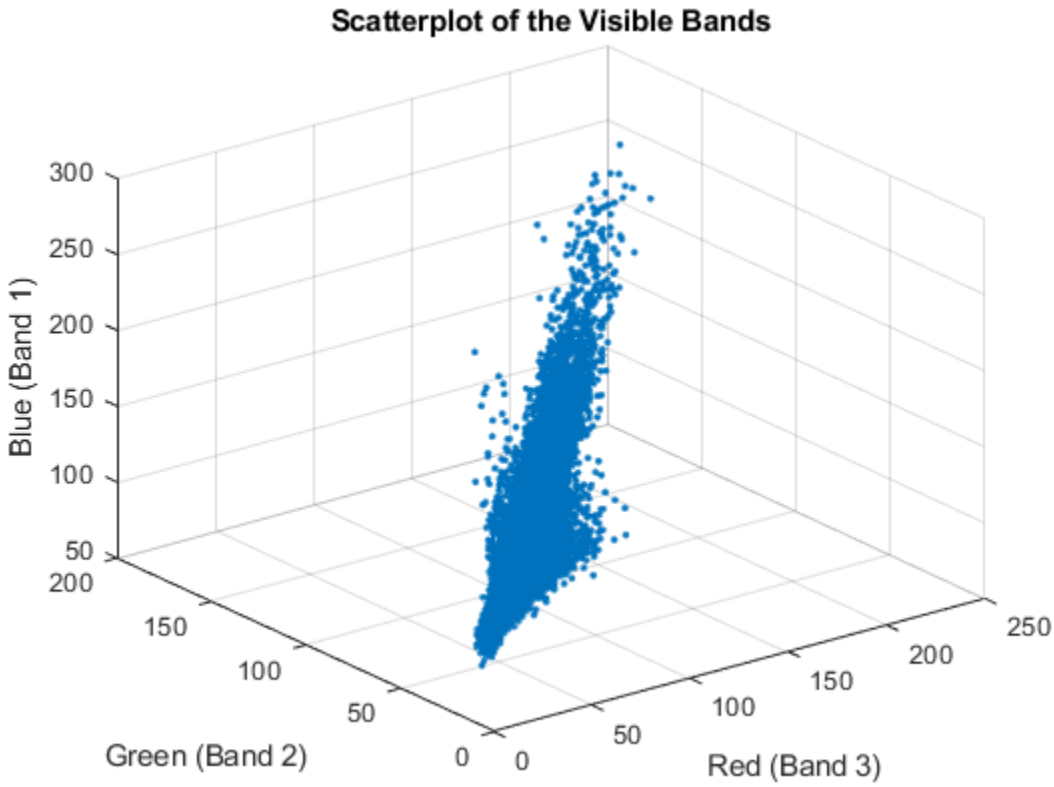


Step 3: Use Correlation to Explore Un-Enhanced Truecolor Composite

Another reason for the dull appearance of the composite is that the visible bands are highly correlated with each other. Two- and three-band scatterplots are an excellent way to gauge the degree of correlation among spectral bands. You can make them easily just by using `plot`.

```
r = truecolor(:,:,1);
g = truecolor(:,:,2);
```

```
b = truecolor(:,:,3);
f3 = figure;
plot3(r(:),g(:),b(:),'.')
grid('on')
xlabel('Red (Band 3)')
ylabel('Green (Band 2)')
zlabel('Blue (Band 1)')
figure(f3);
title('Scatterplot of the Visible Bands')
```



The pronounced linear trend of the red-green-blue scatterplot indicates that the visible bands are highly correlated. This helps explain the monochromatic look of the un-enhanced truecolor composite.

Step 4: Enhance Truecolor Composite with a Contrast Stretch

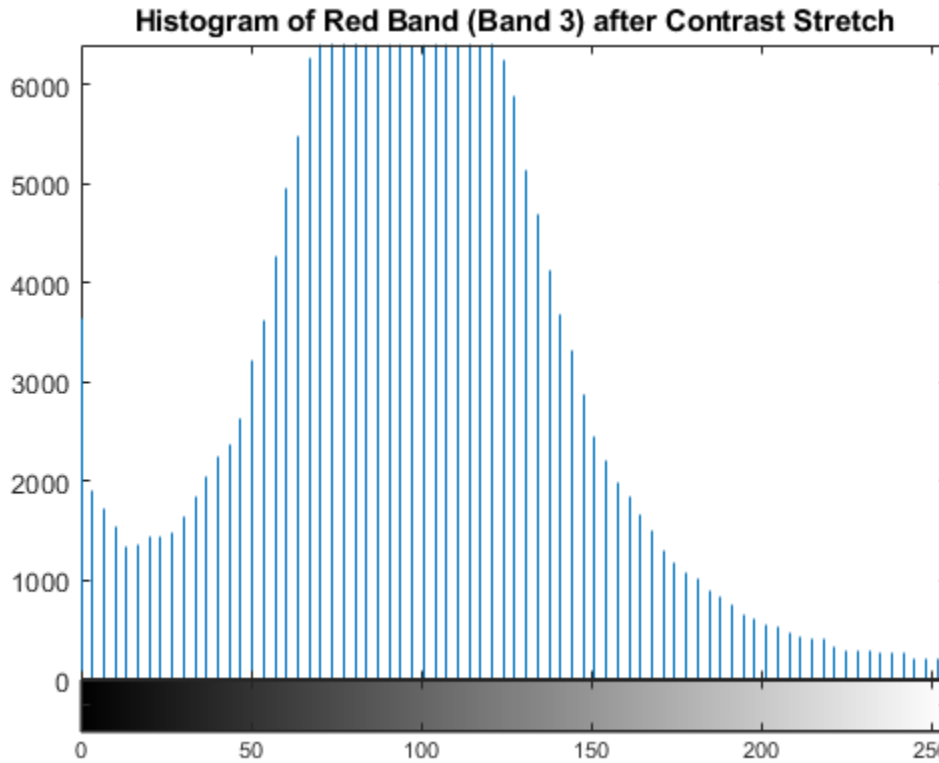
When you use `imadjust` to apply a linear contrast stretch to the truecolor composite image, the surface features are easier to recognize.

```
stretched_truecolor = imadjust(truecolor,stretchlim(truecolor));
f4 = figure;
imshow(stretched_truecolor)
figure(f4);
title('Truecolor Composite after Contrast Stretch')
```

Step 5: Check Histogram Following the Contrast Stretch

A histogram of the red band after applying a contrast stretch shows that the data has been spread over much more of the available dynamic range.

```
f5 = figure;
imhist(stretched_truecolor(:,:,1))
figure(f5);
title('Histogram of Red Band (Band 3) after Contrast Stretch')
```

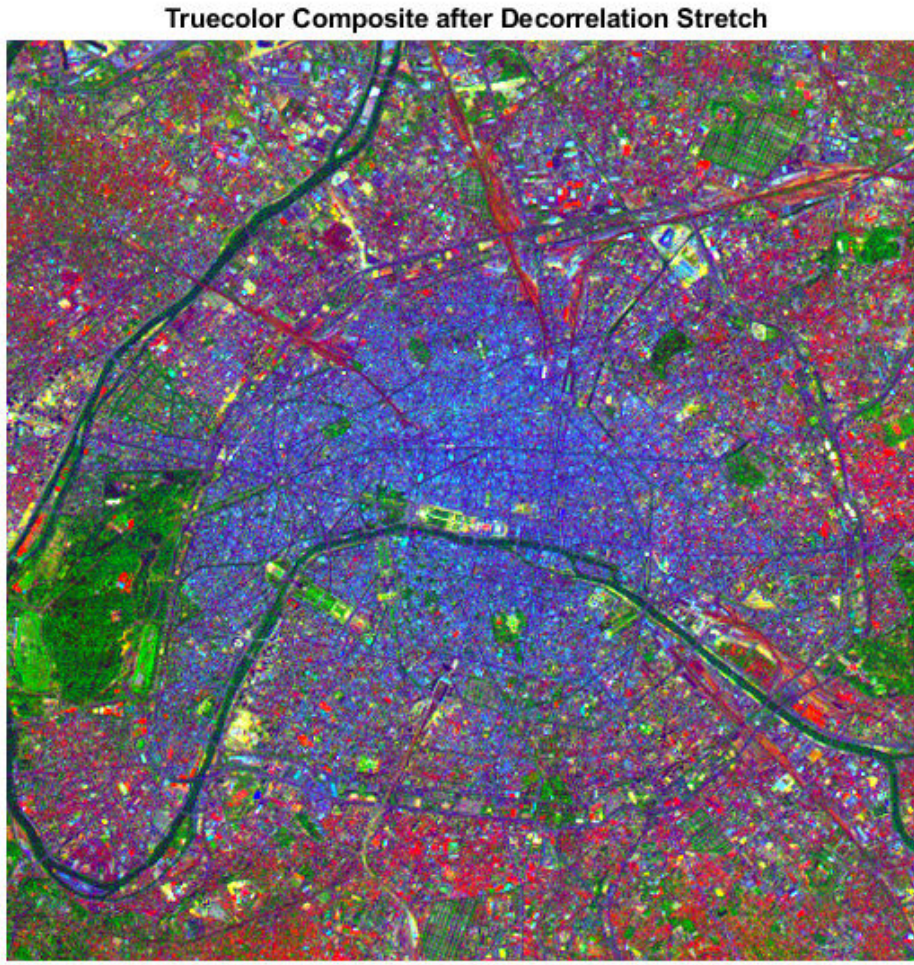


Step 6: Enhance Truecolor Composite with a Decorrelation Stretch

Another way to enhance the truecolor composite is to use a decorrelation stretch, which enhances color separation across highly correlated channels. Use `decorrstretch` to perform the decorrelation stretch (followed by a linear contrast stretch, as specified by the optional parameter-value pair `'Tol'` and `0.1`).

```
decorrstretched_truecolor = decorrstretch(truecolor, 'Tol', 0.01);
f6 = figure;
```

```
imshow(decorrstretched_truecolor)
figure(f6);
title('Truecolor Composite after Decorrelation Stretch')
```

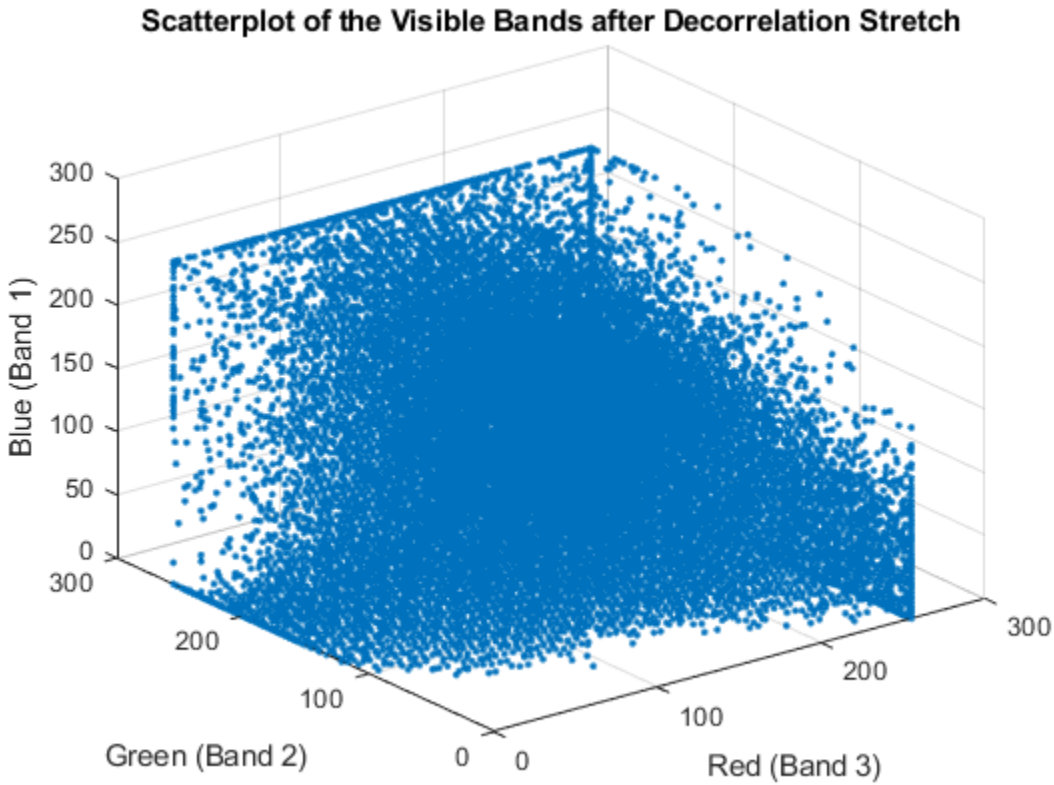


Again, surface features have become much more clearly visible, but in a different way. The spectral differences across the scene have been exaggerated. A noticeable example is the area of green on the left edge, which appears black in the contrast-stretched composite. This green area is the Bois de Boulogne, a large park on the western edge of Paris.

Step 7: Check Correlation Following the Decorrelation Stretch

As expected, a scatterplot following the decorrelation stretch shows a strong decrease in correlation.

```
r = decorrstretched_truecolor(:,:,1);
g = decorrstretched_truecolor(:,:,2);
b = decorrstretched_truecolor(:,:,3);
f7 = figure;
plot3(r(:),g(:),b(:),'.')
grid('on')
xlabel('Red (Band 3)')
ylabel('Green (Band 2)')
zlabel('Blue (Band 1)')
figure(f7);
title('Scatterplot of the Visible Bands after Decorrelation Stretch')
```



Step 8: Construct and Enhance a CIR Composite

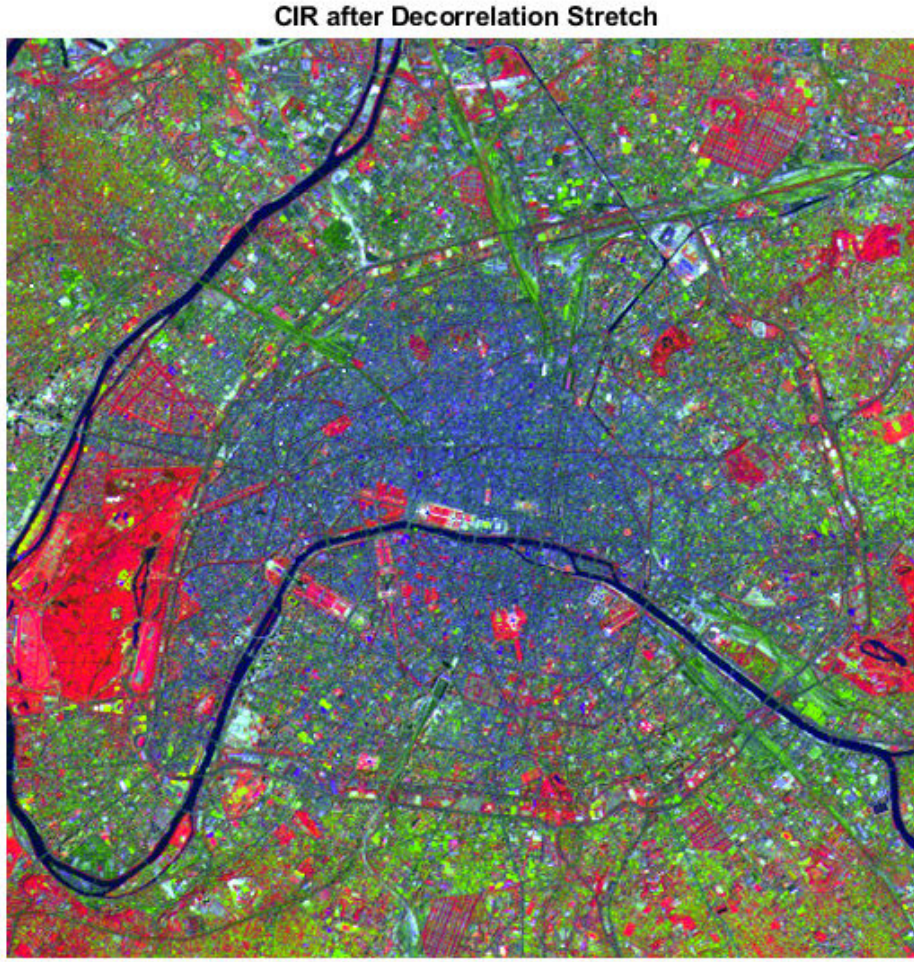
Just as with the visible bands, information from Landsat bands covering non-visible portions of the spectrum can be viewed by constructing and enhancing RGB composite images. The near infrared (NIR) band (Band 4) is important because of the high reflectance of chlorophyll in this part of the spectrum. It is even more useful when combined with visible red and green (Bands 3 and 2, respectively) to form a color infrared (CIR) composite image. Color infrared (CIR) composites are commonly used to identify vegetation or assess its state of growth and/or health.

Construct a CIR composite by reading from the original LAN file and composing an RGB image that maps bands 4, 3, and 2 to red, green, and blue, respectively.

```
CIR = multibandread('paris.lan', [512, 512, 7], 'uint8=>uint8', ...
                    128,  'bil', 'ieee-le', {'Band','Direct',[4 3 2]});
```

Even though the near infrared (NIR) band (Band 4) is less correlated with the visible bands than the visible bands are with each other, a decorrelation stretch makes many features easier to see.

```
stretched_CIR = decorrstretch(CIR, 'Tol', 0.01);
f8 = figure;
imshow(stretched_CIR)
figure(f8);
title('CIR after Decorrelation Stretch')
```



A property of color infrared composites is that they look red in areas with a high vegetation (chlorophyll) density. Notice that the Bois de Boulogne park is red in the CIR composite, which is consistent with its green appearance in the decorrelation-stretched truecolor composite.

See also `decorrstretch`, `imhist`, `imadjust`, `multibandread`, `stretchlim`.

See Also

`decorrstretch` | `imadjust` | `imhist` | `multibandread` | `plot3` | `stretchlim`

Related Examples

- “Enhance Color Separation Using Decorrelation Stretching” on page 11-106

Low-Light Image Enhancement

Images captured in outdoor scenes can be highly degraded due to poor lighting conditions. These images can have low dynamic ranges with high noise levels that affect the overall performance of computer vision algorithms. To make computer vision algorithms robust in low-light conditions, use low-light image enhancement to improve the visibility of an image. The histogram of pixel-wise inversion of low-light images or HDR images is very similar to the histogram of hazy images. Thus, you can use haze removal techniques to enhance low-light images.

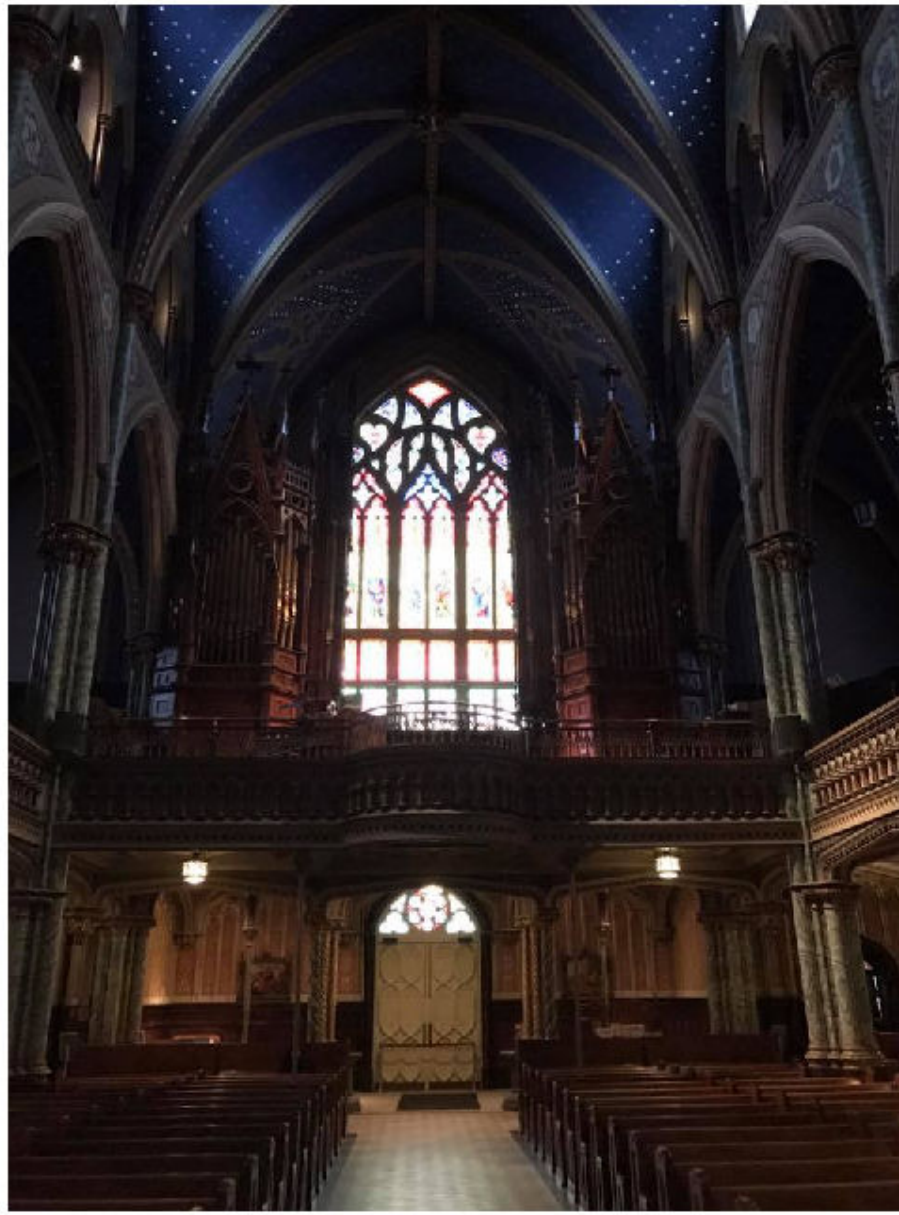
Using haze removal techniques to enhance low-light images comprises three steps:

- Step 1: Invert the low-light image.
- Step 2: Apply the haze removal algorithm to the inverted low-light image.
- Step 3: Invert the enhanced image.

Enhance Low Light Image using Dehazing Algorithm

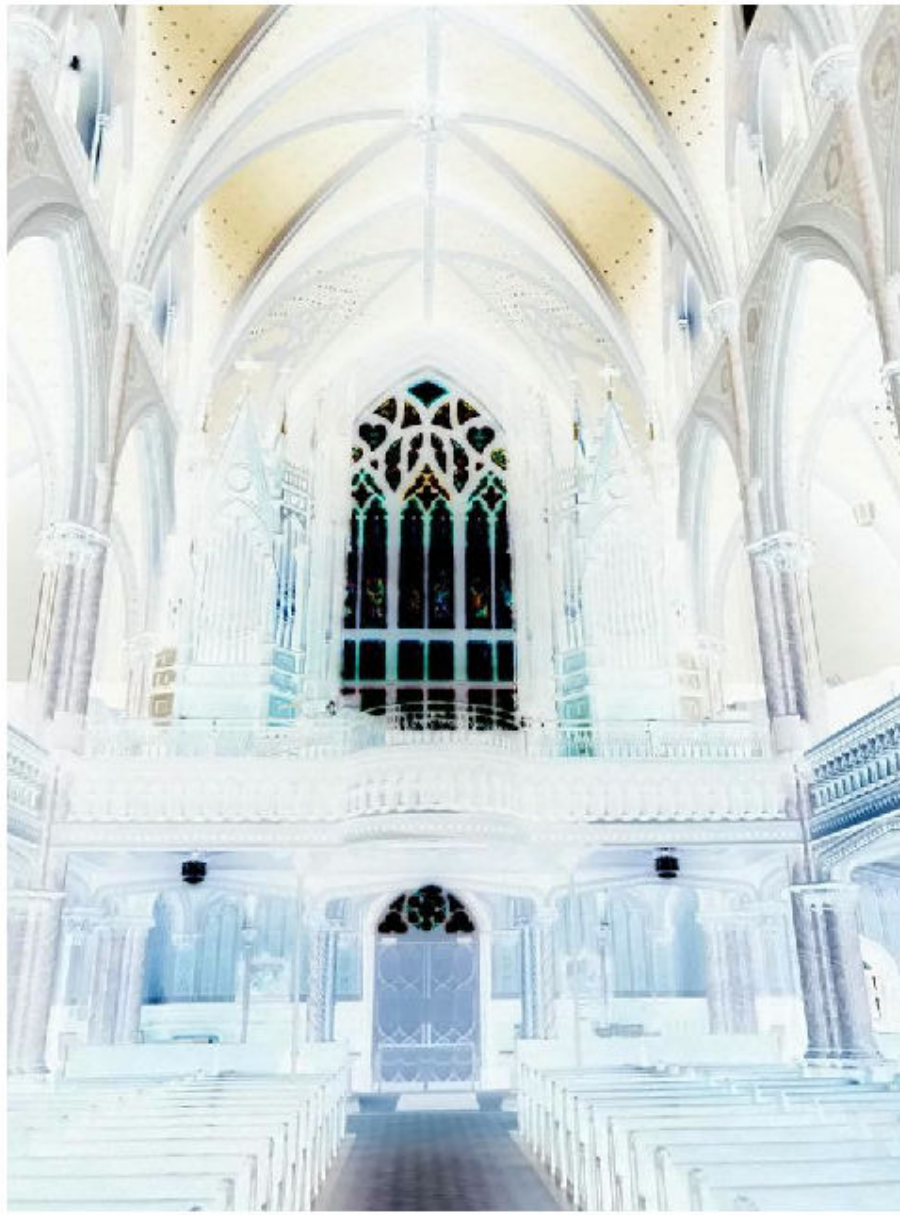
Import an RGB image captured in low light.

```
A = imread('lowlight_11.jpg');
figure, imshow(A);
```



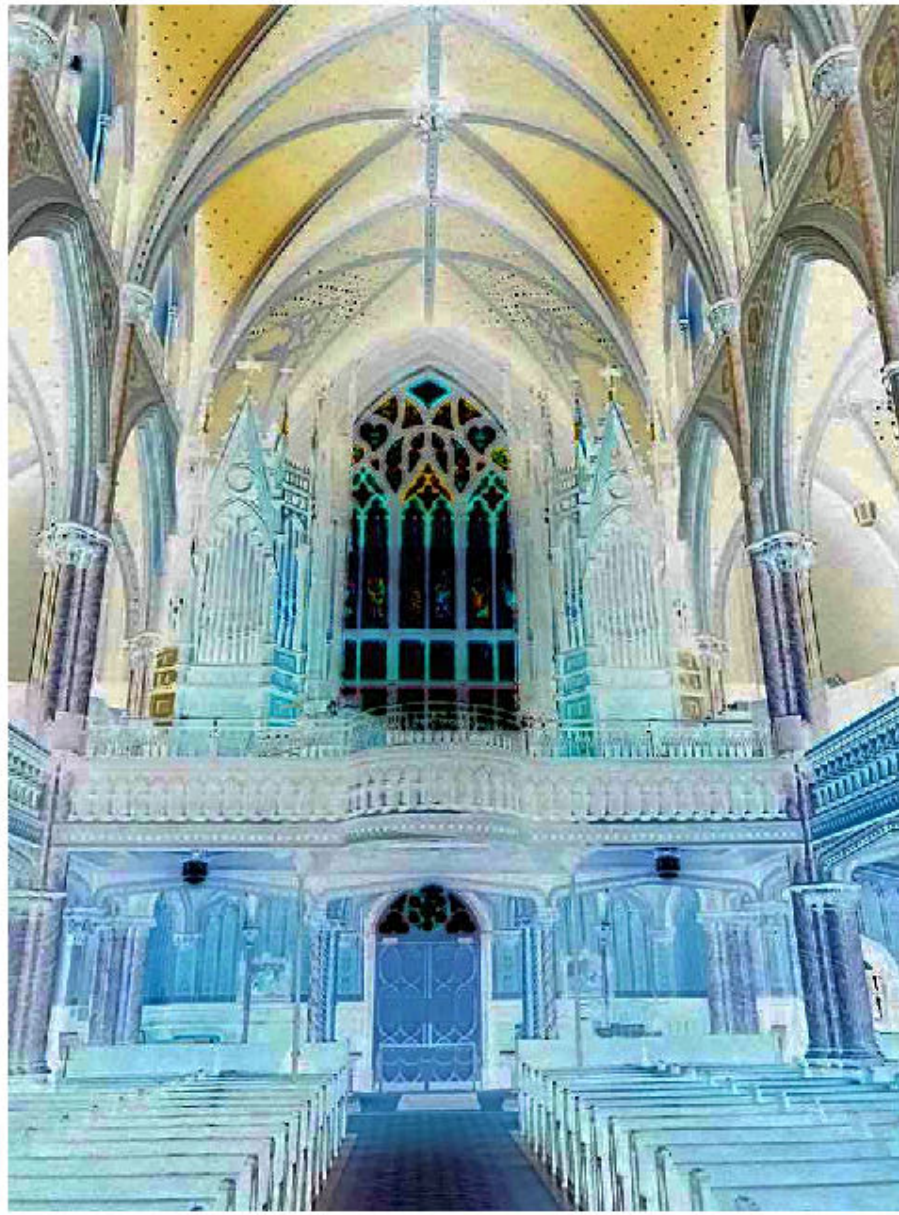
Invert the image and notice how the low-light areas in the original image appear hazy.

```
AInv = imcomplement(A);
figure, imshow(AInv);
```



Reduce the haze using the `imreducehaze` function.

```
BInv = imreducehaze(AInv);
figure, imshow(BInv);
```

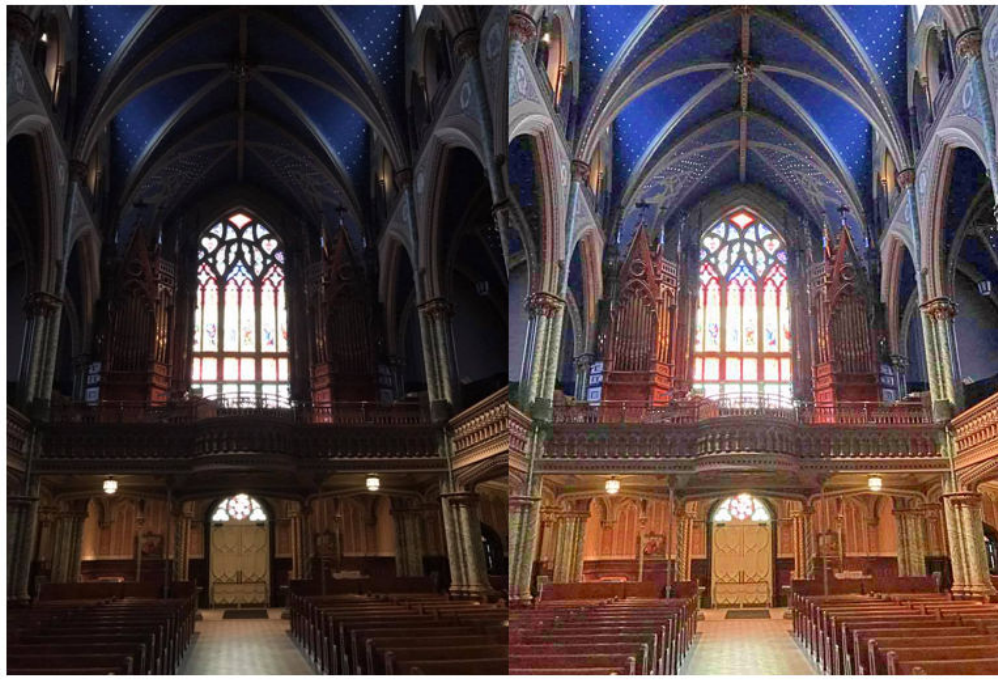


Invert the results to obtain the enhanced image.

```
B = imcomplement(BInv);
```

Display the original image and the enhanced images, side-by-side.

```
figure, montage({A, B});
```



Improve Results Further Using `imreducehaze` Optional Parameters

To get a better result, call `imreducehaze` on the inverted image again, this time specifying some optional parameters.

```
BInv = imreducehaze(AInv, 'Method','approx','ContrastEnhancement','boost');
BImp = imcomplement(BInv);
figure, montage({A, BImp});
```

Another Example of Improving a Poorly Lit Image

Import an RGB image captured in low light.

```
A = imread('lowlight_21.jpg');
```

Invert the image.

```
AInv = imcomplement(A);
```

Apply the dehazing algorithm.

```
BInv = imreducehaze(AInv, 'ContrastEnhancement', 'none');
```

Invert the results.

```
B = imcomplement(BInv);
```

Display the original image and the enhanced images, side-by-side.

```
figure, montage({A, B});
```

Reduce Color Distortion by using a Different Color Space

Convert the input image from the RGB colorspace to the L*a*b* colorspace.

```
Lab = rgb2lab(A);
```

Invert the L*a*b* image.

```
LInv = imcomplement(Lab(:,:,1) ./ 100);
```

Dehaze the inverted image using the `imreducehaze` function.

```
LEnh = imcomplement(imreducehaze(LInv,'ContrastEnhancement','none'));
```

Increase the saturation.

```
LabEnh(:,:,1)   = LEnh .* 100;
LabEnh(:,:,2:3) = Lab(:,:,2:3) * 2; % Increase saturation
```

Convert the image back to an RGB image and display the original and the enhanced image, side-by-side.

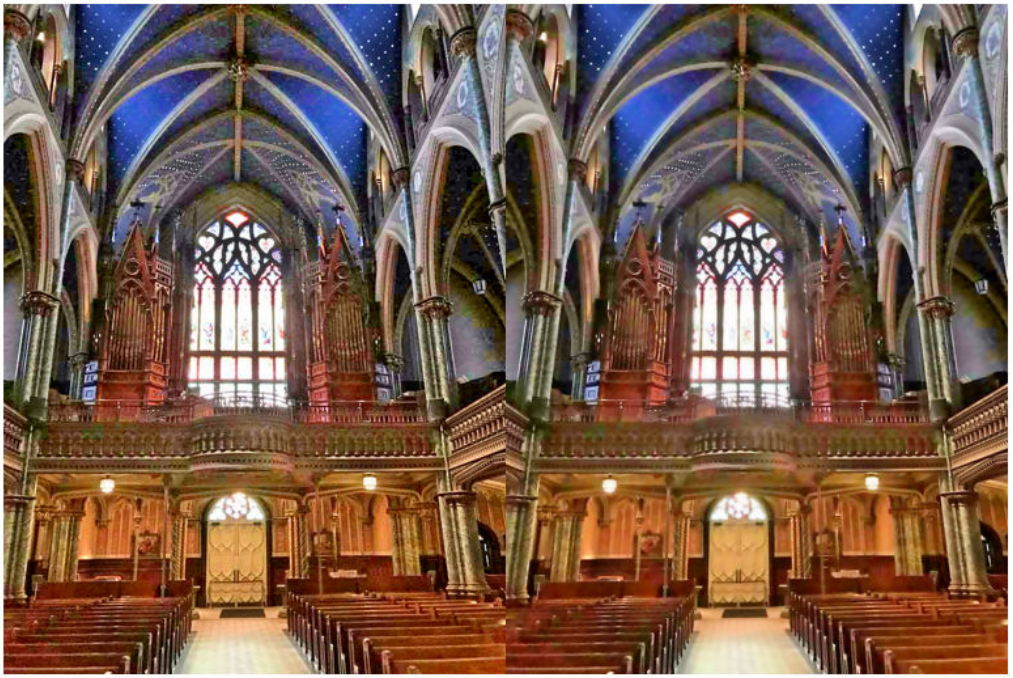
```
AEnh = lab2rgb(LabEnh);
figure, montage({A, AEnh});
```

Improve Results using Denoising

Low-light images can have high noise levels. Enhancing low-light images can increase this noise level. Denoising can be a useful post-processing step.

Use the `imguidedfilter` function to remove noise from the enhanced image.

```
B = imguidedfilter(BImp);
figure, montage({BImp, B});
```



Estimate Illumination Map

Import an RGB image captured in low light.

```
A = imread('lowlight_21.jpg');
```

Invert the image.

```
AInv = imcomplement(A);
```

Apply the dehazing algorithm to the image.

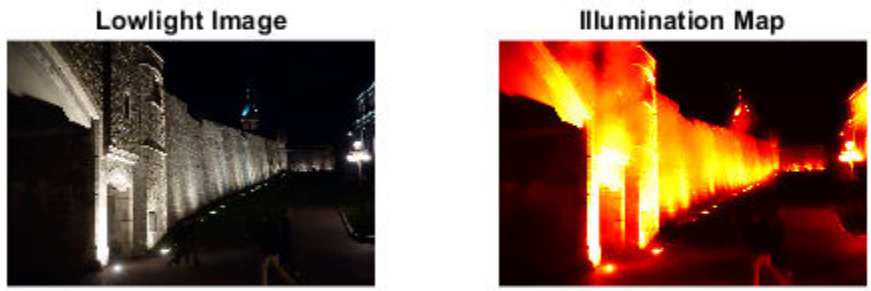
```
[BInv, TInv] = imreducehaze(AInv, 'Method', 'approxdcp', 'ContrastEnhancement', 'none'
```

Invert the enhanced image.

```
T = imcomplement(TInv);
```

Display the original image next to the estimated illumination map in false color.

```
figure,
subplot(1,2,1);
imshow(A), title('Lowlight Image');
subplot(1,2,2);
imshow(T), title('Illumination Map');
colormap(gca, hot(256));
```



Limitations

This method can lose some details or get over-enhanced because of poor adaptability of the dark channel in low-light conditions.

References

Dong, Xuan, et al. "Fast efficient algorithm for enhancement of low lighting video." Multimedia and Expo (ICME), 2011 IEEE International Conference on. IEEE, 2011.

References

[1] Dong, X., G. Wang, Y. Pang, W. Li, J. Wen, W. Meng, and Y. Lu. "Fast efficient algorithm for enhancement of low lighting video." *Proceedings of IEEE® International Conference on Multimedia and Expo (ICME)*. 2011, pp. 1–6.

See Also

`imcomplement` | `imguidedfilter` | `imreducehaze` | `lab2rgb` | `rgb2lab`

Apply Gaussian Smoothing Filters to Images

This example shows how to apply different Gaussian smoothing filters to images using `imgaussfilt`. Gaussian smoothing filters are commonly used to reduce noise.

Read an image into the workspace.

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

Filter the image with isotropic Gaussian smoothing kernels of increasing standard deviations. Gaussian filters are generally isotropic, that is, they have the same standard deviation along both dimensions. An image can be filtered by an isotropic Gaussian filter by specifying a scalar value for `sigma`.

```
Iblur1 = imgaussfilt(I,2);
Iblur2 = imgaussfilt(I,4);
Iblur3 = imgaussfilt(I,8);
```

Display the original image and all the filtered images.

```
figure
imshow(I)
title('Original image')
```

```
figure
imshow(Iblur1)
title('Smoothed image, \sigma = 2')
```

```
figure
imshow(Iblur2)
title('Smoothed image, \sigma = 4')
```

```
figure
imshow(Iblur3)
title('Smoothed image, \sigma = 8')
```

Filter the image with anisotropic Gaussian smoothing kernels. `imgaussfilt` allows the Gaussian kernel to have different standard deviations along row and column dimensions. These are called axis-aligned anisotropic Gaussian filters. Specify a 2-element vector for `sigma` when using anisotropic filters.

```
IblurX1 = imgaussfilt(I,[4 1]);
IblurX2 = imgaussfilt(I,[8 1]);
IblurY1 = imgaussfilt(I,[1 4]);
IblurY2 = imgaussfilt(I,[1 8]);
```

Display the filtered images.

```
figure
imshow(IblurX1)
title('Smoothed image, \sigma_x = 4, \sigma_y = 1')
```

```
figure
imshow(IblurX2)
title('Smoothed image, \sigma_x = 8, \sigma_y = 1')
```

```
figure
imshow(IblurY1)
title('Smoothed image, \sigma_x = 1, \sigma_y = 4')
```

```
figure
imshow(IblurY2)
title('Smoothed image, \sigma_x = 1, \sigma_y = 8')
```

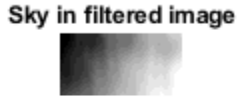
Suppress the horizontal bands visible in the sky region of the original image. Anisotropic Gaussian filters can suppress horizontal or vertical features in an image. Extract a section of the sky region of the image and use a Gaussian filter with higher standard deviation along the X axis (direction of increasing columns).

```
I_sky = imadjust(I(20:50,10:70));
IblurX1_sky = imadjust(IblurX1(20:50,10:70));
```

Display the original patch of sky with the filtered version.

```
figure
imshow(I_sky), title('Sky in original image')
```

```
figure
imshow(IblurX1_sky), title('Sky in filtered image')
```



Noise Removal

In this section...

"Remove Noise by Linear Filtering" on page 11-151
"Remove Noise Using an Averaging Filter and a Median Filter" on page 11-151
"Remove Noise By Adaptive Filtering" on page 11-155

Digital images are prone to various 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?" on page 8-2 for more information about linear filtering using `imfilter`.

Remove Noise Using an Averaging Filter and a Median Filter

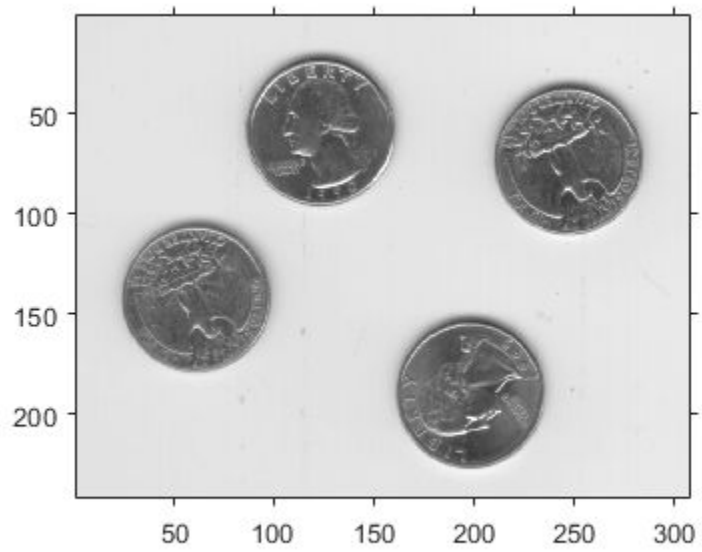
This example shows how to remove salt and pepper noise from an image using an averaging filter and a median filter to allow comparison of the results. These two types of

filtering both set the value of the output pixel to the average of the pixel values in the neighborhood around 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.

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

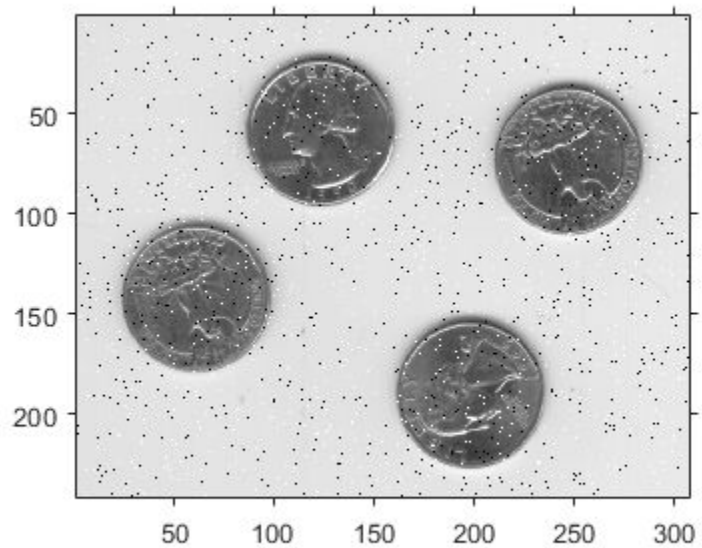
Read image into the workspace and display it.

```
I = imread('eight.tif');
figure
imshow(I)
```



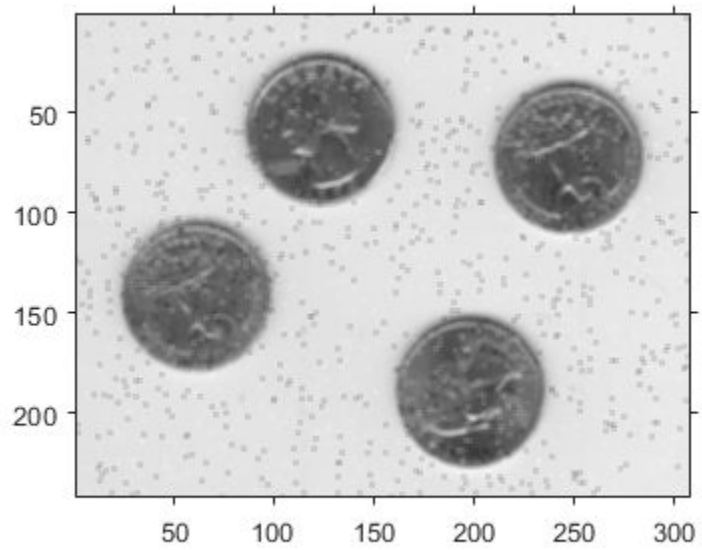
For this example, add salt and pepper noise to the image. This type of noise consists of random pixels being set to black or white (the extremes of the data range).

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



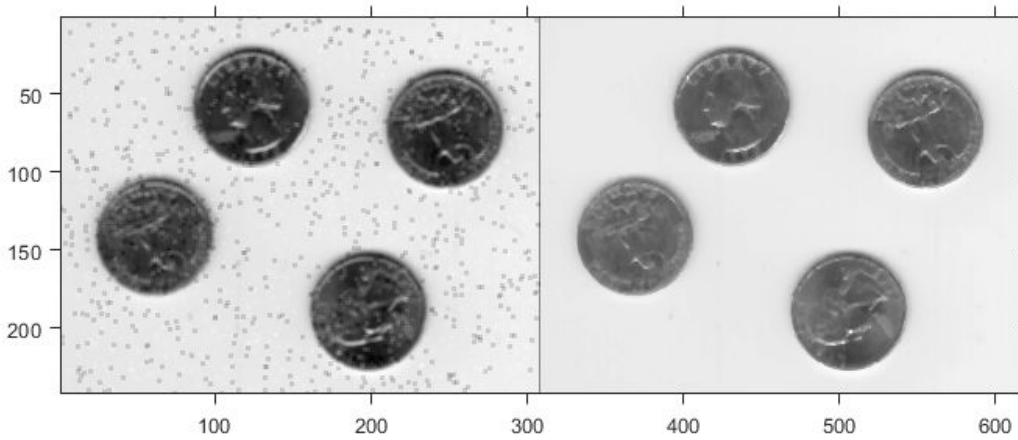
Filter the noisy image, J, with an averaging filter and display the results. The example uses a 3-by-3 neighborhood.

```
Kaverage = filter2(fspecial('average',3),J)/255;
figure
imshow(Kaverage)
```



Now use a median filter to filter the noisy image, `J`. The example also uses a 3-by-3 neighborhood. Display the two filtered images side-by-side for comparison. Notice that `medfilt2` does a better job of removing noise, with less blurring of edges of the coins.

```
Kmedian = medfilt2(J);
imshowpair(Kaverage,Kmedian,'montage')
```



Remove Noise By Adaptive Filtering

This example shows how to use the `wiener2` function to apply a Wiener filter (a type of linear filter) to an image adaptively. The Wiener filter tailors 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 with added Gaussian noise.

Read the image into the workspace.

```
RGB = imread('saturn.png');
```

Convert the image from truecolor to grayscale.

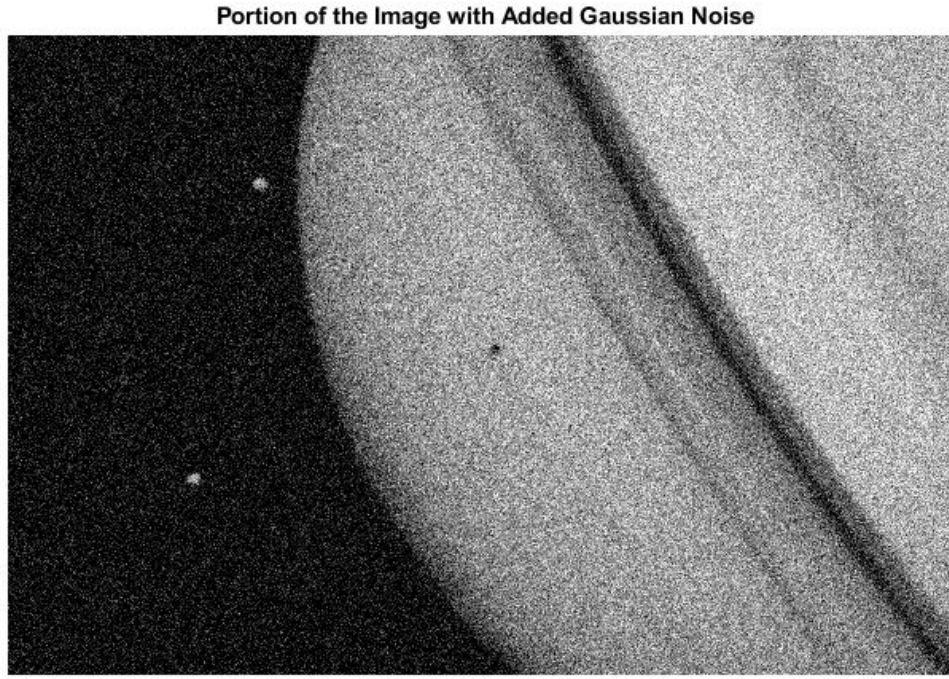
```
I = rgb2gray(RGB);
```

Add Gaussian noise to the image

```
J = imnoise(I,'gaussian',0,0.025);
```

Display the noisy image. Because the image is quite large, display only a portion of the image.

```
imshow(J(600:1000,1:600));
title('Portion of the Image with Added Gaussian Noise');
```

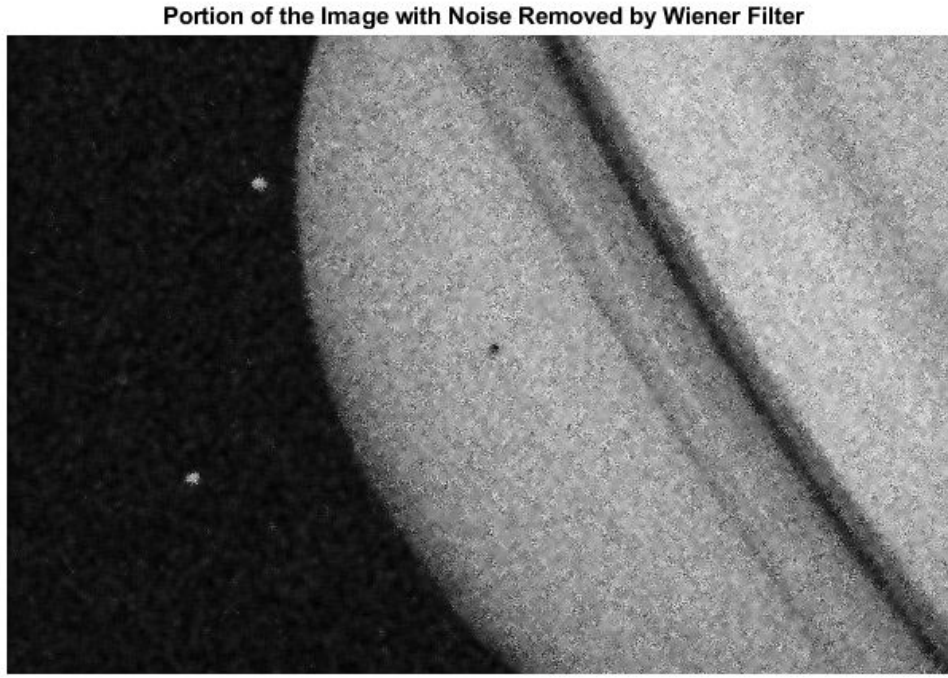


Remove the noise using the `wiener2` function.

```
K = wiener2(J,[5 5]);
```

Display the processed image. Because the image is quite large, display only a portion of the image.

```
figure
imshow(K(600:1000,1:600));
title('Portion of the Image with Noise Removed by Wiener Filter');
```



See Also

imbilatfilt | imfilter | imgaussfilt | imguidedfilter | locallapfilt | nlfilter

More About

- “What Is Image Filtering in the Spatial Domain?” on page 8-2

Compute 3-D Superpixels of Input Volumetric Intensity Image

Load 3-D MRI data, remove any singleton dimensions, and convert the data into a grayscale intensity image.

```
load mri;
D = squeeze(D);
A = ind2gray(D,map);
```

Calculate the 3-D superpixels. Form an output image where each pixel is set to the mean color of its corresponding superpixel region.

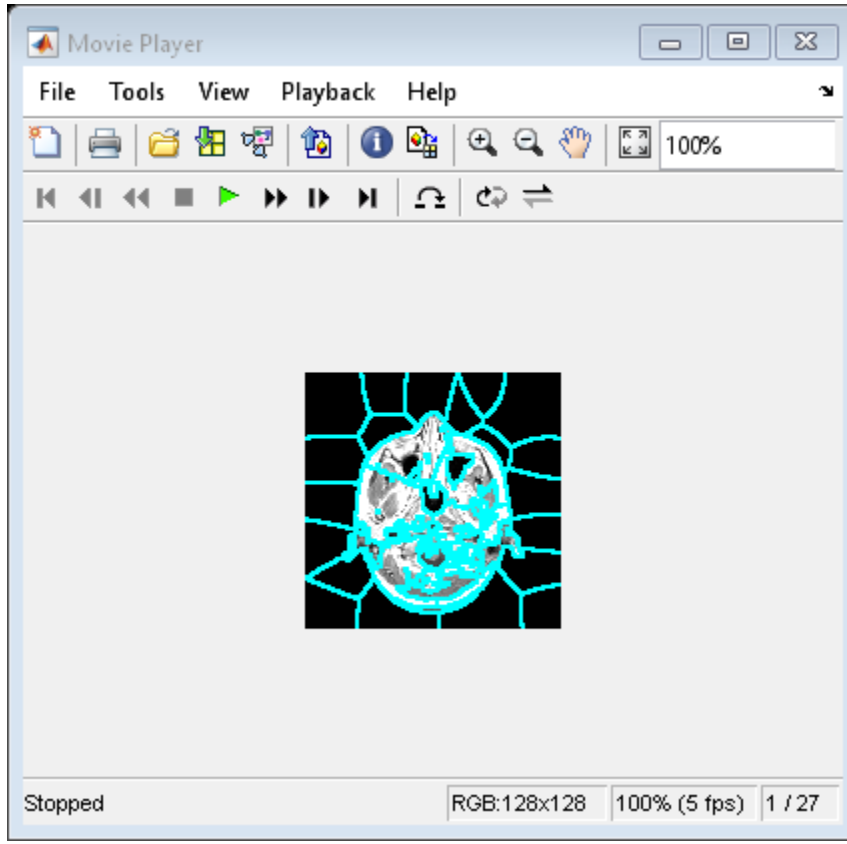
```
[L,N] = superpixels3(A,34);
```

Show all xy-planes progressively with superpixel boundaries.

```
imSize = size(A);
```

Create a stack of RGB images to display the boundaries in color.

```
imPlusBoundaries = zeros(imSize(1),imSize(2),3,imSize(3),'uint8');
for plane = 1:imSize(3)
  BW = boundarymask(L(:, :, plane));
  % Create an RGB representation of this plane with boundary shown
  % in cyan.
  imPlusBoundaries(:, :, :, plane) = imoverlay(A(:, :, plane), BW, 'cyan');
end

implay(imPlusBoundaries,5)
```



Set the color of each pixel in output image to the mean intensity of the superpixel region. Show the mean image next to the original. If you run this code, you can use `implay` to view each slice of the MRI data.

```
pixelIdxList = label2idx(L);
meanA = zeros(size(A),'like',D);
for superpixel = 1:N
     memberPixelIdx = pixelIdxList{superpixel};
     meanA(memberPixelIdx) = mean(A(memberPixelIdx));
end
implay([A meanA],5);
```

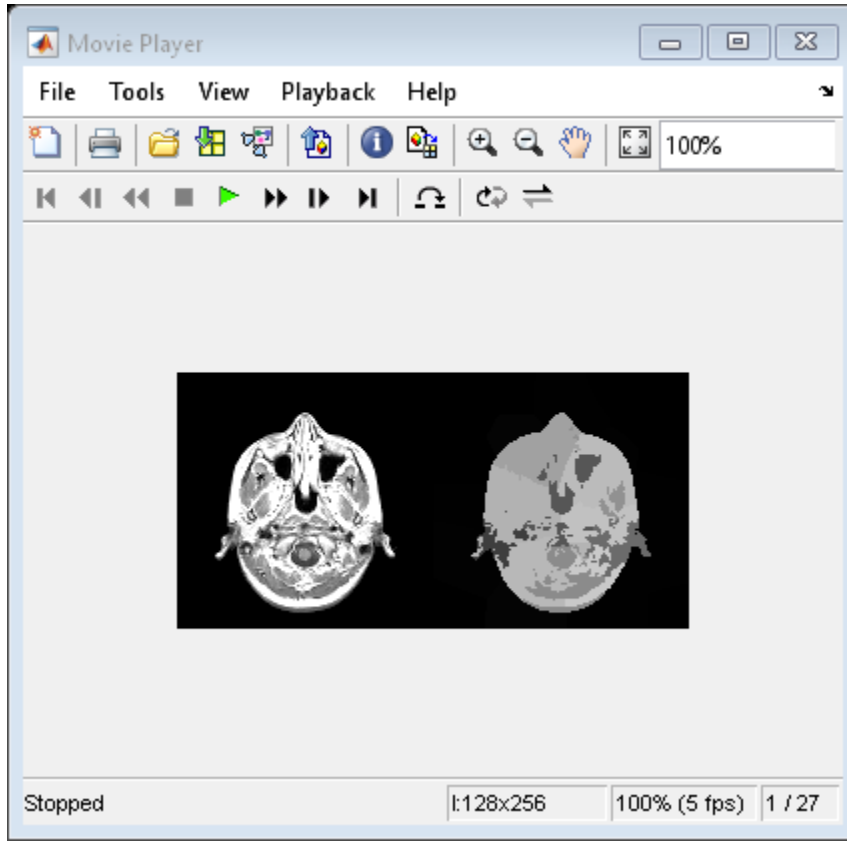

Movie Player
File
Tools
View
Playback
Help
100%
Stopped
I:128x256
100% (5 fps)
1 / 27

Image Quality Metrics

Image quality can degrade due to distortions during image acquisition and processing. Examples of distortion include noise, blurring, ringing, and compression artifacts.

Efforts have been made to create objective measures of quality. For many applications, a valuable quality metric correlates well with the subjective perception of quality by a human observer. Quality metrics can also track unperceived errors as they propagate through an image processing pipeline, and can be used to compare image processing algorithms.

If an image without distortion is available, you can use it as a reference to measure the quality of other images. For example, when evaluating the quality of compressed images, an uncompressed version of the image provides a useful reference. In these cases, you can use full-reference quality metrics to directly compare the target image and the reference image.

If a reference image without distortion is not available. you can use a no-reference image quality metric instead. These metrics compute quality scores based on expected image statistics.

Full-Reference Quality Metrics

Full-reference algorithms compare the input image against a pristine reference image with no distortion. These algorithms include:

- `immse` — Mean-squared error (MSE). MSE measures the average squared difference between actual and ideal pixel values. This metric is simple to calculate but might not align well with the human perception of quality.
- `psnr` — Peak signal-to-noise ratio (pSNR). pSNR is derived from the mean square error, and indicates the ratio of the maximum pixel intensity to the power of the distortion. Like MSE, the pSNR metric is simple to calculate but might not align well with perceived quality.
- `ssim` — Structural Similarity (SSIM) Index. The SSIM metric combines local image structure, luminance, and contrast into a single local quality score. In this metric, structures are patterns of pixel intensities, especially among neighboring pixels, after normalizing for luminance and contrast. Because the human visual system is good at perceiving structure, the SSIM quality metric agrees more closely with the subjective quality score.

Because structural similarity is computed locally, `ssim` can generate a map of quality over the image.

No-Reference Quality Metrics

No-reference algorithms use statistical features of the input image to evaluate the image quality. These no-reference algorithms include:

- `brisque` — Blind/Referenceless Image Spatial Quality Evaluator (BRISQUE). A BRISQUE model is trained on a database of images with known distortions, and BRISQUE is limited to evaluating the quality of images with the same type of distortion. BRISQUE is *opinion-aware*, which means subjective quality scores accompany the training images.
- `niqe` — Natural Image Quality Evaluator (NIQE). Although a NIQE model is trained on a database of pristine images, NIQE can measure the quality of images with arbitrary distortion. NIQE is *opinion-unaware*, and does not use subjective quality scores. The tradeoff is that the NIQE score of an image might not correlate as well as the BRISQUE score with human perception of quality.
- `piqe` — Perception based Image Quality Evaluator (PIQE). The PIQE algorithm is opinion-unaware and *unsupervised*, which means it does not require a trained model. PIQE can measure the quality of images with arbitrary distortion and in most cases performs similar to NIQE. PIQE estimates block-wise distortion and measures the local variance of perceptibly distorted blocks to compute the quality score.

The BRISQUE and the NIQE algorithms calculate the quality score of an image with computational efficiency after the model is trained. PIQE is less computationally efficient, but it provides local measures of quality in addition to a global quality score. All no-reference quality metrics usually outperform full-reference metrics in terms of agreement with a subjective human quality score.

See Also

More About

- “Train and Use a No-Reference Quality Assessment Model” on page 11-164
- “Obtain Local Structural Similarity Index” on page 11-169
- “Compare Image Quality at Various Compression Levels” on page 11-172

Train and Use a No-Reference Quality Assessment Model

The Natural Image Quality Evaluator (NIQE) and Blind/Referenceless Image Spatial Quality Evaluator (BRISQUE) algorithms use a trained model to compute a quality score.

Both algorithms train a model using identical predictable statistical features, called natural scene statistics (NSS). NSS are based on normalized luminance coefficients in the spatial domain, and are modeled as a multidimensional Gaussian distribution. Distortions appear as perturbations to the Gaussian distribution.

The algorithms differ in how they use the NSS features to train a model and compute a quality score.

In this section...
"NIQE Workflow" on page 11-164 "BRISQUE Workflow" on page 11-167

NIQE Workflow

NIQE measures the quality of images with arbitrary distortion. A NIQE model is not trained using subjective quality scores, but the tradeoff is that the NIQE score does not correlate as reliably as the BRISQUE score with human perception of quality.

Train a NIQE Model

Note If the default NIQE model provides a sufficient quality score for your application, you do not need to train a new model. You can skip to "Predict Image Quality Using a NIQE Model" on page 11-165.

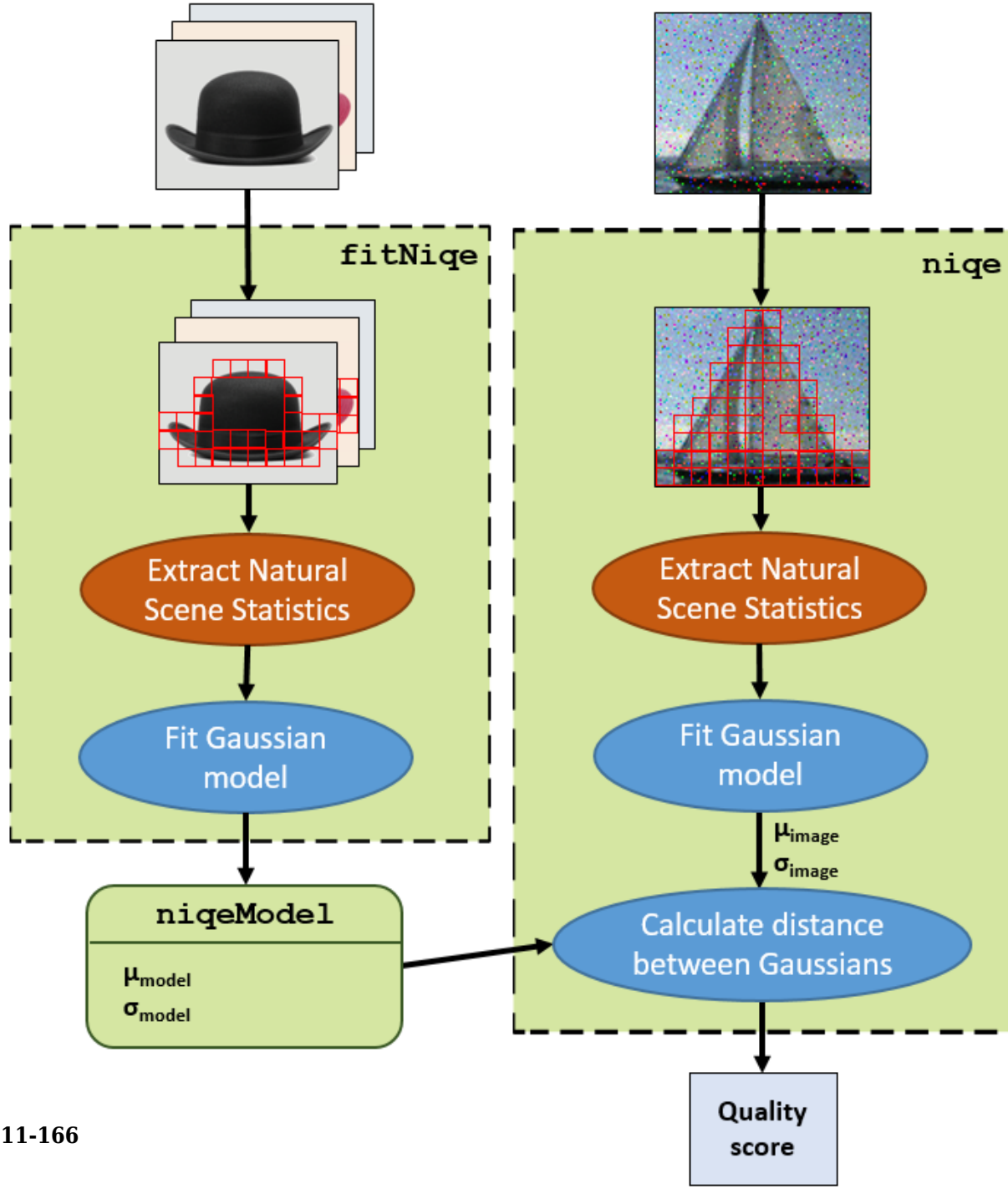
To train a NIQE model, pass a datastore of pristine image to the `fitniqe` function. The function divides each image into blocks and computes the NSS for each block. The training process includes only blocks with statistically significant features.

The returned model, `niqeModel`, stores the multivariate Gaussian mean and standard deviation derived from the NSS features.

Predict Image Quality Using a NIQE Model

Use the `niqe` function to calculate an image quality score for an image with arbitrary distortion. The `niqe` function extracts the NSS features from statistically significant blocks in the distorted image. The function fits a multivariate Gaussian distribution to the image NSS features. The quality score is the distance between the Gaussian distributions.

The diagram shows the full NIQE workflow.


fitNiqe
Extract Natural Scene Statistics
Fit Gaussian model
niqeModel
μmodel
σmodel
niqe
Extract Natural Scene Statistics
Fit Gaussian model
μimage
σimage
Calculate distance between Gaussians
Quality score

BRISQUE Workflow

BRISQUE is limited to measuring the quality of images with the same type of distortion as the model. A BRISQUE model is trained using subjective opinion scores, with the advantage that the BRISQUE score correlates well with human perception of quality.

Train a BRISQUE Model

Note If the default BRISQUE model provides a sufficient quality score for your application, you do not need to train a new model. You can skip to “Predict Image Quality Using a BRISQUE Model” on page 11-167.

To train a BRISQUE model, pass to the `fitbrisque` function:

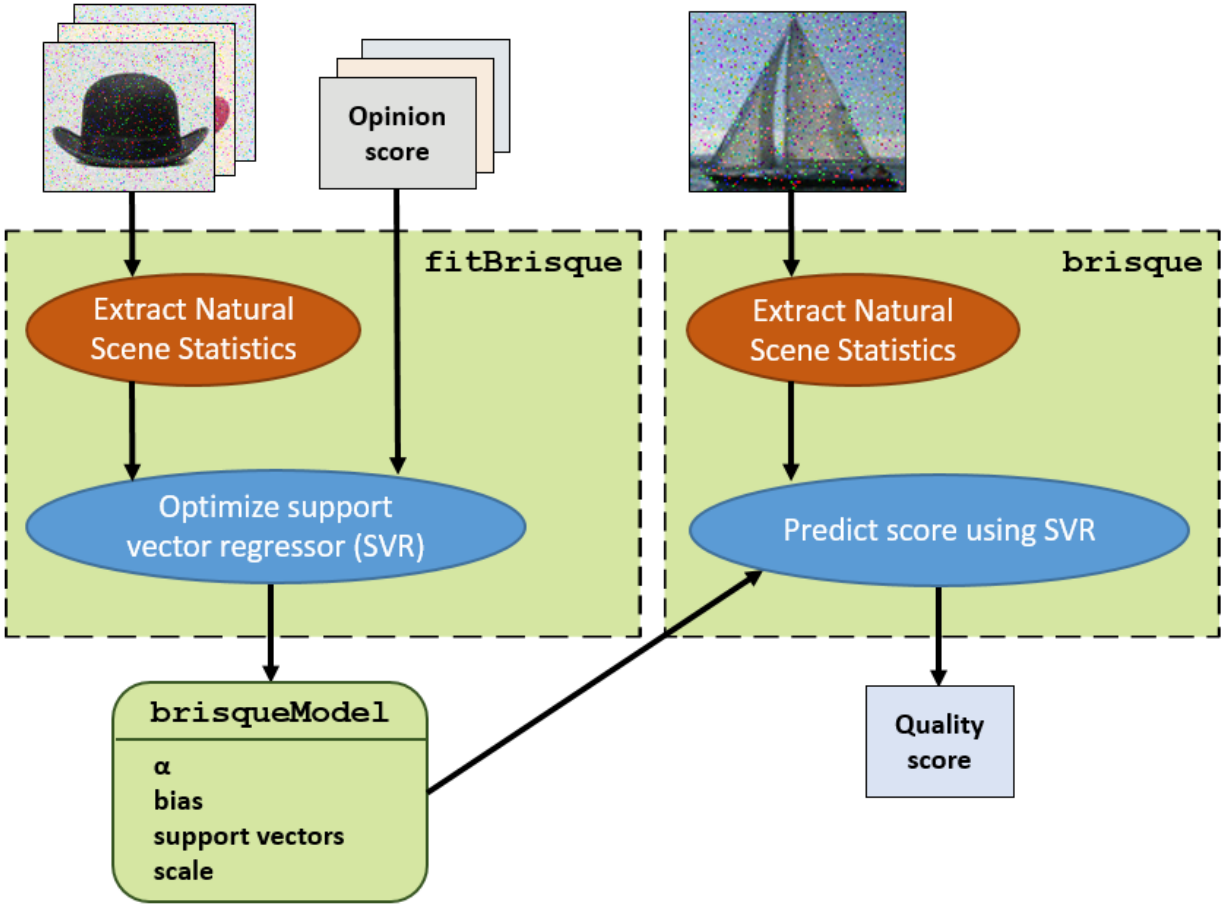
- A datastore containing images with known distortions and pristine copies of those images
- A subjective opinion score for each distorted image in the database

The function computes the NSS features for each image, without dividing the image into blocks. The function uses the NSS features and corresponding opinion scores to train a support vector machine regression model. The returned model, `brisqueModel`, stores the parameters of the support vector regressor.

Predict Image Quality Using a BRISQUE Model

Use the `brisque` function to calculate an image quality score for an image with the same type of distortions as the model. The `brisque` function extracts the NSS features from the distorted image, and predicts a quality score using support vector regression.

The diagram shows the full BRISQUE workflow.



See Also

More About

- “Image Quality Metrics” on page 11-162

Obtain Local Structural Similarity Index

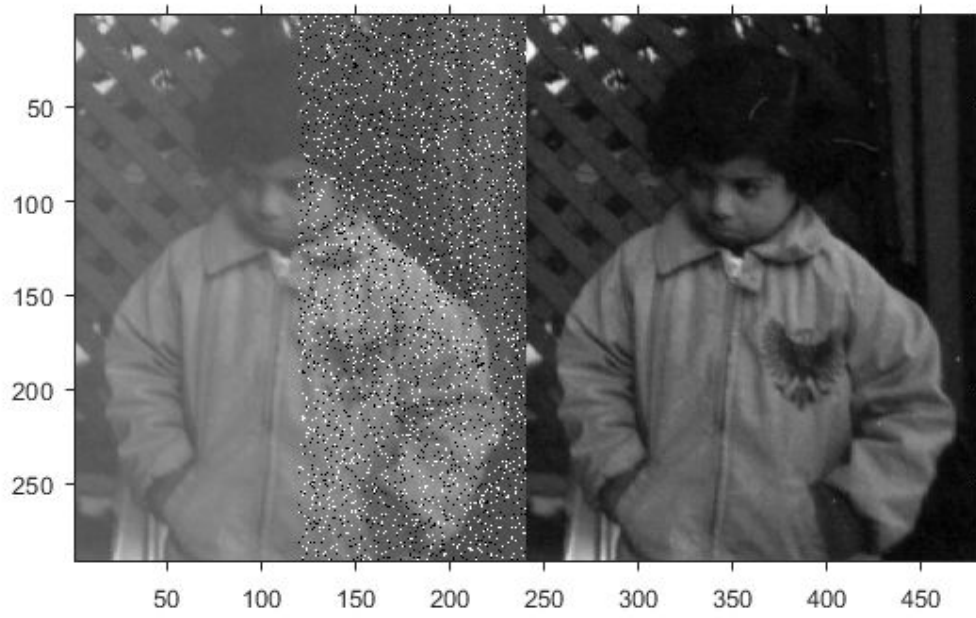
This example shows how to measure the quality of regions of an image when compared with a reference image. The `ssim` function calculates the structural similarity index for each pixel in an image, based on its relationship to other pixels in an 11-by-11 neighborhood. The function returns this information 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');
```

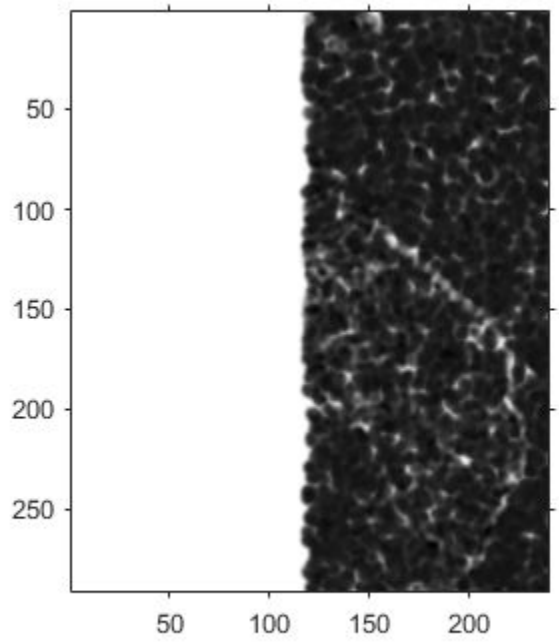
Create an image whose quality is to be measured, by making a copy of the reference image and adding noise. To illustrate local similarity, isolate the noise to half of the image. Display the reference image and the noisy image side-by-side.

```
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,[])
```



See Also

`ssim`

More About

- “Image Quality Metrics” on page 11-162

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 an image into the workspace.

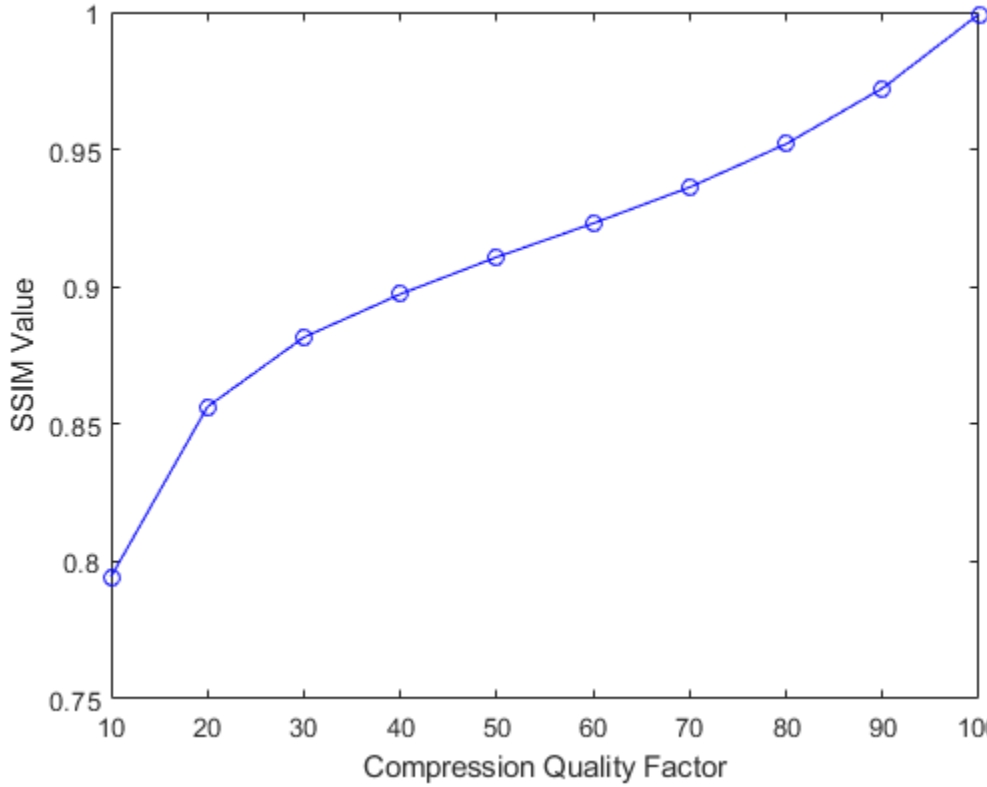
```
I = imread('cameraman.tif');
```

Write the image to a file using various quality values. The JPEG format supports the `'quality'` parameter. Use the `ssim` function 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');
```



See Also

`ssim`

More About

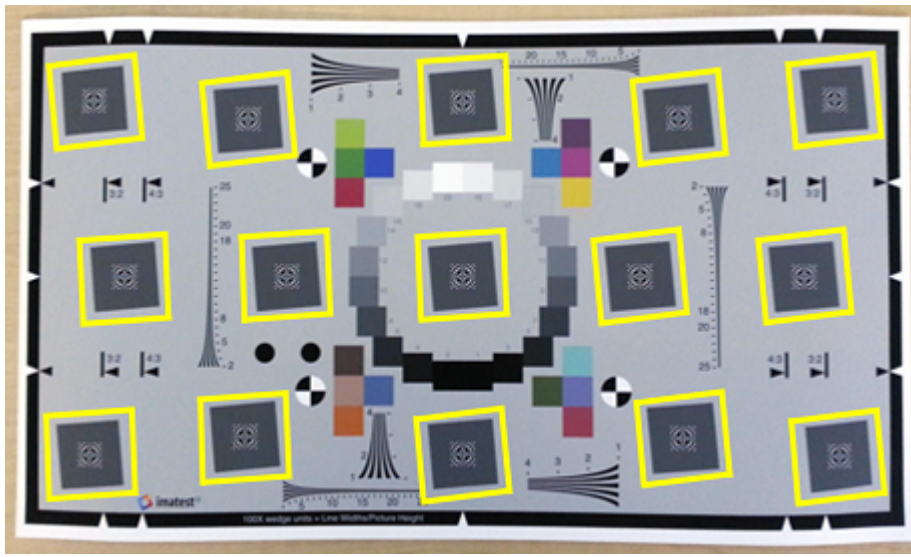
- “Image Quality Metrics” on page 11-162

Anatomy of an eSFR Chart

The Imatest® edge spatial frequency response (eSFR) test charts contain visual features that enable sharpness measurements according to the ISO 12233:2014 standard [1]. You can use a single test chart to measure chromatic aberration, noise, and scene illumination. Registration markers on the chart enable automatic selection of regions of interest (ROIs) for measurements.

Image Processing Toolbox supports the Extended version of the eSFR chart, which has additional features to measure color accuracy. The `esfrChart` object does not analyze other visual features in the chart, such as focusing targets and wedges.

Slanted Edge Features



The Extended eSFR test chart has 15 gray boxes tilted 5° away from vertical. The left, top, right, and bottom edges of each box are used to measure:

- Local SFR, which indicates image sharpness. Sharp edges have better contrast than blurry edges, and they more clearly show the actual position of the edge.
    - In sharp edges, pixel intensity values quickly transition across boundaries in the scene. Most pixels clearly belong to one side of the boundary or the other, and few

pixels have intermediate values. The contrast is high because adjacent pixels on either side of the actual edge have large differences in intensity.

- In blurry edges, the transition happens gradually over many pixels, which makes it unclear where the boundary actually occurs. The contrast is low because adjacent pixels have similar intensity values.

Sharpness is higher toward the center of the imaged region and decreases toward the periphery. Horizontal sharpness is usually higher than vertical sharpness. To measure SFR, use the `measureSharpness` function.

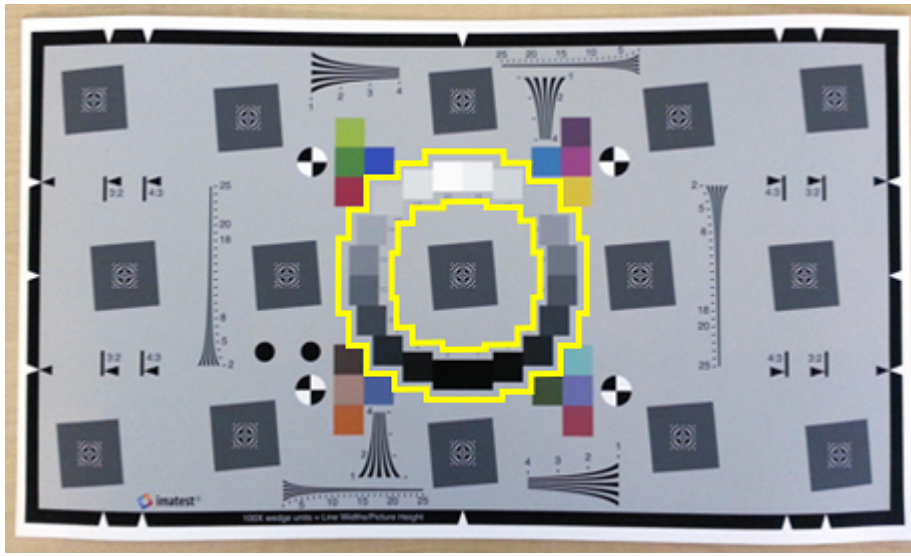
- Local chromatic aberration, or color fringing, which indicates how uniformly the camera optical system focuses light in the red, green, and blue color channels. In captured images, chromatic aberration appears as an artificial strip of color along edges. Chromatic aberration also lowers contrast in the luminance channel, and therefore reduces image sharpness.

  Chromatic aberration increases radially from the center of the image. To measure chromatic aberration, use the `measureChromaticAberration` function. This function also returns the edge profiles for each color channel, which is the averaged projection of pixel intensity values along the edge.

If you capture an image of the test chart at the full 16:9 aspect ratio, then `esfrChart` automatically identifies and labels all 60 available slanted-edge ROIs. You can also image the test chart at the 4:3 and 3:2 aspect ratios, as indicated on the chart. At these ratios, fewer edges are available, and `esfrChart` indexes edges according to the convention used by the 16:9 aspect ratio.

In a proper capture of the test chart, orient the chart without rotation, so that the angle of the slanted edges is close to 5°. The contrast of the edges must be greater than 20%. If the contrast is less than 20%, adjust the scene lighting and the camera exposure.

Gray Patch Features

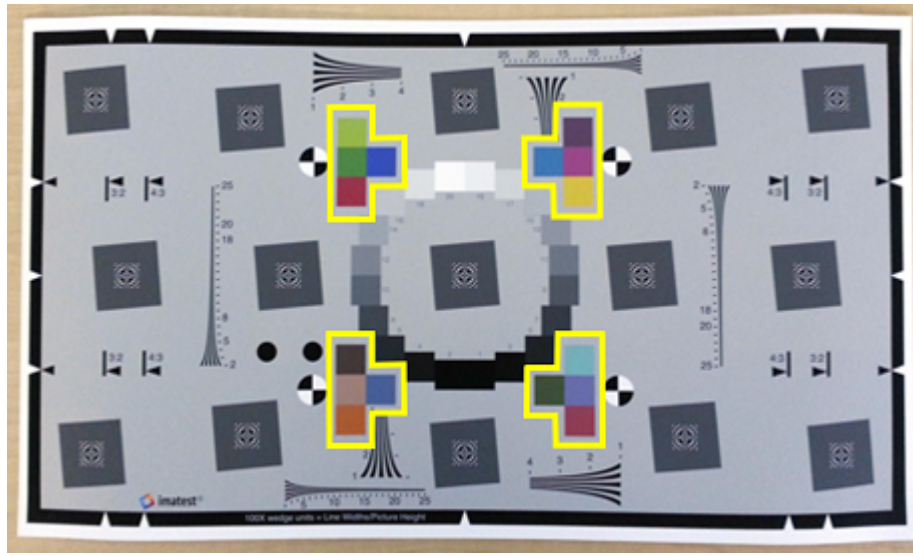


The Extended eSFR test chart has 20 gray patches of increasing brightness, arranged in a ring around the center of the image. The gray patches are used to measure:

- Scene illumination, which estimates the color of the light illuminating the scene. You can use the measured illumination to white-balance images acquired under similar lighting conditions. To measure scene illumination, use the `measureIlluminant` function. To white-balance the image, use `chromadapt`.
- Noise, which quantifies how much the camera electronics generate error in pixel values. To estimate noise in each color channel, use the `measureNoise` function.

In a proper capture of the test chart, set the scene lighting and camera exposure so that each gray patch appears distinct from the other patches and no pixels are clipped. If the darkest patches appear identical, or have values of 0, increase the scene lighting or the exposure. If the brightest patches appear identical, or if the brightest patch is saturated, decrease the scene lighting or the exposure.

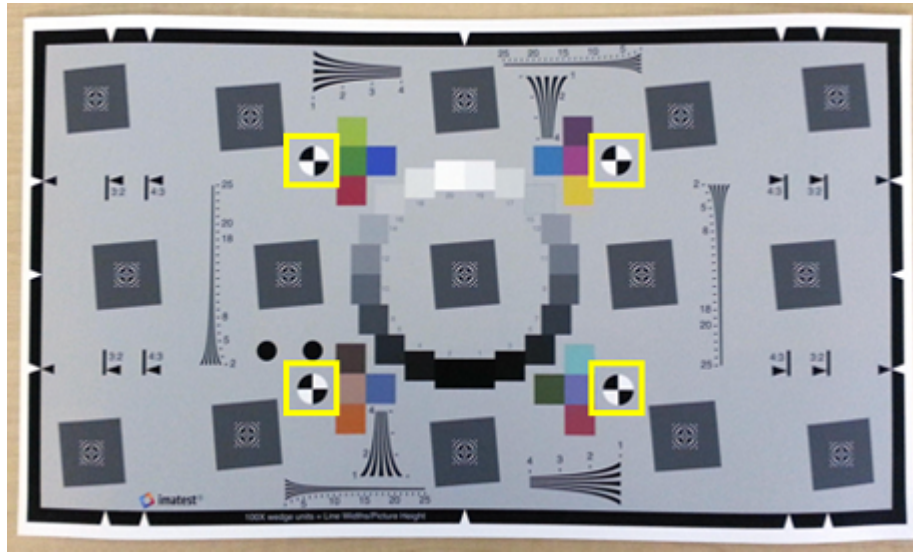
Color Patch Features



The Extended eSFR test chart has 16 color patches arranged in four groups. The color patches are used to measure:

- Color accuracy, which indicates how well the measured red, green, and blue values agree with expected color values. To measure color accuracy, use the `measureColor` function. This function also returns a color correction matrix, which you can use to adjust the image colors toward the expected values.

Registration Markers



The eSFR test charts have registration markers used to orient the image properly. When you import a chart, `esfrChart` detects four black-and-white checkered circles and uses their position to define regions of interest automatically. You can optionally specify the `[x, y]` coordinates of the circle centers manually.

References

[1] ISO 12233:2014. "Photography – Electronic still picture imaging – Resolution and spatial frequency responses." *International Organization for Standardization; ISO/TC 42 Photography*. URL: https://www.iso.org/standard/59419.html.

See Also

`displayChart` | `esfrChart`

More About

- “Evaluate Quality Metrics on eSFR Test Chart” on page 11-179

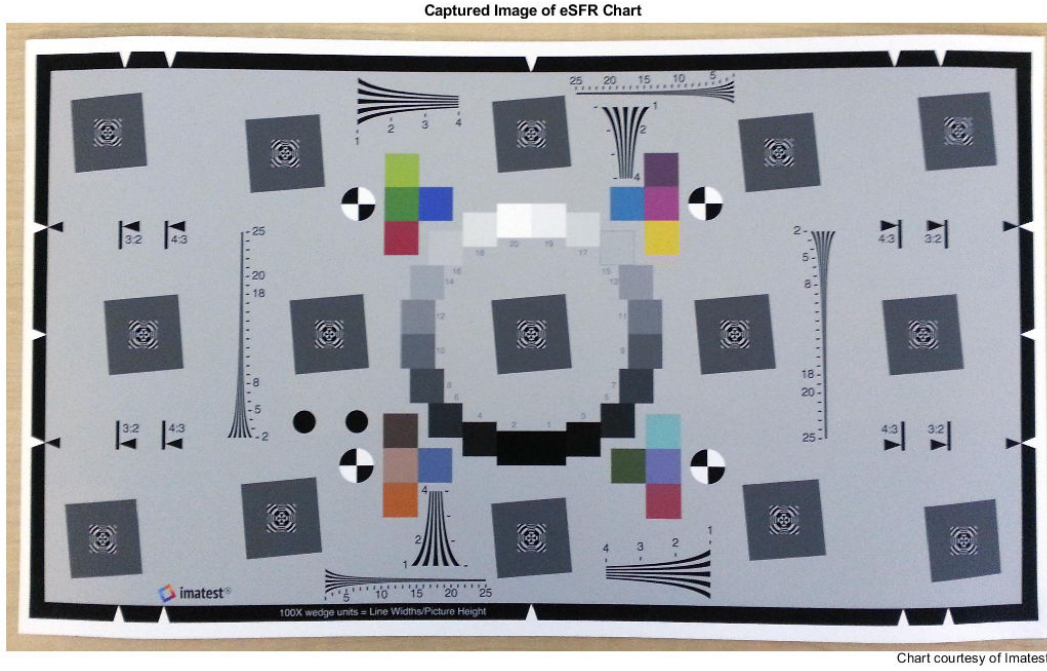
Evaluate Quality Metrics on eSFR Test Chart

This example shows how to perform standard quality measurements on an Imatest® edge spatial frequency response (eSFR) test chart. Measured properties include sharpness, chromatic aberration, noise, illumination, and color accuracy.

Create a Test Chart Object

Read an image of an eSFR chart into the workspace. Display the chart.

```
I = imread('eSFRTestImage.jpg');
figure
imshow(I)
title('Captured Image of eSFR Chart')
text(size(I,2),size(I,1)+15, ...
    ['Chart courtesy of Imatest',char(174)],'FontSize',10,'HorizontalAlignment','right
```

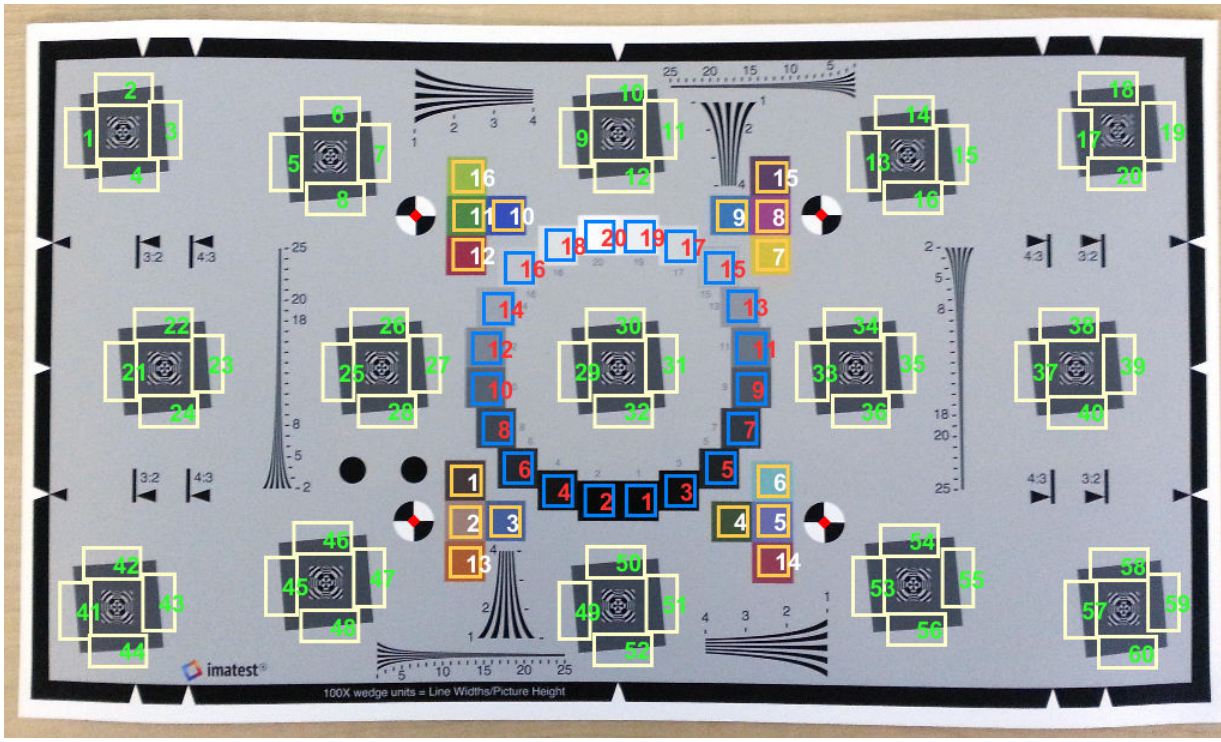


Create an eSFR test chart object that automatically defines regions of interest (ROIs) based on detected registration markers.

```
chart = esfrChart(I);
```

Highlight and label the detected ROIs to visually confirm that the ROIs are suitable for measurements.

```
displayChart(chart)
```



All 60 slanted edge ROIs (labeled in green) are visible and centered on appropriate edges. In addition, 20 gray patch ROIs (labeled in red) and 16 color patch ROIs (labeled in white) are visible and are contained within the boundary of each patch. The chart is correctly imported.

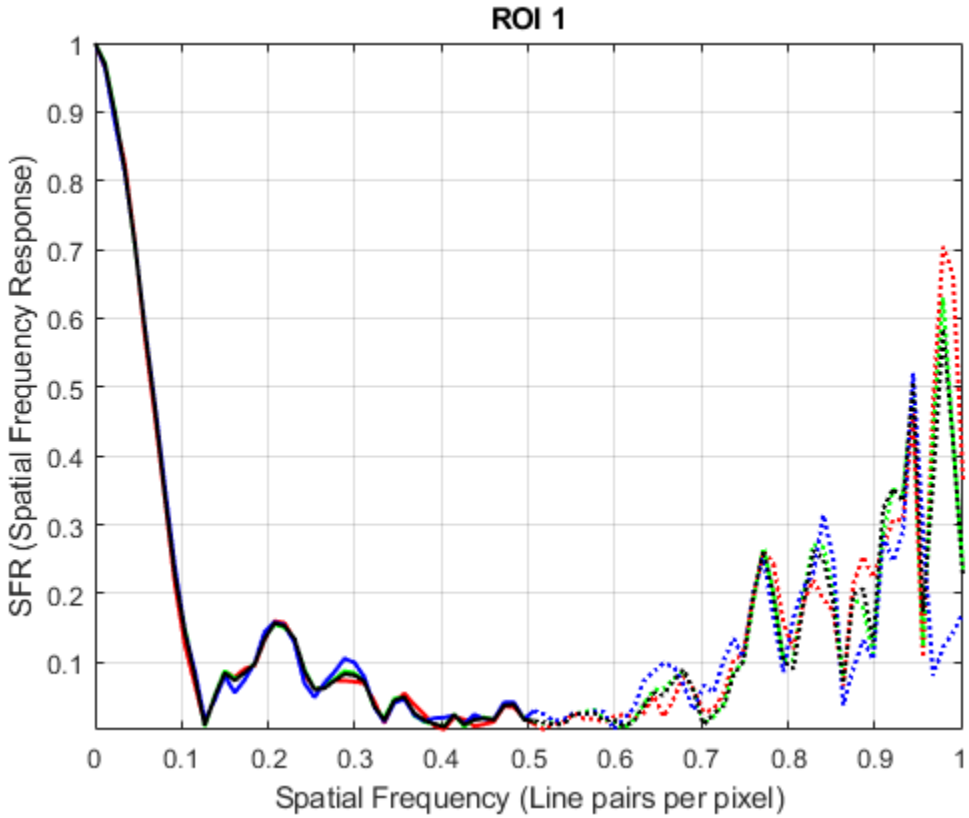
Measure Edge Sharpness

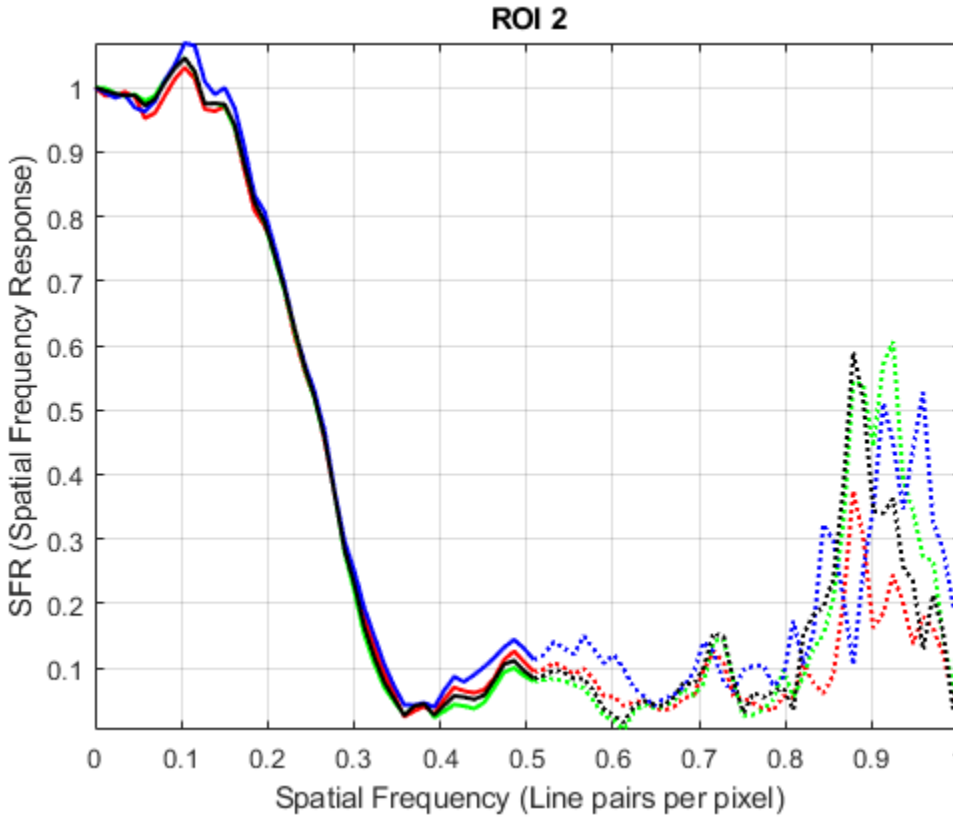
Measure the sharpness of all 60 slanted edge ROIs. Also measure the averaged horizontal and vertical sharpness of these ROIs.

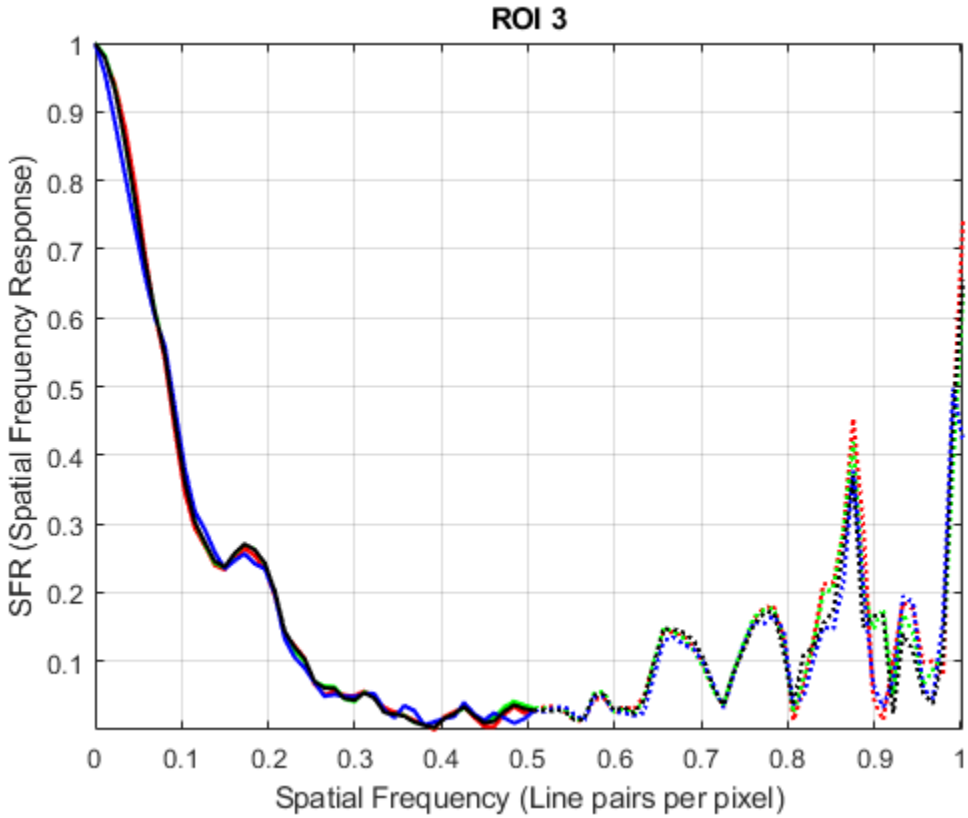
```
[sharpnessTable,aggregateSharpnessTable] = measureSharpness(chart);
```

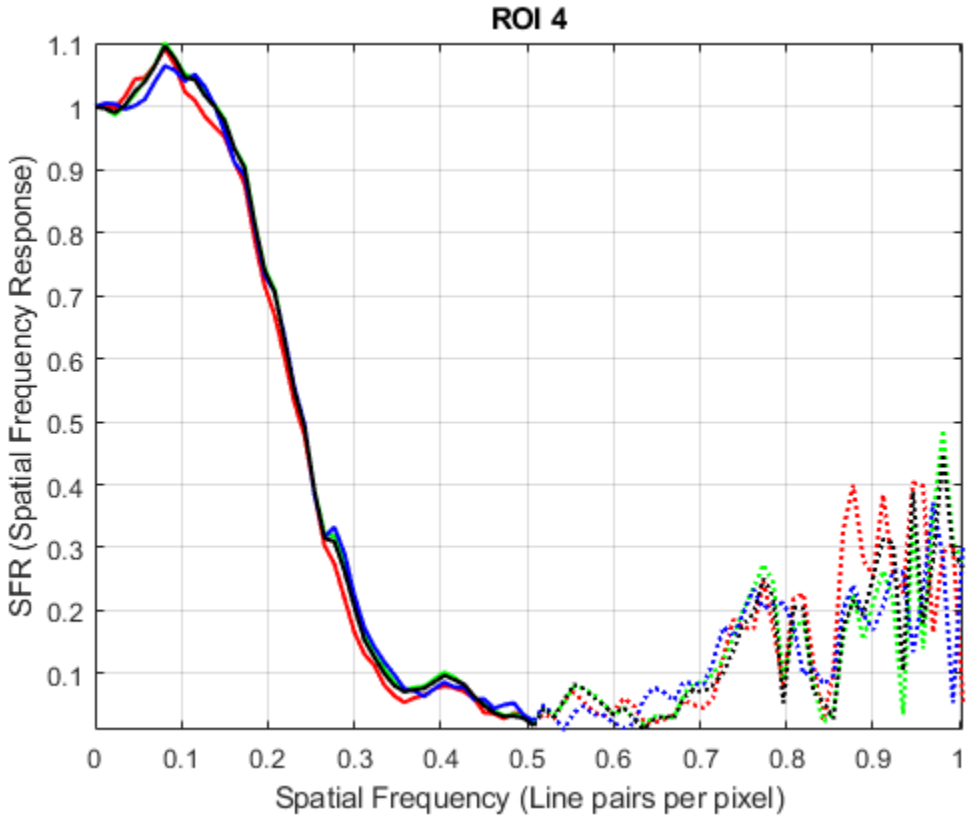
Display the SFR plot for the first four ROIs.

```
plotSFR(sharpnessTable,'ROIIndex',1:4,'displayLegend',false,'displayTitle',true)
```



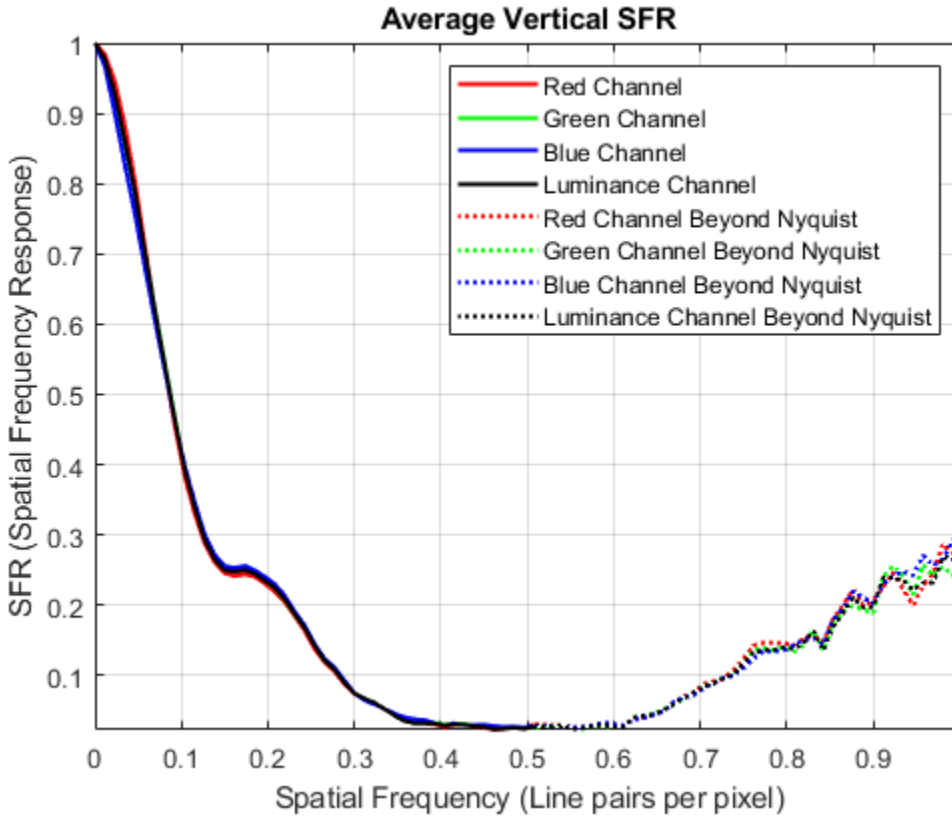

ROI 2
SFR (Spatial Frequency Response)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Spatial Frequency (Line pairs per pixel)

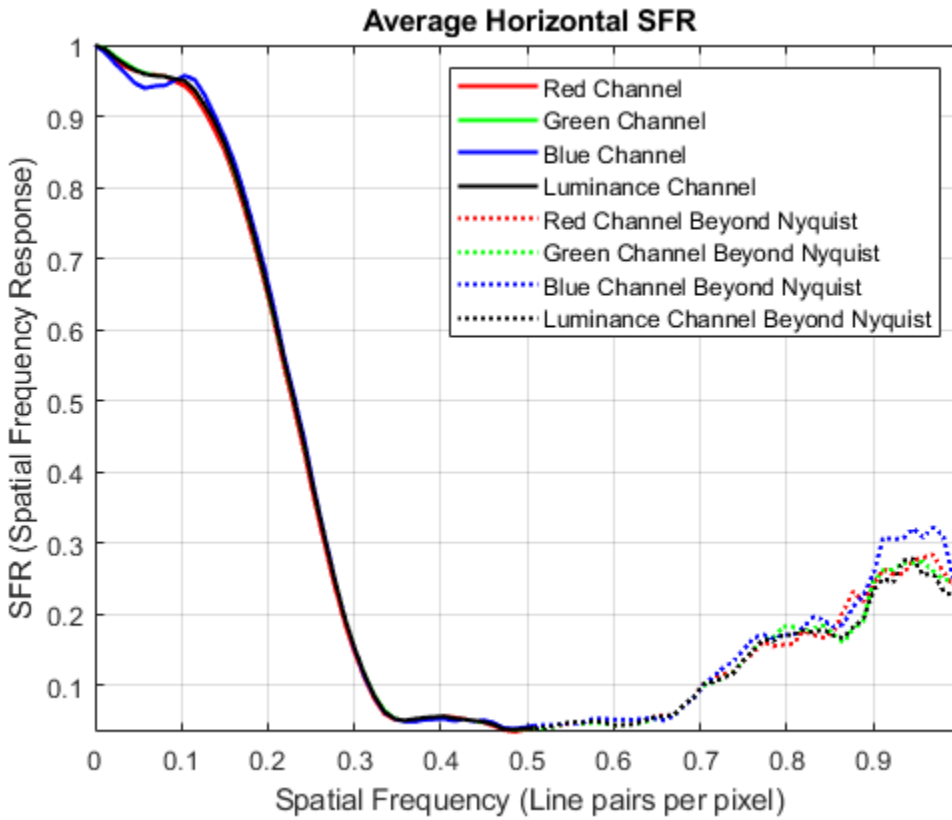

ROI 3
SFR (Spatial Frequency Response)
Spatial Frequency (Line pairs per pixel)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1



Display the average SFR of the averaged vertical and horizontal edges. The average vertical SFR drops off more rapidly than the average horizontal SFR. Therefore, the average vertical edge is less sharp than the average horizontal edge.

```
plotSFR(aggregateSharpnessTable)
```


Average Vertical SFR
SFR (Spatial Frequency Response)
Spatial Frequency (Line pairs per pixel)
Red Channel
Green Channel
Blue Channel
Luminance Channel
Red Channel Beyond Nyquist
Green Channel Beyond Nyquist
Blue Channel Beyond Nyquist
Luminance Channel Beyond Nyquist
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1



Measure Chromatic Aberration

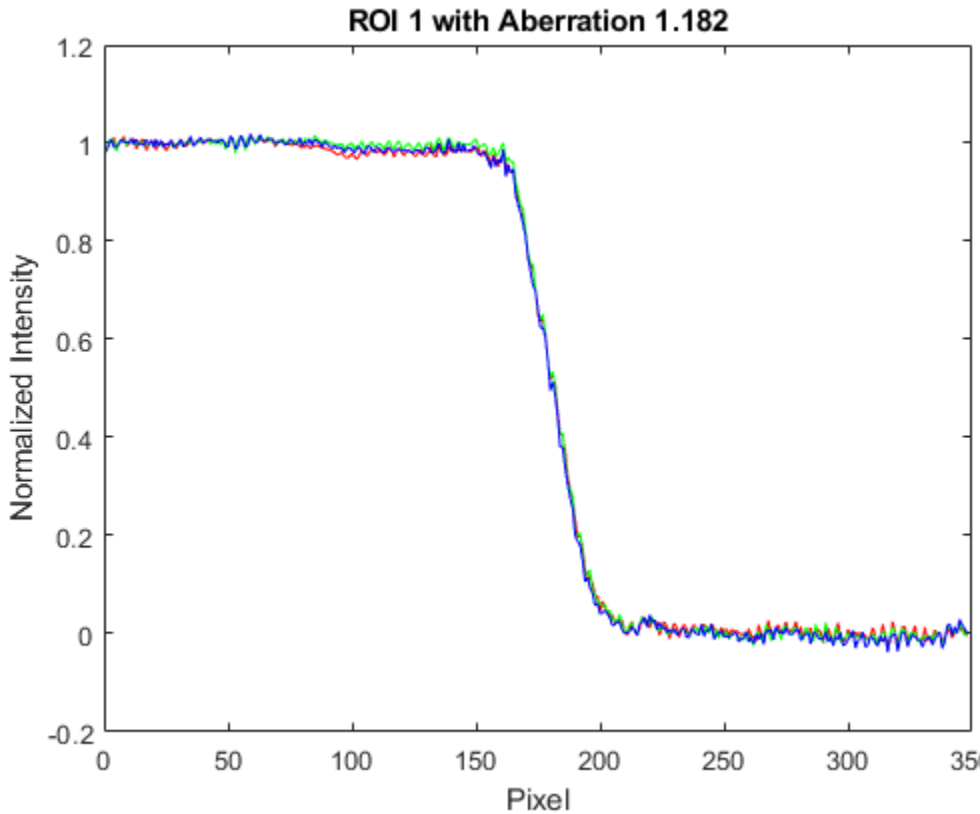
Measure chromatic aberration at all slanted edge ROIs.

```
chTable = measureChromaticAberration(chart);
```

Plot the normalized intensity profile of the three color channels in the first ROI. Store the normalized edge profile in a separate variable, `edgeProfile`, for clarity.

```
roi_index = 1;
edgeProfile = chTable.normalizedEdgeProfile{roi_index};

figure
p = length(edgeProfile.normalizedEdgeProfile_R);
```

```
plot(1:p,edgeProfile.normalizedEdgeProfile_R,'r', ...
     1:p,edgeProfile.normalizedEdgeProfile_G,'g', ...
     1:p,edgeProfile.normalizedEdgeProfile_B,'b')
xlabel('Pixel')
ylabel('Normalized Intensity')
title(['ROI ' num2str(1) ' with Aberration ' num2str(chTable.aberration(1))])
```



The color channels have similar normalized intensity profiles, and not much color fringing is visible along the edge.

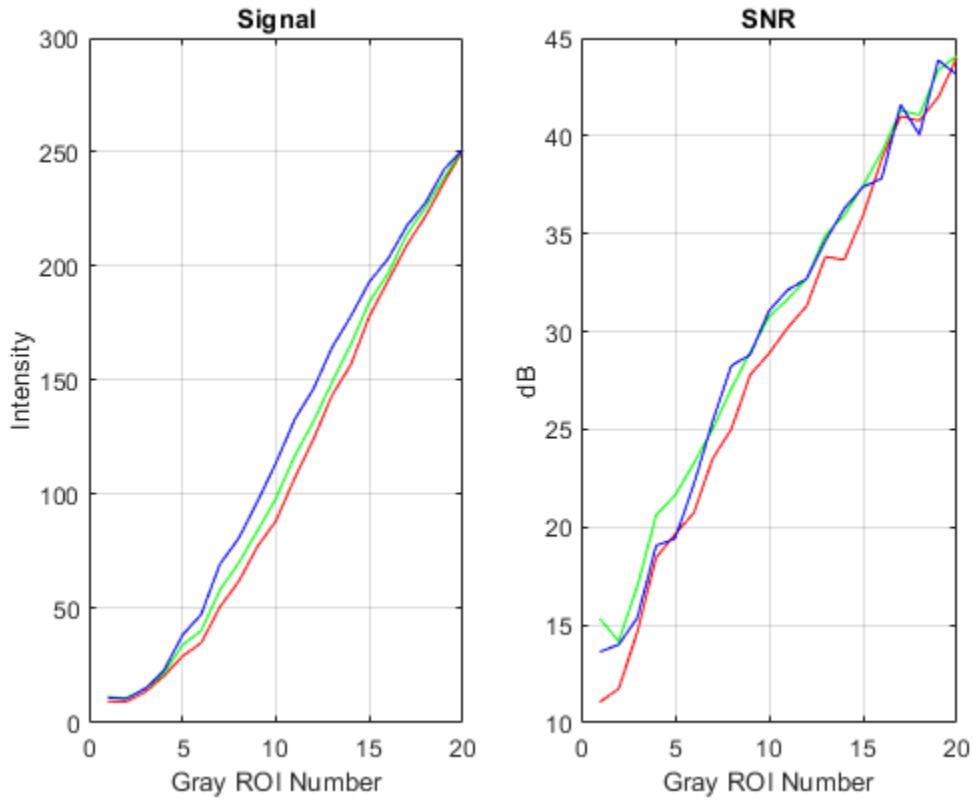
Measure Noise

Measure noise using the 20 gray patch ROIs.

```
noiseTable = measureNoise(chart);
```

Plot the average raw signal and the signal-to-noise ratio (SNR) in each grayscale ROI.

```
figure

subplot(1,2,1)
plot(noiseTable.ROI,noiseTable.MeanIntensity_R,'r', ...
    noiseTable.ROI,noiseTable.MeanIntensity_G,'g', ...
    noiseTable.ROI,noiseTable.MeanIntensity_B,'b')
title('Signal')
ylabel('Intensity')
xlabel('Gray ROI Number')
grid on

subplot(1,2,2)
plot(noiseTable.ROI,noiseTable.SNR_R,'r', ...
    noiseTable.ROI,noiseTable.SNR_G,'g', ...
    noiseTable.ROI,noiseTable.SNR_B,'b')
title('SNR')
ylabel('dB')
xlabel('Gray ROI Number')
grid on
```



Estimate Illuminant

Estimate the scene illumination using the 20 gray patch ROIs. The illuminant has a stronger blue component a weaker red component, which is consistent with the blue tint of the test chart image.

```
illum = measureIlluminant(chart)
```

```
illum = 1×3

  110.9147  116.0008  123.2339
```

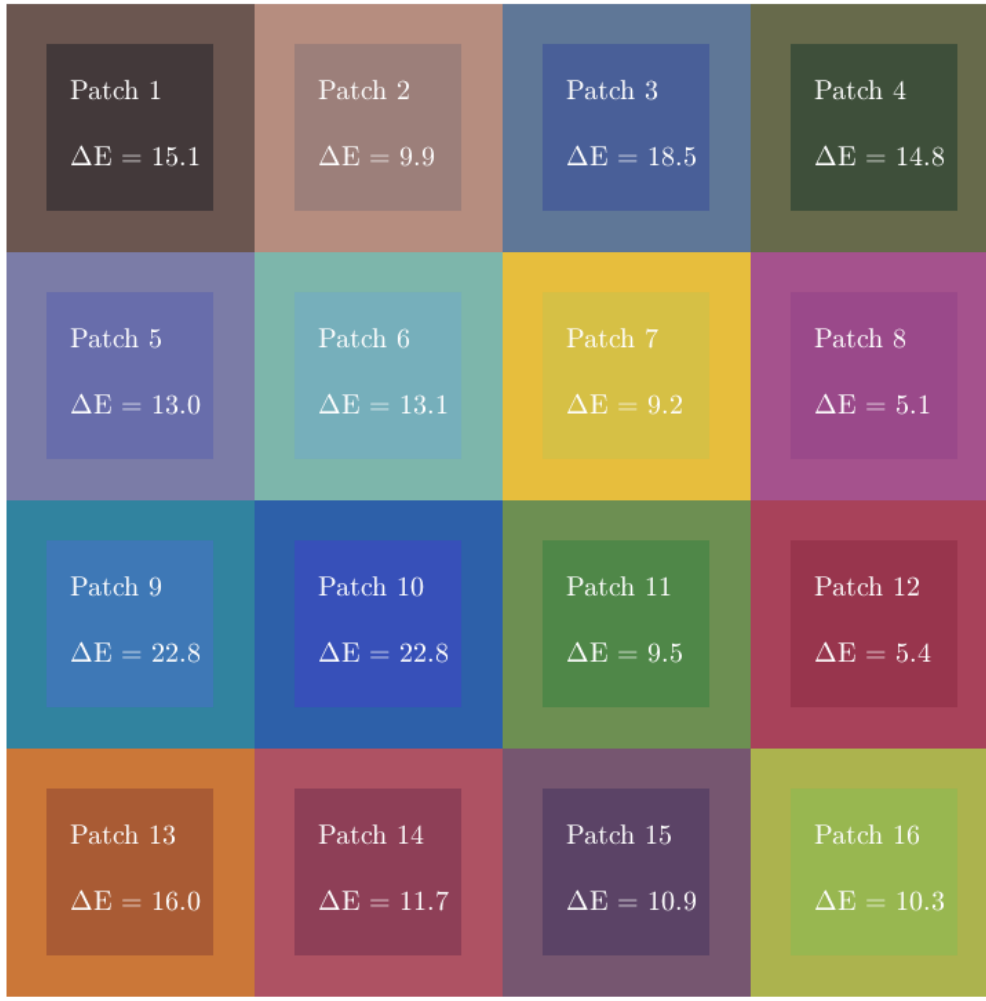
Measure Color Accuracy

Measure color accuracy using the 16 color patch ROIs.

```
[colorTable,ccm] = measureColor(chart);
```

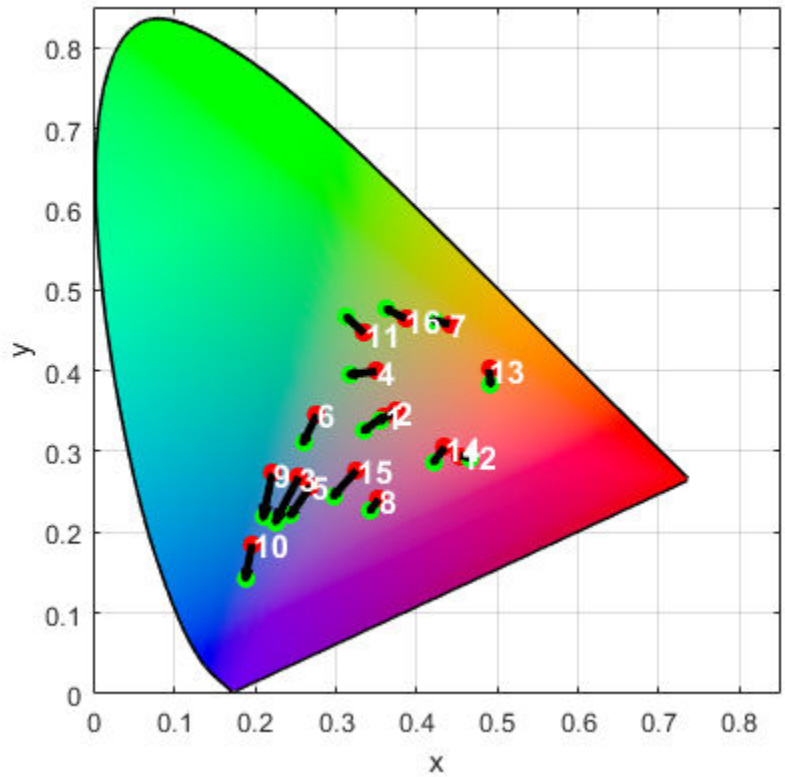
Display the average measured color and the expected color of the ROIs. Display the color accuracy measurement, `Delta_E`. The closer the `Delta_E` value is to 1, the less perceptible the color difference is. Typical values of `Delta_E` range from 3 to 6 for printing, and up to 20 in other commercial applications.

```
figure
displayColorPatch(colorTable)
```



Plot the measured and reference colors in the CIE 1976 L*a*b* color space on a chromaticity diagram. Red circles indicate the reference color. Green circles indicate the measured color of each color patch.

```
figure
plotChromaticity(colorTable)
```



You can use the color correction matrix, `ccm`, to color-correct the test chart images. For an example, see “Correct Colors Using Color Correction Matrix” on page 11-194.

See Also

`displayChart` | `esfrChart` | `measureChromaticAberration` | `measureColor` | `measureIlluminant` | `measureNoise` | `measureSharpness`

More About

- “Anatomy of an eSFR Chart” on page 11-174
- “Correct Colors Using Color Correction Matrix” on page 11-194

Correct Colors Using Color Correction Matrix

This example shows how to adjust the colors of an image to better match a standardized set of colors on an Imatest® edge spatial frequency response (eSFR) test chart.

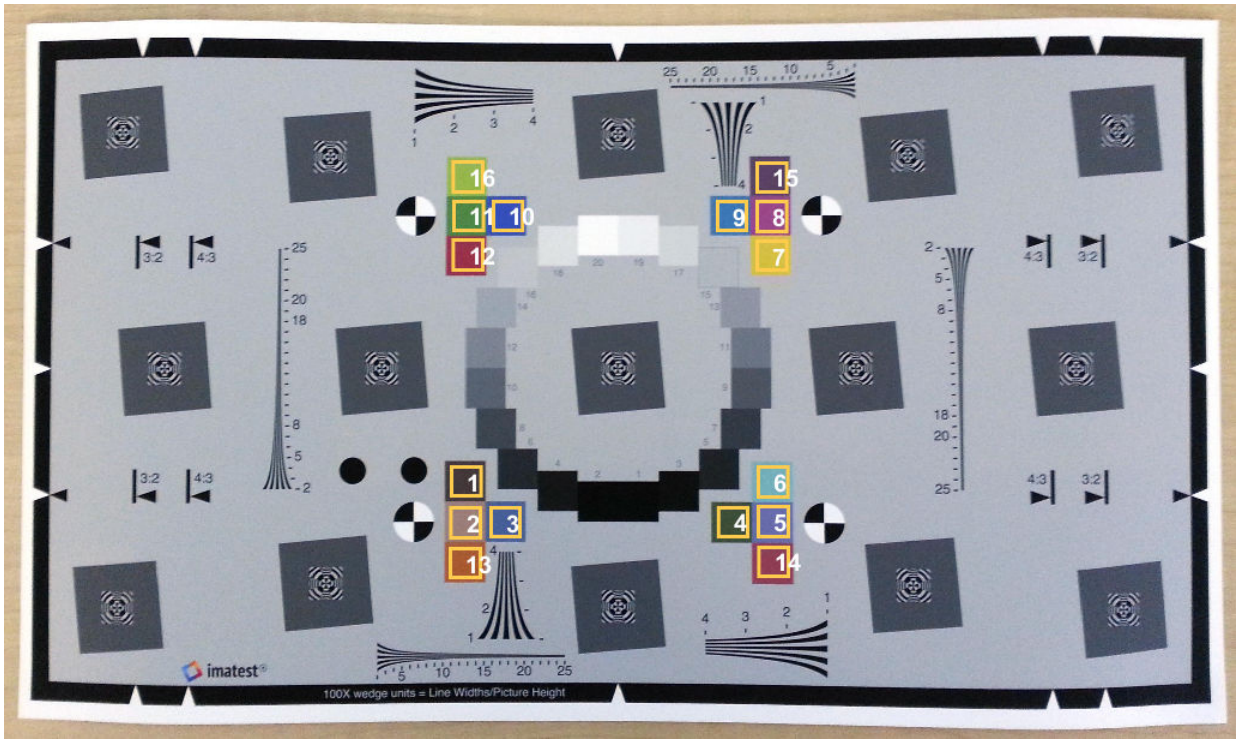
Obtain Color Correction Matrix from the Test Chart Image

Read an image of a test chart.

```
I = imread('eSFRTestImage.jpg');
```

Create an `esfrChart` object that stores information about the test chart. Display the chart, highlighting the 16 color patches. The image has a blue tint.

```
chart = esfrChart(I);
displayChart(chart,'displayEdgeROIs',false, ...
    'displayGrayROIs',false,'displayRegistrationPoints',false)
```

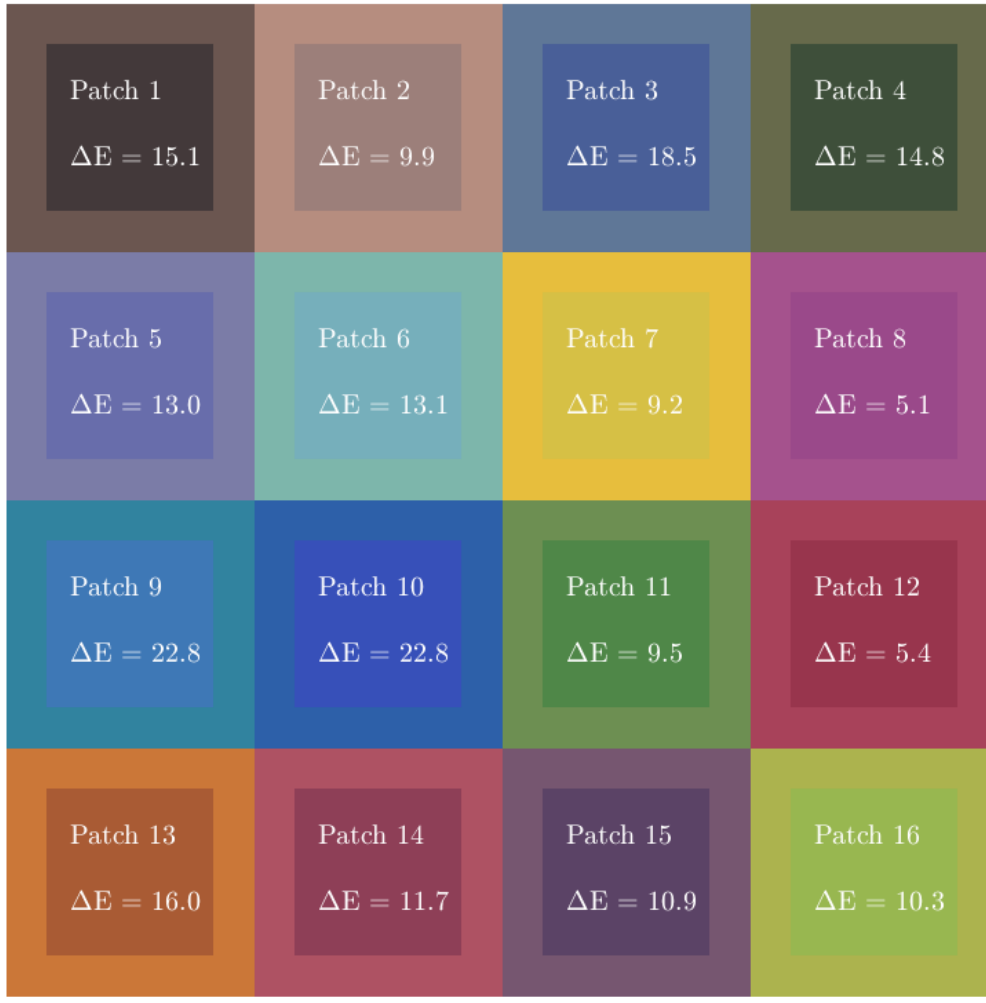


Measure the color accuracy of the 16 color patches by using the `measureColor` function. The function also returns the color correction matrix that is used to perform the color correction.

```
[colorTable,ccm] = measureColor(chart);
```

Compare the measured and reference colors on a color patch diagram. The closer the `Delta_E` value is to 1, the less perceptible the color difference is.

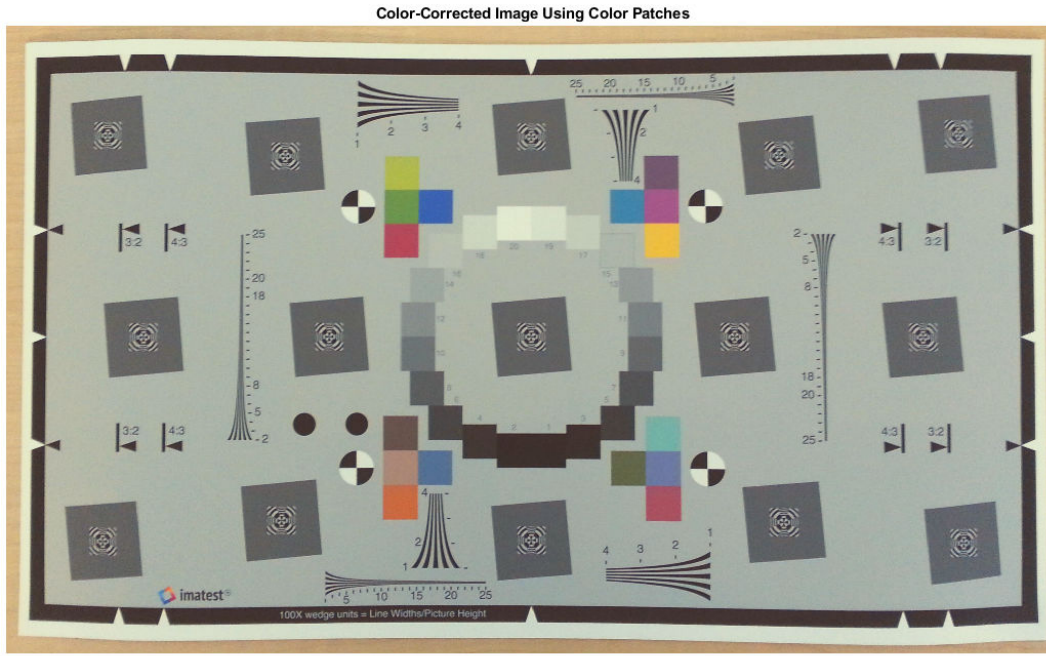
```
displayColorPatch(colorTable)
```



Color-Correct the Test Chart Image

Color-correct the original test chart image and display the result.

```
I_cc = imapplymatrix(ccm(1:3,:)',I,ccm(4,:));
imshow(I_cc)
title('Color-Corrected Image Using Color Patches')
```

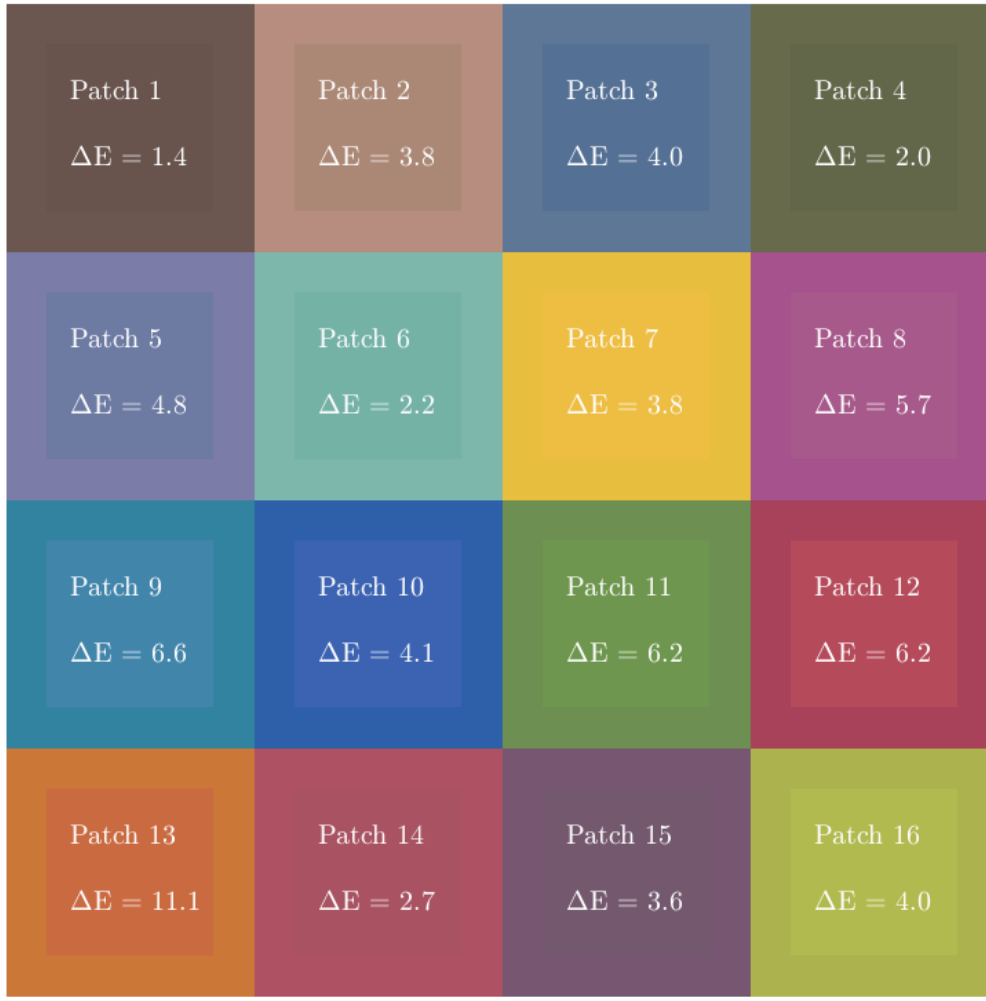


Create an `esfrChart` object that stores information about the color-corrected test chart. Measure the color accuracy of the 16 color-corrected color patches.

```
chart_cc = esfrChart(I_cc);
colorTable_cc = measureColor(chart_cc);
```

Compare the corrected and reference colors on a color patch diagram. The measured color errors, `delta_E`, are smaller for the color-corrected image than for the original image. Therefore, the colors in this image better agree with the reference colors. However, the chart now has an overall yellow tint and the contrast of the image has decreased.

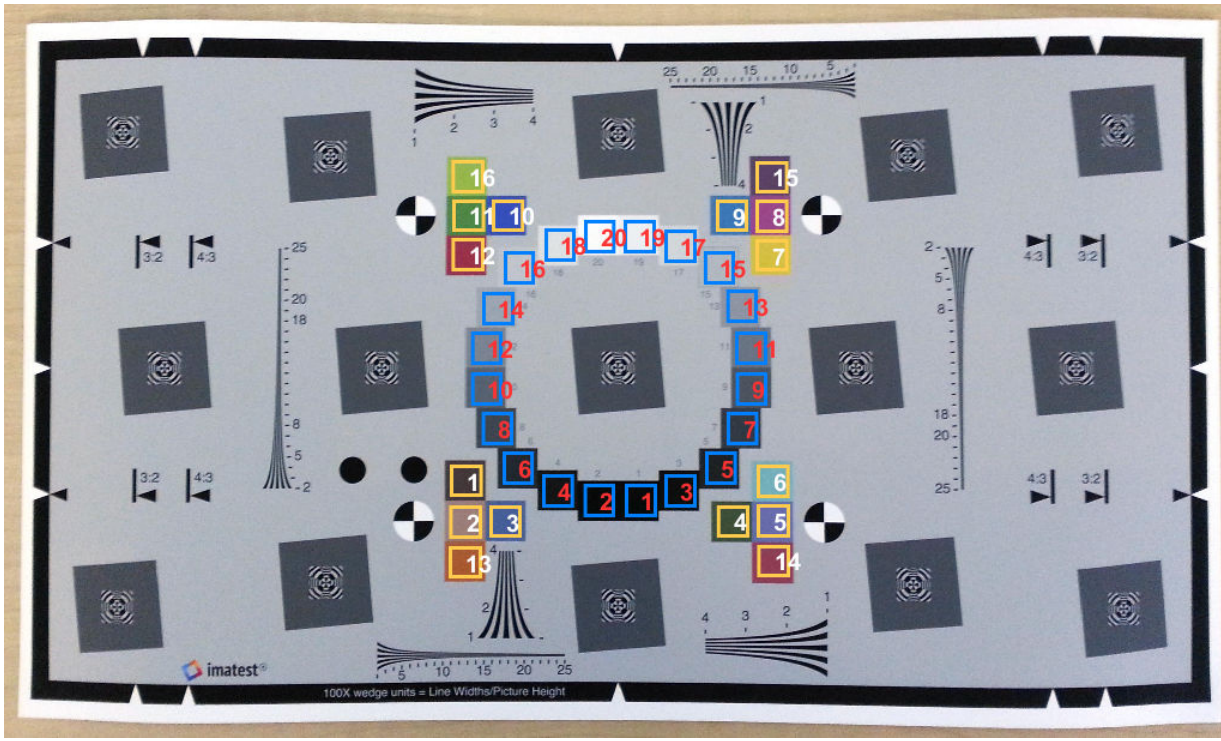
```
displayColorPatch(colorTable_cc)
```


Patch 1
ΔE = 1.4
Patch 2
ΔE = 3.8
Patch 3
ΔE = 4.0
Patch 4
ΔE = 2.0
Patch 5
ΔE = 4.8
Patch 6
ΔE = 2.2
Patch 7
ΔE = 3.8
Patch 8
ΔE = 5.7
Patch 9
ΔE = 6.6
Patch 10
ΔE = 4.1
Patch 11
ΔE = 6.2
Patch 12
ΔE = 6.2
Patch 13
ΔE = 11.1
Patch 14
ΔE = 2.7
Patch 15
ΔE = 3.6
Patch 16
ΔE = 4.0

Improve Color Correction Using Gray Patches

You can improve the color correction by including the gray patches as well as the color patches in the least squares fit. Display the original chart, highlighting the 20 gray patches and 16 color patches.

```
displayChart(chart,'displayEdgeROIs',false, ...
    'displayRegistrationPoints',false)
```



Get the reference L*a*b* values of the color and grayscale patches, which are stored in the `ReferenceColorLab` and `ReferenceGrayLab` properties of the eSFR chart object. Convert these values to the RGB color space.

```
referenceLab = [chart.ReferenceColorLab; chart.ReferenceGrayLab];
referenceRGB = lab2rgb(referenceLab,'outputtype','uint8');
```

Measure the mean gray value in each of the 20 gray patches by using the `measureNoise` function.

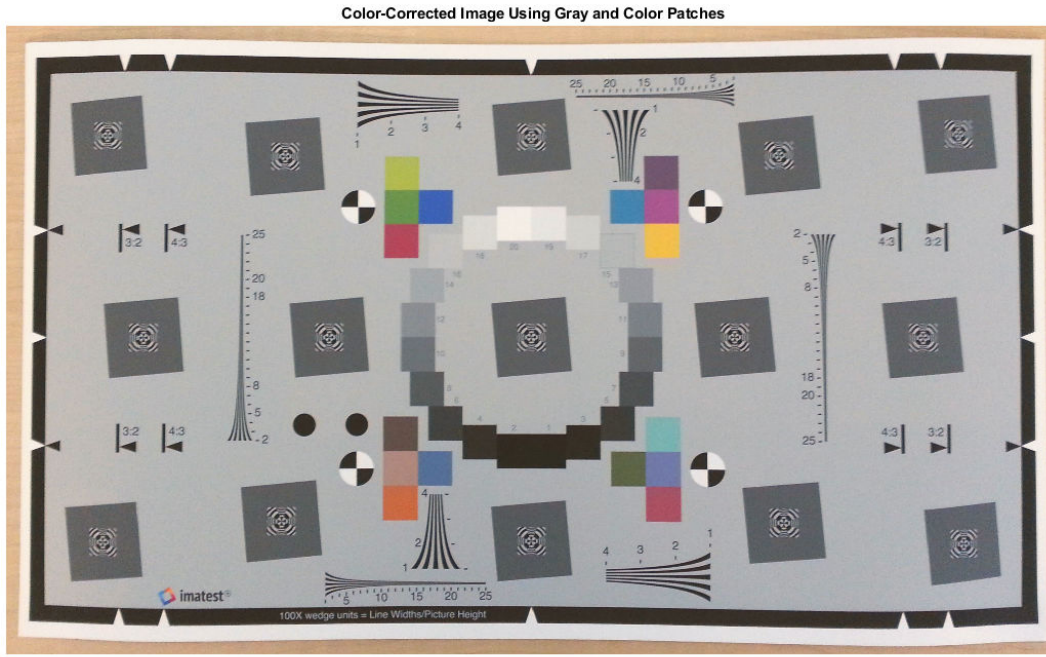
```
noiseTable = measureNoise(chart);
measuredGrayRGB = [noiseTable.MeanIntensity_R, ...
    noiseTable.MeanIntensity_G, ...
    noiseTable.MeanIntensity_B];
```

Concatenate all measured RGB color values of the color and grayscale patches.

```
measuredColorRGB = [colorTable.Measured_R, ...
    colorTable.Measured_G, ...
    colorTable.Measured_B];
measuredRGB = [measuredColorRGB; measuredGrayRGB];
ccm_cc = double([measuredRGB ones(36,1)]) \ double(referenceRGB);
```

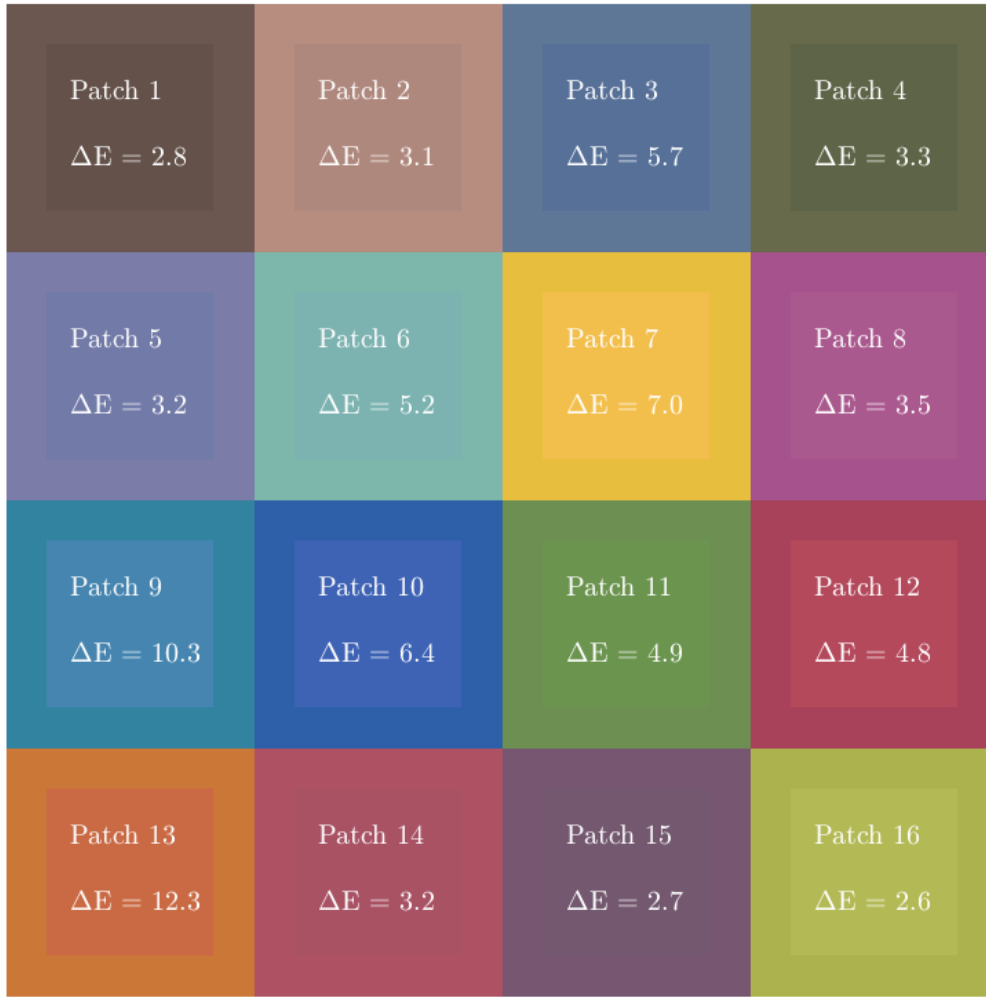
Perform the color correction and display the result. The chart no longer has a yellow tint and the overall appearance of the chart has improved.

```
I_cc2 = imapplymatrix(ccm_cc(1:3,:)',I,ccm_cc(4,:)');
imshow(I_cc2)
title('Color-Corrected Image Using Gray and Color Patches')
```



Compare the corrected and reference colors on a color patch diagram. Some of the measured color errors have decreased, while others have increased.

```
chart_cc2 = esfrChart(I_cc2);
colorTable_cc2 = measureColor(chart_cc2);
displayColorPatch(colorTable_cc2)
```



See Also

`displayColorPatch` | `esfrChart` | `measureColor` | `plotChromaticity`

More About

- “Evaluate Quality Metrics on eSFR Test Chart” on page 11-179

Install Sample Data Using the Add-Ons Explorer

Image Processing Toolbox Sample Image Data file contains the sample 3-D volumetric data used by the example “Segment Lungs from 3-D Chest Scan and Calculate Lung Volume” on page 13-146.

To install the support file, select **Get Add-ons** from the **Add-ons** drop-down menu from the MATLAB desktop. The support file is in the **MathWorks Features** section. You must have write privileges for the installation folder.

ROI-Based Processing

- “Create a Binary Mask” on page 12-2
- “ROI Creation Overview” on page 12-5
- “Overview of ROI Filtering” on page 12-9
- “Sharpen Region of Interest in an Image” on page 12-10
- “Apply Custom Filter to Region of Interest in Image” on page 12-13
- “Fill Region of Interest in an Image” on page 12-16
- “Image Region Properties” on page 12-20
- “Calculate Region Properties Using Image Region Analyzer” on page 12-21
- “Filter Images on Region Properties Using Image Region Analyzer App” on page 12-29
- “Create Image Comparison Tool Using ROIs” on page 12-36
- “Use Freehand ROIs to Refine Segmentation Masks” on page 12-44
- “Rotate Rectangle ROI to interact with imrotate” on page 12-50
- “Subsample or Simplify a Freehand ROI” on page 12-56
- “Measure Distances in an Image” on page 12-67
- “Use Polyline to Create An Angle Measurement Tool” on page 12-75
- “Create Freehand ROI Editing Tool” on page 12-78

Create a Binary Mask

Image Processing Toolbox supports four methods to generate a binary mask. The binary mask defines a region of interest (ROI) of the original image. Mask pixel values of `1` indicate the image pixel belongs to the ROI. Mask pixel values of `0` indicate the image pixel is part of the background.

Any binary image can be used as a mask, provided that the binary image is the same size as the image being filtered.

Create a Binary Mask from a Grayscale Image

You can create a mask from a grayscale image by classifying each pixel as belonging to either the region of interest or the background. 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)
```

Create Binary Mask Using an ROI Object

You can use the `createMask` function 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 `1`s inside the ROI and `0`s everywhere else.

This example illustrates how to use the `createMask` function:

```
img = imread('pout.tif');
h_im = 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.

Create Binary Mask Based on Color Values

You can use the `roicolor` function to define an ROI based on color or intensity range.

Create Binary Mask Without an Associated Image

You can use the `poly2mask` function to create a binary mask without having an associated image. 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);
imshow(BW)
```

ROI Creation Overview

The toolbox provides a set of classes to create regions of interest of various shapes, including circles, ellipses, polygons, lines, polylines, points, and rectangles. You can also draw freehand shapes and freehand shapes that snap to edges in the image. In this figure, you see a polygonal ROI drawn over an image.

ROI Creation Classes

The toolbox includes nine classes that support ROI creation. You use these classes to create ROIs of certain shapes on an axes. These classes have properties, methods, and events that provide flexibility to ROI creation. Using these classes, you can draw ROIs interactively, using the mouse, or programmatically by specifying information about the size and shape of the ROI. Once you create the ROI object, you can perform operations on the ROI such as creating a mask. You can also control aspects of its behavior and appearance and respond to events that occur in the ROI, such as mouse clicks and movement.

The basic workflow when using the ROI classes interactively: is to instantiate an object of the class and then call the `draw` method.

1. Instantiate an object of the class. You can specify many different properties when you create the object, such as `LineWidth` and `Color`.

   ```
   circROI = images.roi.Circle;
   ```

2. Call the `draw` method of the `Circle` object to start the interactive drawing of the circular ROI on the axes.

   ```
   draw(circROI)
   ```

When you create the ROI programmatically, you do not need to call the `draw` method. For more information, see the examples of ROI creation and look at the class for the shape you want to create, such as, `images.roi.Circle`.

ROI Creation Functions

As a convenience, the toolbox provides nine functions that let you draw these ROIs without directly calling the ROI classes. These functions create instances of the ROI classes.

The basic workflow when using these ROI convenience functions interactively is to call the function. The function instantiates the class and calls the draw method to start interactive placement of the ROI. You can specify the value many different properties when you create the object using convenience functions, such as `LineWidth` and `Color`.

See Also

Related Examples

- “Create Binary Mask Using an ROI Object” on page 12-2

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`.

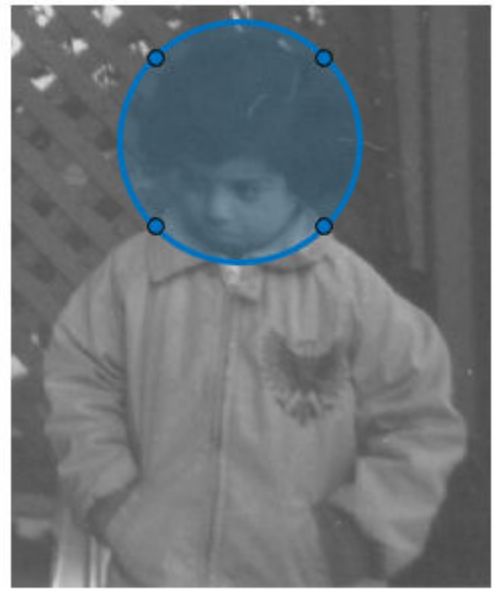
Sharpen Region of Interest in an Image

Read a grayscale image into the workspace.

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

Draw a region of interest over the image to specify the area you want to filter. Use the `drawcircle` function to create the region of interest, specifying the center of the circle and the radius of the circle. Alternatively, if you want to draw the circle interactively, then do not specify the center or radius of the circle.

```
hax = drawcircle(gca,'Center',[115 69],'Radius', 60);
```



Create the mask using the `createMask` function and specifying the ROI.

```
mask = createMask(hax);
```

Define the function you want to use as a filter. This function, named `f`, passes the input image `x` to the `imsharpen` function and specifies the strength of the sharpening effect by using the `'Amount'` name-value pair argument.

```
f = @(x)imsharpen(x,'Amount',3)
```

```
f = function_handle with value:
    @(x)imsharpen(x,'Amount',3)
```

Filter the ROI using the `roifilt2` function and specifying the image, mask, and filtering function.

```
J = roifilt2(I,mask,f);
```

Display the result.

```
imshow(J)
```

See Also

`createMask` | `drawcircle` | `images.roi.Circle` | `imsharpen` | `roifilt2`

Apply Custom Filter to Region of Interest in Image

This example shows how to filter a region of interest (ROI), using the `roifilt2` function to specify the filter. `roifilt2` enables you to specify your own function to operate on the ROI. This example uses the `imadjust` function to lighten parts of an image.

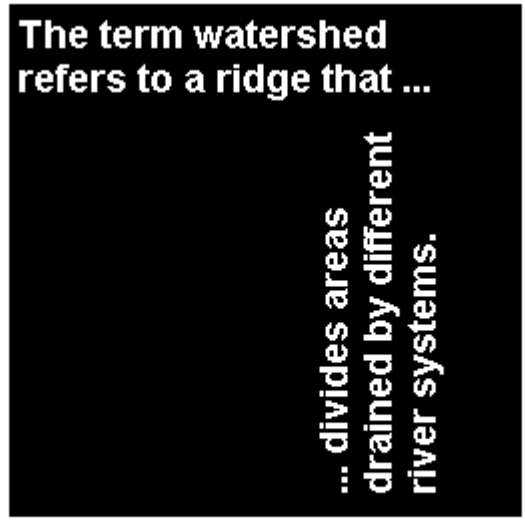
Read an image into the workspace and display it.

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

Create the mask image. This example uses a binary image of text as the mask image. All the 1-valued pixels define the regions of interest. The example crops the image because a mask image must be the same size as the image to be filtered.

```
BW = imread('text.png');
mask = BW(1:256,1:256);
```

```
figure
imshow(mask)
```



Create the function you want to use as a filter.

```
f = @(x) imadjust(x,[],[],0.3);
```

Filter the ROI, specifying the image to be filtered, the mask that defines the ROI, and the filter that you want to use.

```
I2 = roifilt2(I,mask,f);
```

Display the result.

```
figure
imshow(I2)
```

See Also

imadjust | roifilt2

Fill Region of Interest in an Image

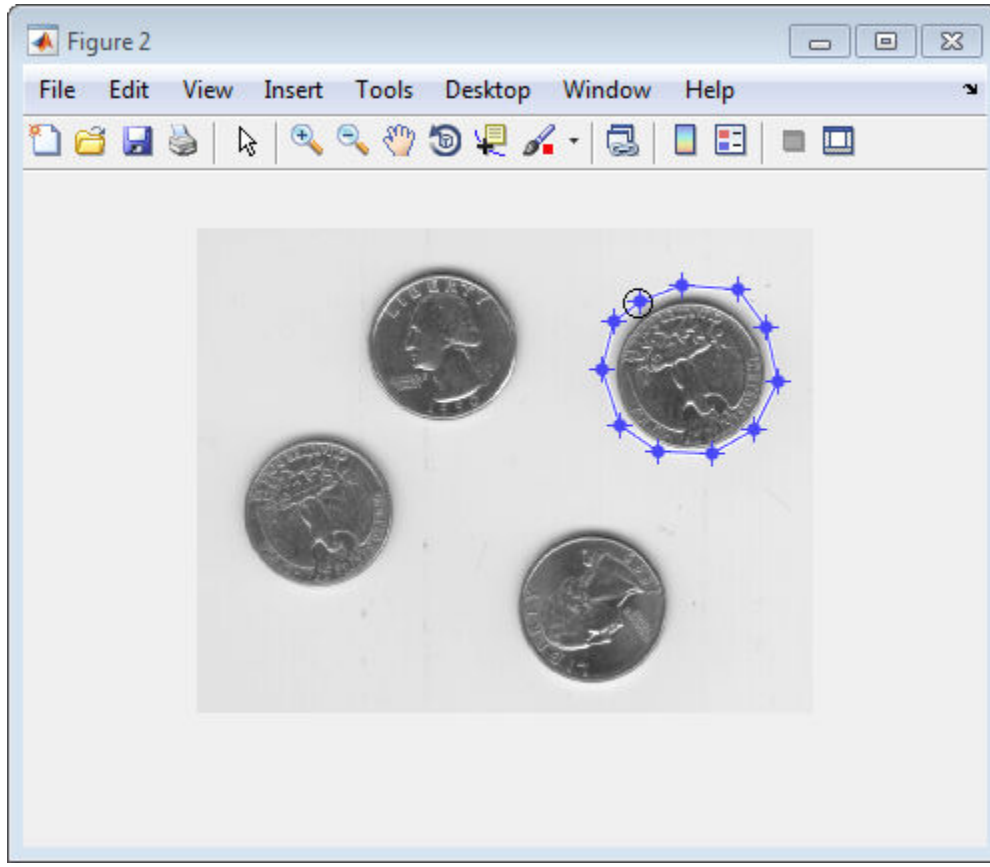
This example shows how to use `regionfill` to fill a region of interest (ROI) in an image. The example uses the `roipoly` function to define the region of interest interactively with the mouse. `regionfill` smoothly interpolates inward into the region from the pixel values on the boundary of the polygon. You can use this function for image editing, including removal of extraneous details or artifacts. The filling process replaces values in the region with values that blend with the background.

Read an image into the MATLAB workspace and display it.

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

Create a mask image to specify the region of interest (ROI) you want to fill. Use the `roipoly` function to specify the region interactively. Call `roipoly` and 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.

```
mask = roipoly(I);
```



Double-click to finish defining the region. `roipoly` creates a binary image with the region filled with `1`-valued pixels.

Display the mask image.

```
figure
imshow(mask)
```

Fill the region, using `regionfill`, specifying the image to be filled and the mask image as inputs. Display the result. Note the image contains one less coin.

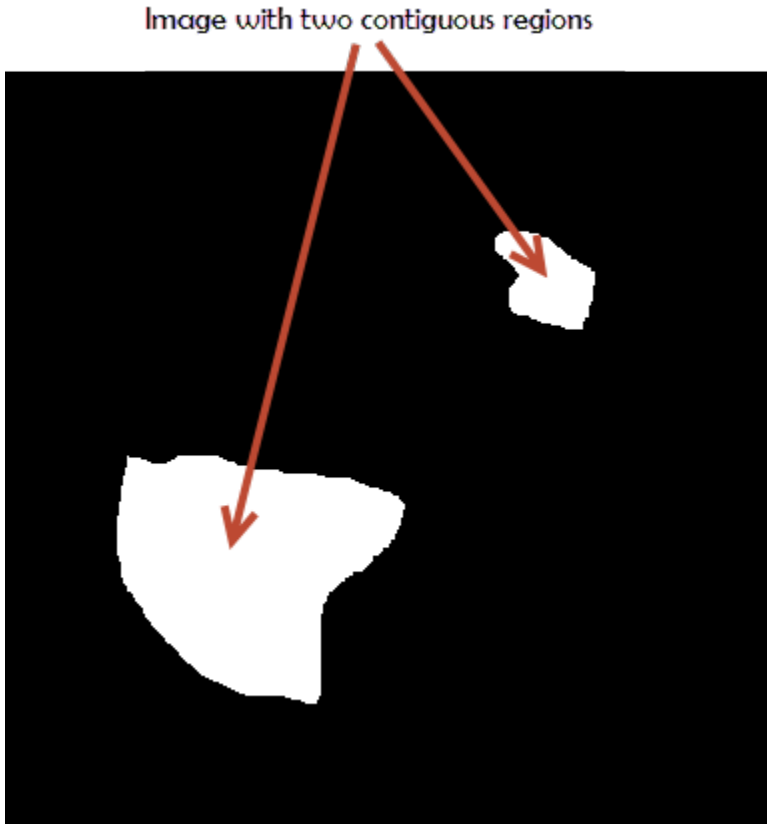
```
J = regionfill(I,mask);
figure
imshow(J)
```

See Also

`createMask` | `drawpolygon` | `regionfill` | `roipoly`

Image Region Properties

Image regions, also called objects, connected components, or blobs, can be contiguous or discontiguous. 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

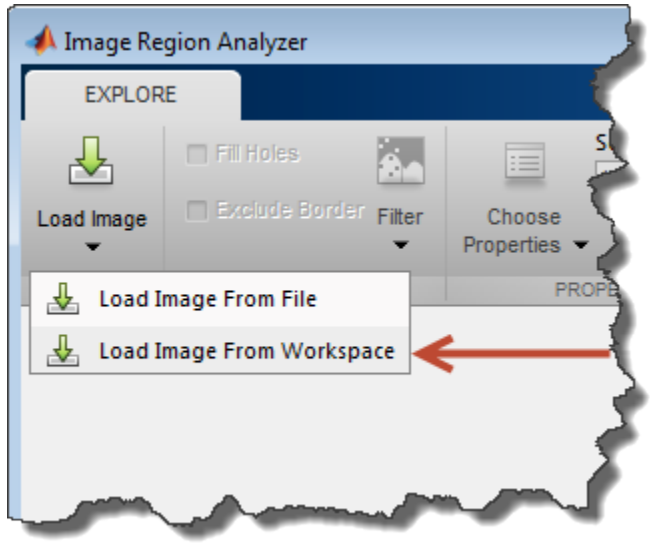
This example shows how to calculate the properties of regions in binary images using the Image Region Analyzer app. This example finds the 10 largest regions in the image as measured by their area.

Read a binary image into the MATLAB workspace.

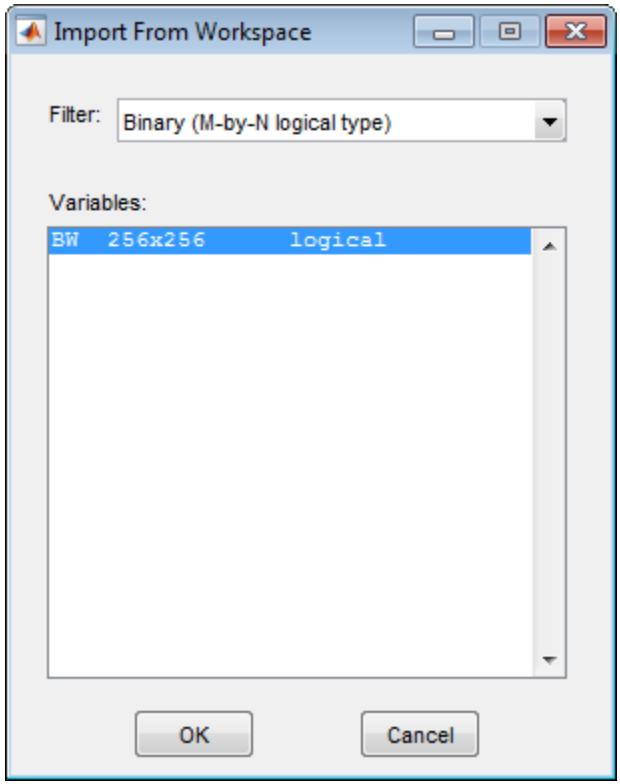
```
BW = imread('text.png');
```

Open the Image Region Analyzer app from the MATLAB Toolstrip. On the Apps tab, in the Image Processing and Computer Vision group, click .

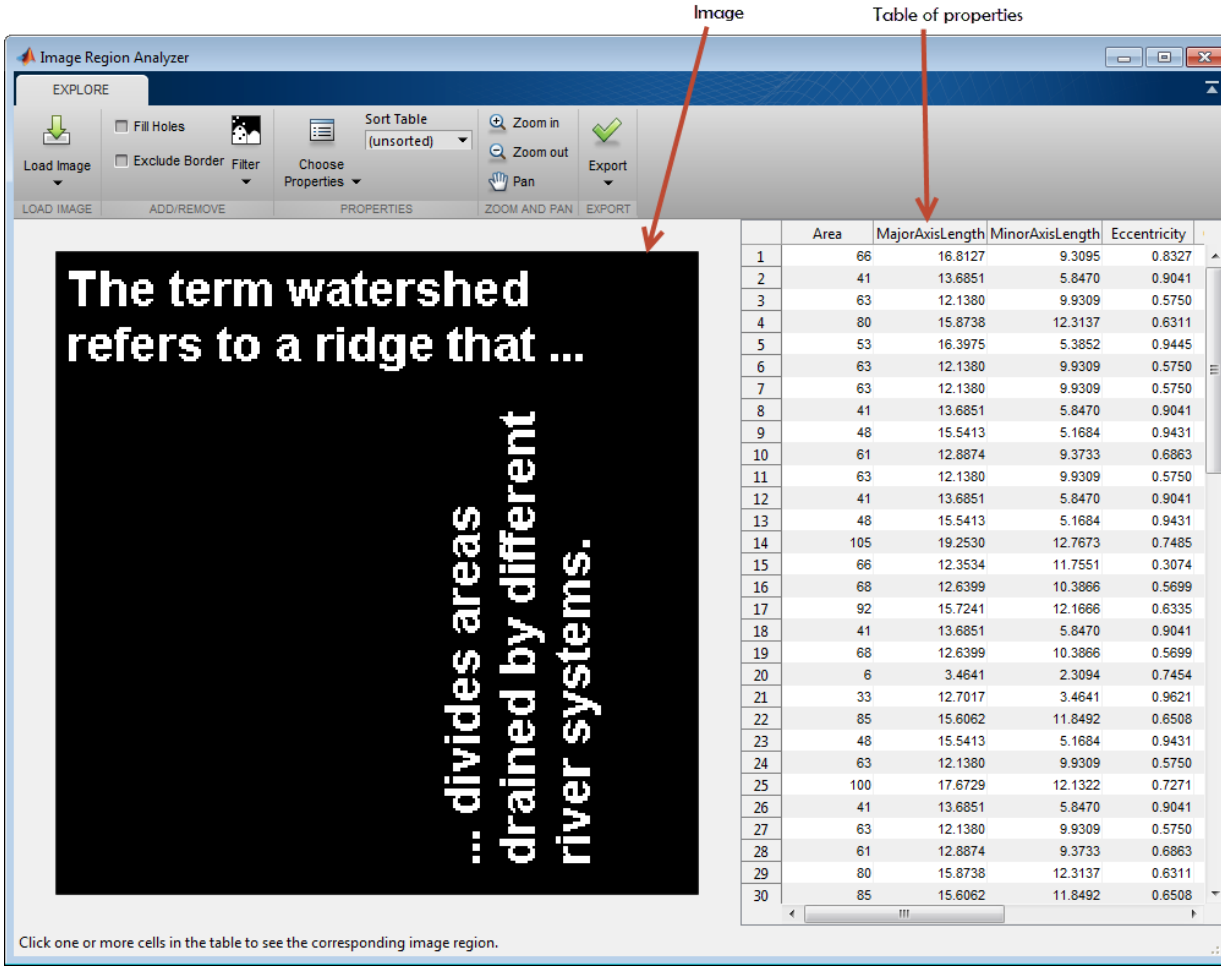
In the Image Region Analyzer app, click **Load Image**, and then select **Load Image from Workspace**, since you have already read the image into the workspace. You can also open the app from the command line using the `imageRegionAnalyzer` command.



In the Import From Workspace dialog box, select the image you read into the workspace, and click **OK**.

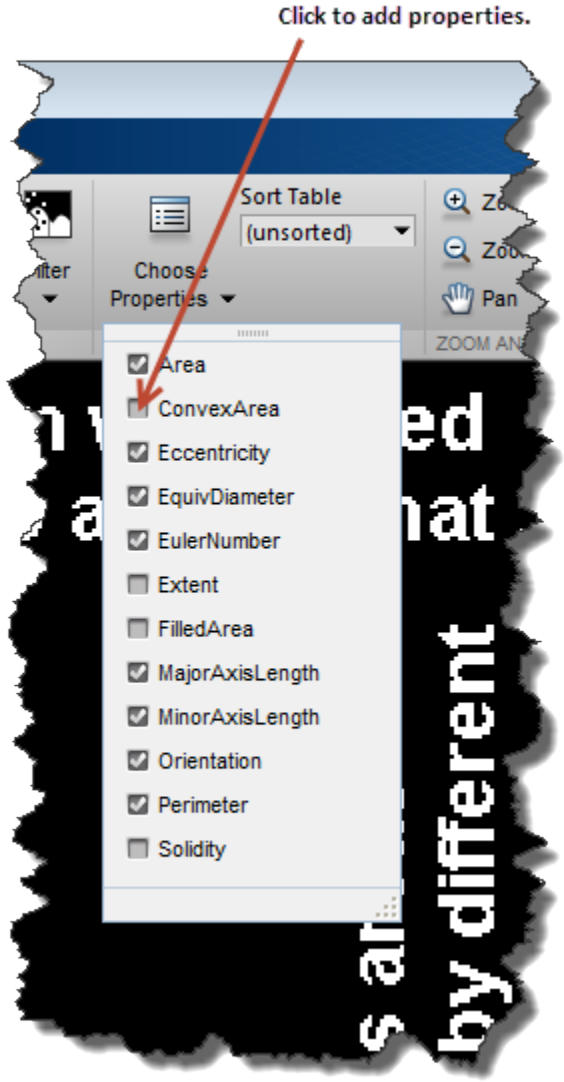


The Image Region Analyzer app displays the image you selected next to a table where every row is a region identified in the image and every column is a property of that region, such as the area of the region, perimeter, and orientation. (The Image Region Analyzer app uses `regionprops` to identify regions in the image and calculate properties of these regions.)

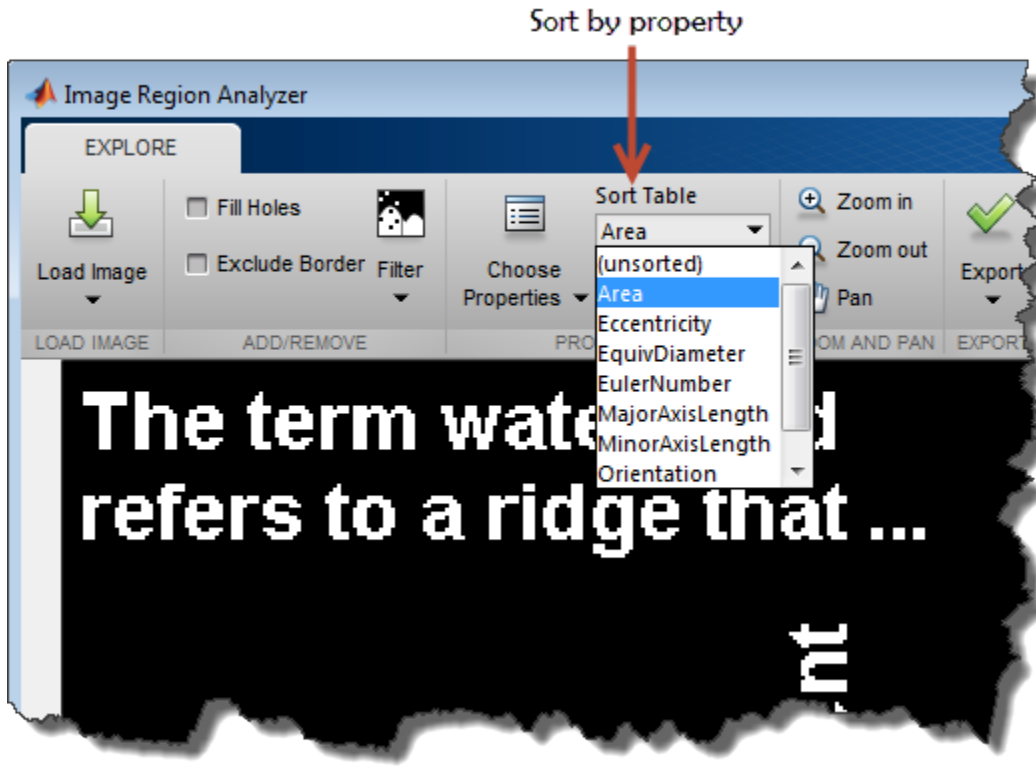


	Area	MajorAxisLength	MinorAxisLength	Eccentricity
1	66	16.8127	9.3095	0.8327
2	41	13.6851	5.8470	0.9041
3	63	12.1380	9.9309	0.5750
4	80	15.8738	12.3137	0.6311
5	53	16.3975	5.3852	0.9445
6	63	12.1380	9.9309	0.5750
7	63	12.1380	9.9309	0.5750
8	41	13.6851	5.8470	0.9041
9	48	15.5413	5.1684	0.9431
10	61	12.8874	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

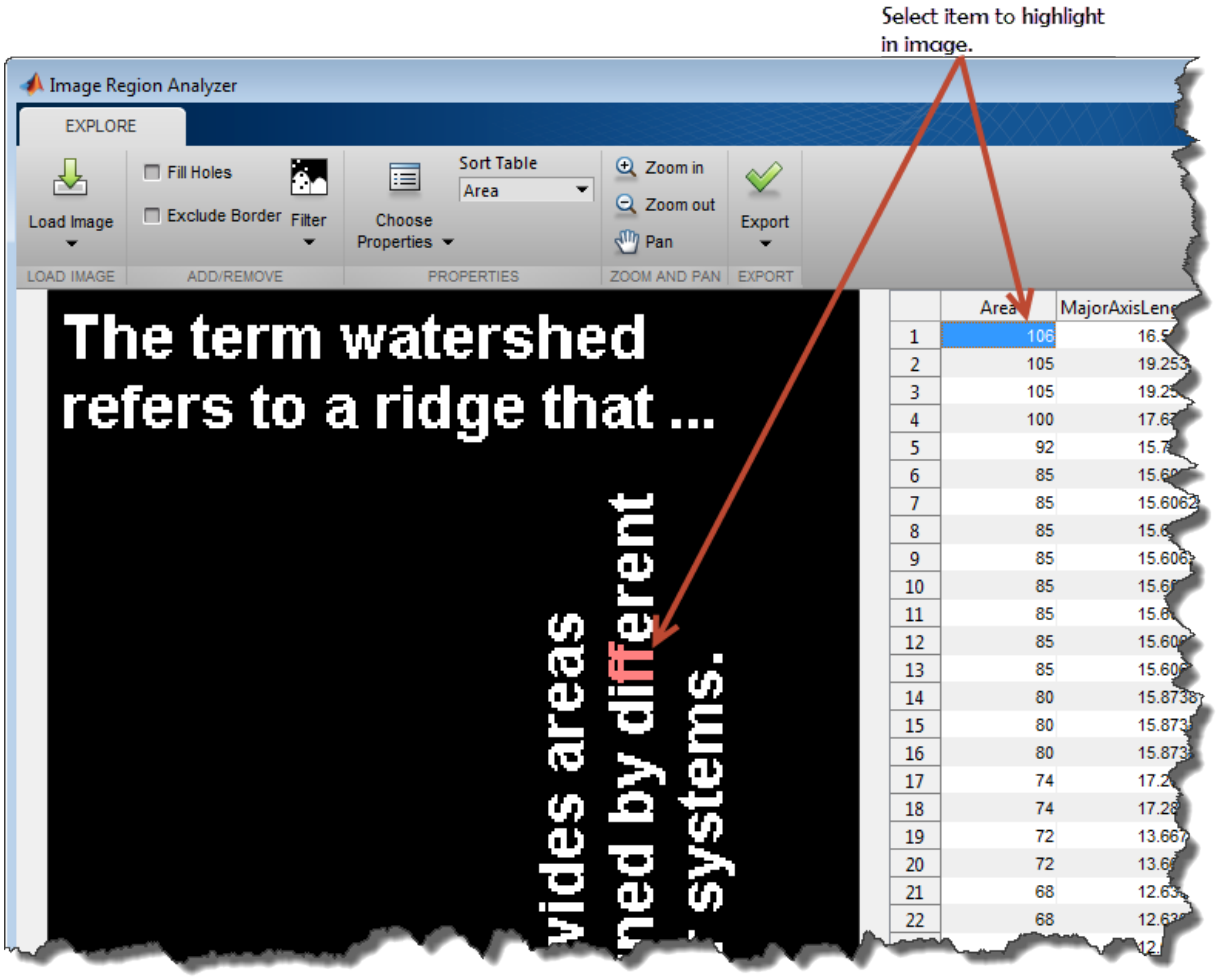
The app calculates more properties than are displayed initially. To view more properties in the table, click **Choose Properties** and select the properties you want to view. Properties displayed are marked with a check. The app updates the table automatically, adding a new column to the table.



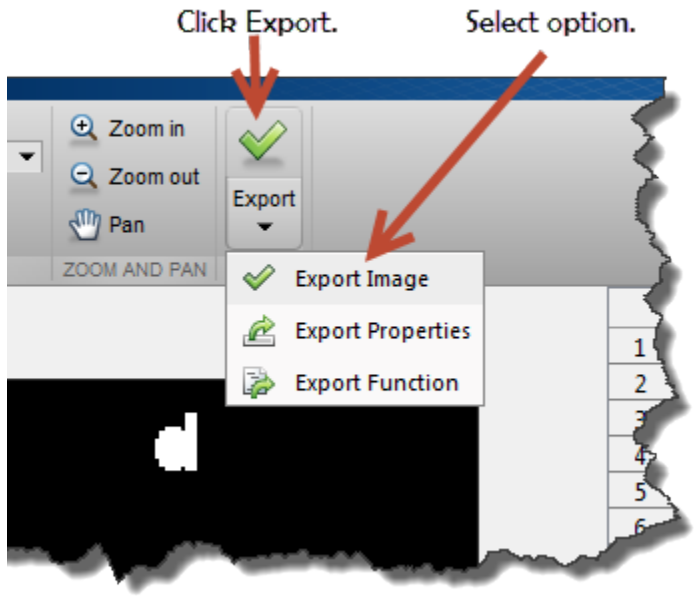
To explore the image, sort the information in the table. For example, if you sort on the Area property, the table lists the regions in order by size. Click the **Sort Table** button in the Properties group and select the property you want to sort on.



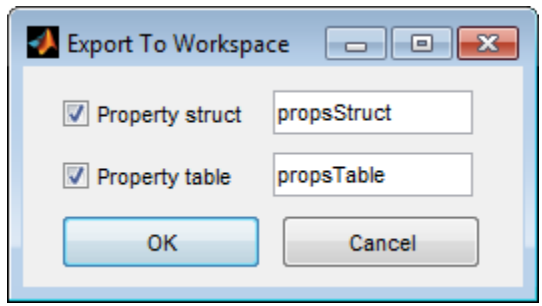
To view the region in the image with the largest area, click the item in the table. The app highlights the corresponding region in the image.



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**.



View the results returned in an array of MATLAB structures, named `propsStruct`. The following code displays the first structure in the array.

```
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
```

View the results returned in a MATLAB table, named `propsTable`. The following code displays the first four elements of the first table row.

```
propsTable(1,1:4)
```

```
ans =

    Area    MajorAxisLength    MinorAxisLength    Eccentricity
    ____    _______________    _______________    ____________

    106     16.598             12.9               0.62925
```

Filter Images on Region Properties Using Image Region Analyzer App

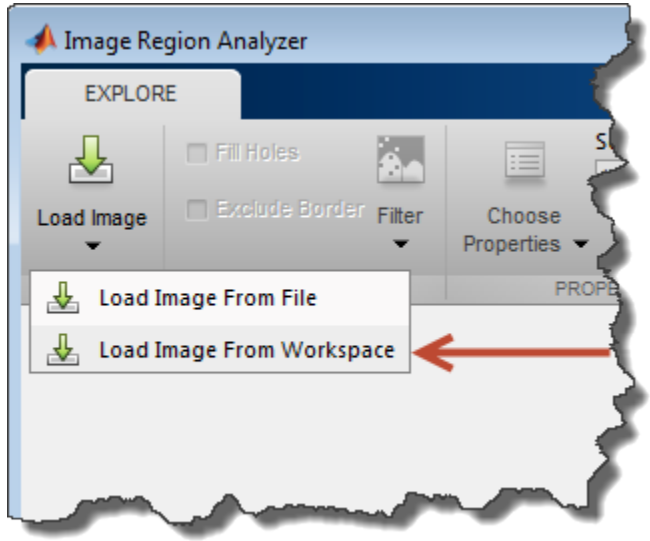
This example shows how to create a new binary image by filtering an existing binary image based on the properties of regions in the image.

Read a binary image into the MATLAB workspace.

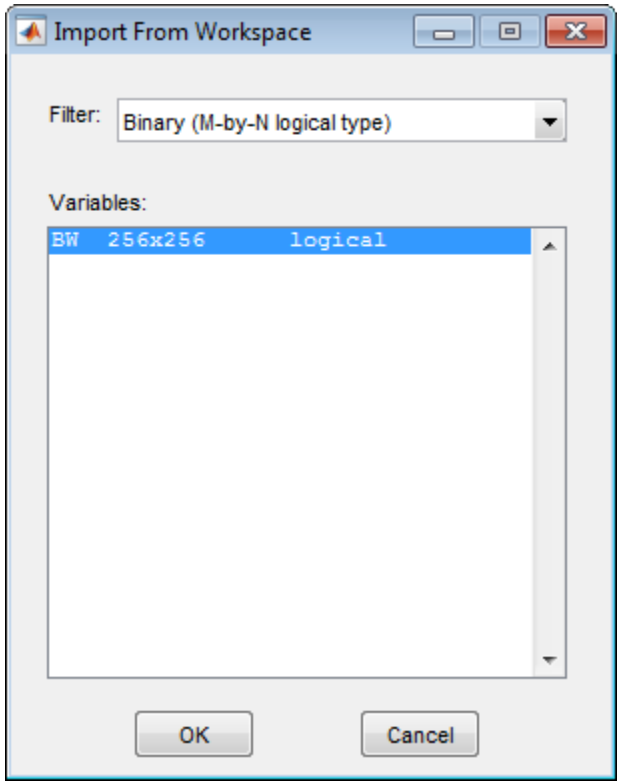
```
BW = imread('text.png');
```

Open the Image Region Analyzer app from the MATLAB Toolstrip. On the Apps tab, in the Image Processing and Computer Vision group, click .

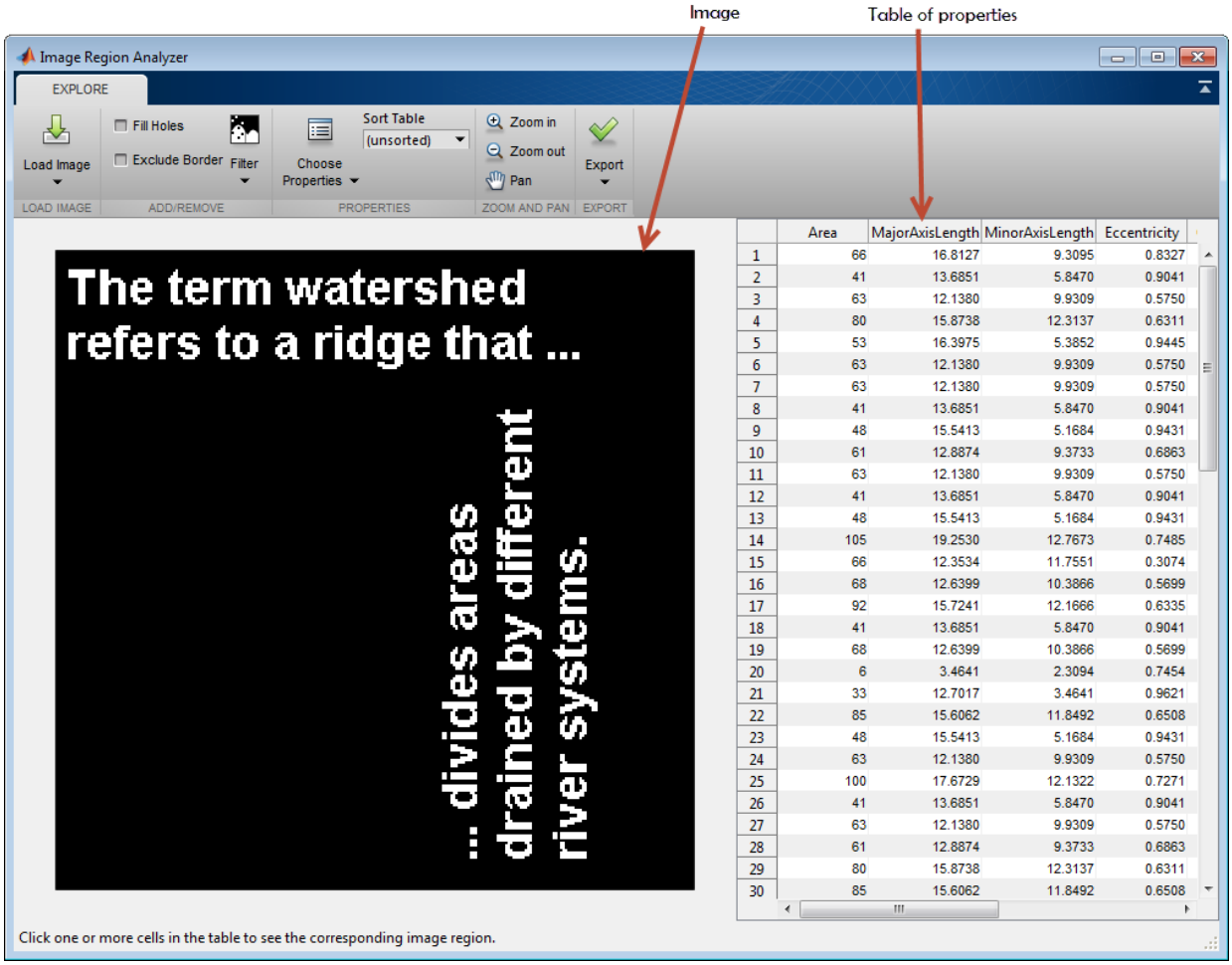
In the Image Region Analyzer app, click **Load Image**, and then select **Load Image from Workspace**, since you have already read the image into the workspace.



In the Import From Workspace dialog box, select the image you read into the workspace, and click **OK**.

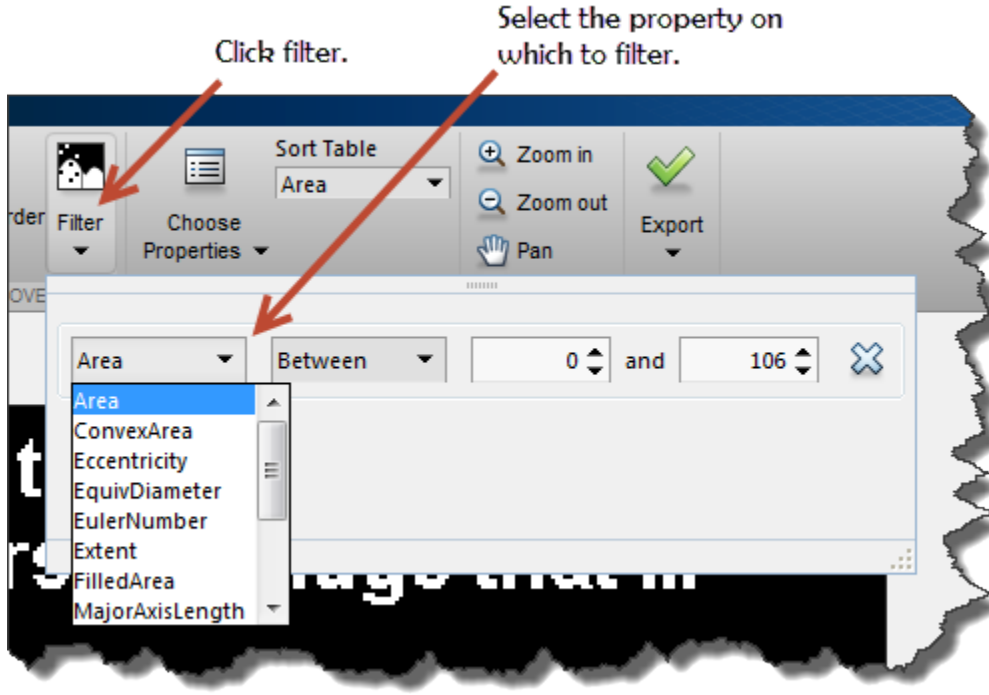


The Image Region Analyzer app displays the image you selected next to a table where every row is a region identified in the image and every column is a property of that region, such as the area, perimeter, and orientation. The Image Region Analyzer app uses `regionprops` to identify regions in the image and calculate properties of these regions.

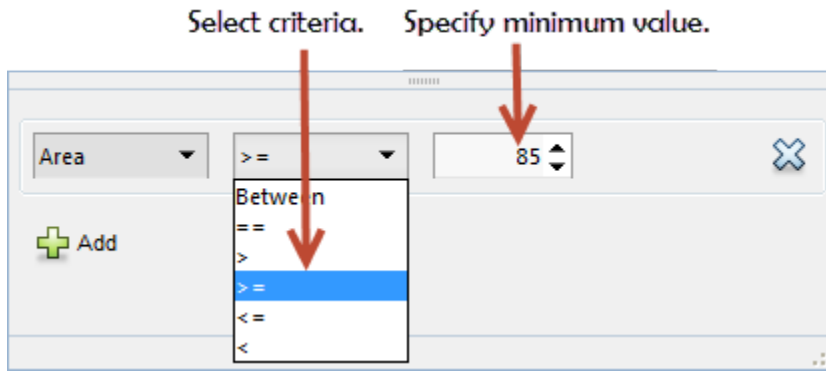


	Area	MajorAxisLength	MinorAxisLength	Eccentricity
1	66	16.8127	9.3095	0.8327
2	41	13.6851	5.8470	0.9041
3	63	12.1380	9.9309	0.5750
4	80	15.8738	12.3137	0.6311
5	53	16.3975	5.3852	0.9445
6	63	12.1380	9.9309	0.5750
7	63	12.1380	9.9309	0.5750
8	41	13.6851	5.8470	0.9041
9	48	15.5413	5.1684	0.9431
10	61	12.8874	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

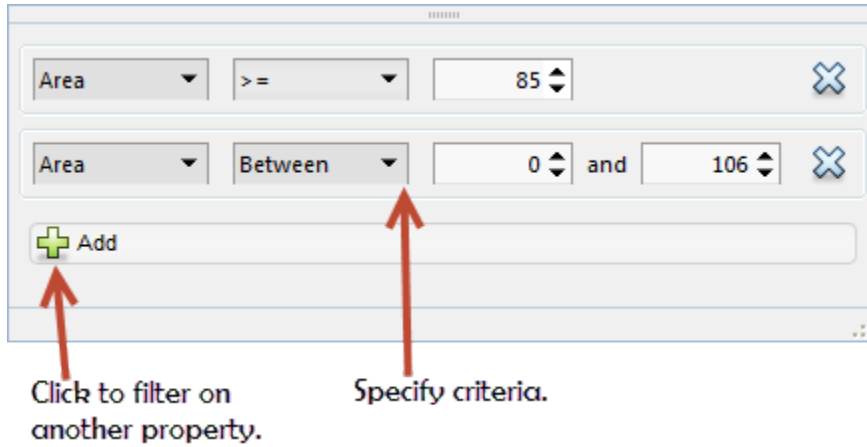
To filter on the value of a region property, click **Filter** and select the property on which you want to filter.



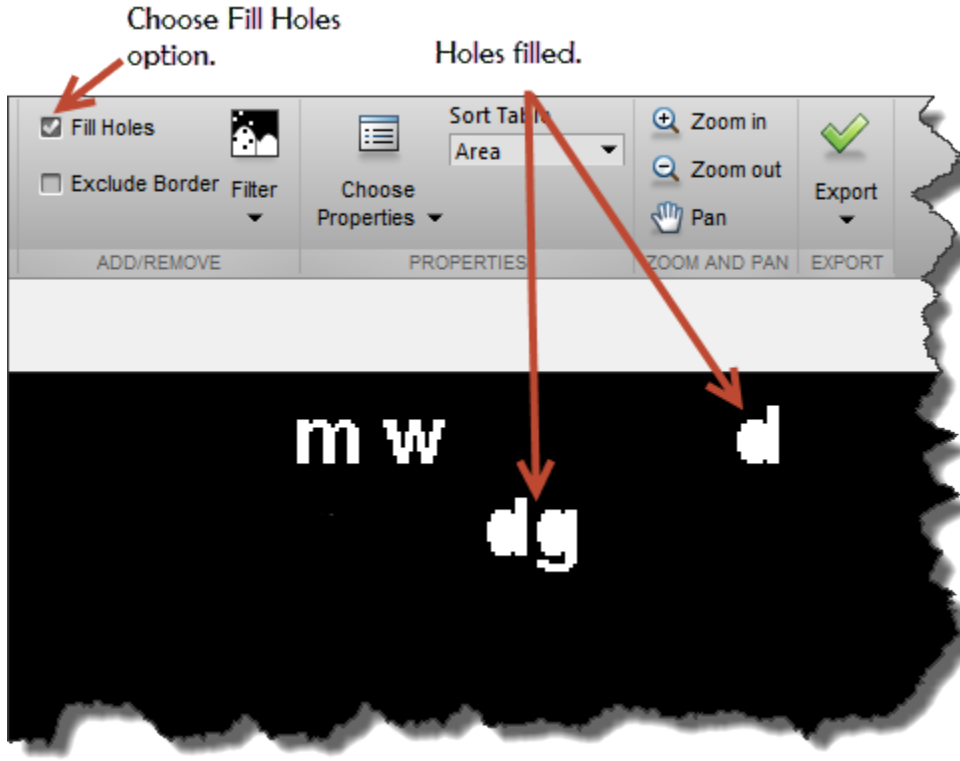
Next, specify the filter criteria. For example, to create an image that removes all but the largest regions, choose the greater than or equal to symbol (>=) and then specify the minimum value. Sometimes it's useful to sort the values in the table by the property you are interested in to determine what 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.



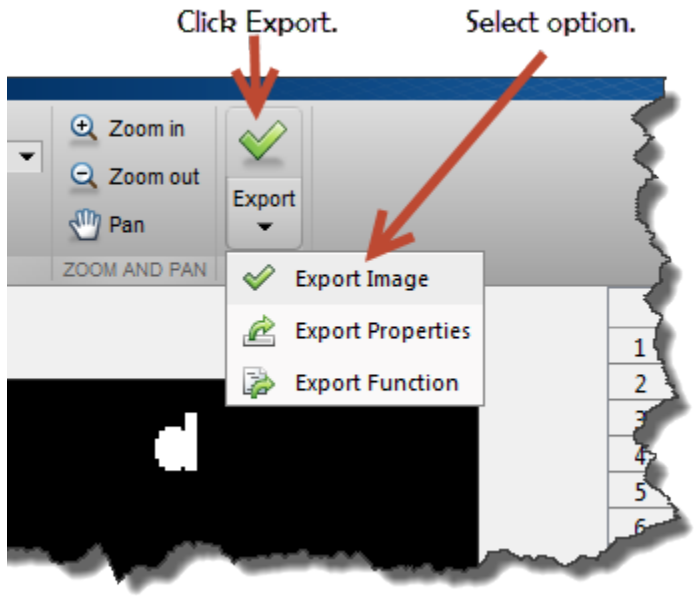
To filter on another property, click **Add**. The app displays another row in which you can select a property and specify filter criteria. The result is the intersection (logical AND) of the two filtering operations.



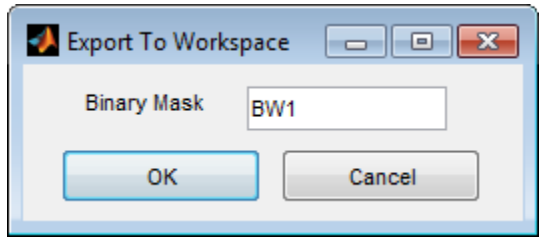
If you are creating a mask image, you can optionally perform some cleanup operations on the mask, such as clearing all foreground pixels that touch the border and filling holes in objects.



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**.



Create Image Comparison Tool Using ROIs

This example shows how to use the new ROI functions to create an interactive image comparison tool. When working with images, we often need to assess visually the impact a function has on an image. Some effects are clearly distinguishable, like an edge filter. But some impacts are more subtle and need closer inspection.

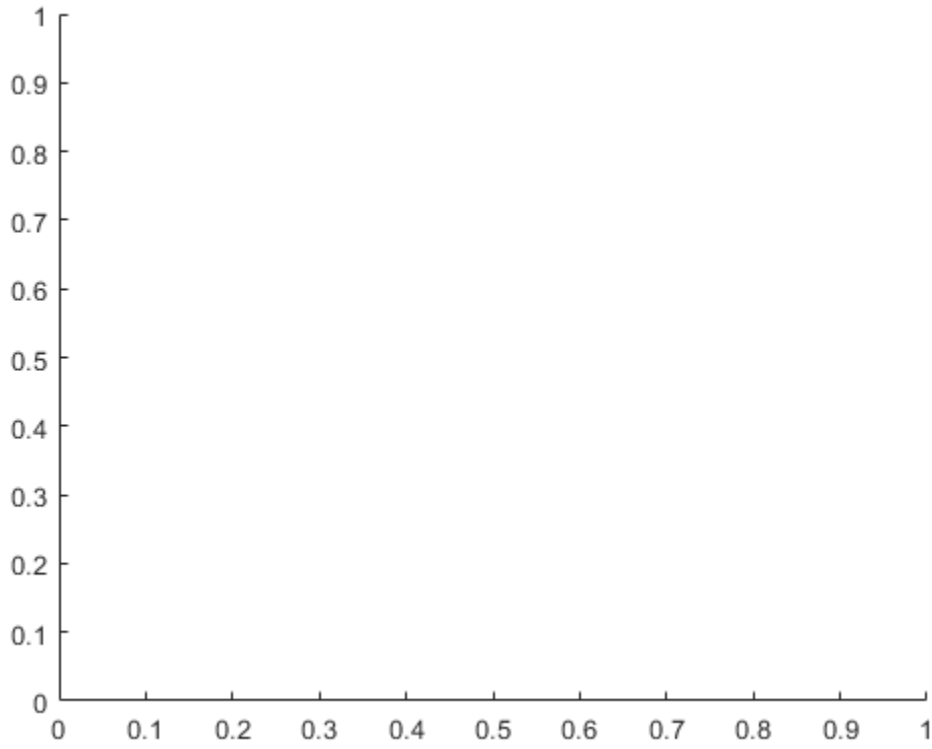
Read Sample Image into the Workspace

Read a sample image into the workspace and then create a grayscale version of the image. Use the `imshowpair` function to view the two images. The `montage` option shows the images side-by-side.

```
im = imread('peppers.png');
imgray = rgb2gray(im);
figure;
imshowpair(im,imgray,'montage')
```

Using an ROI, set the alpha layer (transparency) of two stacked images so that one image shows through only inside the ROI. This selective view follows the ROI so it can be moved interactively. Create a new figure and an axes.

```
hFigure = figure();
hAxes = axes('Parent', hFigure);
```



Stack both images on the same axes.

```
hImage1 = imshow(im, 'Parent', hAxes);
drawnow; % Ensure the image gets drawn.
hold on;
hImage2 = imshow(imgray, 'Parent', hAxes);
hold off;
```

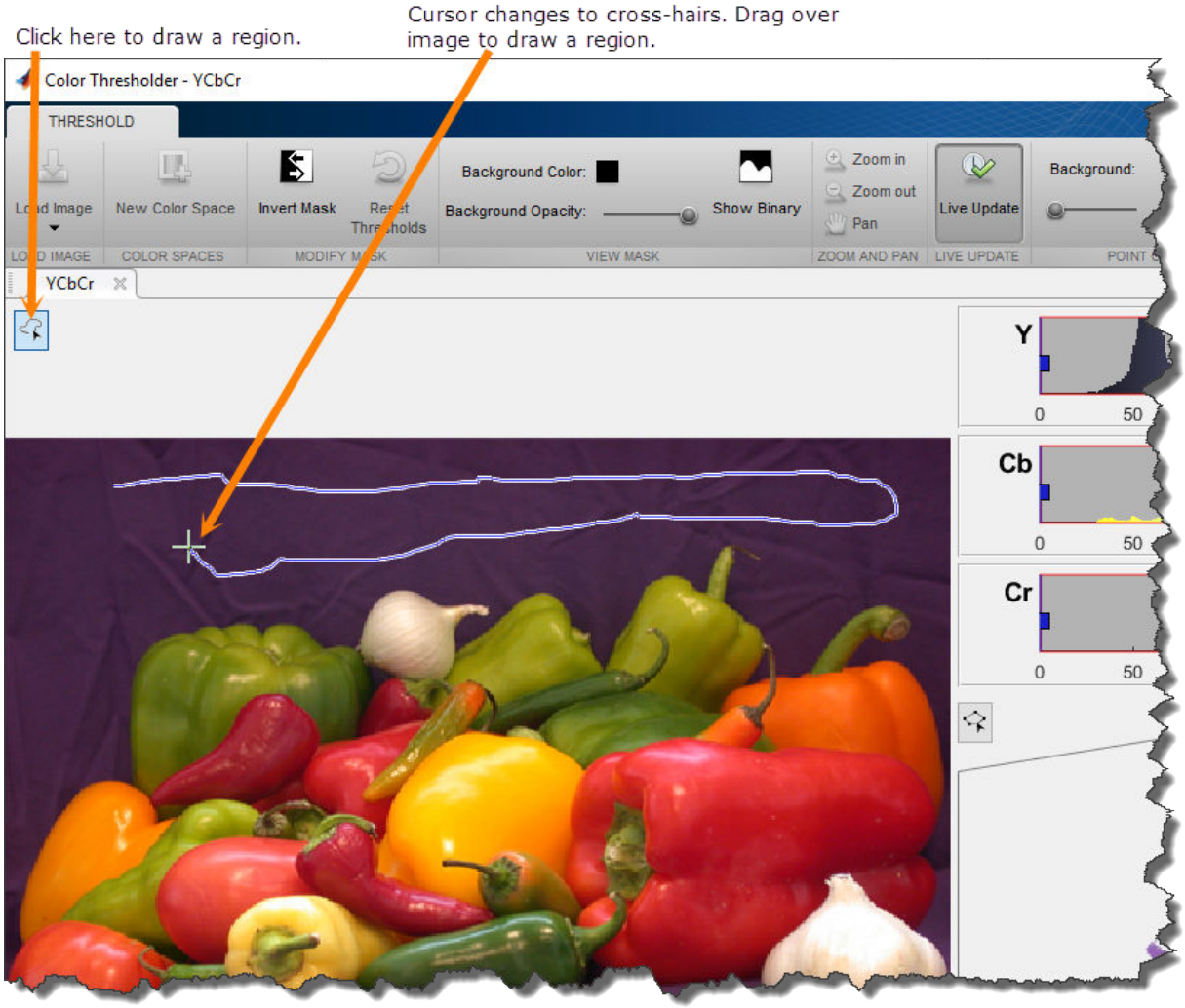
values. You can also rotate the color cloud to find the best isolation of the color you are interested in segmenting.

Segment Image Using the Color Selector in the Color Thresholder

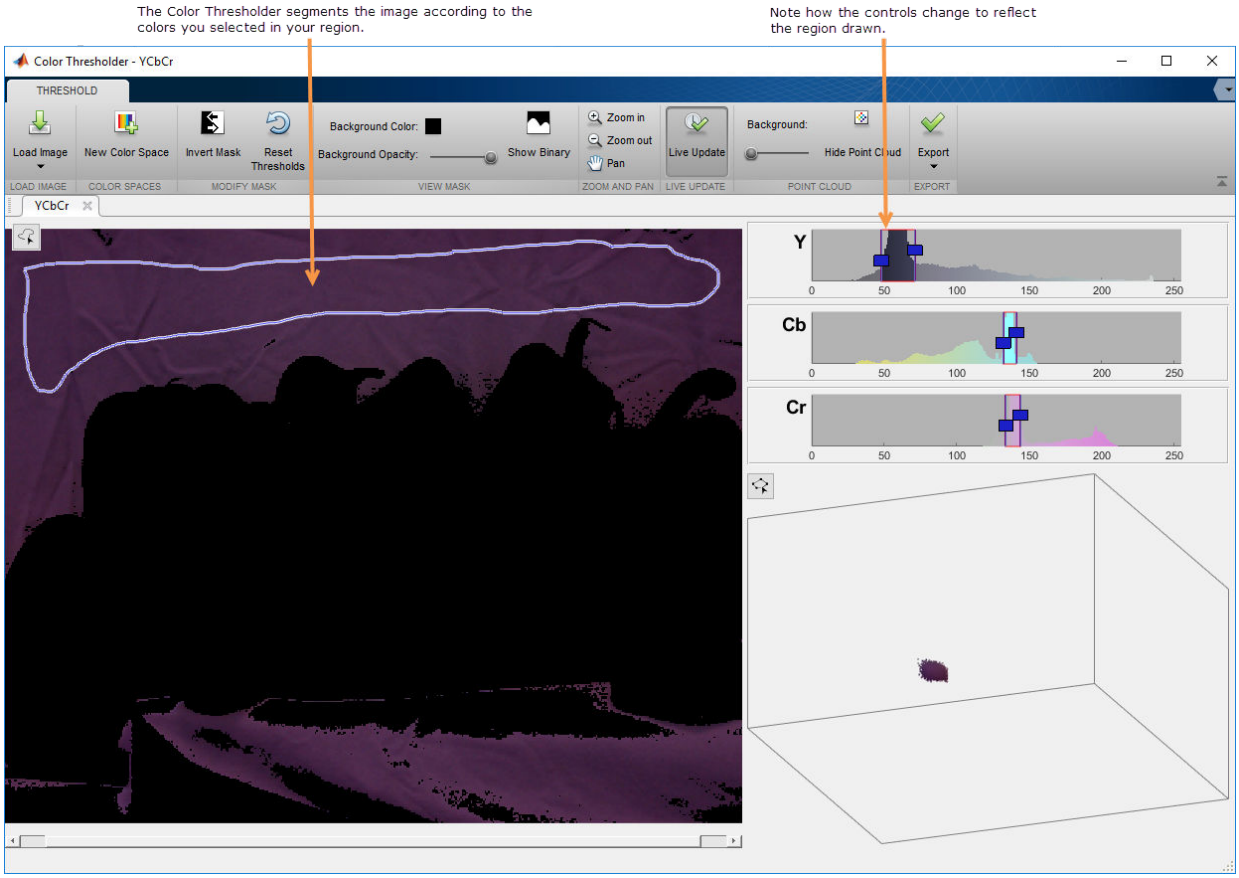
This part of the example shows how to segment an image automatically by selecting colors in the image. With this option, you draw a freehand region in the image to select a

color in either the foreground or the background. This example draws the region on the purple background to segment the vegetables from the background. 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 in the Color Thresholder” on page 13-65.

To segment the image automatically based on a color selection, click the button to draw a region on the image. When you move the cursor over the image, the cursor changes to a cross-hair shape. Drag the cursor over the image to draw regions to specify the colors you want to segment. You can draw multiple regions. If you want to delete a region you drew and start over, right-click the line you drew and select **Delete**.



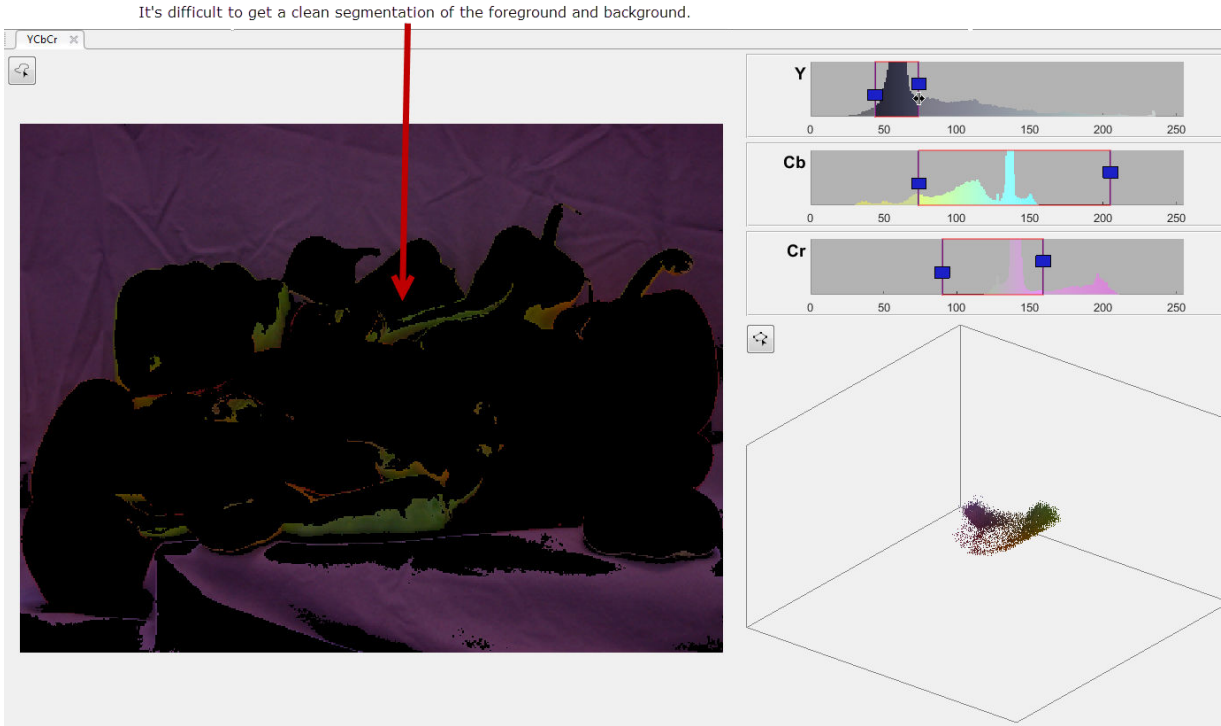
After drawing the regions, the Color Thresholder automatically thresholds the image based on the colors you selected in the region you drew. The color controls change to reflect the segmentation. This automatic segmentation did not create a well-defined edge between foreground and background. The background color is lighter near the bottom of the image. You can refine the thresholding by moving the controls.



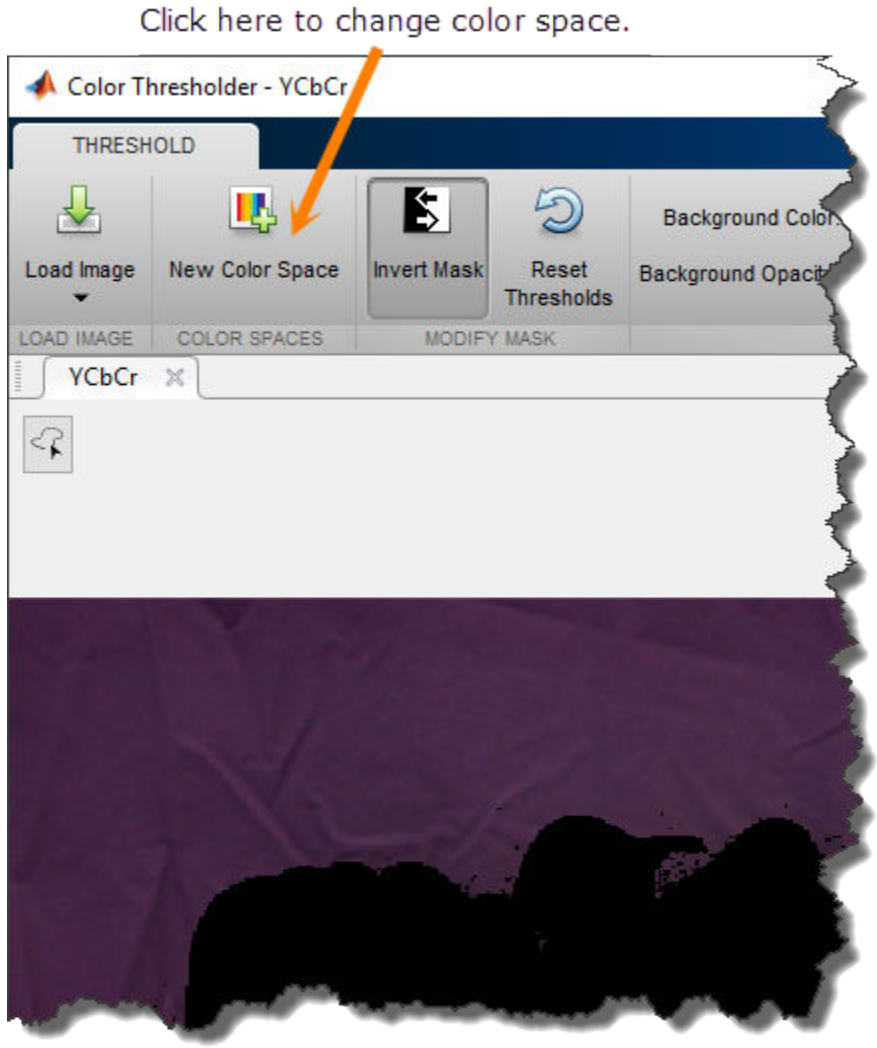
Segment Image Using Color Component Controls in the Color Thresholder

This part of the 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 the Color Selector in the Color Thresholder” on page 13-62.

Segment the image interactively using the color component controls. For example, use the histogram sliders to select the colors associated with each 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 image. In this case, you segment the foreground but you can invert the segmentation when you are done. Using the controls, It's hard to achieve a clean segmentation of the background without including part of the foreground image, as shown in the following figure. Redo the segmentation using another color space.

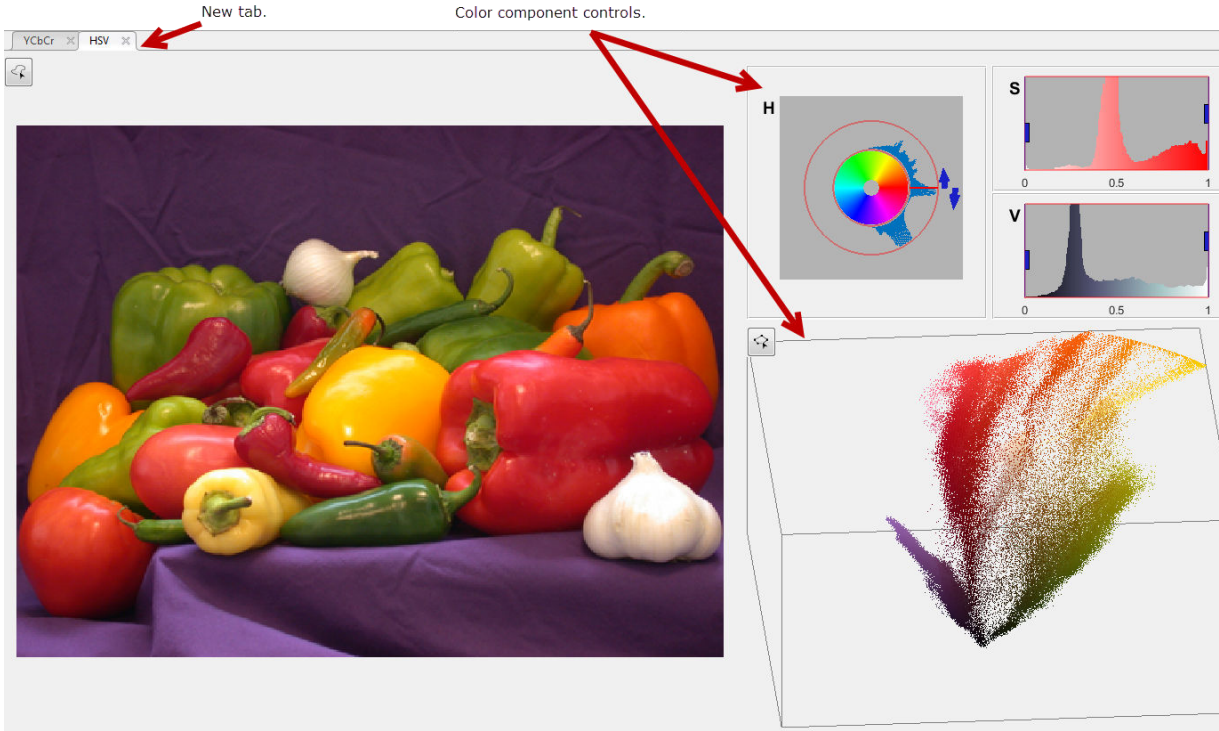


To use another color space, click **New Color Space**. The app displays the **Choose a color space** dialog box again.

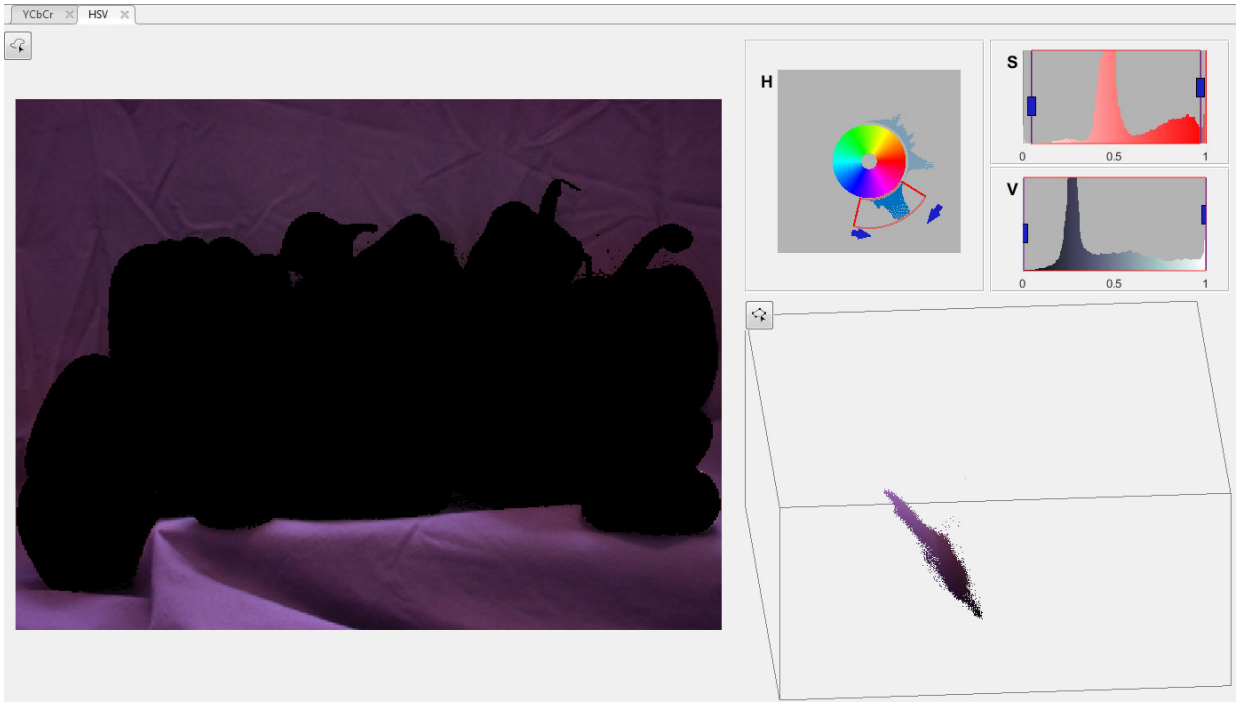


Select a new color space in the Choose a Color Space dialog box. For this example, choose the **HSV** color space. The Color Thresholder creates a new tab displaying the image and the color component controls for this color space. The HSV color space uses a dual-direction knob for the **H** component 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. The tab also contains the point cloud representation of the colors in the image.



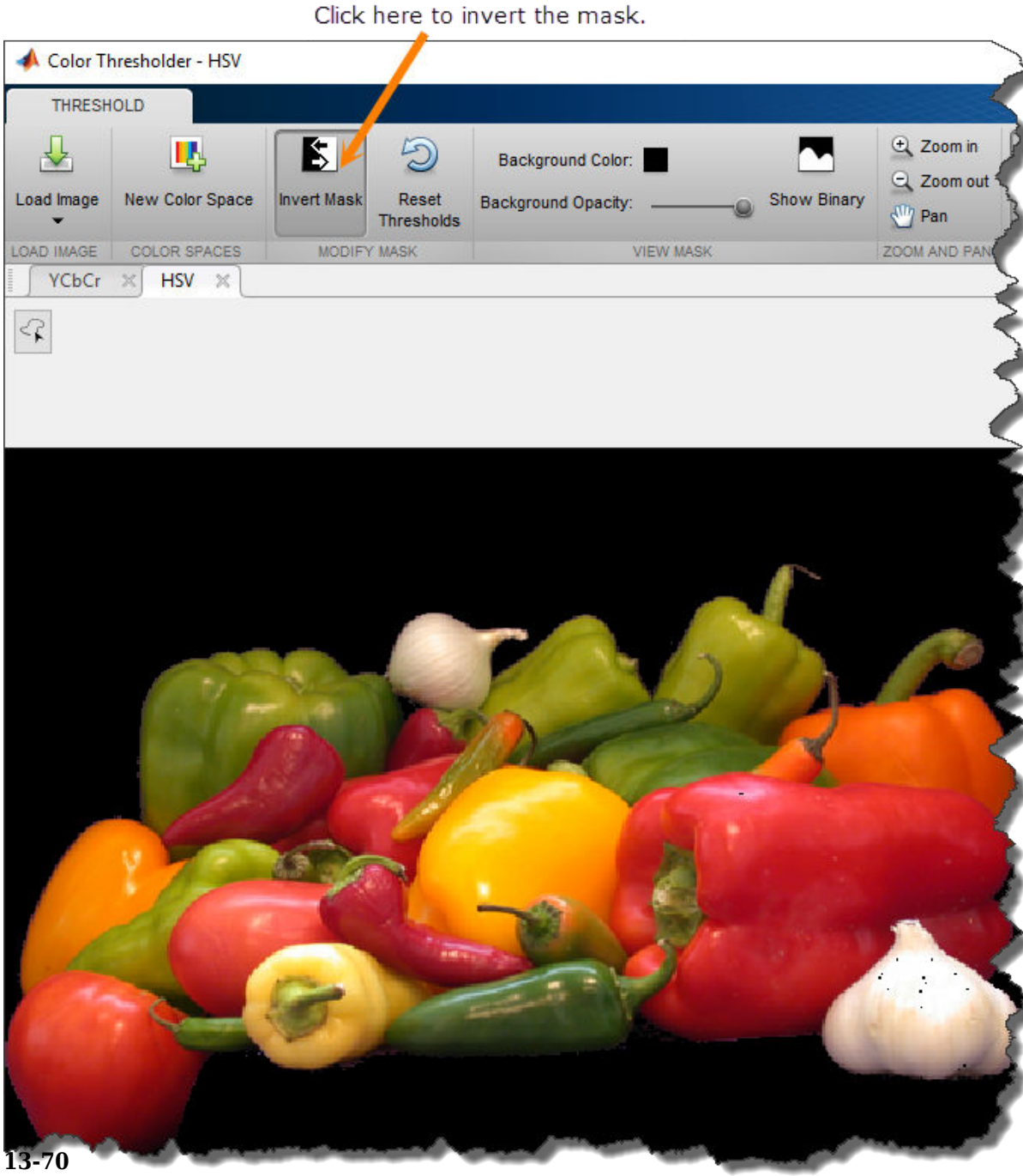
As you did before with the YCbCr color space, use the color component controls to segment the image interactively. As you use the controls, you can see the segmentation in progress. Using the mouse, grab one of the handles on the **H** control and move it in the direction of the arrow. Experiment with 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 Using the Color Thresholder” on page 13-69.



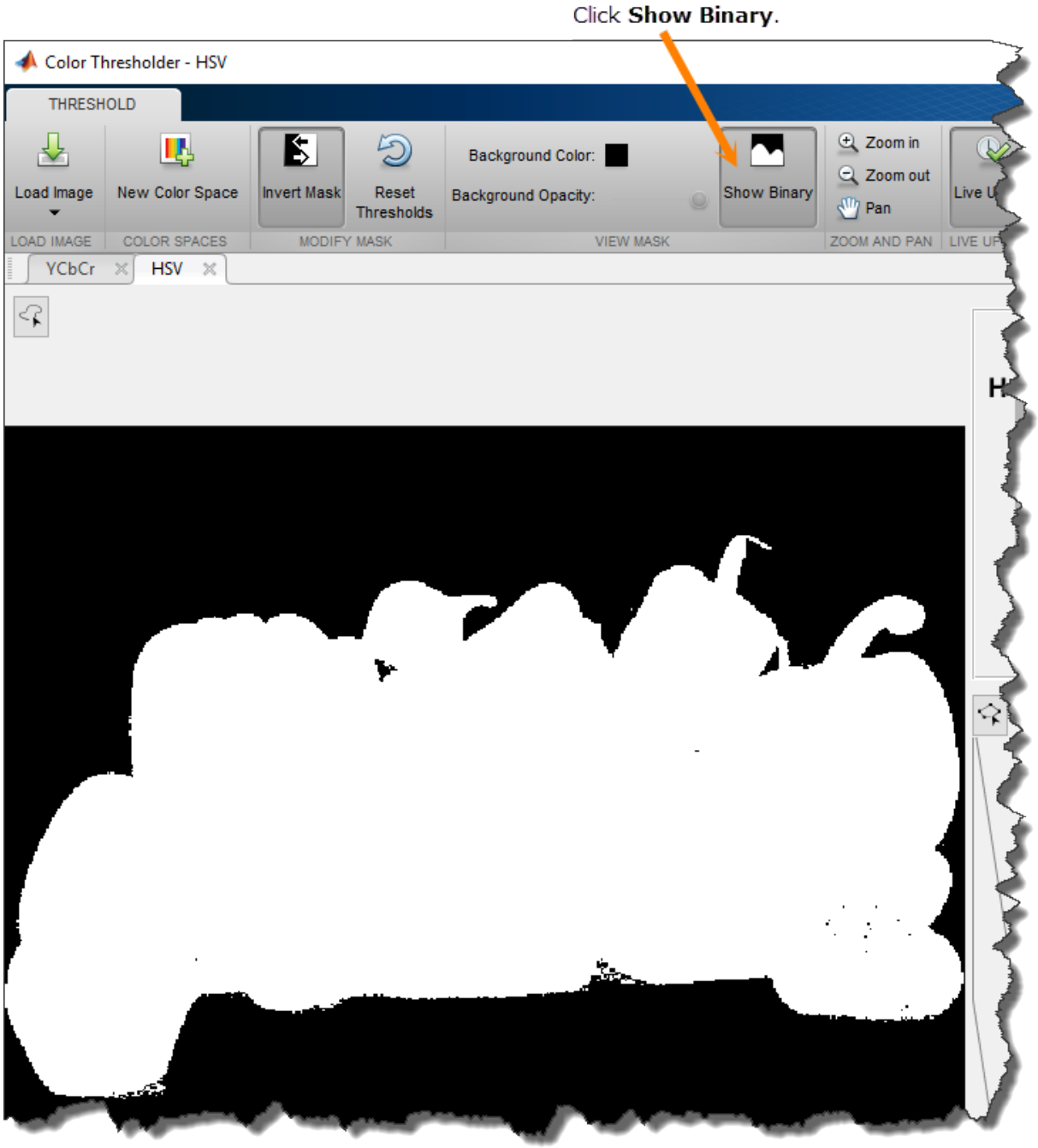
Create an Image Mask Using the Color Thresholder

This part of the 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.

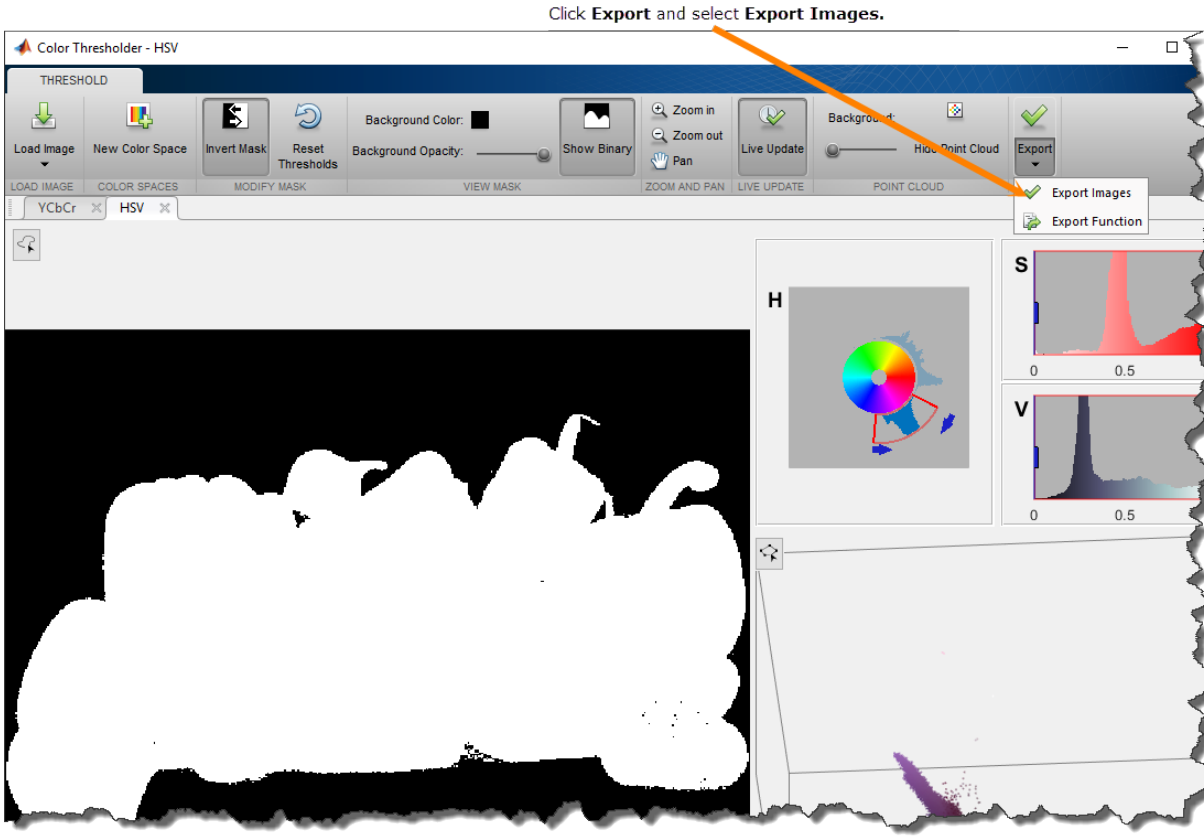
After segmenting out the foreground, you can swap the foreground and background by clicking **Invert Mask**. Inverting the mask can be helpful when, for example, it is easier to get a clean separation working with the foreground but you want a mask of the foreground. Perform the segmentation of the foreground, and then invert the mask.


Click here to invert the mask.
Color Thresholder - HSV
THRESHOLD
Load Image
New Color Space
Invert Mask
Reset Thresholds
Background Color:
Background Opacity:
Show Binary
Zoom in
Zoom out
Pan
LOAD IMAGE
COLOR SPACES
MODIFY MASK
VIEW MASK
ZOOM AND PAN
YCbCr
HSV

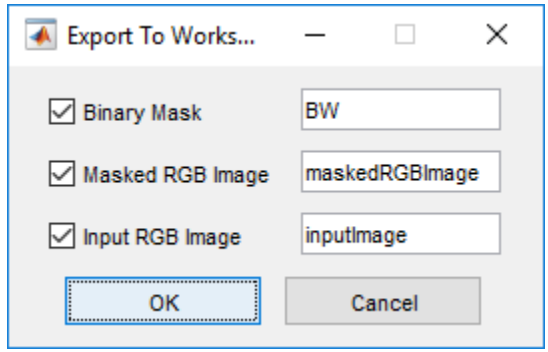
View the binary mask image that you created by clicking **Show Binary**.


Click **Show Binary**.
Color Thresholder - HSV
THRESHOLD
Load Image
New Color Space
Invert Mask
Reset Thresholds
Background Color:
Background Opacity:
Show Binary
Zoom in
Zoom out
Pan
LOAD IMAGE
COLOR SPACES
MODIFY MASK
VIEW MASK
ZOOM AND PAN
YCbCr
HSV

To save the mask image in the workspace, when you are satisfied with the segmentation, click **Export** and choose the **Export Images** option.



In the Export to Workspace dialog box, specify the name of the variables for the binary mask image. You can also save the original image and the segmented version of the original image.



To save the MATLAB code required to recreate the segmentation you just performed, click **Export** 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 22-Jun-2016
%------------------------------------------------------


% Convert RGB image to chosen color space
I = rgb2hsv(RGB);

% Define thresholds for channel 1 based on histogram settings
channel1Min = 0.713;
channel1Max = 0.911;

% Define thresholds for channel 2 based on histogram settings
channel2Min = 0.049;
channel2Max = 0.971;

% Define thresholds for channel 3 based on histogram settings
channel3Min = 0.005;
```

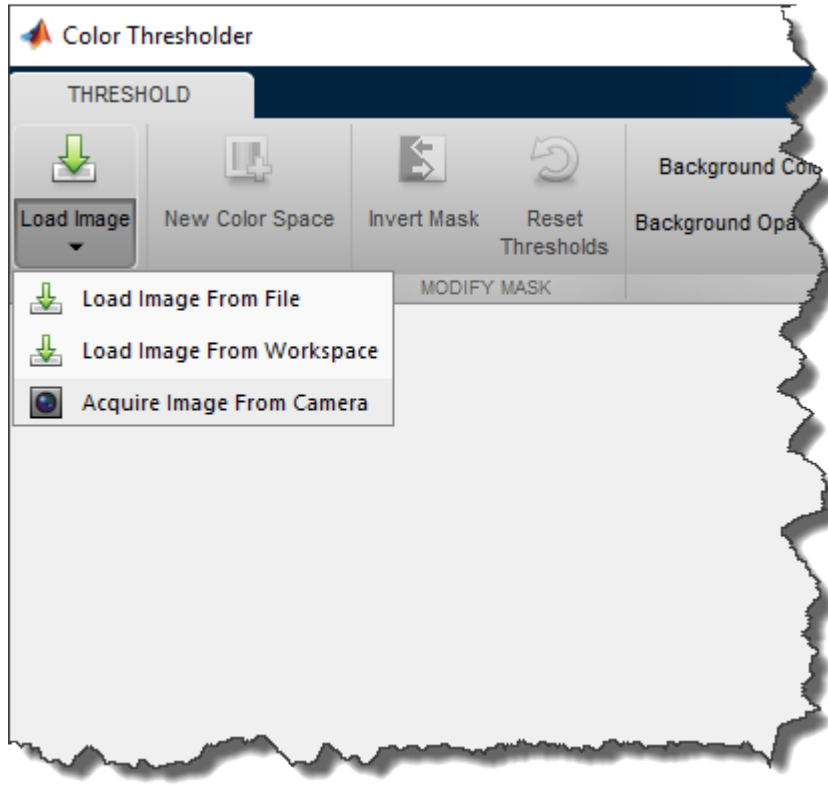
```
channel3Max = 1.000;

% Create mask based on chosen histogram thresholds
sliderBW = (I(:,:,1) >= channel1Min ) & (I(:,:,1) <= channel1Max) & ...
    (I(:,:,2) >= channel2Min ) & (I(:,:,2) <= channel2Max) & ...
    (I(:,:,3) >= channel3Min ) & (I(:,:,3) <= channel3Max);
BW = sliderBW;

% 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;

end
```

Acquire Live Images in the Color Thresholder App

You can color threshold an image acquired from a webcam using the Color Thresholder app. The **Image Capture** tab enables you to bring a live image from USB webcams into the app.

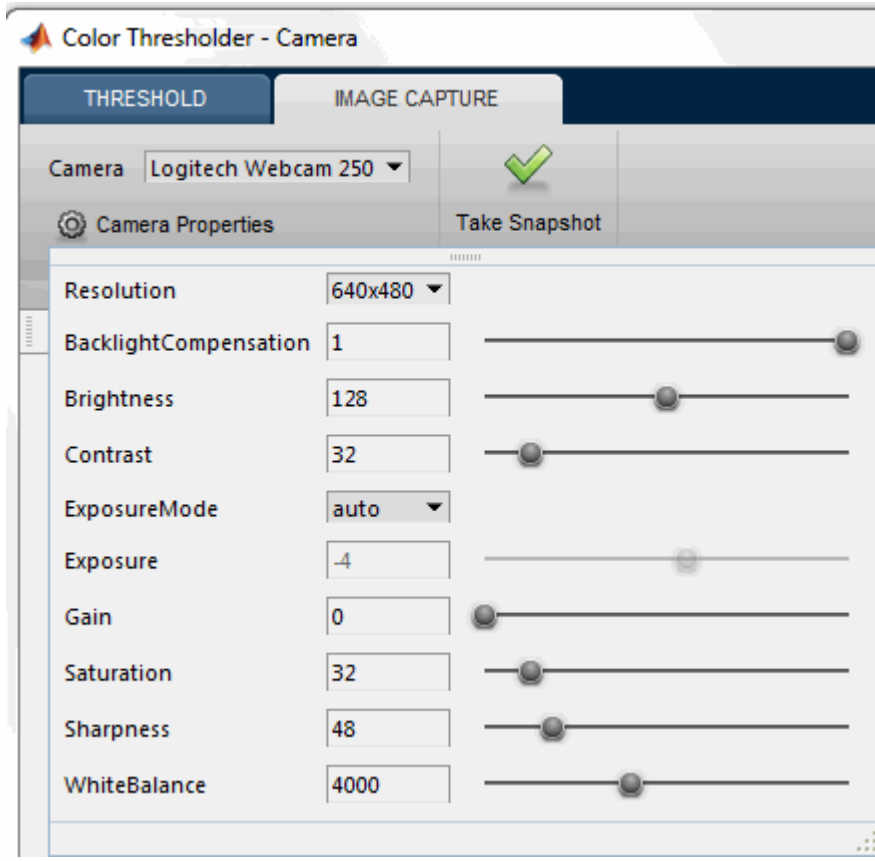
To begin color thresholding, add images that you acquire live from a webcam using the MATLAB Webcam support. Install the MATLAB Support Package for USB Webcams to use this feature. See “Install the MATLAB Support Package for USB Webcams” (Image Acquisition Toolbox) for information on installing the support package.

1. Open the app using the `colorThresholder` function.
2. To add a live image from your webcam, select **Load Image > Acquire Image From Camera** to open the Image Capture tab.



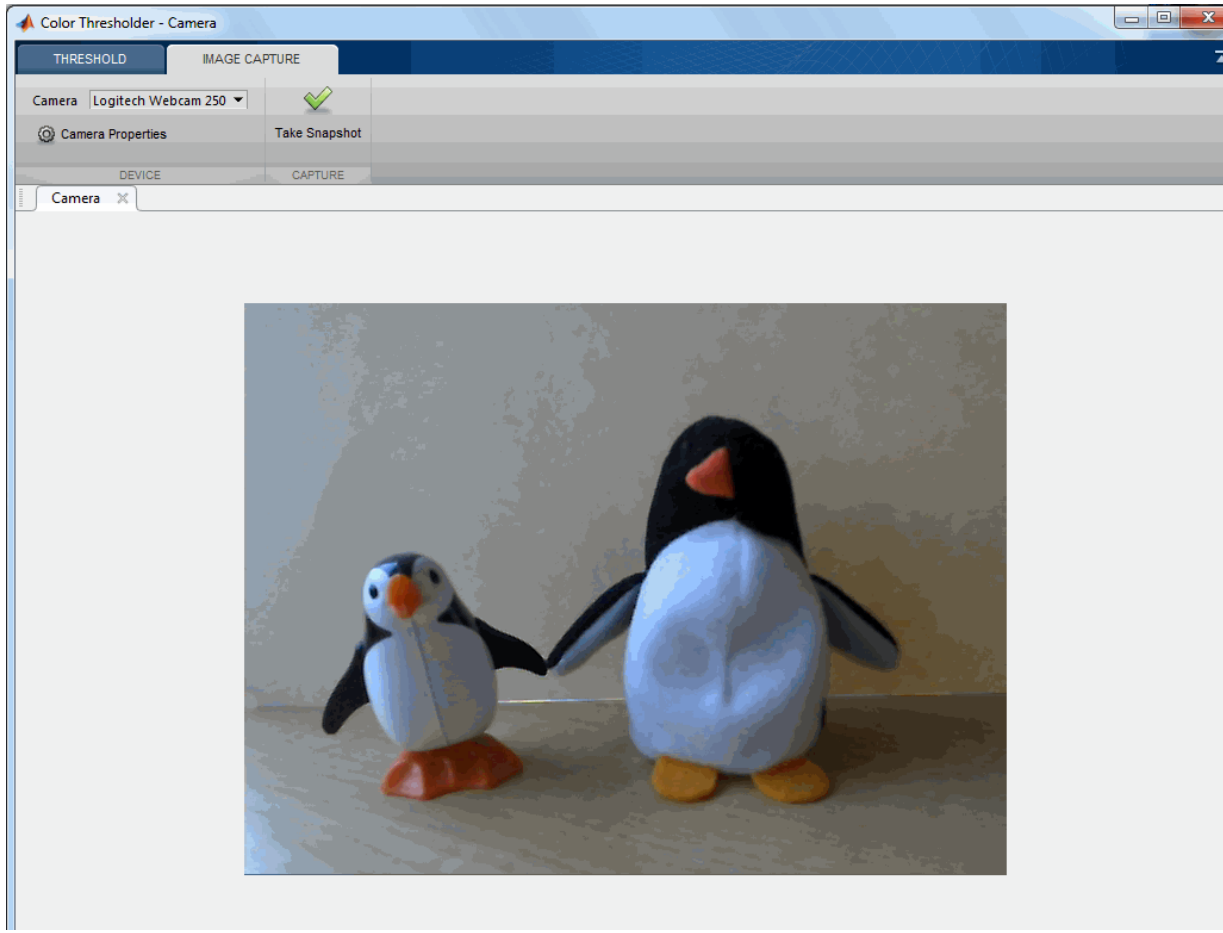
3 On the **Image Capture** tab, if you have only one webcam connected to your system, it is selected by default, and a live preview window opens. If you have multiple cameras connected and want to use a different one, select the camera in the **Camera** list.

4 Set properties for the camera to control the image. Click the **Camera Properties** field to open a list of your camera's properties. This list varies, depending on your device.

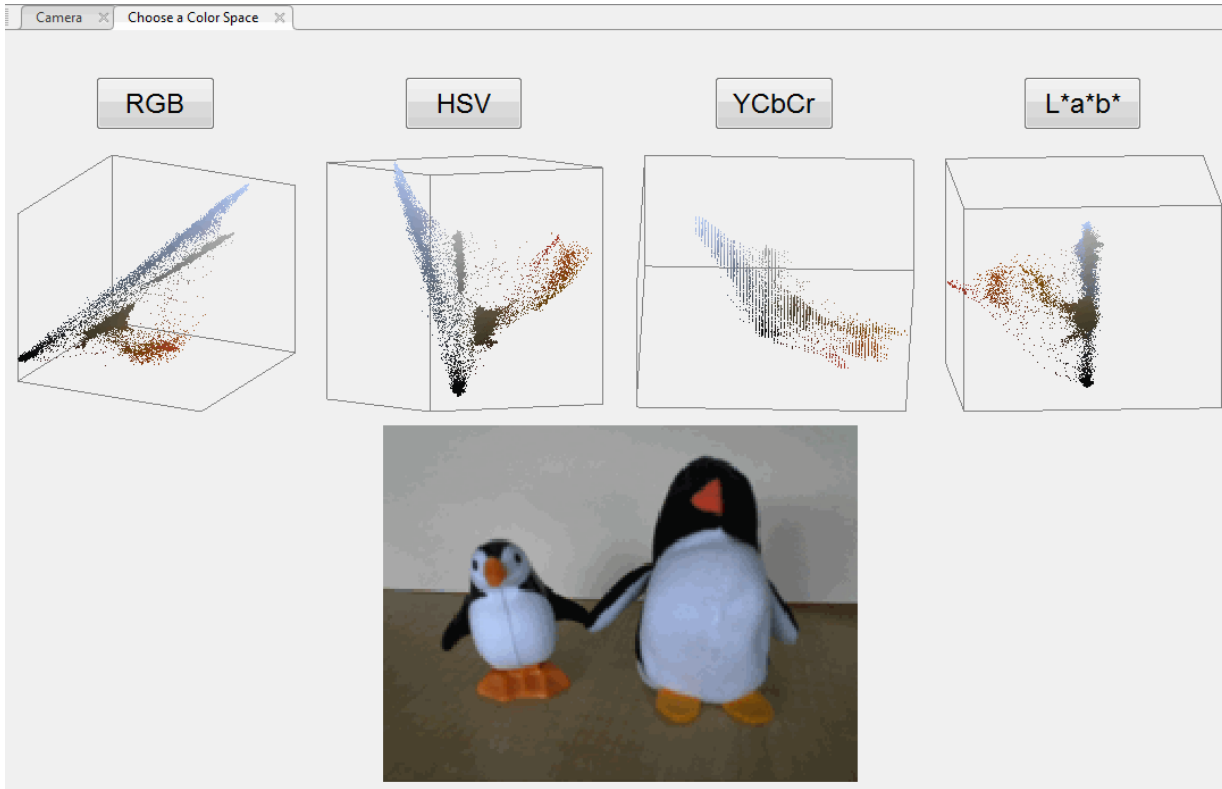


Use the sliders or drop-downs to change property settings. The **Preview** window updates dynamically when you change a setting. When you are done setting properties, click outside of the box to close the properties list.

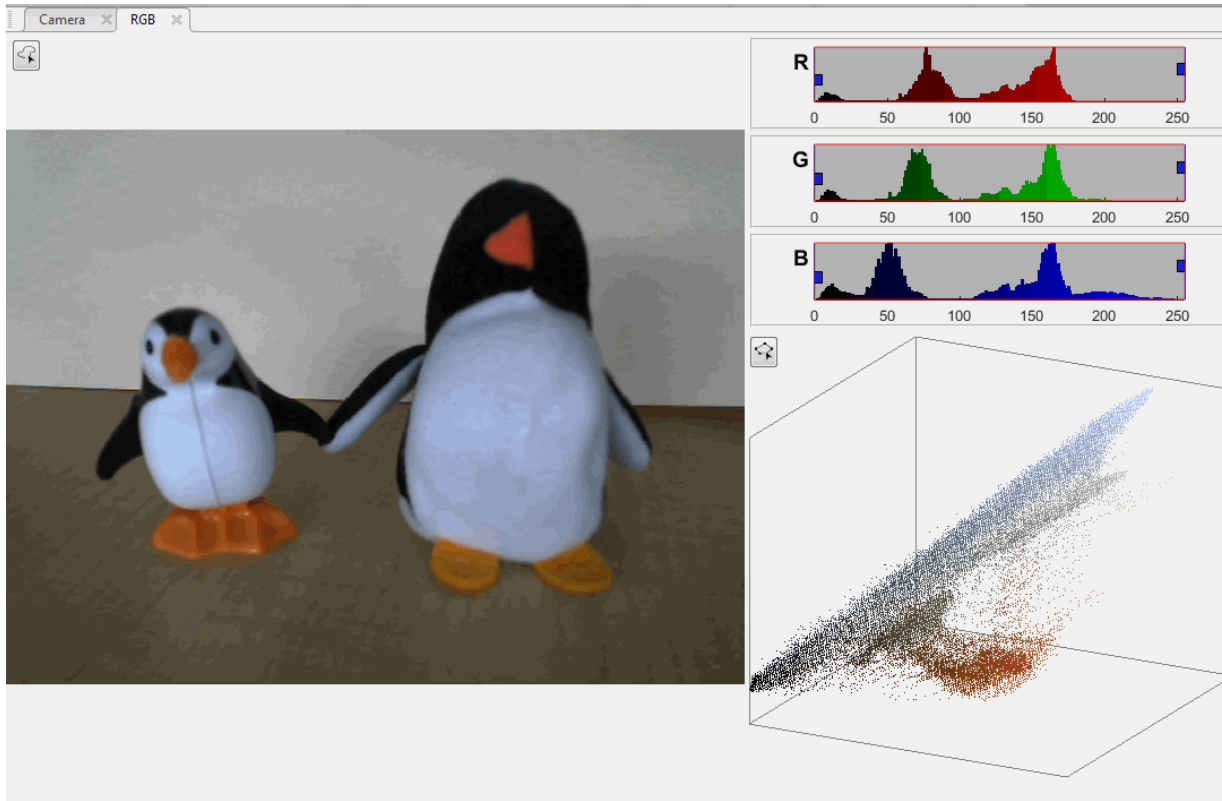
5 Use the **Preview** window as a guide to set up the image you want to acquire. The **Preview** window shows the live images streamed as RGB data.



6 Click **Take Snapshot**. The **Choose a Color Space** dialog box opens and displays the four color space options: `RGB`, `HSV`, `YCbCr`, and `L*a*b*`.

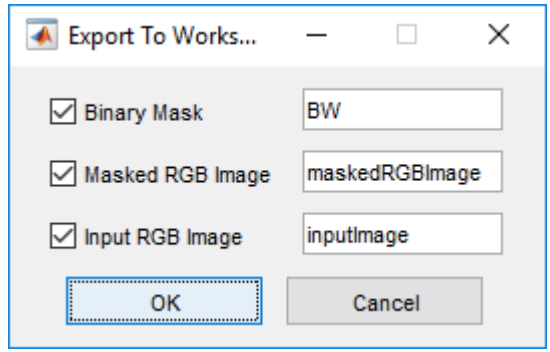


7 Choose a color space by clicking the button of your choice. Your image displays in a new tab using the selected color space.



8 You can now perform color thresholding on the image. See “Open Image in Color Thresholder” on page 13-57 for information about processing the image.

9 If you want to save the image that you captured, click **Export** and select **Export Images**.



In the Export To Workspace dialog box, you can save the binary mask, segmented image, and the captured image. The **Input RGB Image** option saves the RGB image captured from the webcam. For more details about exporting to the workspace, see Create an Image Mask.

10 You can take more snapshots, preview, or change camera properties at any time by clicking the **Camera** tab.

Image Segmentation Using Point Clouds in the Color Thresholder App

This example shows how to segment an image to create a binary mask image using point cloud controls in the Color Thresholder app. The example segments the human face from the background.

In this section...

"Load Image into the Color Thresholder App" on page 13-83

"Choose a Color Space" on page 13-87

"Segment the Image Using the Color Cloud" on page 13-89

Load Image into the Color Thresholder App

This part of the example shows how to load an image in the Color Thresholder app.

Read an image into the workspace. For this example, read the sample image mandi.tif into the workspace. The image is a Bayer pattern-encoded image that needs to be converted into an RGB image to use with the app. Display the image.

```
X = imread('mandi.tif');
rgb = demosaic(X, 'bggr');
imshow(rgb)
```

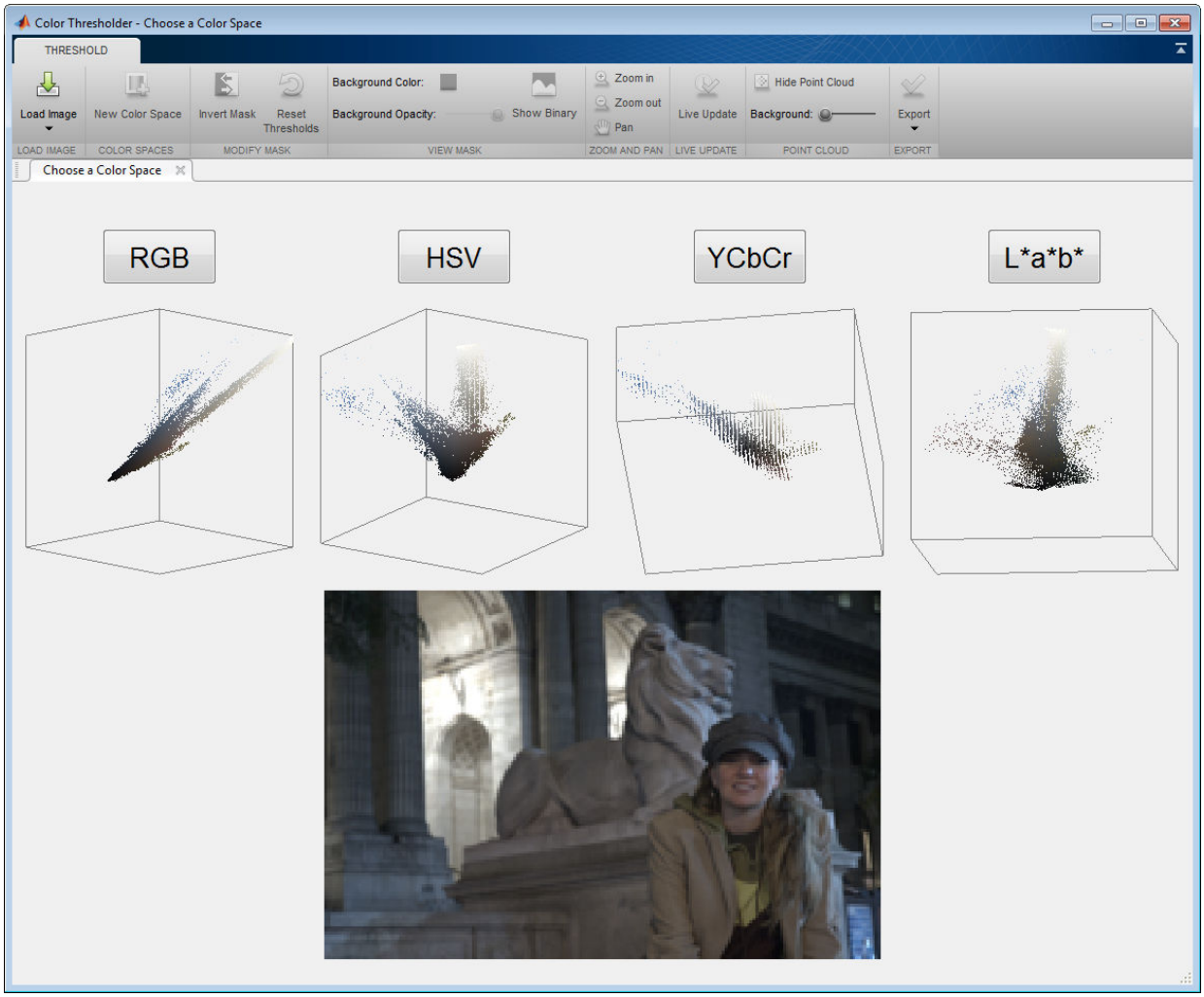
Open the Color Thresholder app. From the MATLAB Toolstrip, open the Apps tab and

under Image Processing and Computer Vision, click . You can also open the app using the `colorThresholder` command.

Bring the image into the Color Thresholder app. Click **Load Image**. You can load an image by specifying its file name or you can read an image into the workspace and load the variable. You can also acquire an image from a camera (see "Acquire Live Images in the Color Thresholder App" on page 13-76).

From the **Load Image** menu, click **Load Image from Workspace**. In the Import from Workspace dialog box, select the variable you created and click **OK**.

When it opens, the Color Thresholder app displays the **Choose a color space** tab which displays your image as a point cloud in several popular color spaces: RGB, HSV, YCbCr, and *L***a***b**.

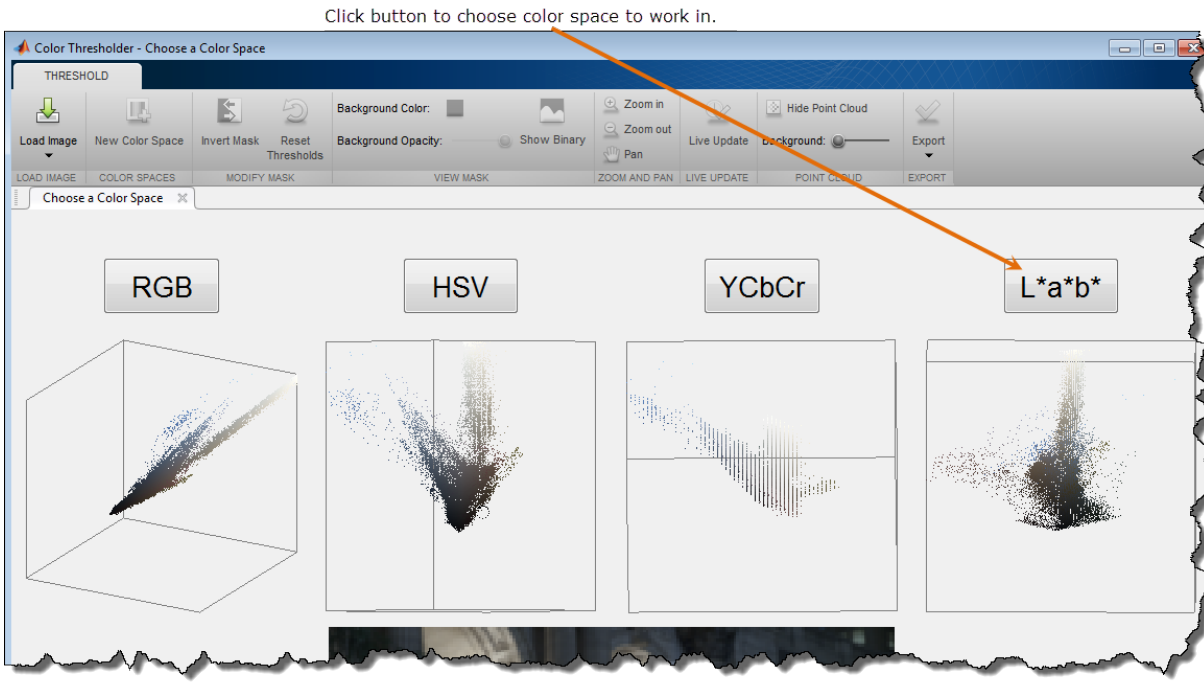


Choose a Color Space

This part of the example shows how to choose the color space to work in while segmenting the image. 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

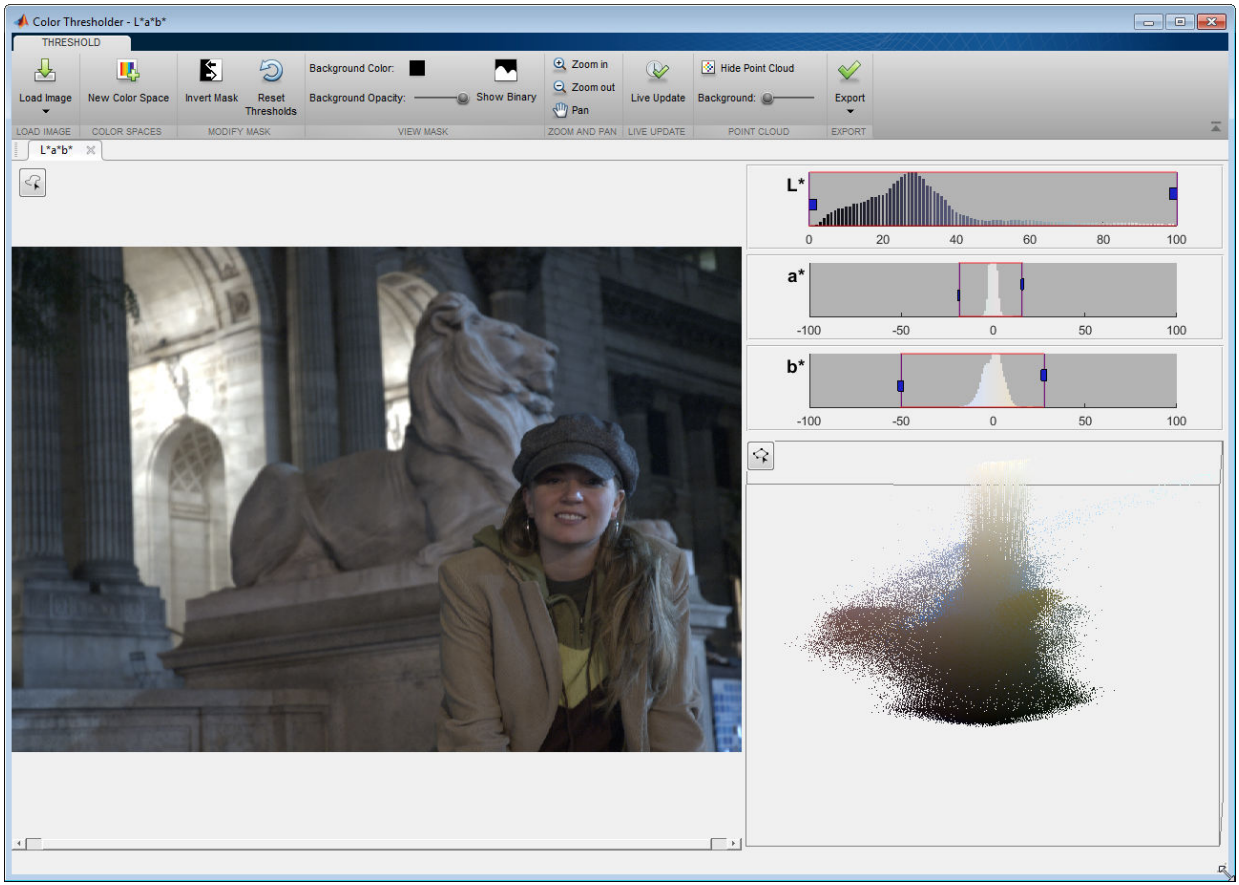
colors you are interested in segmenting appear near each other in the color model. You can always change the color space you choose later, using **New Color Space**.

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** tab which displays your image as a point cloud in several popular color spaces: RGB, HSV, YCbCr, and *L*a*b**. Using the mouse, choose the color space. Examine the representation of the image in each color space, rotating the 3-D depiction of each color space, to see how well the colors are differentiated. You select the color to segment from this 3-D display, so it is important to choose a representation that allows you select the colors of the area you want to segment. For this example, choose the *L*a*b** color space.



When you choose a color space, the app opens a new tab, displaying the image along with a set of controls for each color component of the color space you chose. For the *L*a*b** color space, the Color Thresholder displays three histograms representing the three components in the color space and a 3-D point cloud representation of the colors of the

image in the color space. You can use these components together to create the segmentation of the image. Other color spaces use different types of controls.

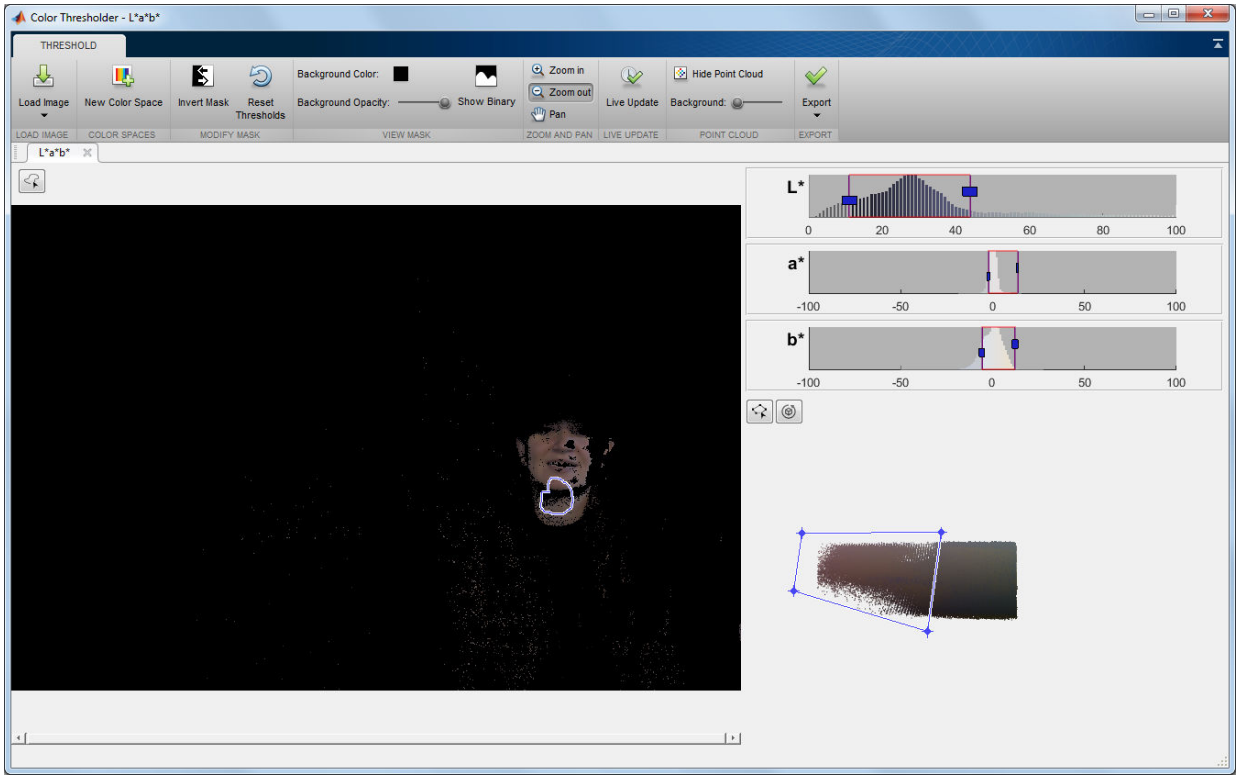


Segment the Image Using the Color Cloud

This part of the example shows how to segment the image using the color cloud.

Rotate the color cloud, using the mouse, to find a view of the color cloud that displays the colors you want to segment in isolation. To select the colors you want to use to segment the image by drawing a polygon around the colors, click the button at the top left of the color cloud and start clicking points to create a polygon around the colors you want to use

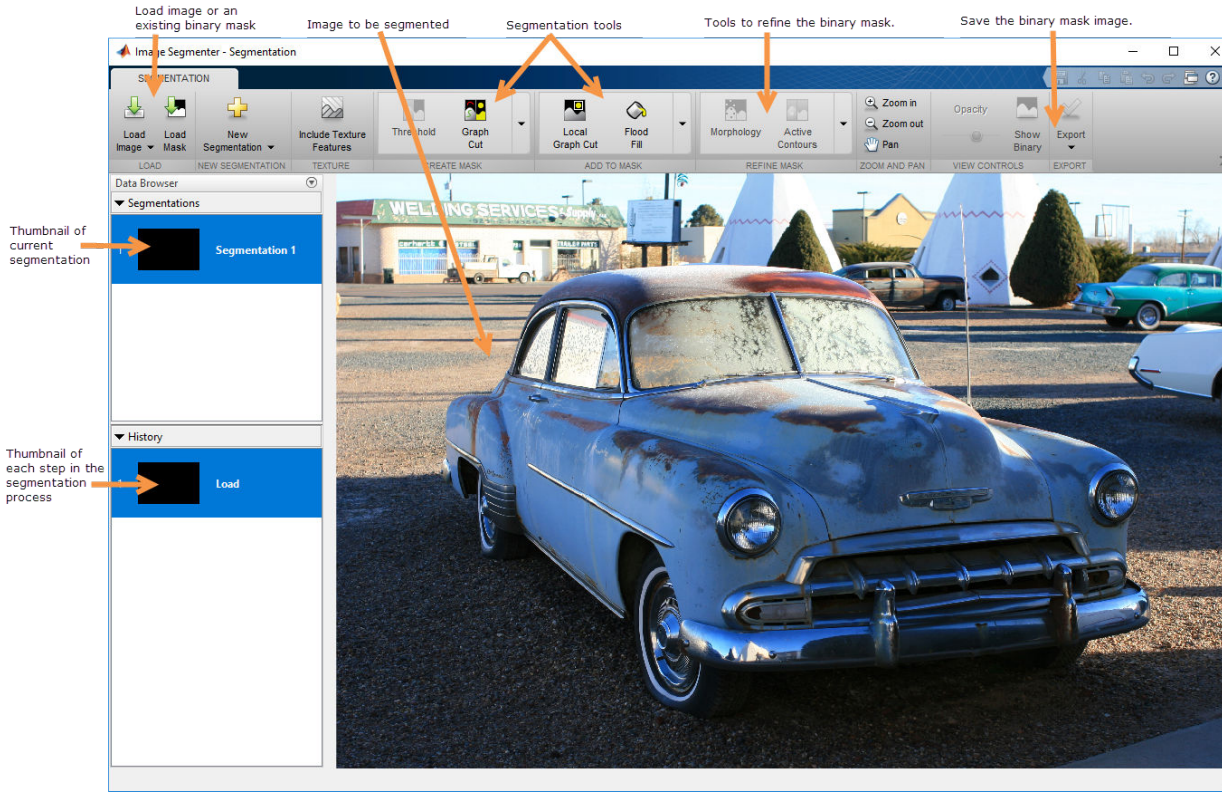
for the segmentation. When you close the polygon, the Color Thresholder app performs the segmentation based on the colors you selected. You can use the histograms to refine your segmentation, as shown in the following figure.



For information about saving your segmentation or the code required to create it, see “Create an Image Mask Using the Color Thresholder” on page 13-69.

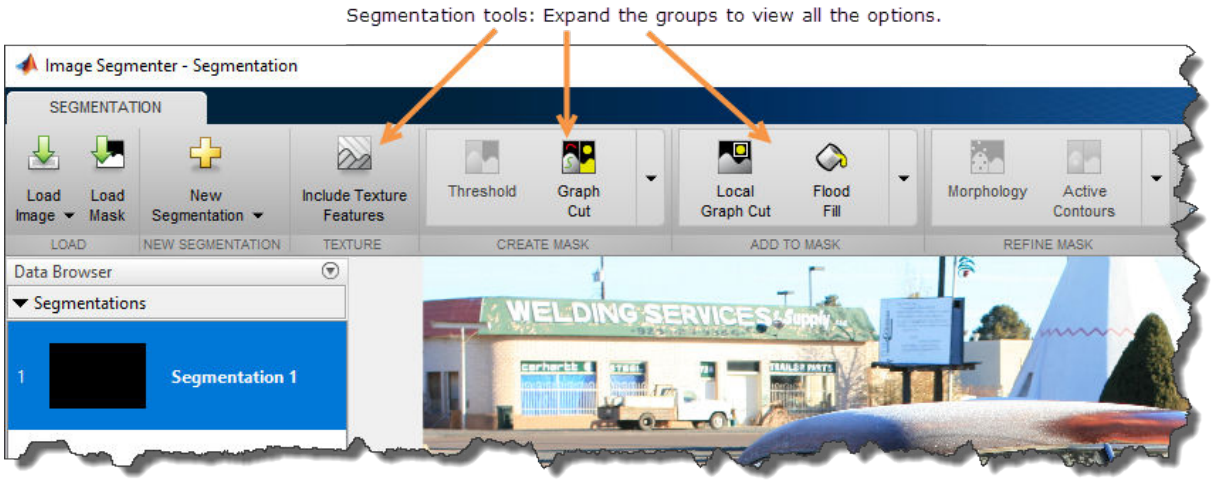
Image Segmentation Using the Image Segmenter App

The Image Segmenter app provides access to many different ways to segment an image. Using the Image Segmenter can be an iterative process where you might try several of the segmentation options. Some segmentation techniques might work better with certain types of images than others. After segmenting an image, you can save the binary mask. You can also retrieve the code the Image Segmenter used to create the mask.The following is an overview of the Image Segmenter app. For information about opening the app, see “Open Image Segmenter App and Load Image” on page 13-97.



Segmentation Tools Available in the Image Segmenter App

The Image Segmenter app provides many segmentation tools, available in the app toolstrip.



Segmentation Tools

Tool	Description
Threshold	An automatic technique where you specify an intensity value that you want to isolate. This technique can be useful if 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. For more information, see “Segment Using Threshold Technique in Image Segmenter” on page 13-103.
Graph Cut	A semi-automatic technique that can segment foreground and background. This technique does not require careful placement of seed points and you can refine the segmentation interactively. For more information, see “Segmentation Using Graph Cut in Image Segmenter” on page 13-115.
Auto Cluster	An automatic technique where the app groups image features into a binary segmentation. This option is only available if you have the Statistics and Machine Learning Toolbox. For more information, see “Segmentation Using Auto Cluster in Image Segmenter” on page 13-136.
Find Circles	An automatic technique where you specify the minimum and maximum diameter of the circular objects you want to detect. For more information, see “Segmentation Using Find Circles in Image Segmenter” on page 13-129

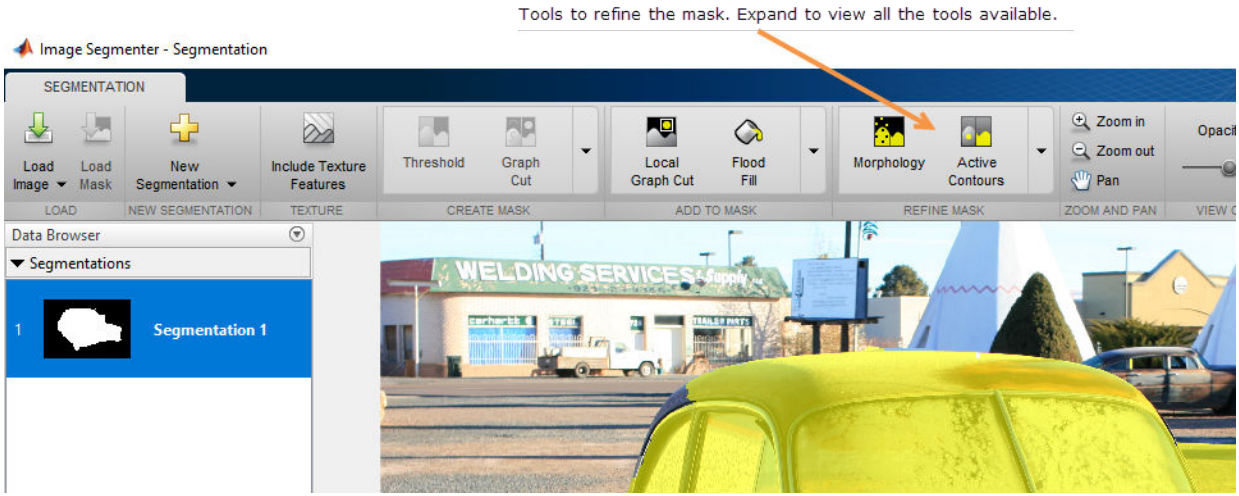
Tool	Description
Local Graph Cut (grabcut)	A semi-automatic technique, similar to the Graph Cut method, that can segment foreground and background. With Local Graph Cut (grabcut), you first define an ROI that encompasses the object in the image that you want to segment. The Image Segmenter automatically segments the object in the ROI. You can refine the segmentation by drawing lines on the image to identify the foreground and the background within the ROI. Everything outside the ROI is considered background. For more information, see “Segmentation Using Local Graph Cut (Grabcut) in Image Segmenter” on page 13-122.
Flood Fill	An automatic technique where you specify starting points and the method segments areas with similar intensity values.
Draw ROI	A manual technique where you draw shapes that outline the region the objects you want to segment. Using the mouse, you can draw rectangles, ellipses, polygons, or freehand shapes. For more information, see “Segment By Drawing Regions in Image Segmenter” on page 13-105.

Including Texture Features in a Segmentation

When using the **Auto Cluster**, **Graph Cut**, and **Flood Fill** segmentation tools, you can also include texture as an additional consideration in your segmentation. Click **Include Texture Features** to turn the texture option on and off. When enabled, the Image Segmenter uses Gabor filters to analyze the texture of the image as a preprocessing step in the segmentation. Texture filtering can help distinguish foreground from background. For an example of using Gabor filters in an image segmentation, see “Texture Segmentation Using Gabor Filters” on page 13-2.

Tools to Refine the Segmentation in the Image Segmenter

In addition to these segmentation techniques, the Image Segmenter app provides access to several tools that you can use to refine the mask you created.


Tools to refine the mask. Expand to view all the tools available.
Image Segmenter - Segmentation
SEGMENTATION
Load Image
Load Mask
New Segmentation
Include Texture Features
Threshold
Graph Cut
Local Graph Cut
Flood Fill
Morphology
Active Contours
Zoom in
Zoom out
Pan
LOAD
NEW SEGMENTATION
TEXTURE
CREATE MASK
ADD TO MASK
REFINE MASK
ZOOM AND PAN
Data Browser
Segmentations
1
Segmentation 1
WELDING SERVICES

Tools to Refine the Binary Mask

Tool	Description
Morphology	Many morphological techniques, such as dilation and erosion. For an example, view “Use Morphology to Refine Mask in Image Segmenter” on page 13-111.
Active contours (also known as snakes)	An iterative method that grows (or shrinks) regions in an image. You identify the regions with seed points. For an example, view “Use Active Contours to Refine Segmentation in Image Segmenter” on page 13-107.
Clear borders	A fast way to remove small regions on the edge of the image.
Fill holes	A fast way to fill small holes in foreground regions. For an example, view “Use Morphology to Refine Mask in Image Segmenter” on page 13-111.
Invert mask	Sometimes the segmentation is easier to evaluate if you invert the foreground and background. For an example, view “Segmentation Using Auto Cluster in Image Segmenter” on page 13-136

See Also

Image Segmenter

Related Examples

- “The Image Segmenter Workflow” on page 13-103

Open Image Segmenter App and Load Image

This example shows how to open the Image Segmenter app and load an image.

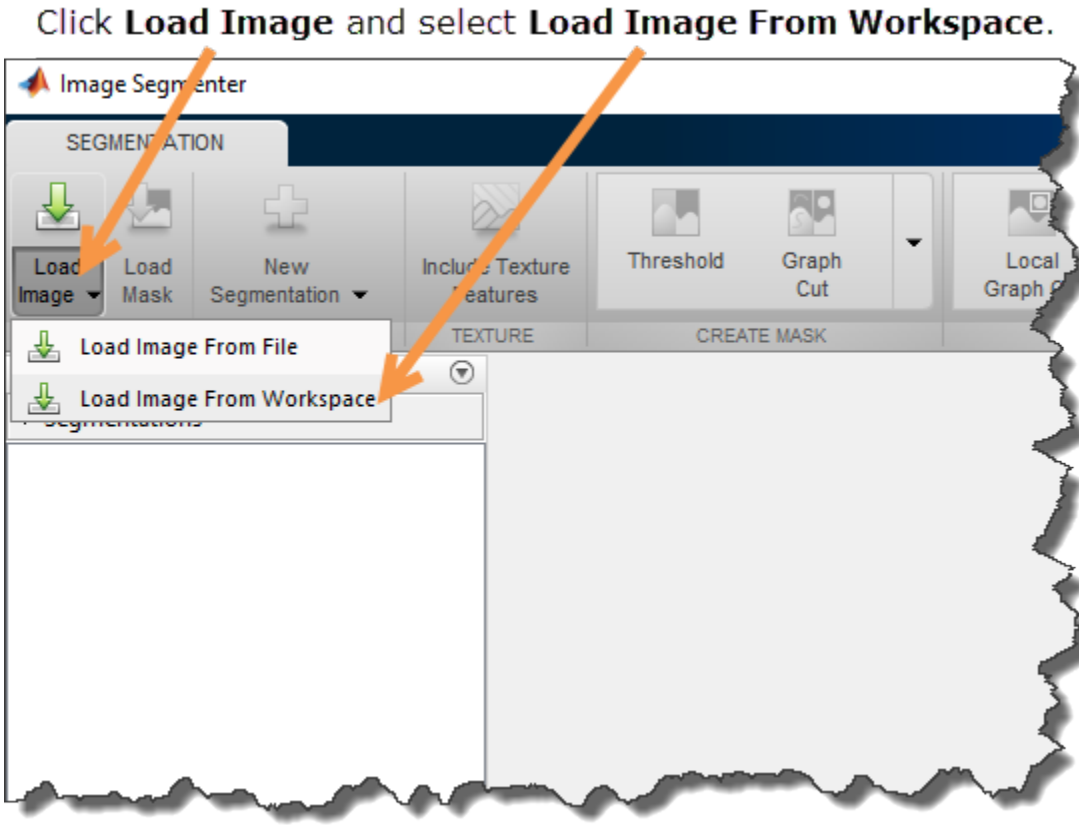
Read an image into the MATLAB workspace. The example uses the `imread` function to import an image of coins.

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

Open the Image Segmenter app. From the MATLAB Toolstrip, open the Apps tab and under Image Processing and Computer Vision, click **Image Segmenter** . You can also open the Image Segmenter from the command line:

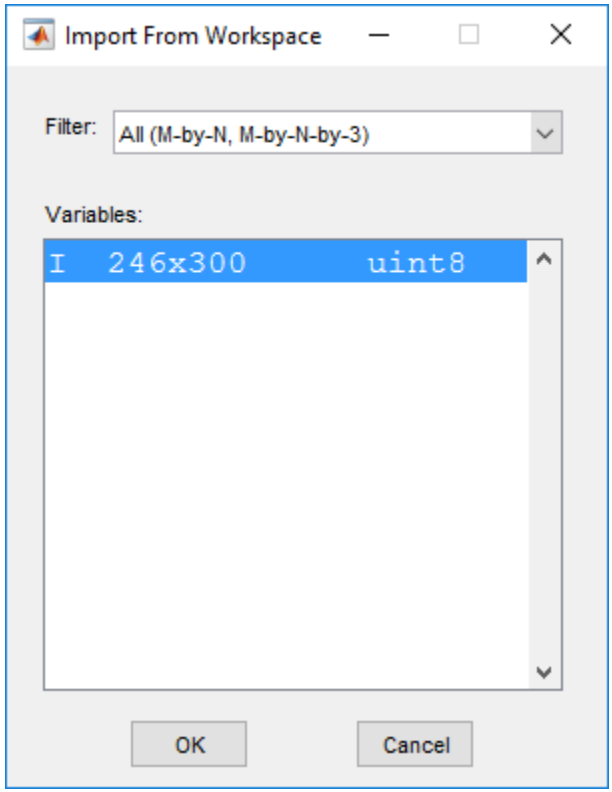
```
imageSegmenter
```

Load an image into the Image Segmenter app. Click **Load Image** and choose whether you want to specify the name of the file containing the image or the name of a workspace variable. The Image Segmenter app can open any file that can be read by `imread`.

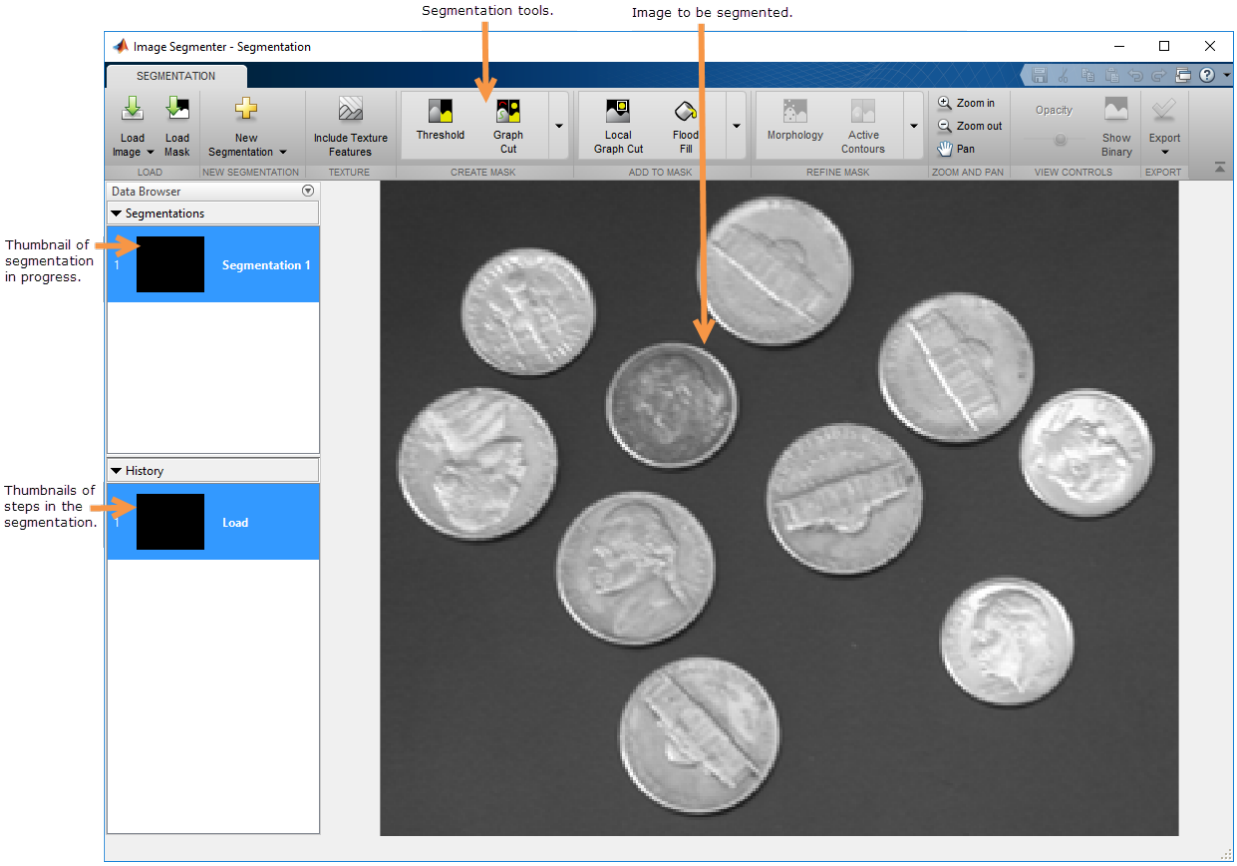


Note If you want to use Image Segmenter tools to refine an existing binary mask for an image that you have already segmented, first load the image. The Image Segmenter does not enable the **Load Mask** button until you load an image. After you load the image, use **Load Mask** to load the binary mask. See "Load Existing Binary Mask into Image Segmenter" on page 13-114 for more information.

To load an image from the workspace, select the **Load Image From Workspace** option. In the Import From Workspace dialog box, select the variable you created.



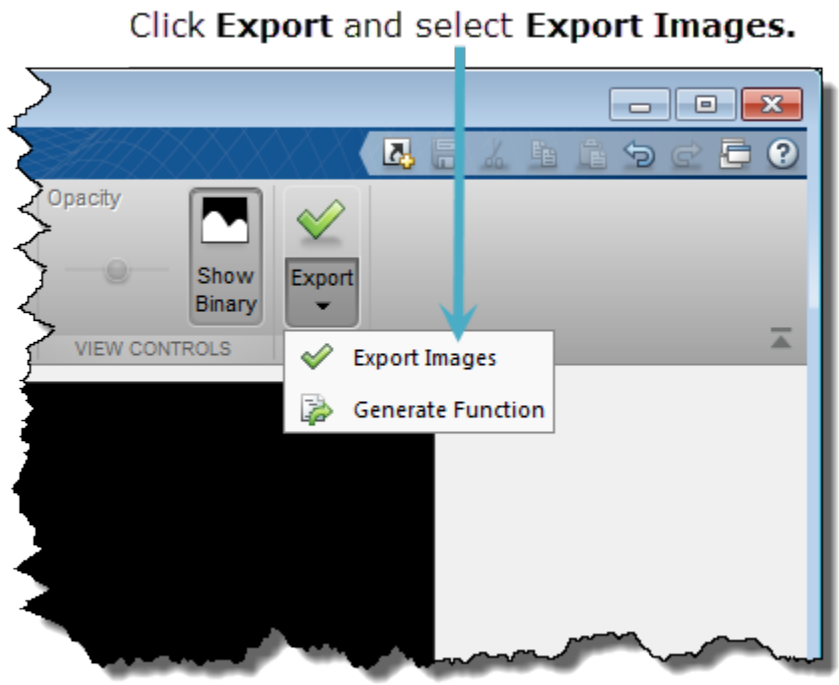
The app displays the image you loaded and displays a thumbnail of the result of the segmentation (the mask image) in the Data Browser. Initially, the thumbnail is completely black because you have not started yet. You can perform multiple segmentations using the app and each segmentation appears, with a thumbnail, in the Data Browser. To start a new segmentation click **New Segmentation**. The app displays the steps you take while creating the segmentation in the History part of the Data Browser.


Segmentation tools.
Image to be segmented.
Image Segmenter - Segmentation
SEGMENTATION
Load Image
Load Mask
New Segmentation
Include Texture Features
Threshold
Graph Cut
Local Graph Cut
Flood Fill
Morphology
Active Contours
Zoom in
Zoom out
Pan
Opacity
Show Binary
Export
LOAD
NEW SEGMENTATION
TEXTURE
CREATE MASK
ADD TO MASK
REFINE MASK
ZOOM AND PAN
VIEW CONTROLS
EXPORT
Data Browser
Segmentations
Thumbnail of segmentation in progress.
Segmentation 1
History
Thumbnails of steps in the segmentation.
Load

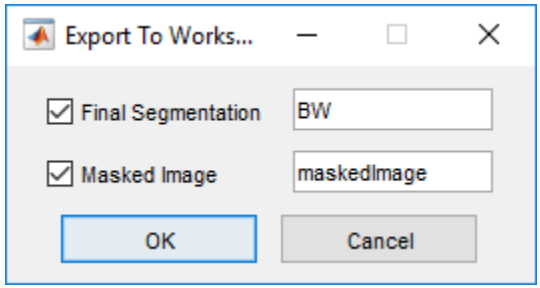
Save Mask Image Created Using Image Segmenter

This example shows how to save the binary mask image you created using the Image Segmenter app.

Click **Export** and select **Export Images**.



In the Export to Workspace dialog box, you can assign names to the final segmentation mask image and the segmented version of the original image.



To get the MATLAB code the app used to segment the image, click **Export** and select the **Generate Function** option. The app opens the MATLAB editor containing a function with the code required to segment the image.

The Image Segmenter Workflow

This example shows how to use an iterative approach to image segmentation, trying several different methods until you achieve the results you want.

For information about opening the app and loading an image, see “Open Image Segmenter App and Load Image” on page 13-97. This example uses an image from an MRI of a knee. Use the following code to read the image into the workspace and load it into the Image Segmenter.

```
I = dicomread('knee1');
knee = mat2gray(I);
imageSegmenter(knee)
```

Segment Using Threshold Technique in Image Segmenter

As a first attempt at the segmentation of the knee image, try thresholding.

Click **Threshold** in the Create Mask toolstrip group. The app displays the Threshold tab. You can choose between several thresholding methods: Global, Adaptive, and Manual. It is often to try each option and see which one works best with your image. Experiment with the optional parameters available with each option. For example, with the Manual option you can use a slider to specify the threshold value. The knee image does not have well-defined pixel intensity differences between foreground and background. Thresholding does not seem like the best choice to segment this image.

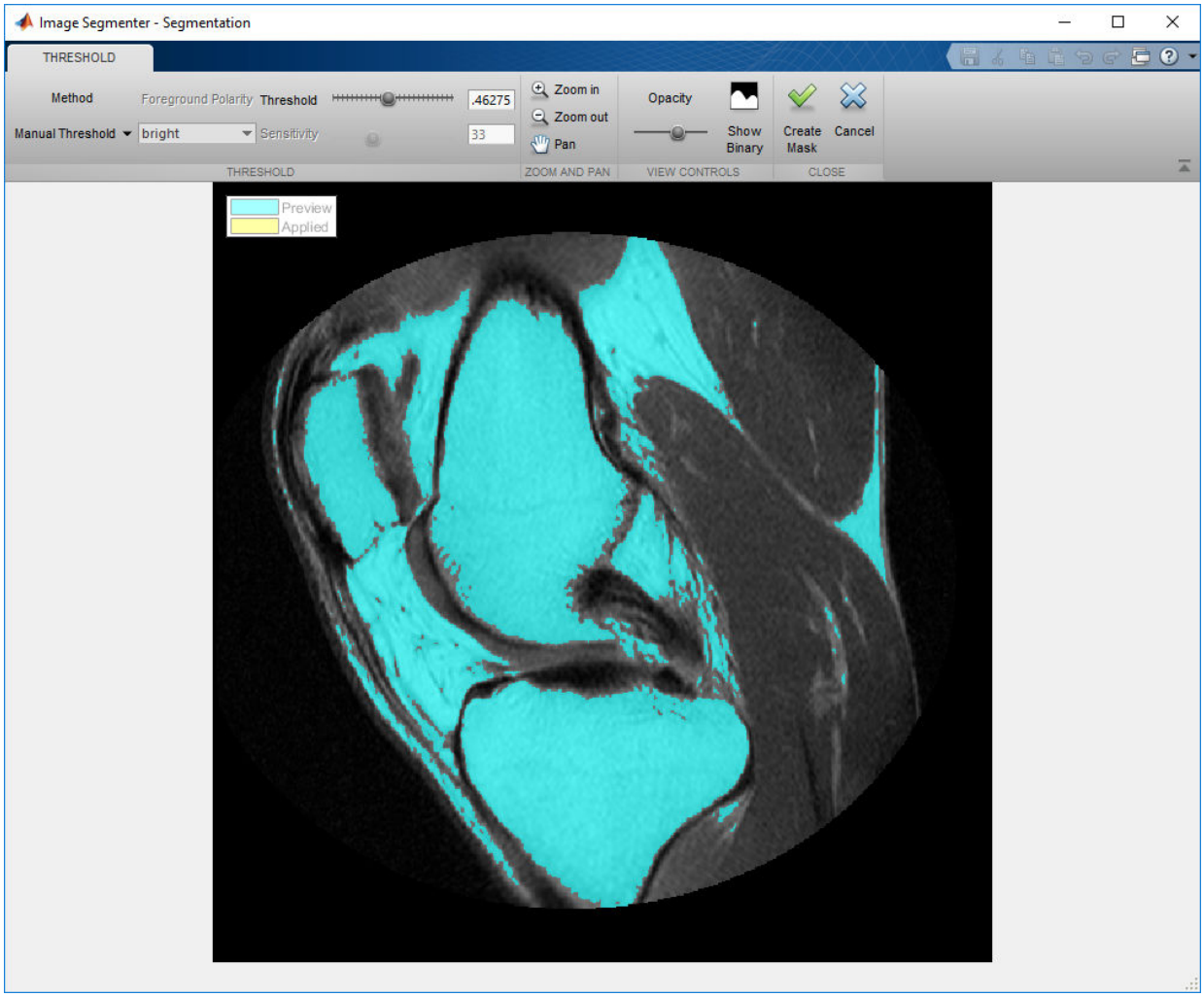


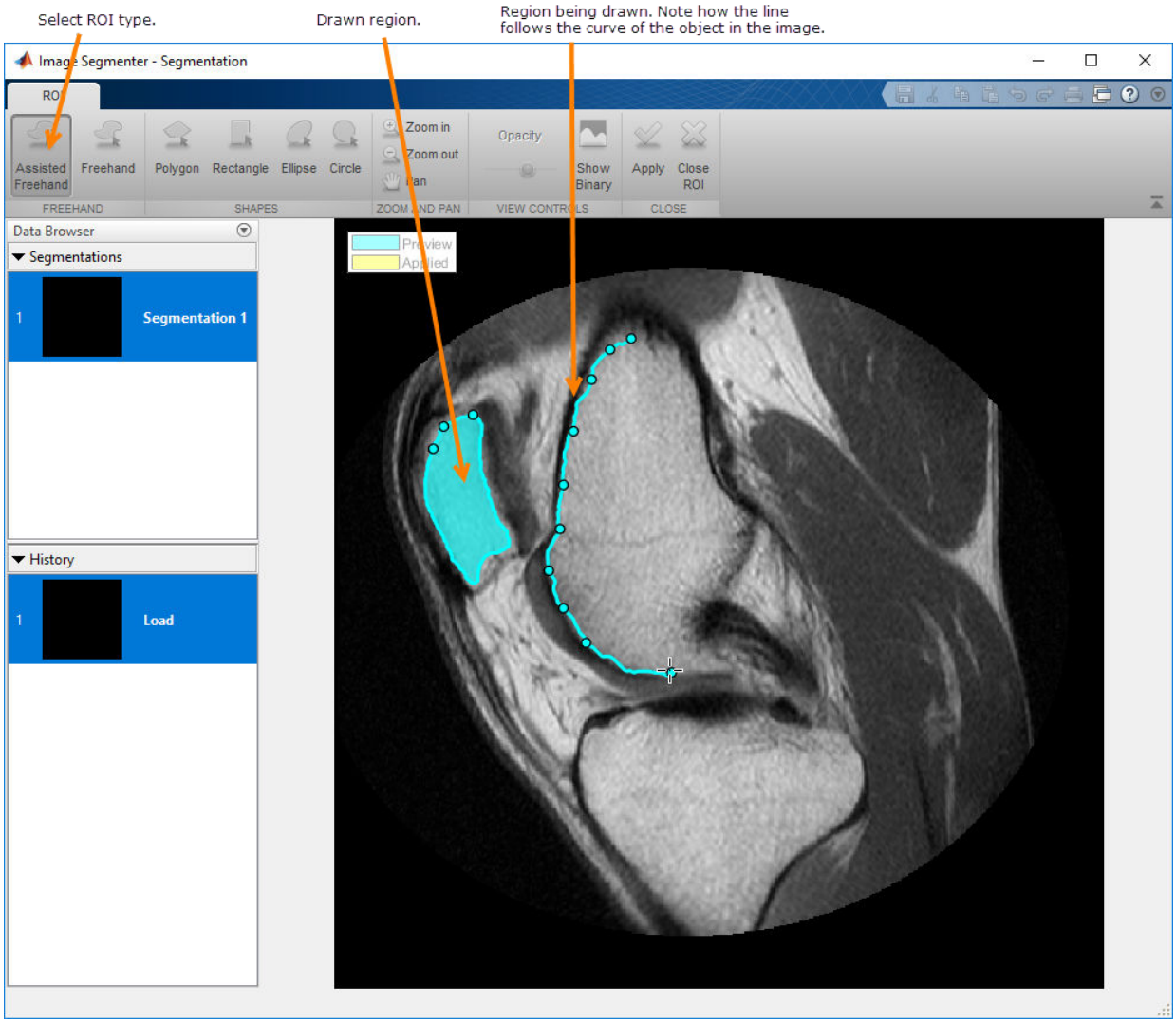
Image Segmenter - Segmentation
THRESHOLD
Method
Foreground Polarity
Threshold
.46275
Manual Threshold
bright
Sensitivity
33
THRESHOLD
Zoom in
Zoom out
Pan
ZOOM AND PAN
Opacity
Show Binary
VIEW CONTROLS
Create Mask
Cancel
CLOSE
Preview
Applied

Click **Cancel** to return to the main segmentation app window without accepting the result and try one of the other segmentation options. If you had wanted to keep the thresholded mask image, click **Create Mask**.

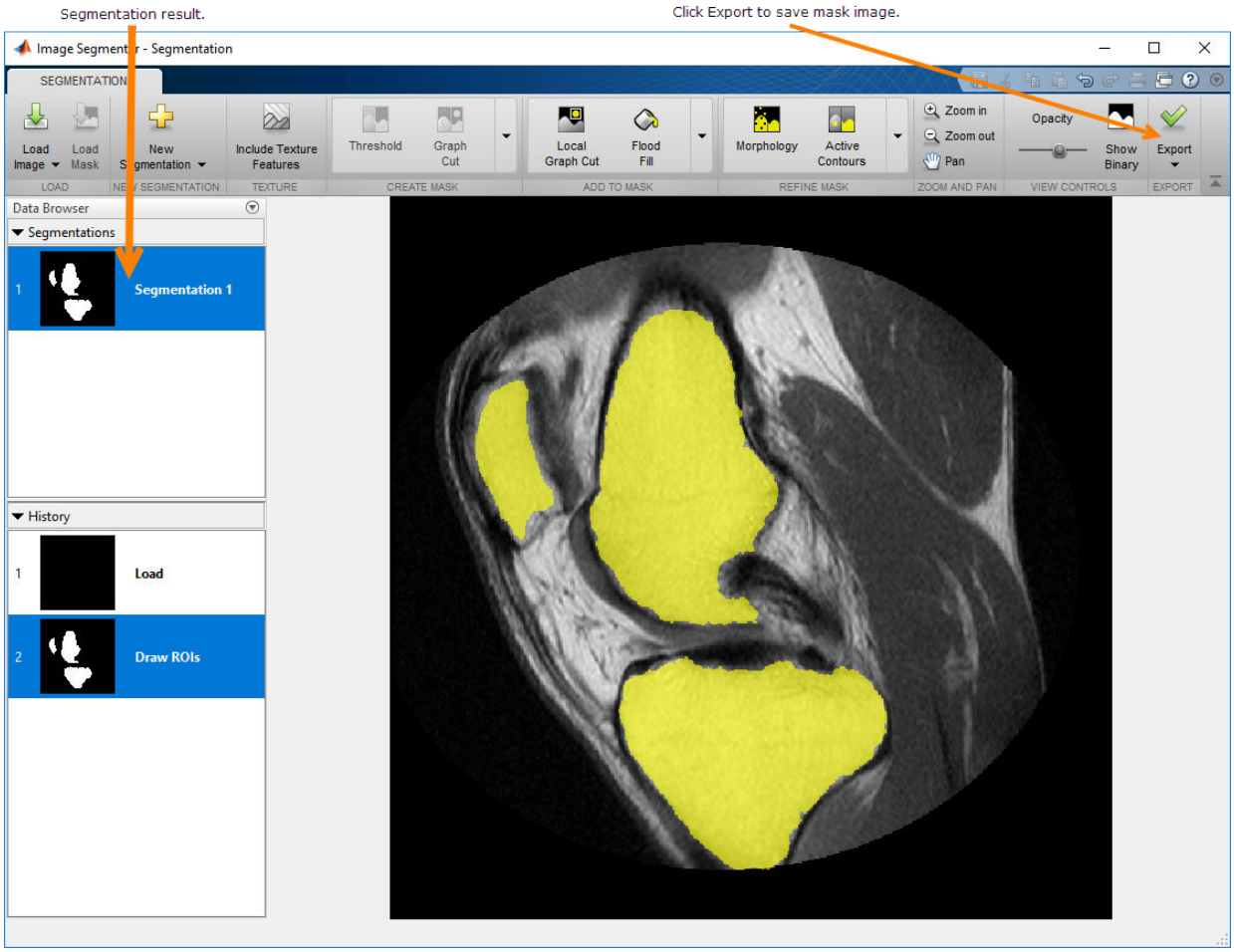
Segment By Drawing Regions in Image Segmenter

Another technique that you can try is to draw the regions that you want to include in the mask image. The Image Segmenter provides tools you can use to draw rectangles, ellipses, polygons, or freehand shapes.

Expand the Add to Mask group and click the **Draw ROIs** option. Select the type of ROI you want to draw and the cursor changes to the cross hairs shape when you move it over the image. Press the mouse button, and draw a shape over the image that outlines the object you want to segment. With the Assisted Freehand ROI option, which is preselected, you can draw a freehand shape that automatically follows edges in the underlying image to help you draw a more accurate ROI. As you draw, click the mouse to create waypoints for accurate drawing. After you finish drawing the ROI you can add waypoints to adjust the borders of the ROI.



Click **Apply** to save the mask image. To save this mask image, click **Export**.



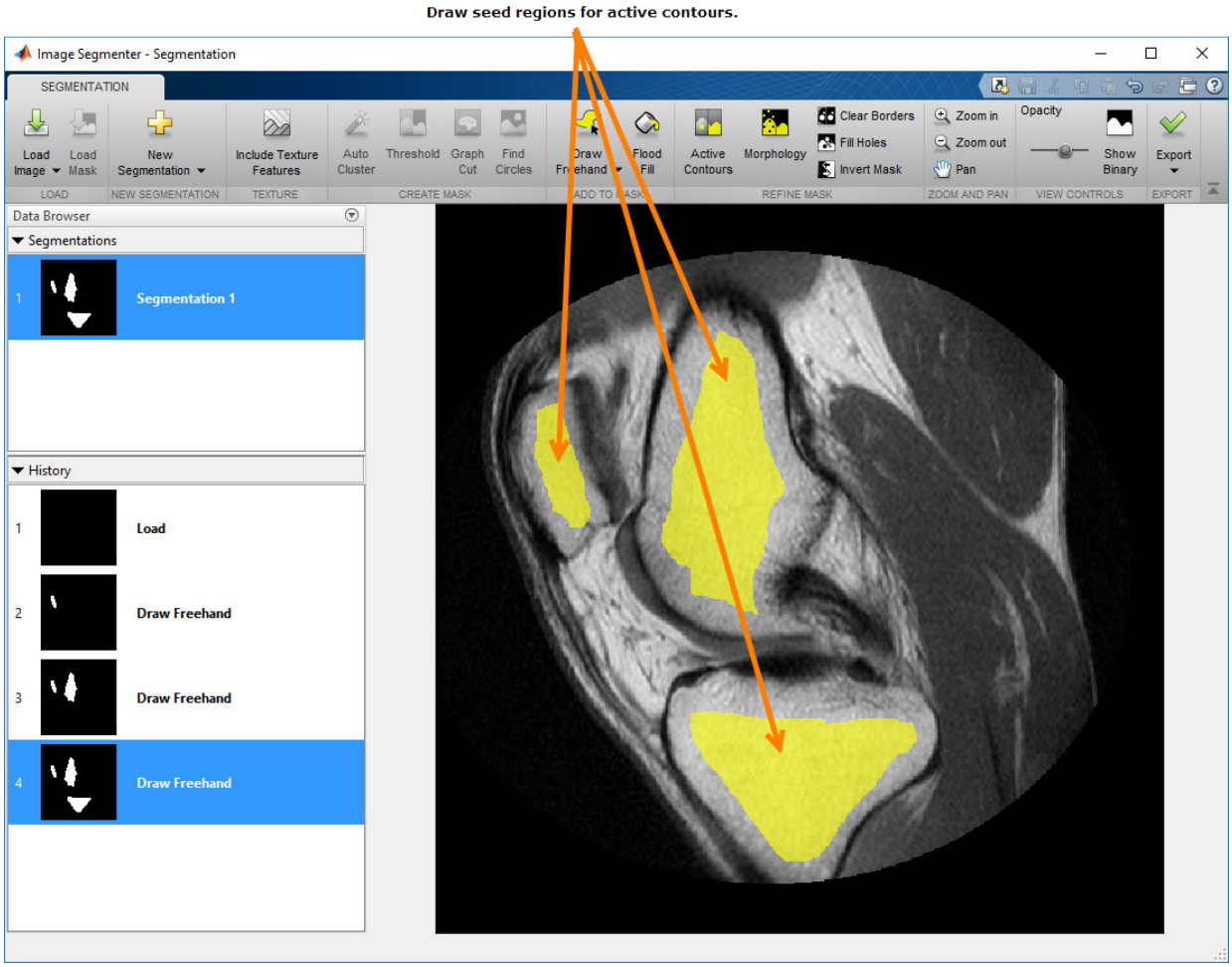
Use Active Contours to Refine Segmentation in Image Segmenter

This part of the example shows how to refine a binary mask created using the Image Segmenter. The Image Segmenter app provides several tools that you can use to fill holes, finish a rough approximation using Active Contours, and other operations. This example shows how to load an existing binary mask image. Active contours (snakes) is an automatic, iterative method where you mark locations in the image and active contours

grows (or shrinks) the regions identified in the image. To use active contours, you must have a rough segmentation already. The accuracy of this initial seed mask can impact the final result after active contours. You can use the **Include Texture Features** option with Active Contours.

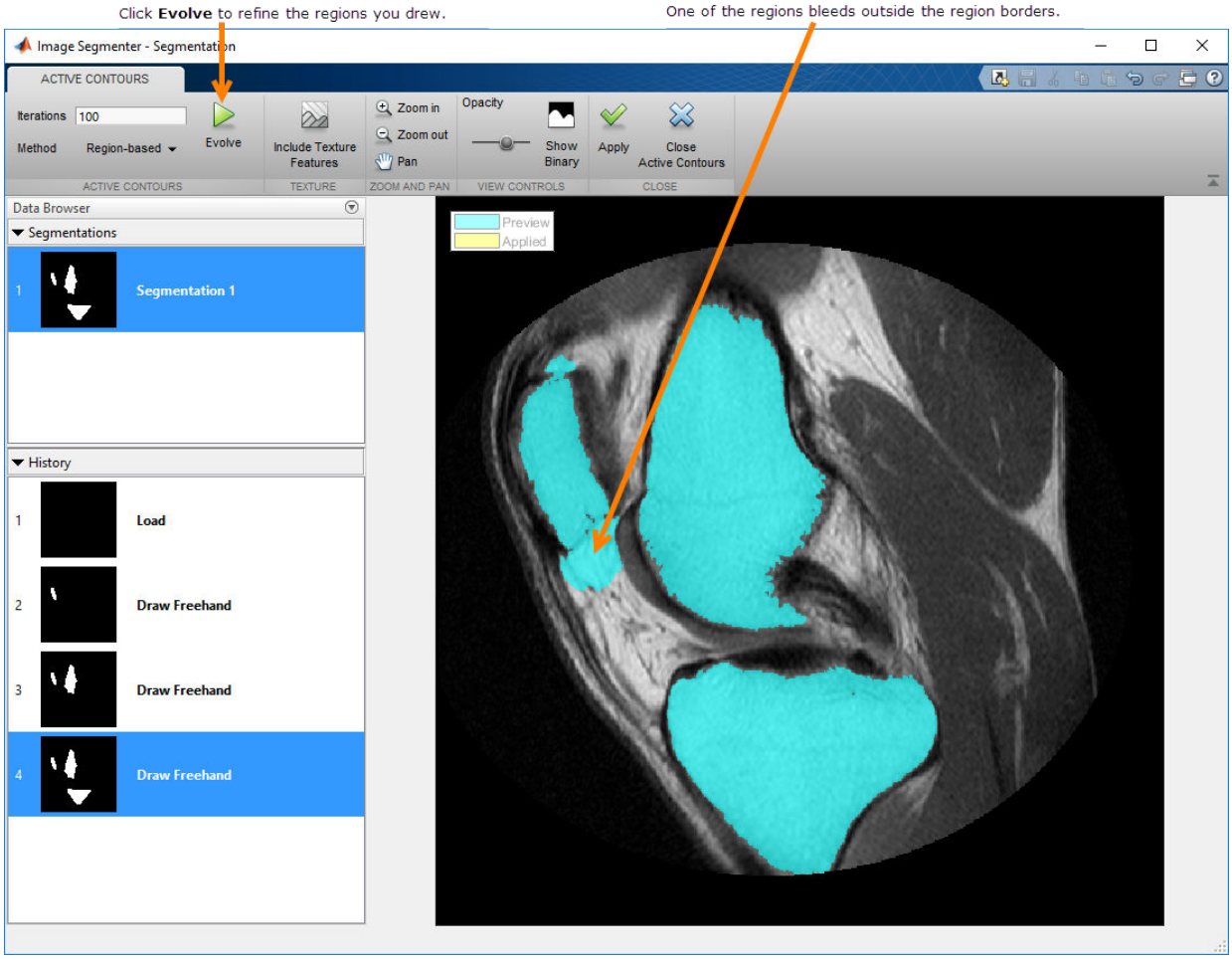
Draw seed shapes in the regions you want to segment. You can use the freehand regions drawn using the freehand tool (see “Segment By Drawing Regions in Image Segmenter” on page 13-105). To use an existing binary mask, first load the original image that you segmented. After loading the original image, click **Load Mask** and specify the segmented image result from the example “Segment By Drawing Regions in Image Segmenter” on page 13-105.

```
I = dicomread('knee1');
knee = mat2gray(I);
imageSegmenter(knee)
```

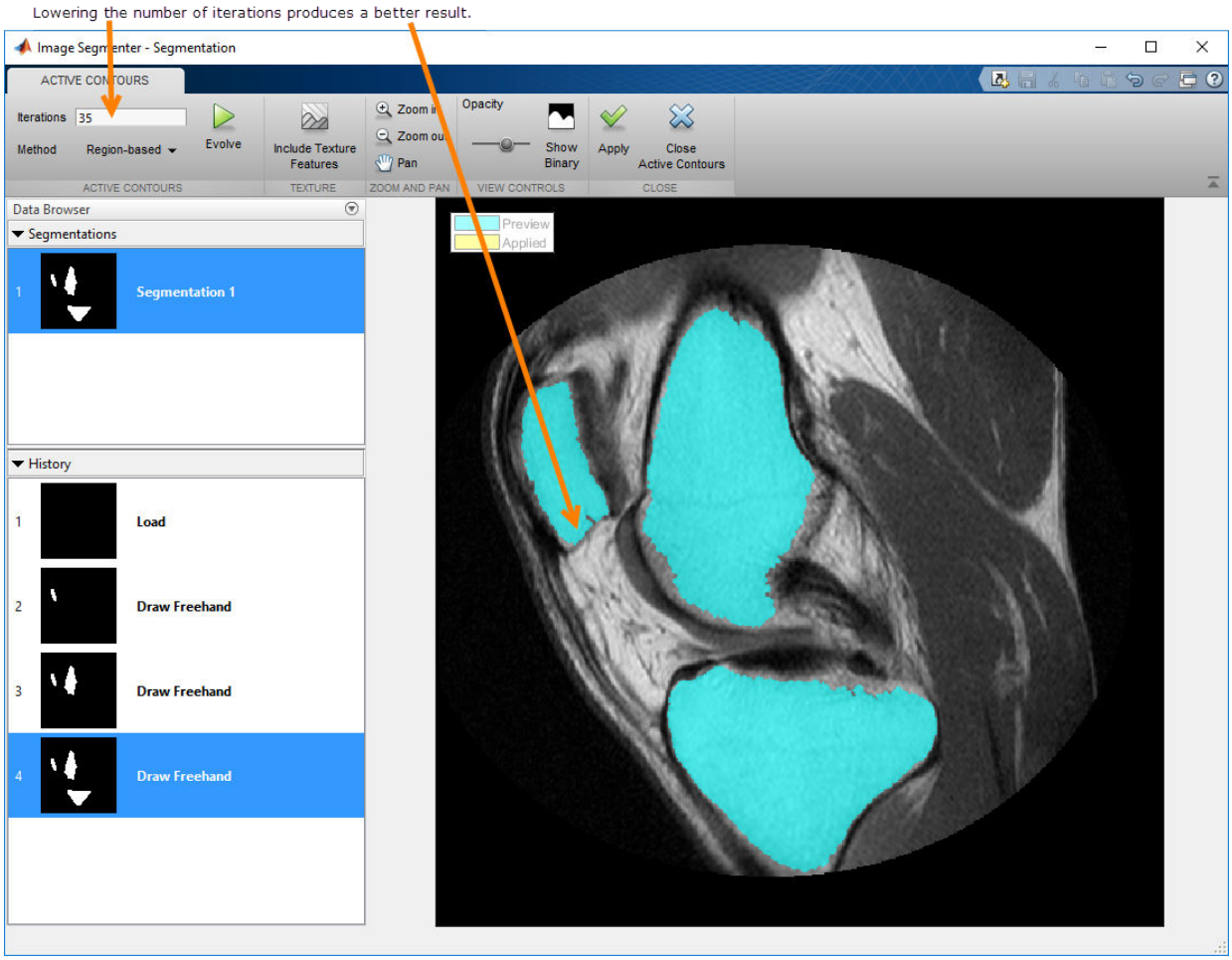


Click the **Active Contours** option. The Image Segmenter opens the Active Contours tab.

Click **Evolve** to use active contours to grow the regions to fill the objects to their borders. Initially, use the default active contours method (Region-based) and the default number of iterations (100). The Image Segmenter displays the progress of the processing in the lower right corner. 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.



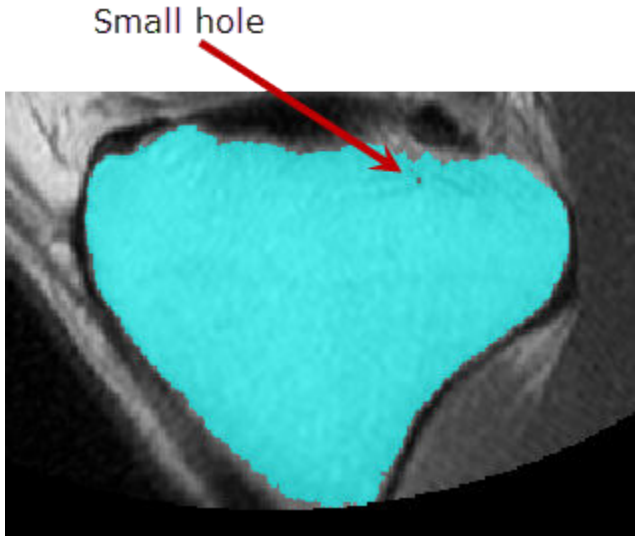
Repeat the active contours segmentation, this time changing the number of iterations. To redo the operation, change the number of iterations in the iterations box, specifying 35, and click **Evolve** again. When you are satisfied with the segmentation, click **Apply**. The color of the regions changes from blue to yellow, indicating that the changes have been applied. To see how to remove the small imperfection in the one of the regions, see “Use Morphology to Refine Mask in Image Segmenter” on page 13-111.



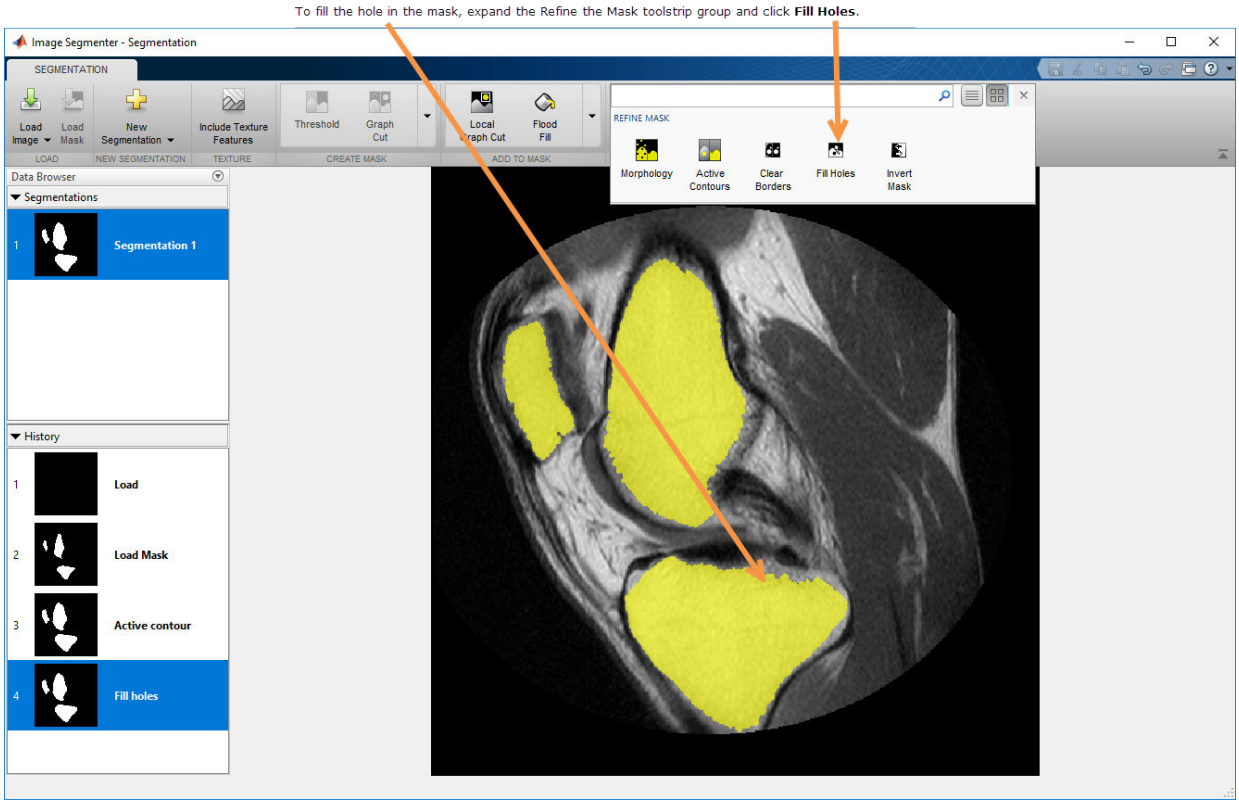
Use Morphology to Refine Mask in Image Segmenter

The segmentation mask image you created in the segmentation step (“The Image Segmenter Workflow” on page 13-103) might have slight imperfections that you'd like to fix. The Image Segmenter includes morphological tools, such as dilation and erosion, on the **Morphology** tab, and options like **Fill Holes** and **Clear Borders** on the **Segmentation** tab. You can use these tools to improve your mask image.

Upon close examination, one of the mask regions (created in “The Image Segmenter Workflow” on page 13-103) contains a small hole.



Click **Fill Holes** and the Image Segmenter fills the hole in the region.



See Also

Image Segmenter | `activecontour`

Related Examples

- "Image Segmentation Using the Image Segmenter App" on page 13-91

Load Existing Binary Mask into Image Segmenter

Load an existing image mask into the Image Segmenter. When you first load an image into the Image Segmenter, and before you start defining objects, the Image Segmenter enables the **Load Mask** option. You can load an existing segmentation mask image into the Image Segmenter and refine the segmentation using tools in the refine **Refine Mask** toolbar group. The segmentation mask image must be a logical image the same size as the image you are segmenting.

For example, you might have previously created a mask of an RGB image in the Color Thresholder and you want to refine the segmentation. Bring the RGB image into the Image Segmenter and then load the mask image. You can then use the tools available in the Image Segmenter to refine the mask.

See Also

Image Segmenter

Related Examples

- “Image Segmentation Using the Image Segmenter App” on page 13-91
- “The Image Segmenter Workflow” on page 13-103

Segmentation Using Graph Cut in Image Segmenter

This example shows how to use the Graph Cut option in the Image Segmenter app to segment an image. Graph cut is a semiautomatic segmentation technique that you can use to segment an image into foreground and background elements. Graph cut segmentation does not require good initialization. You draw lines on the image, called scribbles, to identify what you want in the foreground and what you want in the background. The Image Segmenter segments the image automatically based on your scribbles and displays the segmented image. You can refine the segmentation by drawing more scribbles on the image until you are satisfied with the result.

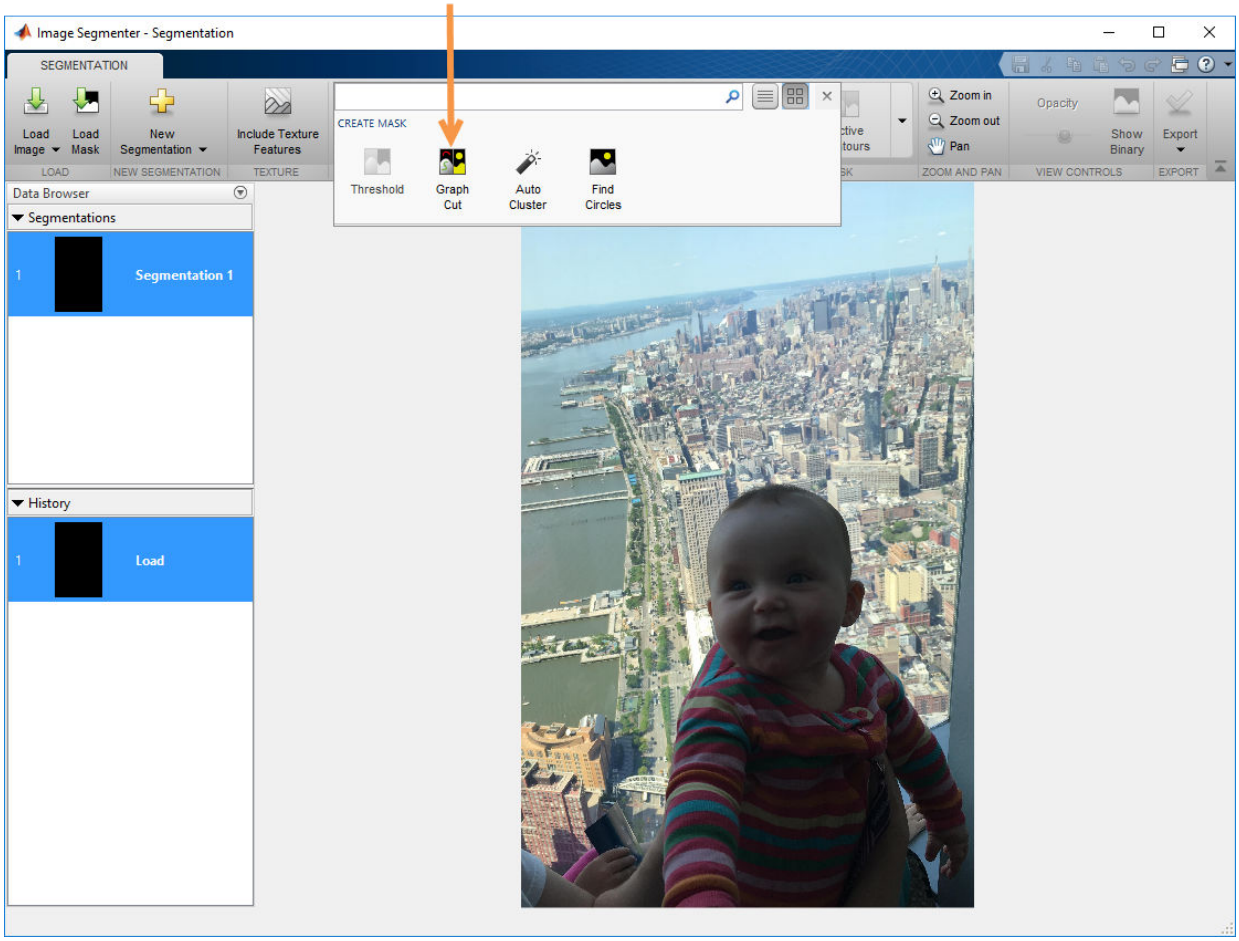
The Graph Cut technique applies graph theory to image processing to achieve fast segmentation. The techniques creates a graph of the image where each pixel is a node connected by weighted edges. The higher the probability that pixels are related the higher the weight. The algorithm cuts along weak edges, achieving the segmentation of objects in the image. The Image Segmenter uses a particular variety of the Graph Cut algorithm called lazysnapping. For information about another segmentation technique that is related to graph cut, see “Segmentation Using Local Graph Cut (Grabcut) in Image Segmenter” on page 13-122.

Read an image into the MATLAB workspace and load it into the Image Segmenter app. For more information about opening the Image Segmenter app and loading an image, see “Open Image Segmenter App and Load Image” on page 13-97.

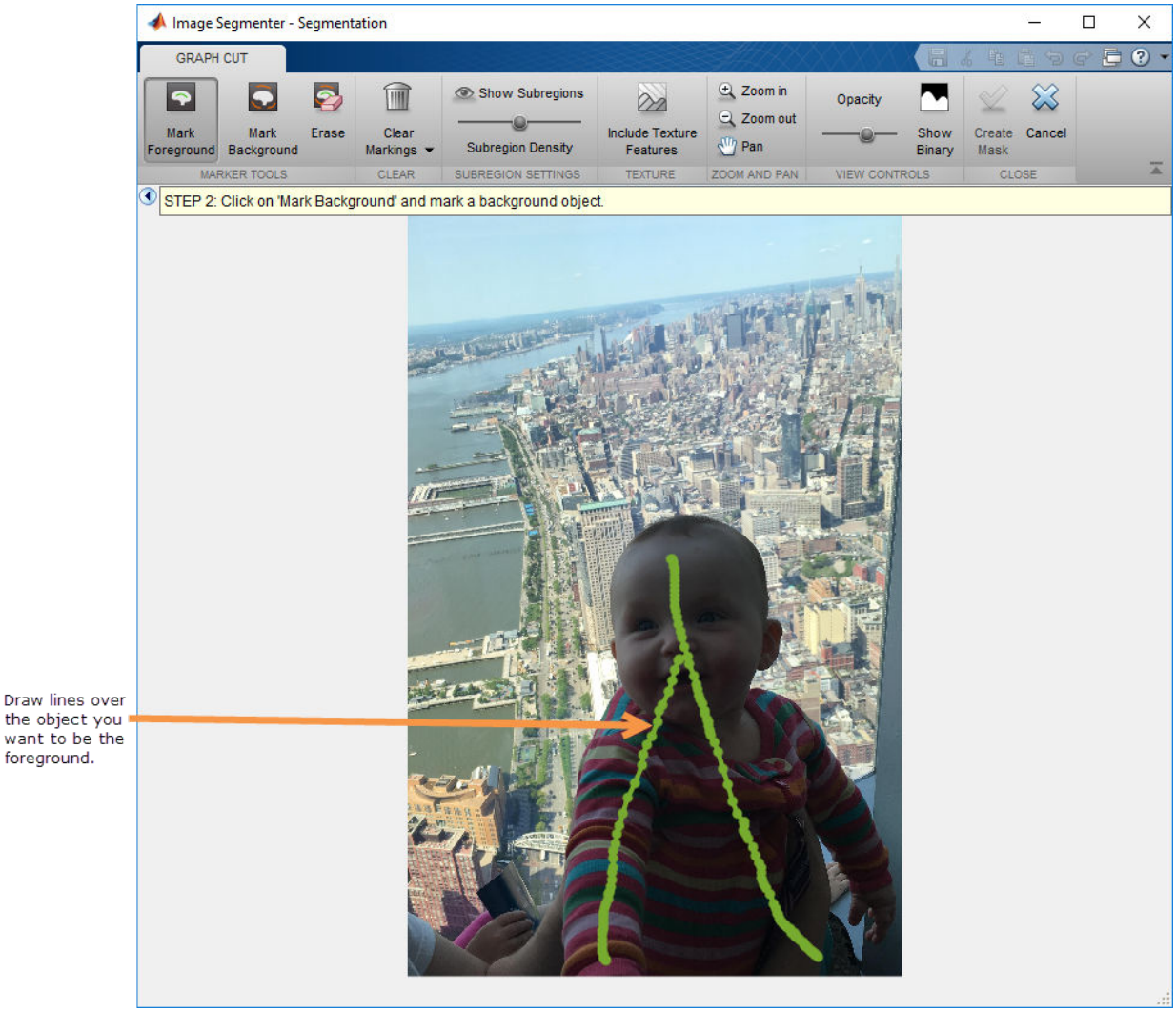
```
b = imread('baby.jpg');
imageSegmenter(b)
```

Expand the Create Mask toolstrip group and click **Graph Cut**. The Image Segmenter app opens the Graph Cut tab.

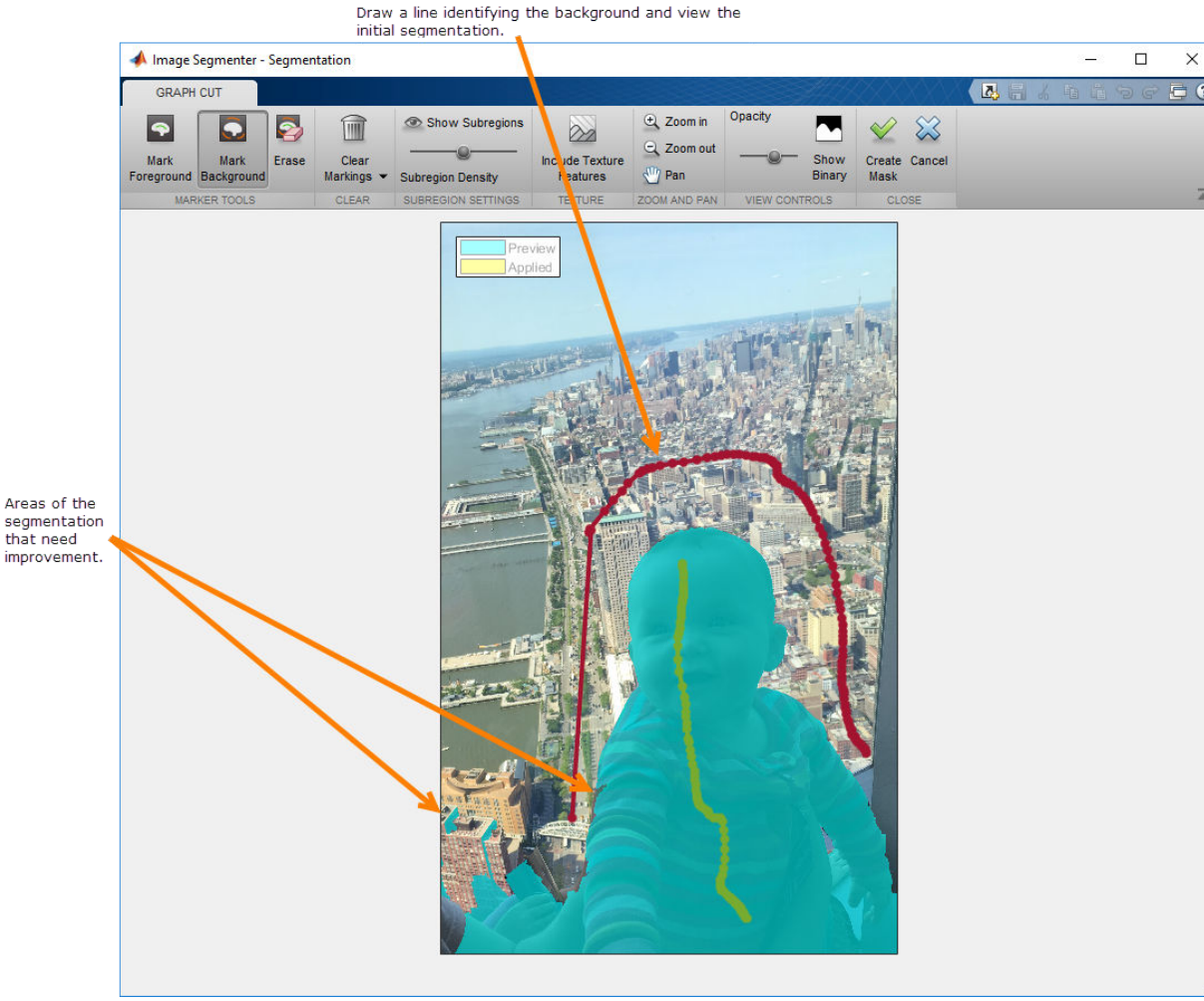
Expand the Create Mask group and select Graph Cut.



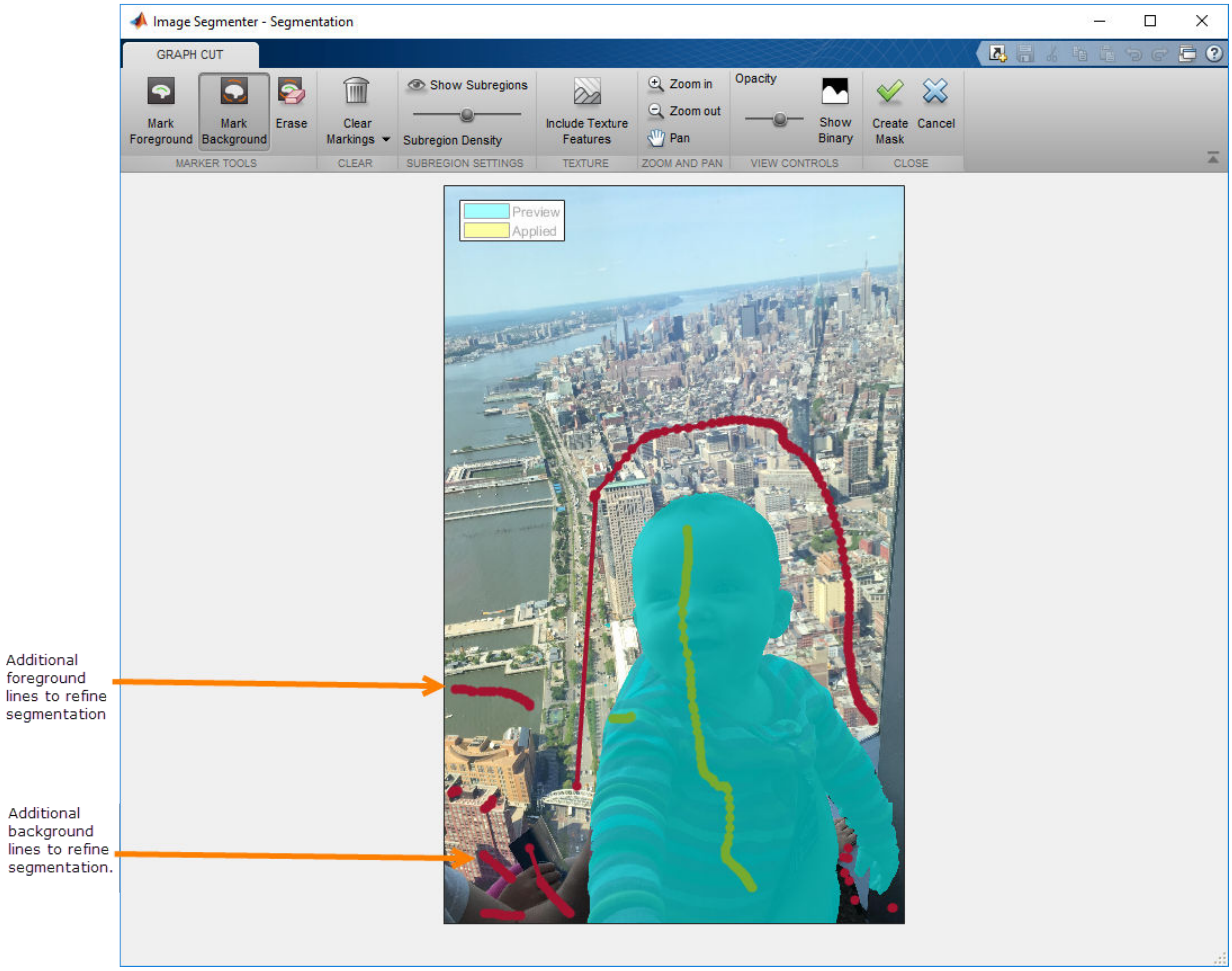
As a first step, you must mark the elements of the image that you want to be in the foreground. (When the Graph Cut opens, the **Mark Foreground** option is preselected.) Marking an object is simply drawing a line (also called a scribble) over the object. When you draw a line, try to include all the different values in the object. You can draw as many separate lines as you like. If you are not satisfied with the lines you draw, you can always edit them. Click **Erase** and move the cursor over any part of the line you want to remove. If you have drawn many lines and want to start over, click **Clear Markings**.



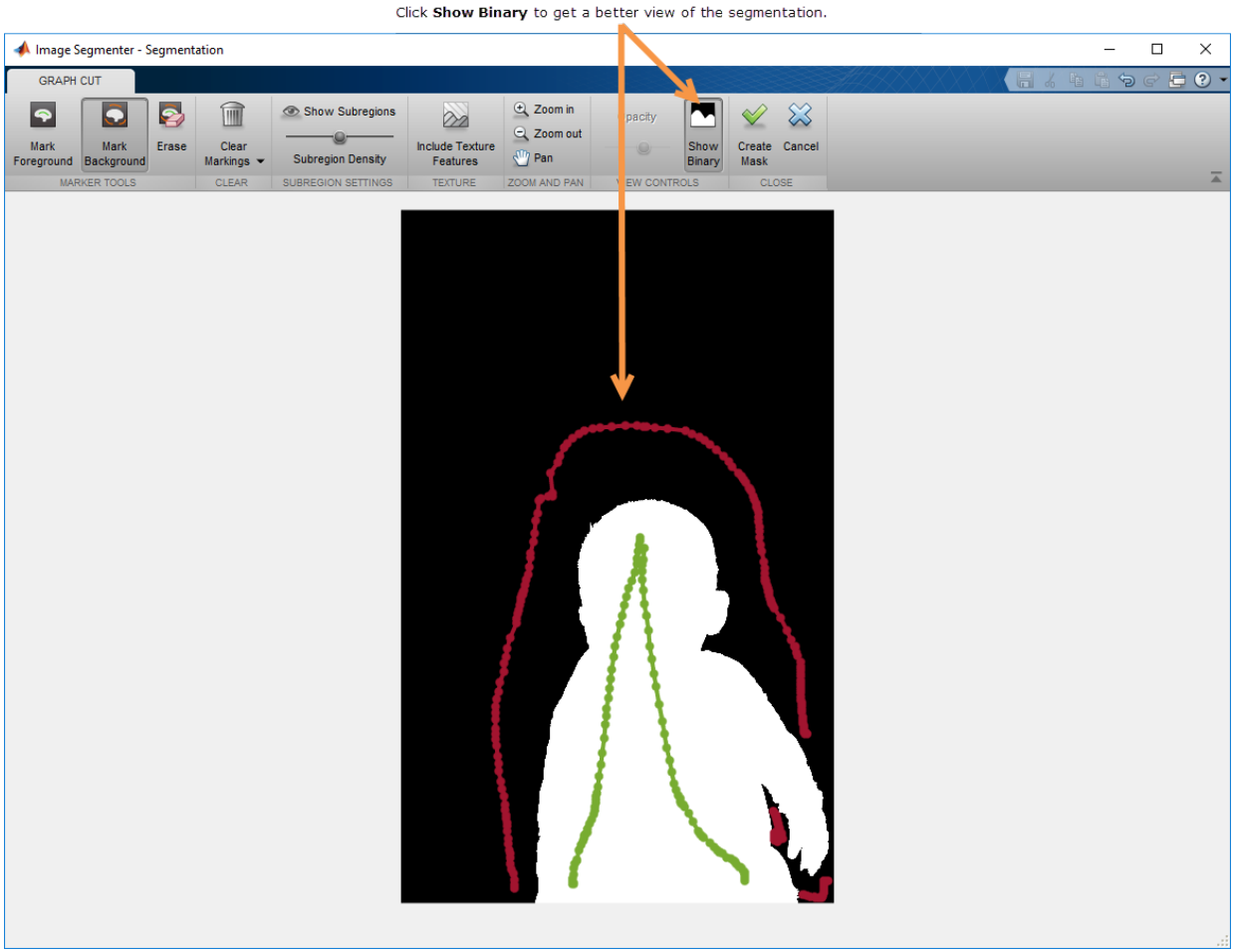
Next, mark the elements of the image you want to be in the background. Again, simply draw a line over the image. When you finish drawing the line, the Image Segmenter immediately performs the segmentation (shown in blue).



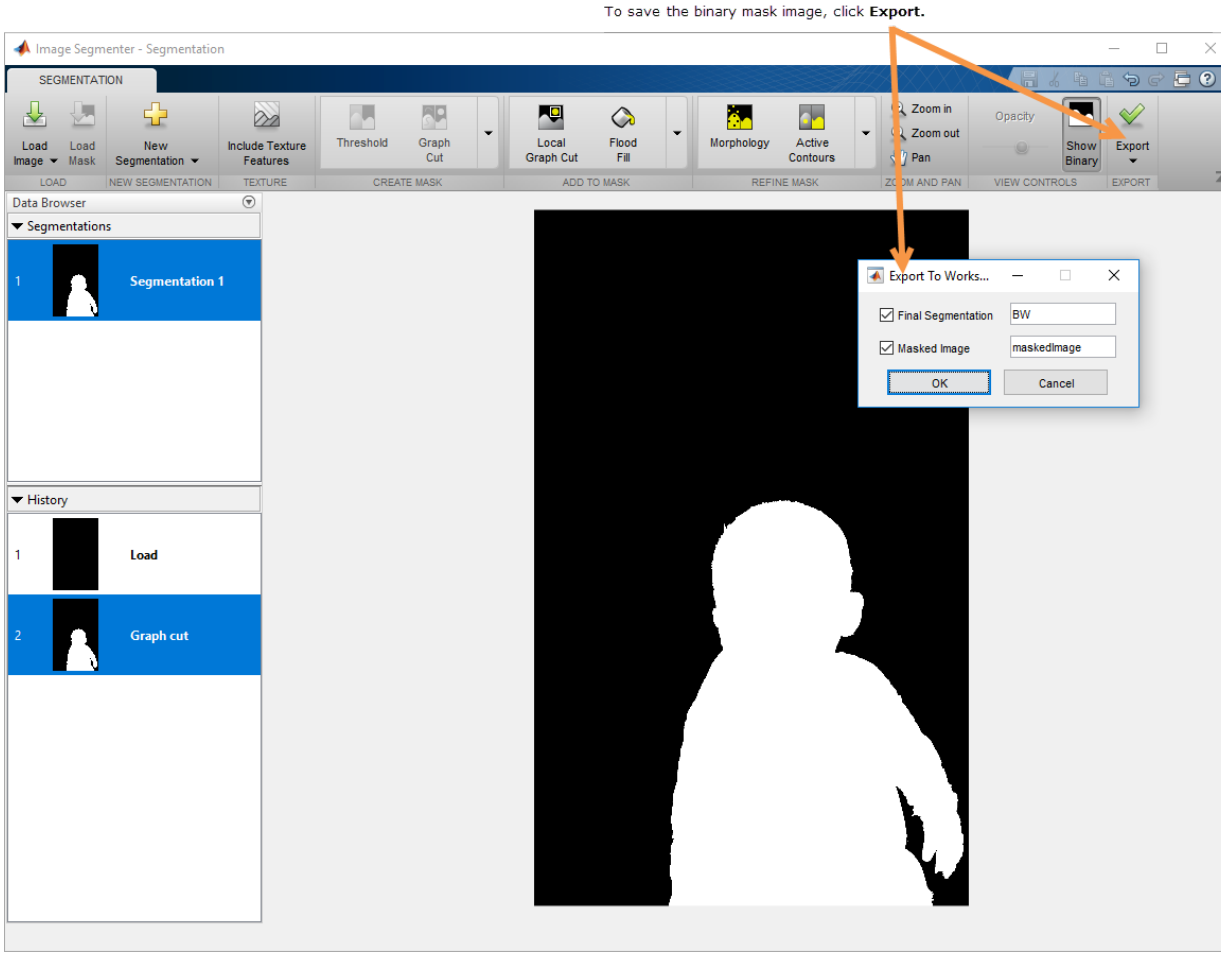
Continue drawing foreground and background lines to improve the segmentation. For example, the baby's left hand (lower right corner of the image) is not well-defined. There are also several spots on the baby's right arm that need to be included in the foreground. To fix these problems, draw additional foreground and background lines on these parts of the image.



To get a better look at the segmentation, click **Show Binary**.



When you are satisfied with the segmentation, click **Create Mask**. The Image Segmenter closes the Graph Cut tab. You can use tools in the Refine the Mask toolstrip group to perfect the mask image, such as Morphology and Active Contours. To save the mask image, click **Export**. You can also use the Export option to obtain the code used for the segmentation. For more information about saving the mask image, see “Save Mask Image Created Using Image Segmenter” on page 13-101.



See Also

Image Segmenter | `lazysnapping`

Related Examples

- "Image Segmentation Using the Image Segmenter App" on page 13-91

Segmentation Using Local Graph Cut (Grabcut) in Image Segmenter

This example shows how to segment an image using the Local Graph Cut option (known as grabcut) in the Image Segmenter app. Like the Graph Cut option, Local Graph Cut is a semiautomatic segmentation technique that you can use to segment an image into foreground and background elements. With Local Graph Cut, you first draw a region-of-interest around the object you want to segment. The Image Segmenter segments the image automatically based on the contents of the ROI. To get a good initial segmentation, make sure the ROI you draw completely surrounds the object, leaving a small amount of space between the object and the ROI boundary. Ensure that the object is completely inside the ROI.

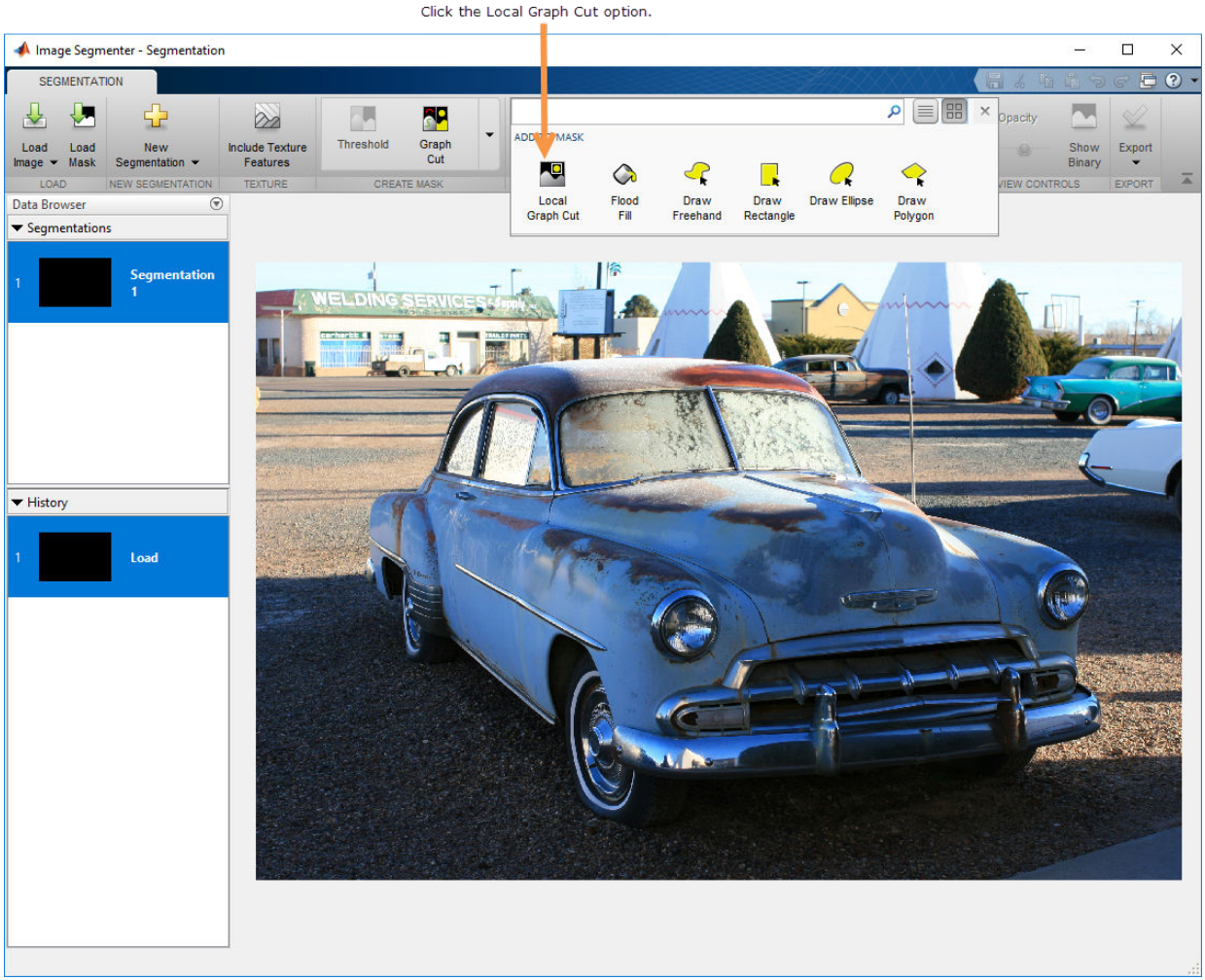
Then, as with Graph Cut, you can refine the automatic segmentation by drawing lines, called scribbles, on the image inside the ROI. The lines you draw identify what you want in the foreground and what you want in the background. The Local Graph Cut option only segments elements within the boundaries of the ROI.

The Local Graph Cut technique, similar to the Graph Cut technique, applies graph theory to image processing to achieve fast segmentation. The algorithm creates a graph of the image where each pixel is a node connected by weighted edges. The higher the probability that pixels are related the higher the weight. The algorithm cuts along weak edges, achieving the segmentation of objects in the image. For information about the Graph Cut technique, see “Segmentation Using Graph Cut in Image Segmenter” on page 13-115.

Read an image into the MATLAB workspace and load it into the Image Segmenter. For more information about loading images into the app, see “Open Image Segmenter App and Load Image” on page 13-97.

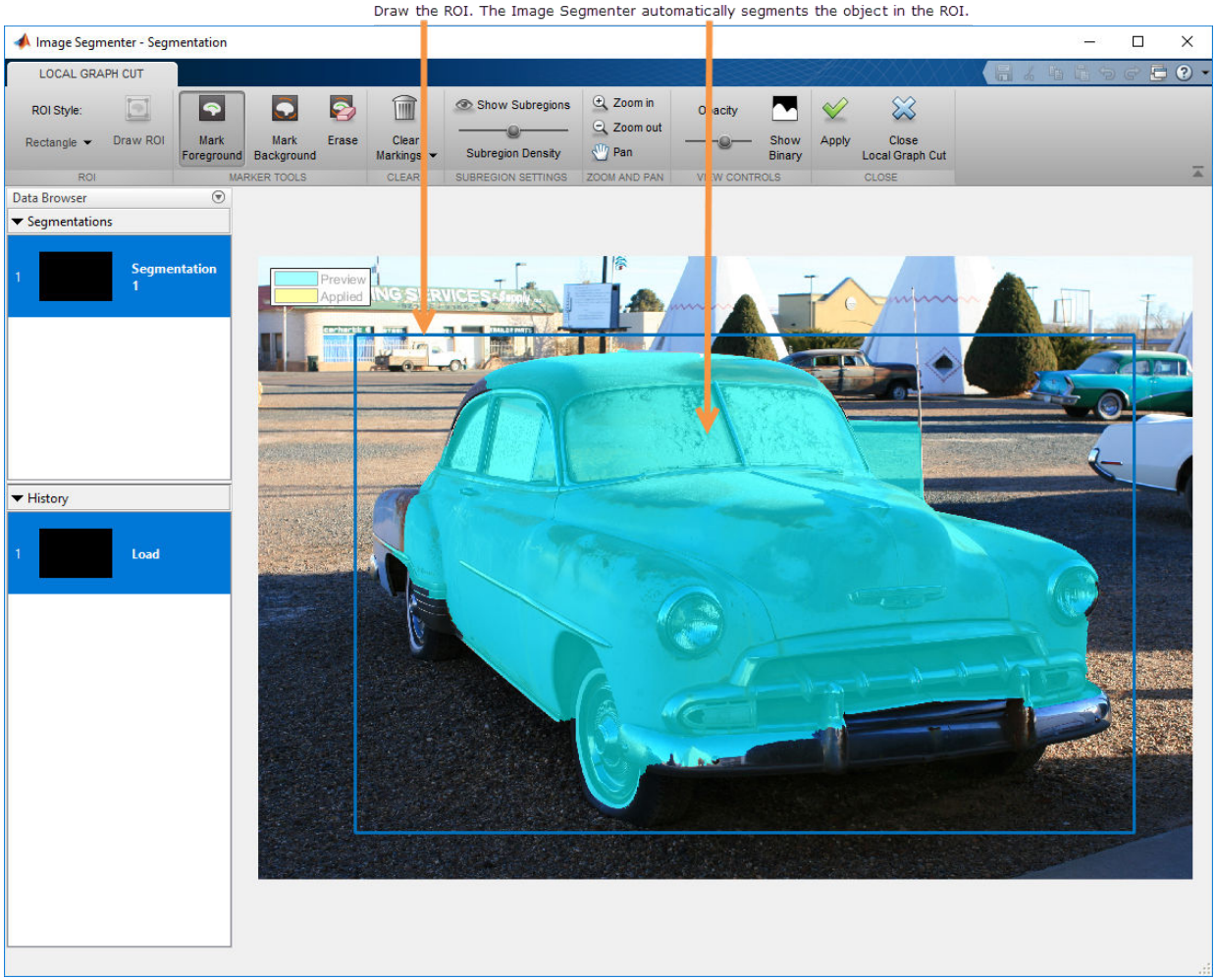
```
car = imread('car2.jpg');
imageSegmenter(car)
```

Select **Local Graph Cut** in the Add to Mask section of the toolstrip. You can expand this group to see all of the options available.

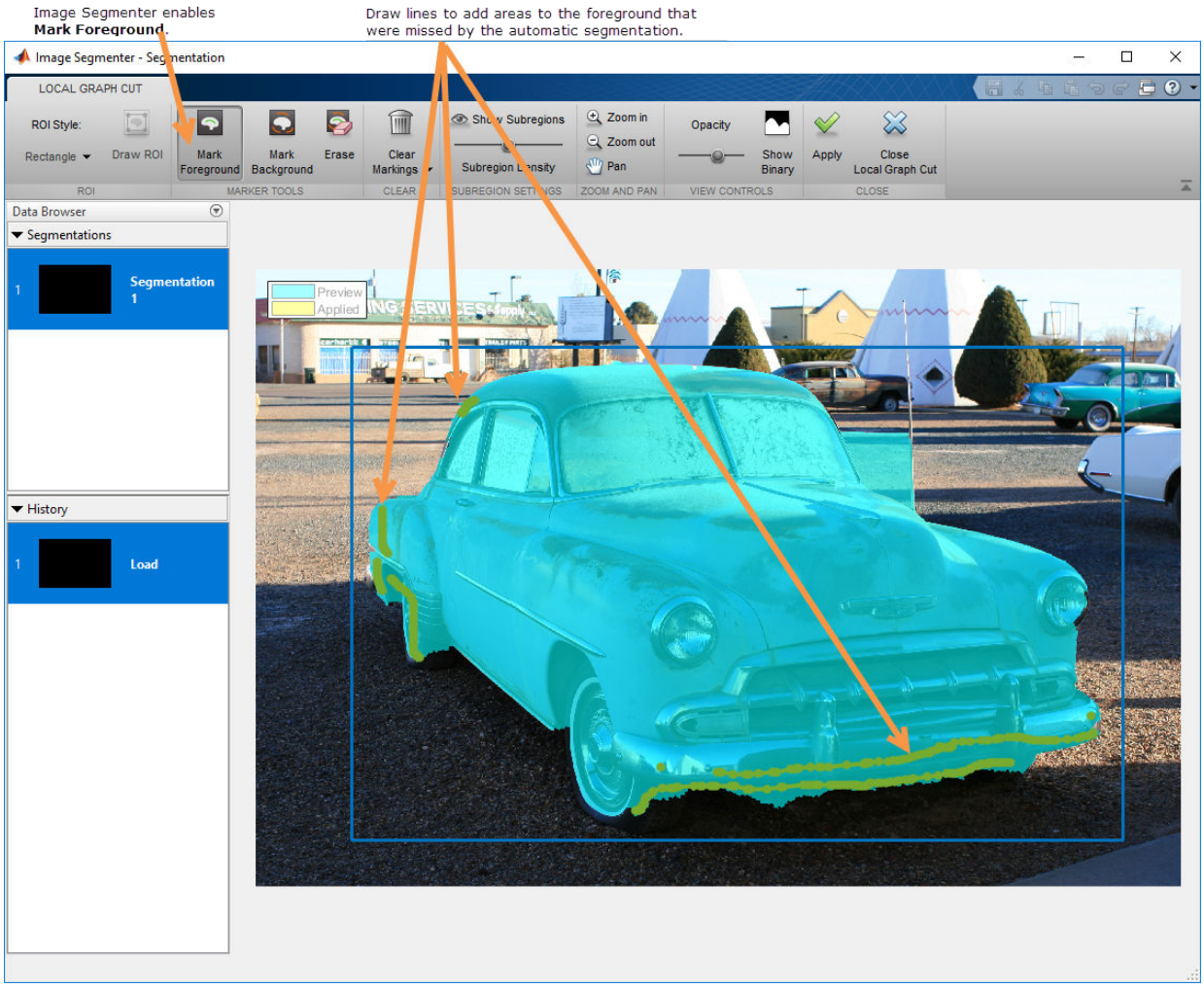


As a first step, draw an ROI around the object in the image that you want to segment. When the Image Segmenter opens the Local Graph Cut tab, the **Draw ROI** button is enabled. Position the cursor over the image and draw an ROI that encompasses the entire object you want to segment. To get a good initial segmentation, make sure the ROI you draw completely surrounds the object, leaving a small amount of space between the object and the ROI boundary. Make sure the object you want to segment is completely inside the ROI.

You can choose to draw a rectangle or a polygon ROI. Use the **ROI Style** menu to choose. To draw a rectangle, position the cursor over the image and then click and drag.To draw a polygon, click and drag the mouse, create a vertex at each click. Double-click to finish the polygon. As soon as you finish the ROI, the Image Segmenter segments the object in the ROI. If you are not satisfied with the shape you drew, you can always edit it. Right-click on the ROI and choose **Delete**.

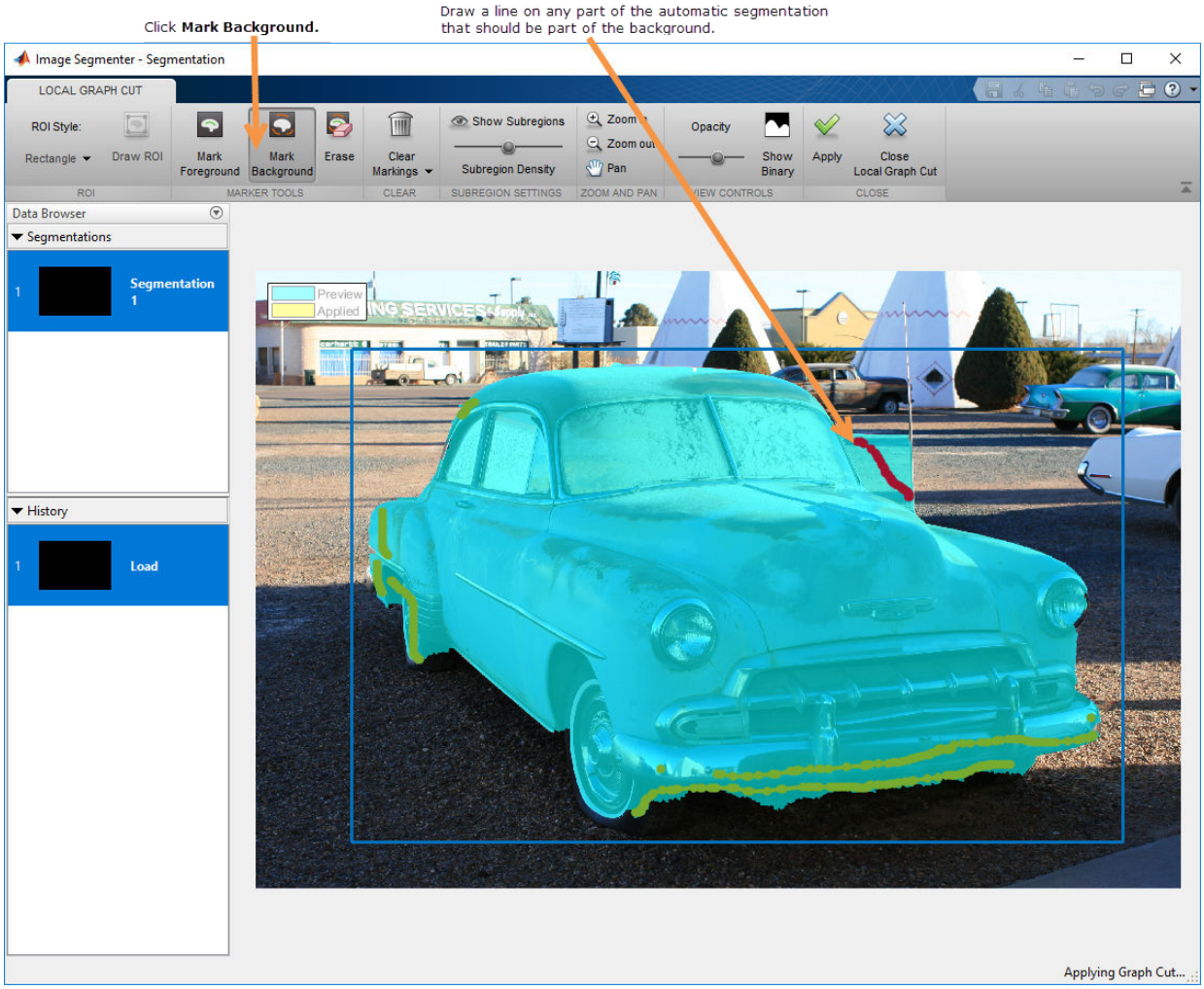


Next, draw scribbles to mark any parts of the foreground that weren't included in the automatic segmentation. After you draw the ROI, the Image Segmenter activates the **Mark Foreground** button automatically. Using the mouse, draw lines to identify parts of the image that you want to be included in the foreground.

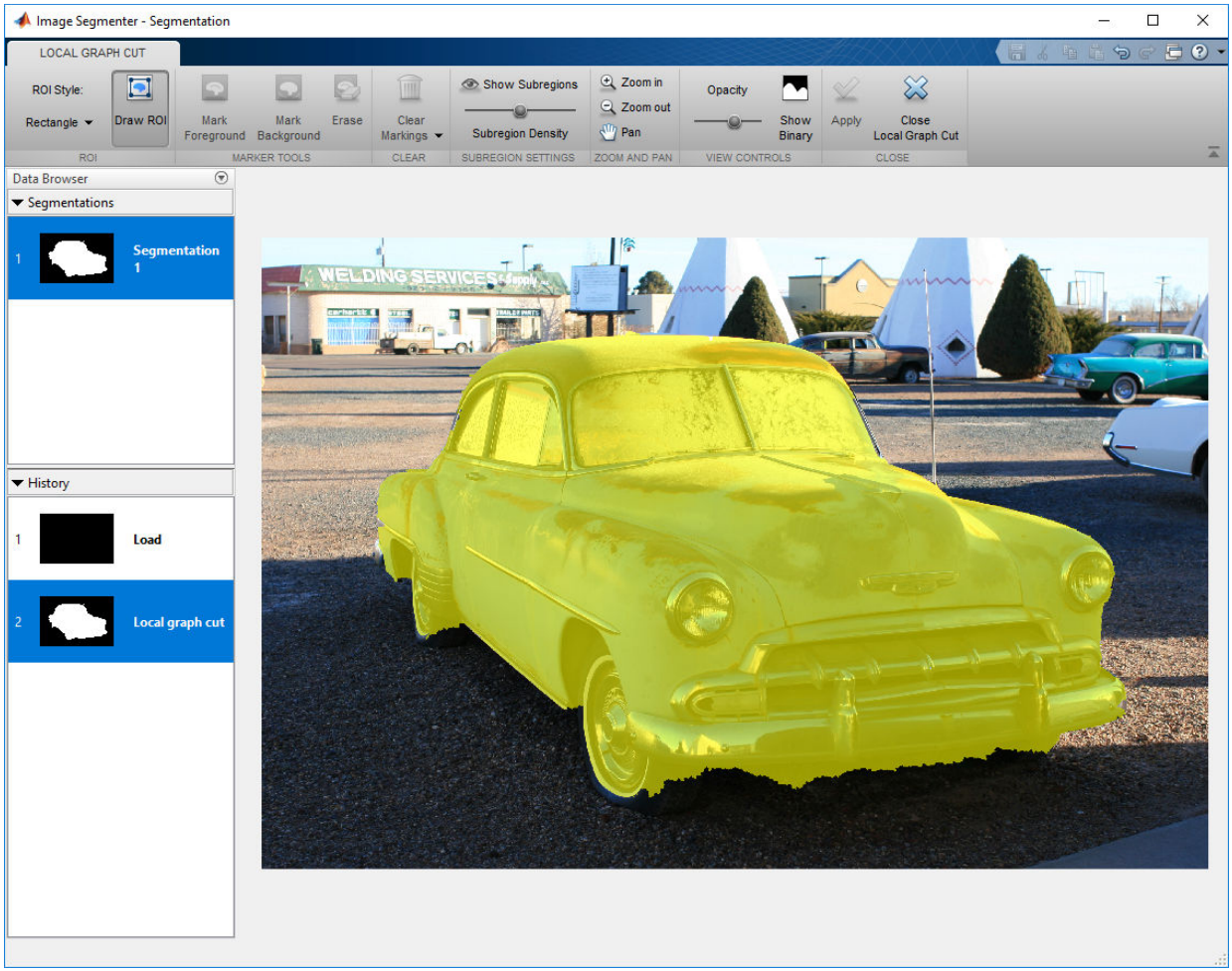


Continue by marking elements that you want to be parts of the background. Click **Mark Background** and draw lines inside the ROI to identify which parts are the background.

You can trim parts of the automatic segmentation that should be part of the background by drawing a line through them.



When you are satisfied with the segmentation, click **Apply**. The Image Segmenter changes the color of the segmented part of the image to yellow.



To view the mask image, click **Show Binary**. You can also view the binary mask image in the main Segmentation tab. To return to the main Image Segmenter app, click **Close Local Graph Cut**.

When you are done segmenting the image, you can save the binary mask, using the **Export** option. You can also obtain the code used for the segmentation. For more

information about saving the mask image, see “Save Mask Image Created Using Image Segmenter” on page 13-101.

See Also

Image Segmenter | `grabcut`

Related Examples

- “Image Segmentation Using the Image Segmenter App” on page 13-91

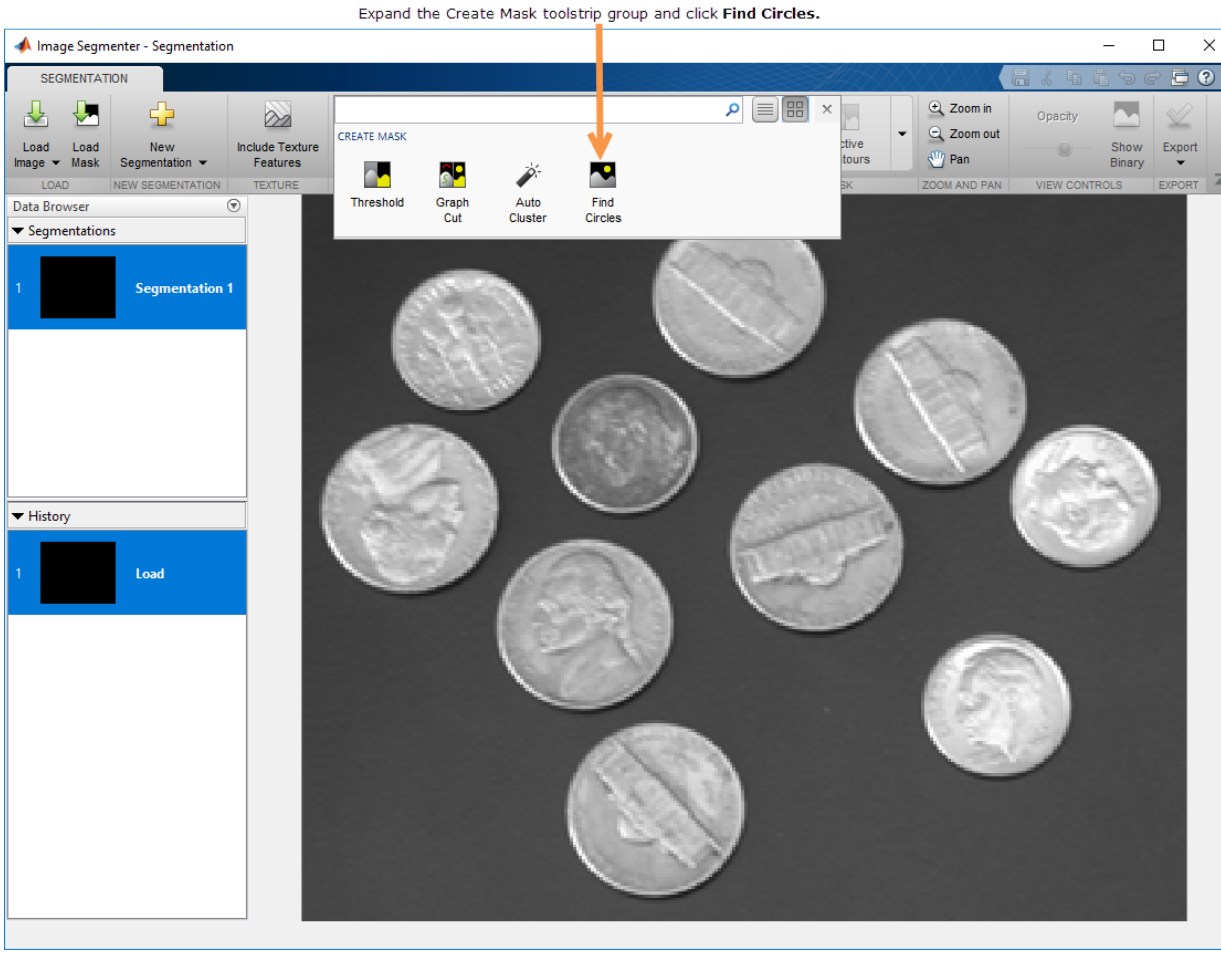
Segmentation Using Find Circles in Image Segmenter

This example shows how to use the **Find Circles** option in the Image Segmenter app to segment an image. The Find Circles option is an automatic segmentation technique that you can use to segment an image into foreground and background elements. The Find Circles option does not require initialization.

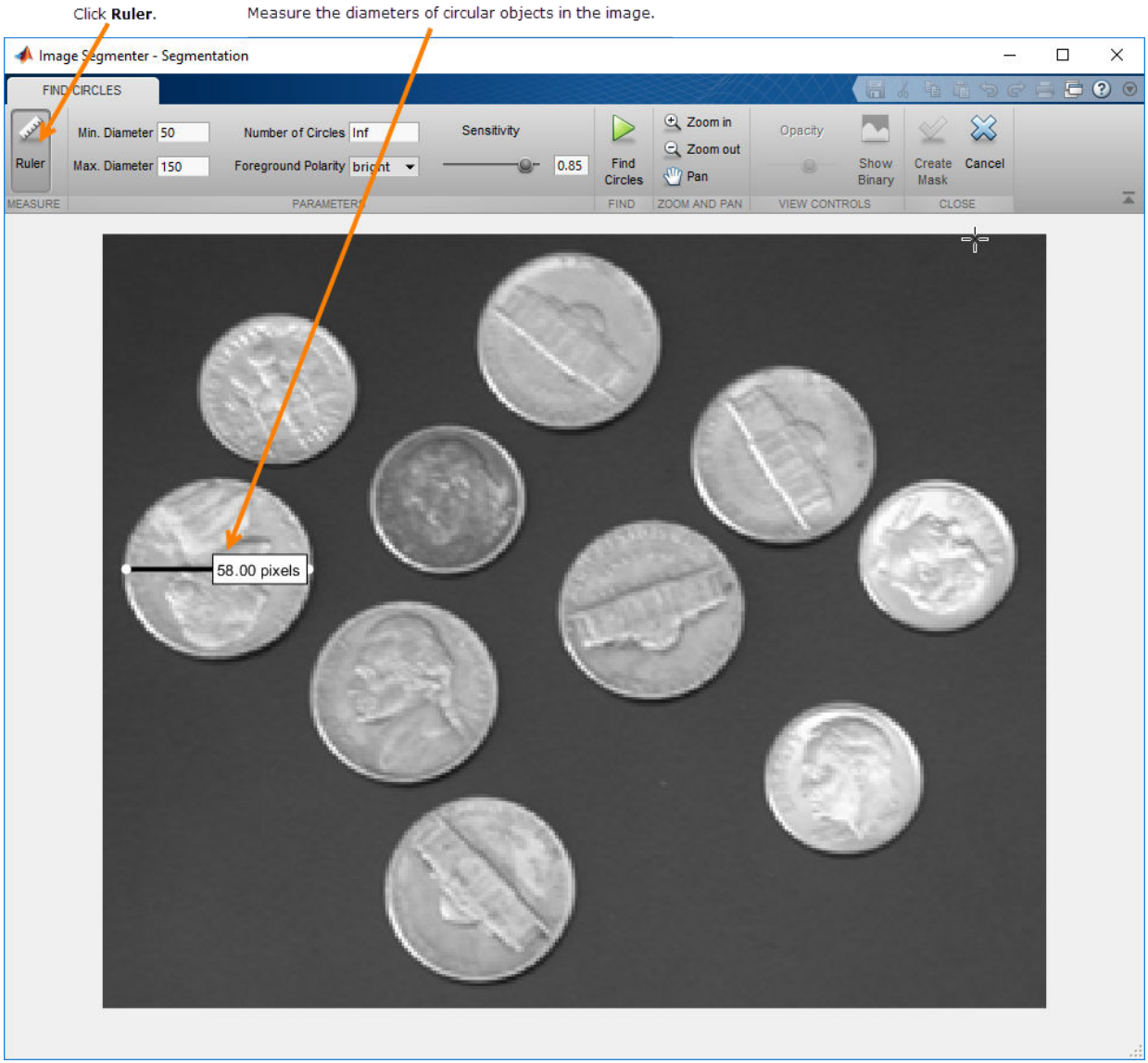
Read an image into the MATLAB workspace and load it into the Image Segmenter app. For more information about opening the Image Segmenter app and loading an image, see “Open Image Segmenter App and Load Image” on page 13-97.

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

Expand the Create Mask toolstrip group and click **Find Circles**. The Image Segmenter opens the Find Circles tab.

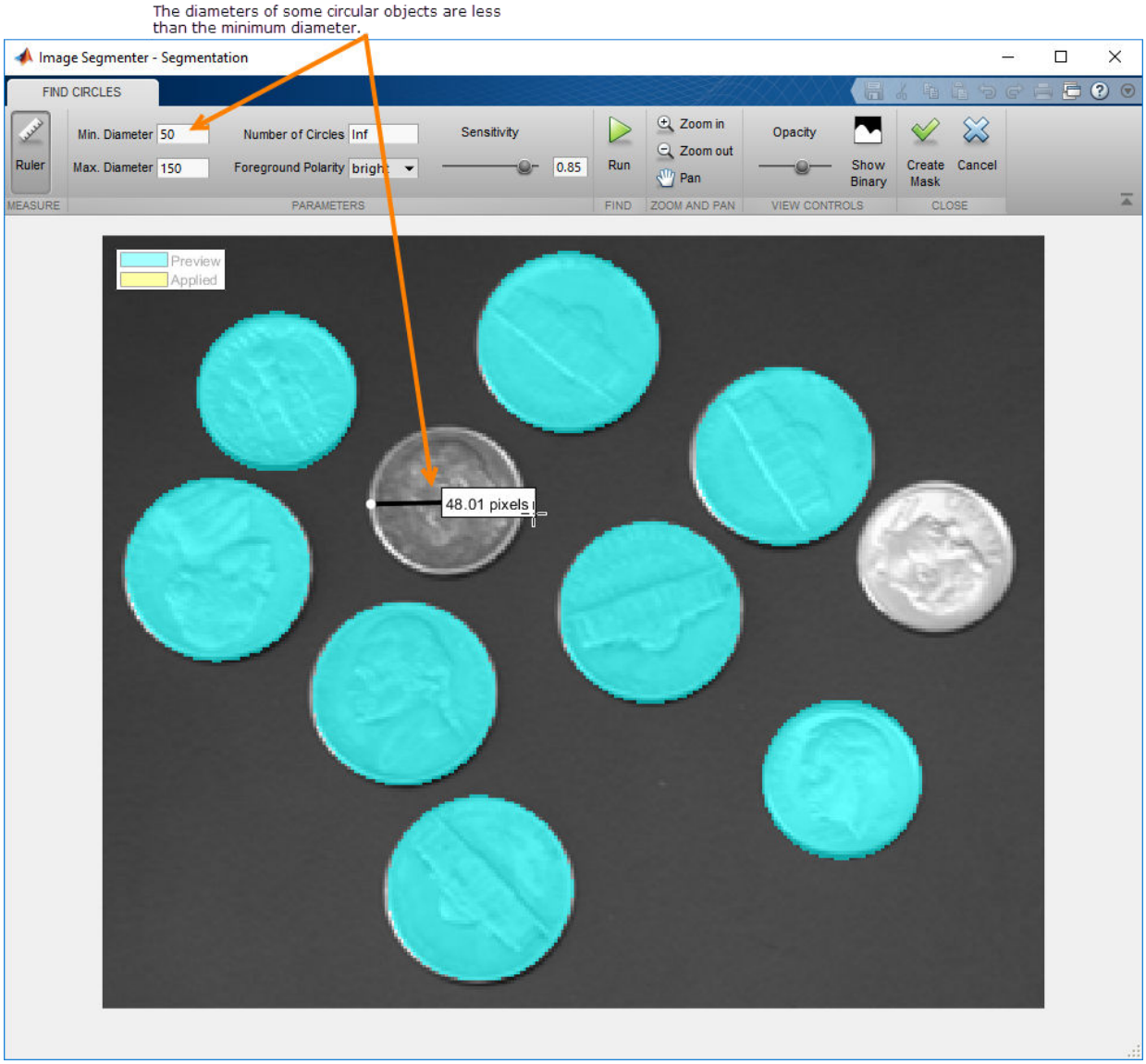


In the Find Circles tab, first click **Ruler** and measure the diameters of circles in the image to determine the range of sizes. To find circles, you must specify the lower and upper bounds on the diameters. You can also specify the number of circles you want to detect. Set the values in the **Min. Diameter** and **Max. Diameter** fields to values that you think include all the objects, 50 and 150.

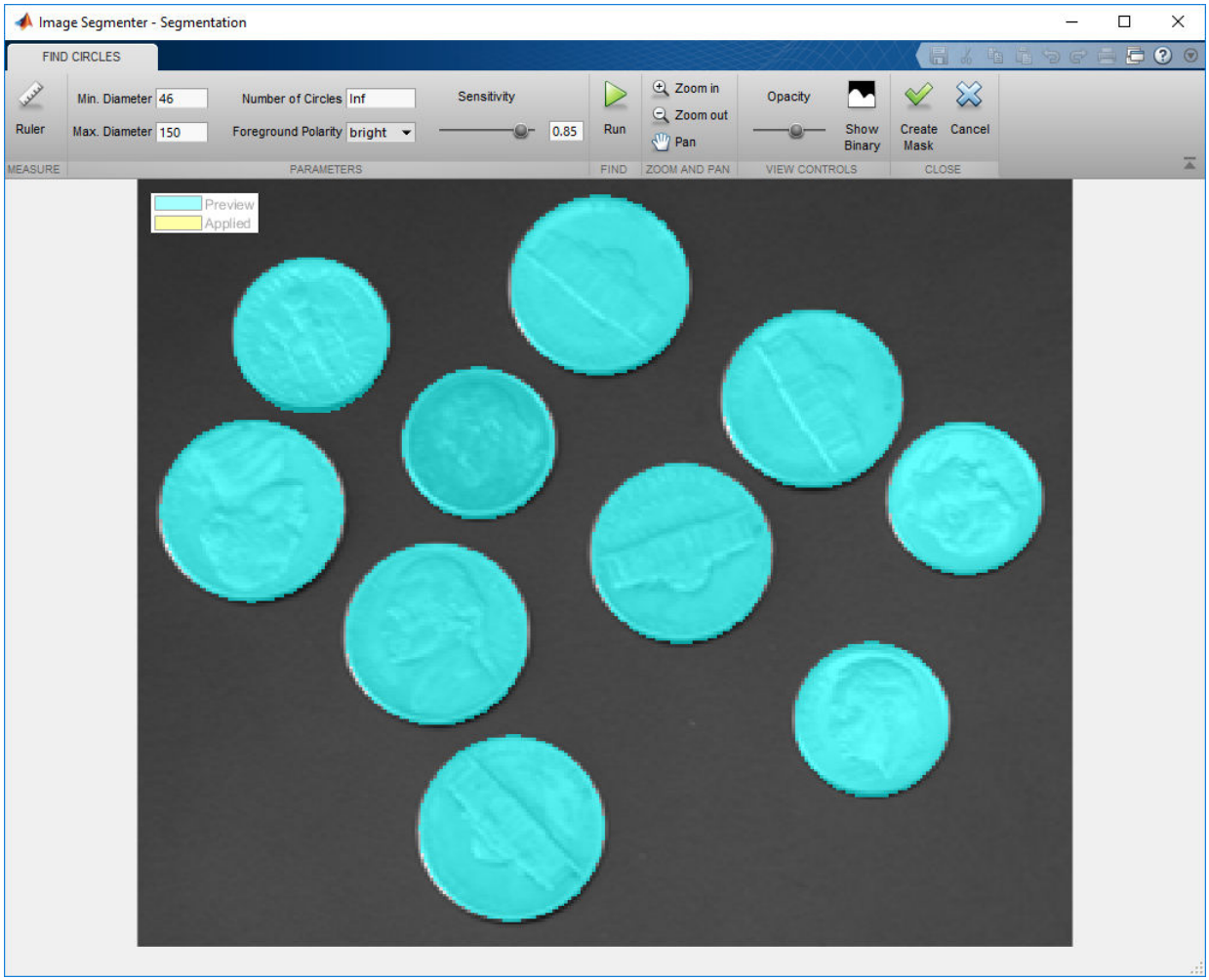


Click **Run**. The Image Segmenter fills the circles it finds. However, in the result, you can see that **Find Circles** does not find two of the circles. Examining the diameter of the

objects not found, you can see that their diameter is slightly less than the specified minimum diameter.

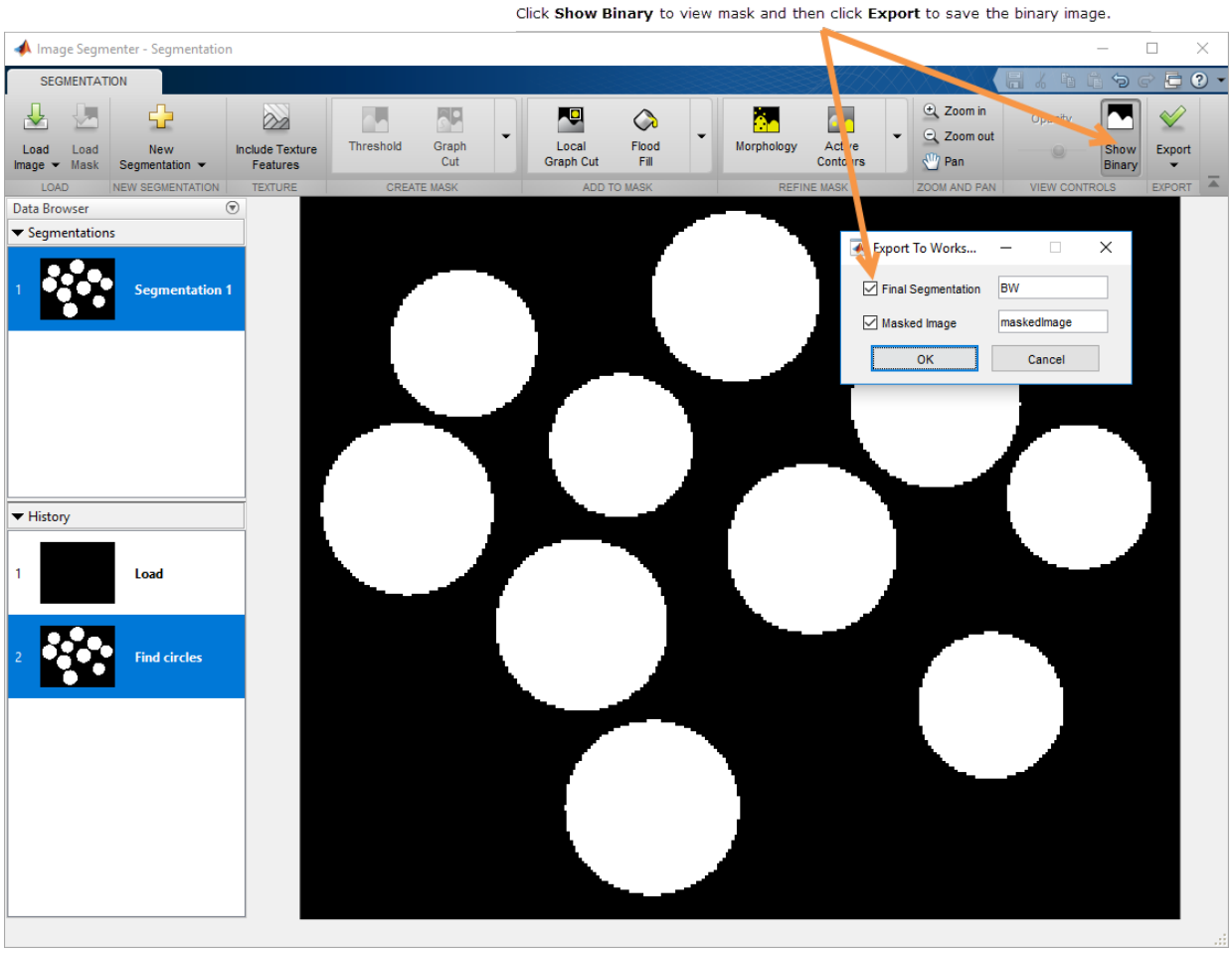


Change the minimum value to accommodate the sizes of the objects that were not segmented and click **Run** again. This time, Find Circles segments all the objects in the image.



When you are satisfied with the segmentation, click **Create Mask** to create the mask image. The Image Segmenter creates the segmentation, changing the color of the segmented part of the image from blue to yellow, and closes the Find Circles tab. To view the mask image, click **Show Binary**.

When you are done segmenting the image, you can save the binary mask, using the **Export** option. You can also obtain the code used for the segmentation. For more information about saving the mask image, see “Save Mask Image Created Using Image Segmenter” on page 13-101.



See Also

Image Segmenter | `imfindcircles`

Related Examples

- “Image Segmentation Using the Image Segmenter App” on page 13-91

Segmentation Using Auto Cluster in Image Segmenter

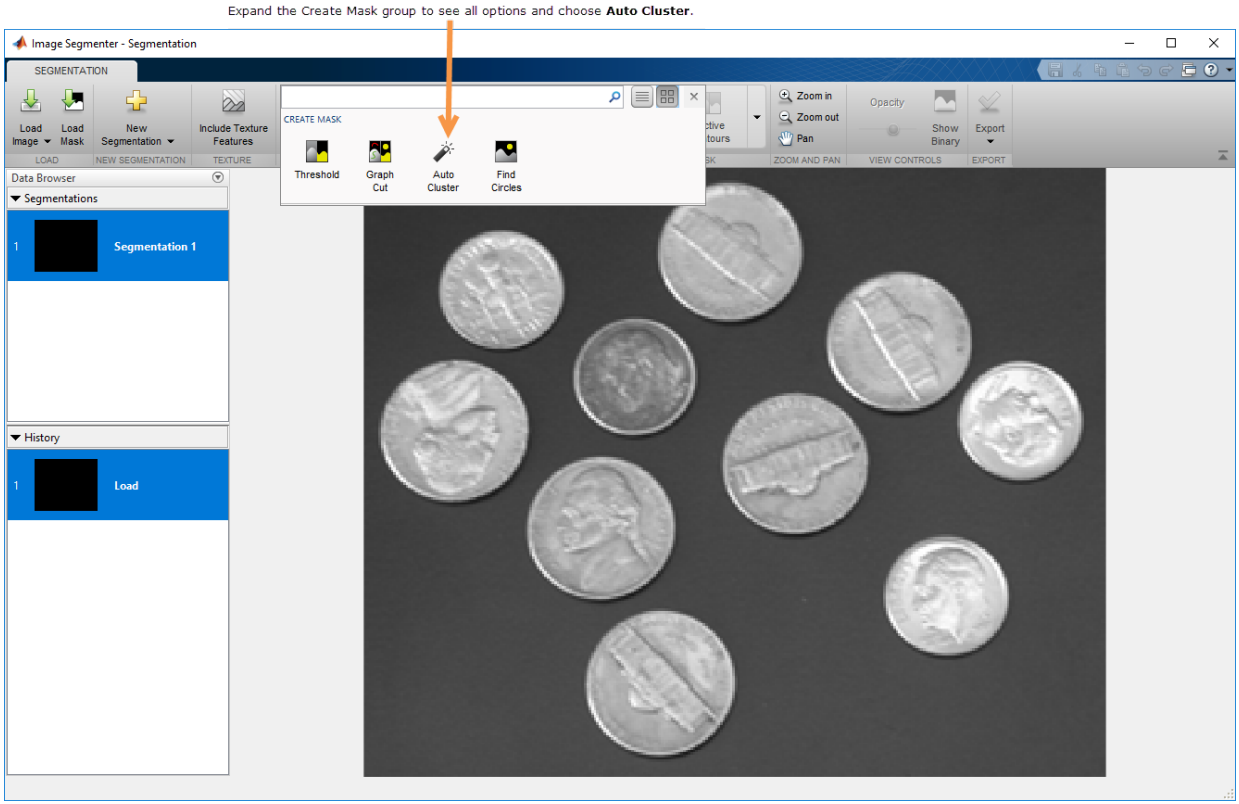
This example shows how to use the **Auto Cluster** option in the Image Segmenter app to segment an image. The **Auto Cluster** option is an automatic segmentation technique that you can use to segment an image into foreground and background elements. The **Auto Cluster** option does not require initialization.

The Auto Cluster option requires the Statistics and Machine Learning Toolbox. If you don't have this toolbox, the Image Segmenter does not include this option. Generated code also requires Statistics and Machine Learning Toolbox.

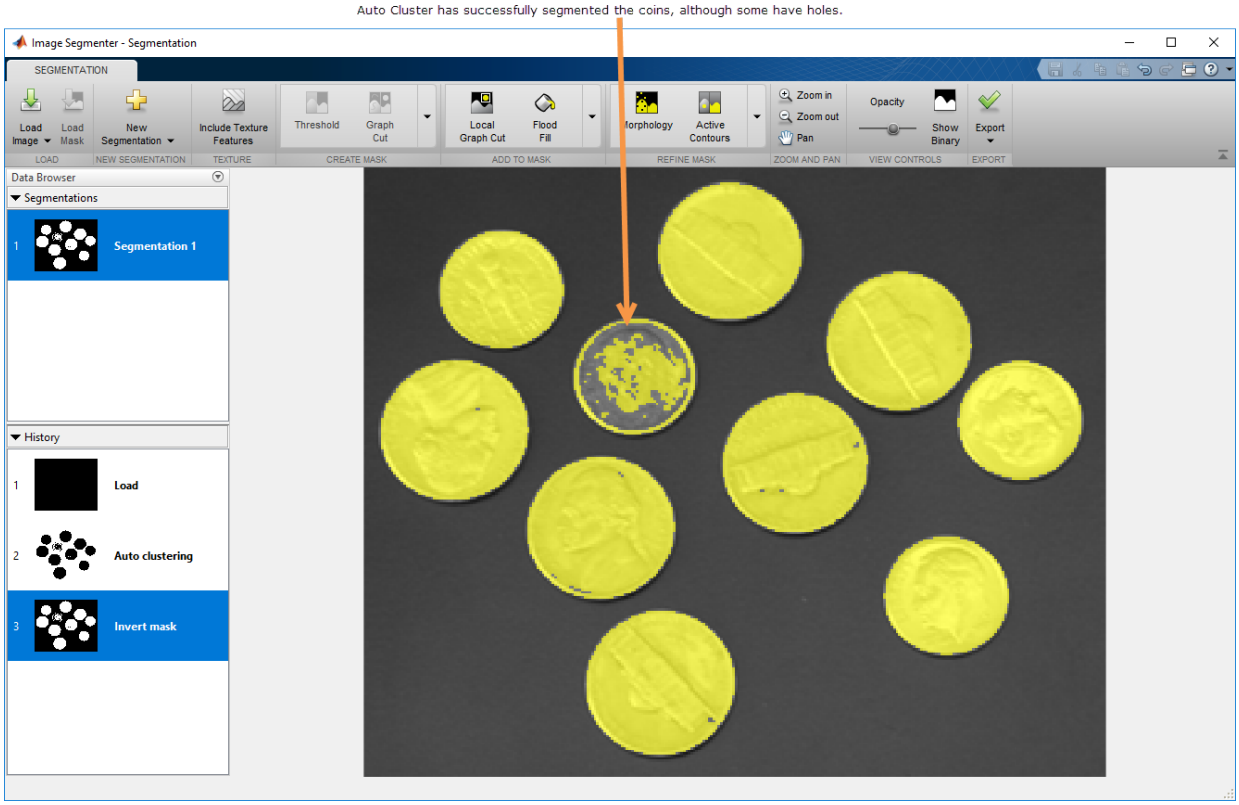
Read an image into the MATLAB workspace and load it into the Image Segmenter app. For more information about opening the Image Segmenter app and loading an image, see “Open Image Segmenter App and Load Image” on page 13-97.

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

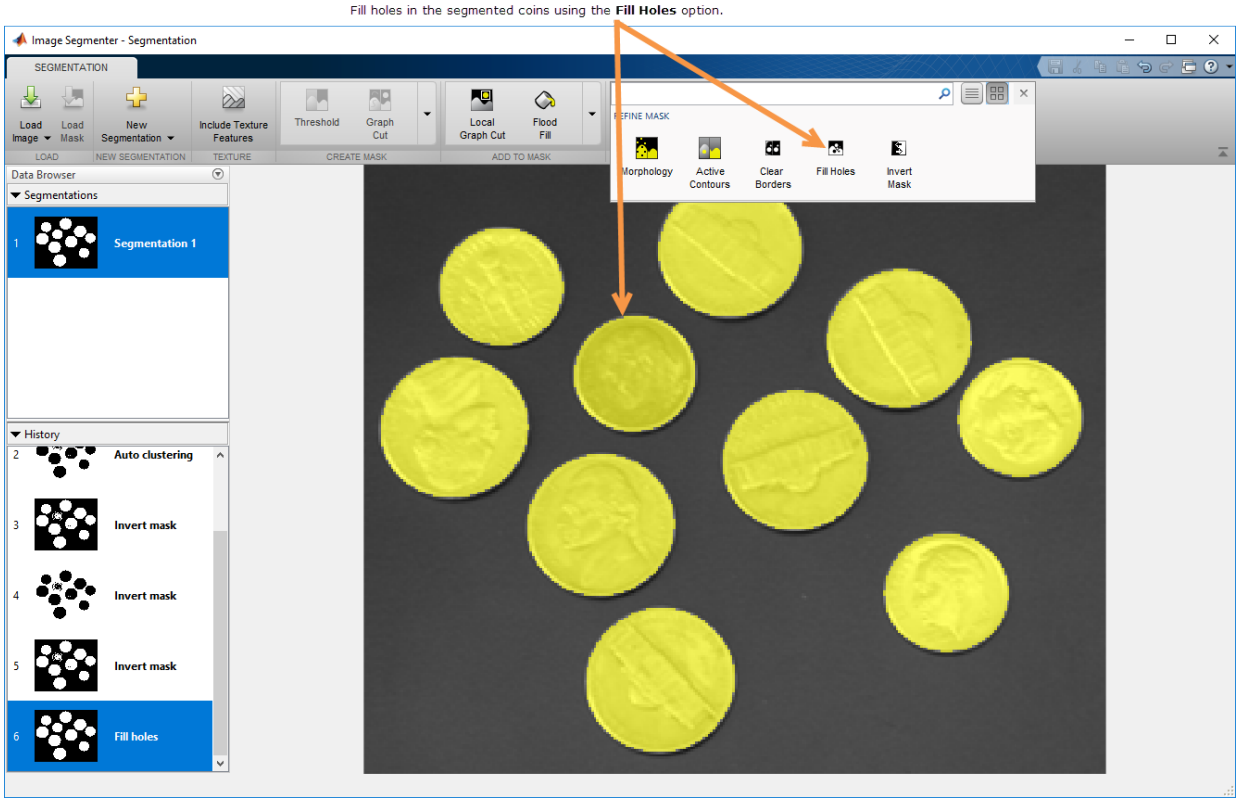
Expand the Create Mask group of toolstrip buttons and click **Auto Cluster**.



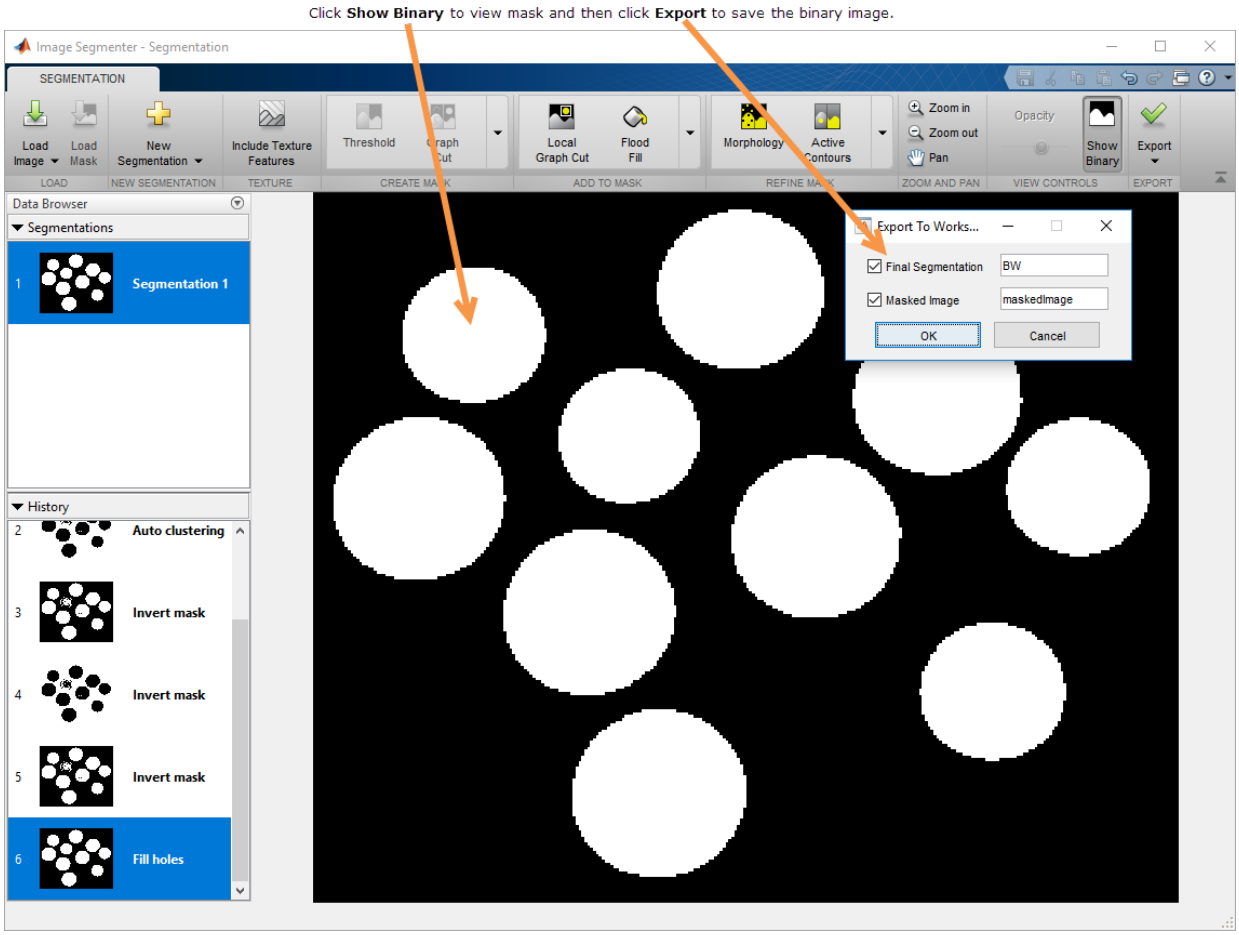
The Image Segmenter app segments the image, displaying the result. To get a better view of the segmentation, invert the mask, using the **Invert Mask** option in the Refine Mask toolstrip group. The Auto Cluster option has correctly segmented all the circles. However, some of the circles have holes.



Clean up the holes in the segmentation using the **Fill Holes** option in the Refine Mask toolstrip group.



When you are satisfied with the segmentation, click **Show Binary** to see the mask. and then click **Export** to save the mask image. For more information about saving the mask image, see “Save Mask Image Created Using Image Segmenter” on page 13-101.



See Also

Image Segmenter

Related Examples

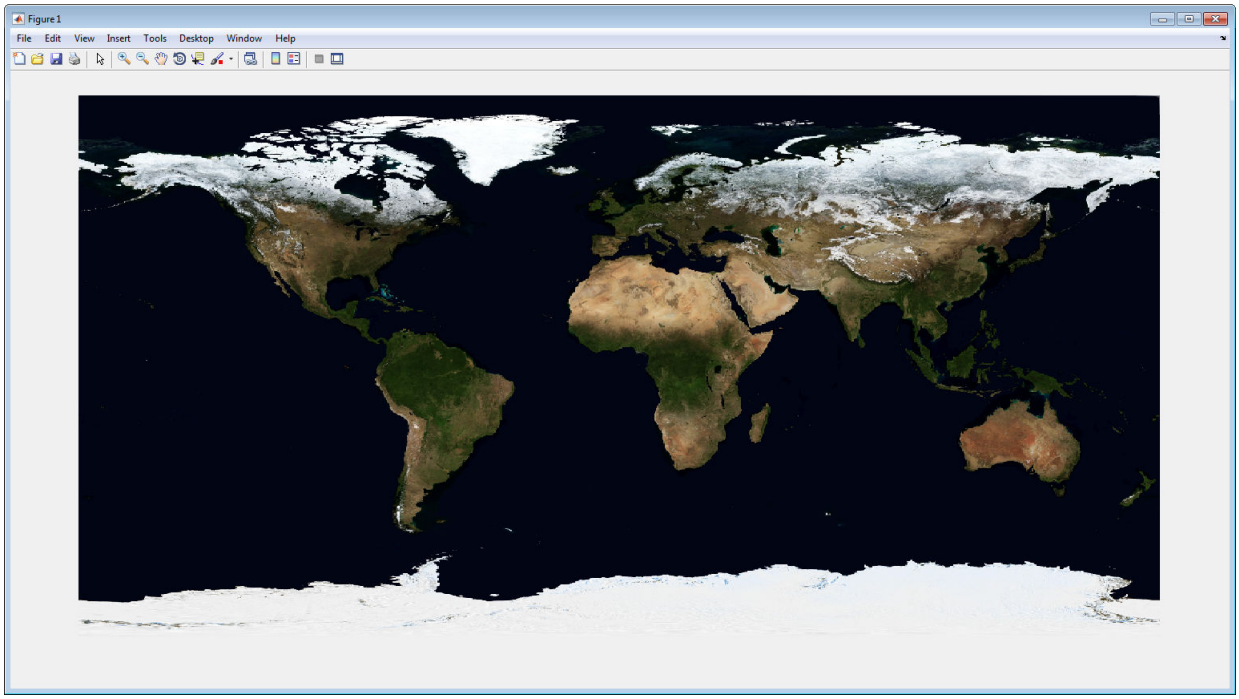
- “Image Segmentation Using the Image Segmenter App” on page 13-91

Plot Land Classification with Color Features and Superpixels

This example shows how to perform land type classification based on color features using K-means clustering and superpixels. Superpixels can be a very useful technique when performing segmentation and classification, especially when working with large images. Superpixels enable you to break an image into a set of structurally meaningful regions, where the boundaries of each region take into account edge information in the original image. Once you break an image into superpixel regions, classification algorithms can be used to classify each region, rather than having to solve the classification problem over the full original image grid. The use of superpixels can provide large performance advantages in solving image classification problems while also providing a high quality segmentation result. This example requires the Statistics and Machine Learning Toolbox

Read an image into the workspace. For better performance, the example reduces the size of the image by half.

```
A = imread('http://eoimages.gsfc.nasa.gov/images/imagerecords/74000/74192/world.200411

A = imresize(A,0.5);
imshow(A)
```

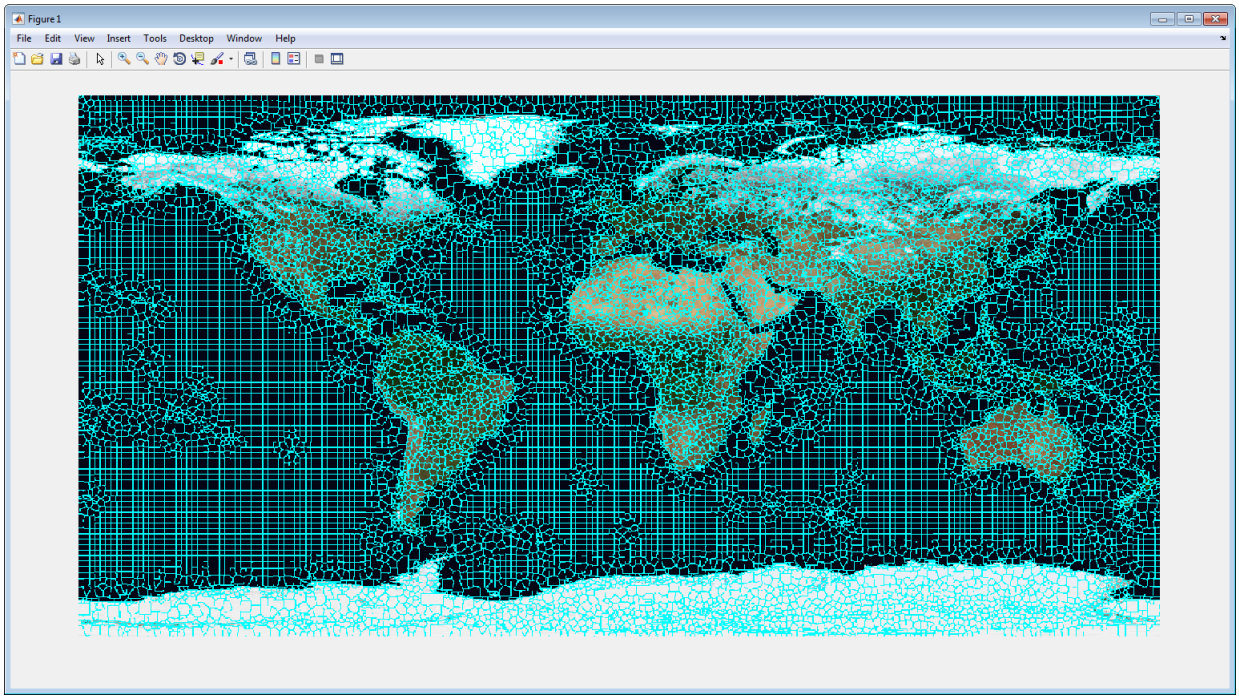


Convert the image to the $L^*a^*b^*$ color space.

```
Alab = rgb2lab(A);
```

Compute the superpixel oversegmentation of the original image and display it.

```
[L,N] = superpixels(Alab,20000,'isInputLab',true);
BW = boundarymask(L);
imshow(imoverlay(A,BW,'cyan'))
```



Create a cell array of the set of pixels in each region.

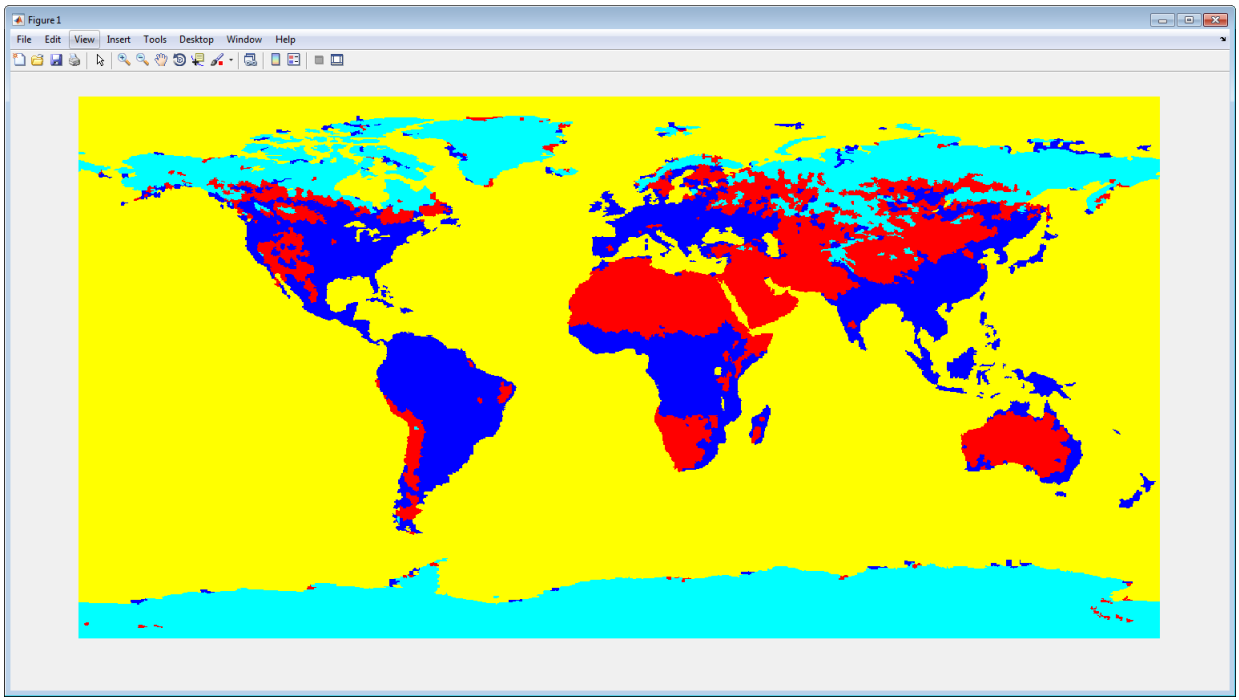
```
pixelIdxList = label2idx(L);
```

Determine the median color of each superpixel region in the *L*a*b** color space.

```
meanColor = zeros(N,3);
[m,n] = size(L);
for  i = 1:N
       meanColor(i,1) = mean(Alab(pixelIdxList{i}));
    meanColor(i,2) = mean(Alab(pixelIdxList{i}+m*n));
    meanColor(i,3) = mean(Alab(pixelIdxList{i}+2*m*n));
end
```

Cluster the color feature of each superpixel, using the `kmeans` function. Visually, there are 4 types of land that are distinguishable in the blue marble image based only on color features: forested regions, dry/desert regions, ice covered regions, and water. When solving the clustering problem with `kmeans`, use four clusters. This step requires the Statistics and Machine Learning Toolbox.

```
numColors = 4;
[idx,cmap] = kmeans(meanColor,numColors,'replicates',2);
cmap = lab2rgb(cmap);
Lout = zeros(size(A,1),size(A,2));
for i = 1:N
    Lout(pixelIdxList{i}) = idx(i);
end
imshow(label2rgb(Lout))
```



Use cluster centers as the colormap for a thematic map. The mean colors found during `kmeans` clustering can be used directly as a colormap to give a more natural visual interpretation of the land classification assignments of forest, ice, dry land, and water.

```
imshow(Lout,cmap)
```

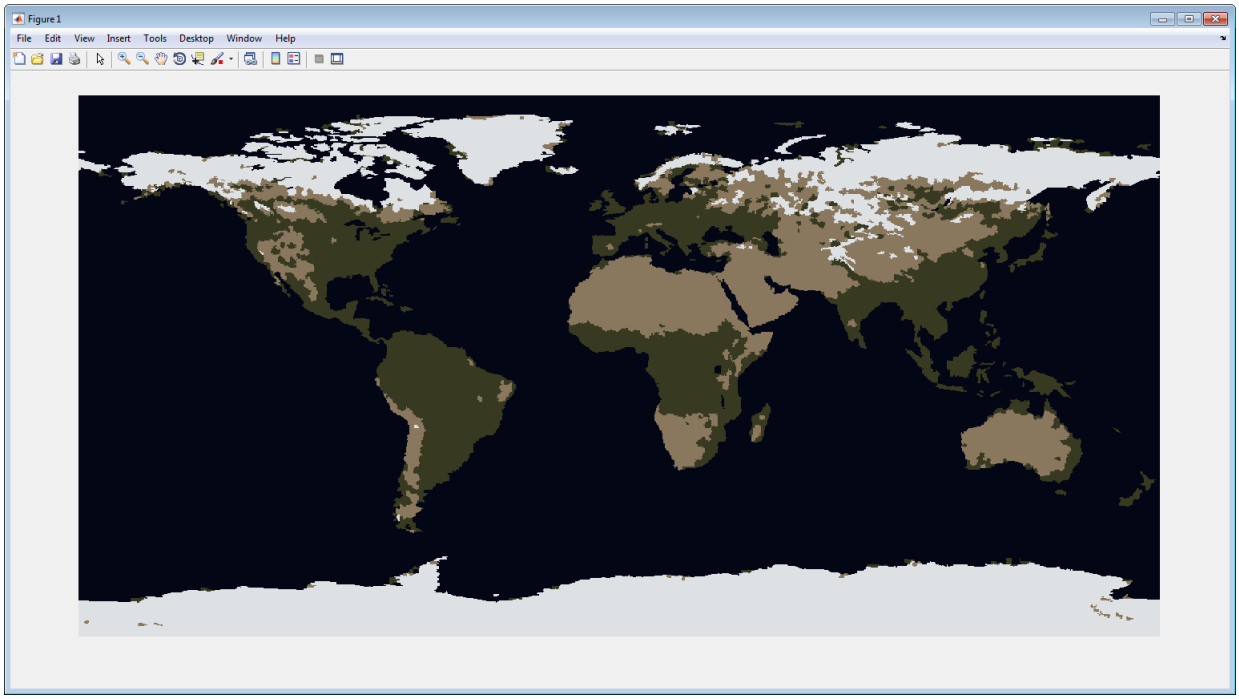


Figure 1
File Edit View Insert Tools Desktop Window Help

Segment Lungs from 3-D Chest Scan and Calculate Lung Volume

In this section...

"Prepare the Data" on page 13-146
"Step 1: Segment the Lungs" on page 13-147
"Step 2: Compute the Volume of the Segmented Lungs" on page 13-155

This example shows how to perform a 3-D segmentation using active contours (snakes) and view the results using the Volume Viewer app.

Prepare the Data

This part of the example loads the human chest CT scan data into the MATLAB workspace. To run this example, you must download the sample data from the MathWorks using the Add-Ons Explorer. See "Install Sample Data Using the Add-Ons Explorer" on page 11-204.

Load the 3-D volumetric CT scan data into the MATLAB workspace.

```
load chestVolume
```

```
whos
```

```
Name        Size                     Bytes  Class    Attributes

  V        512x512x318            166723584  int16
```

Convert the CT scan data from `int16` to `single` to normalize the values between `[0 1]`.

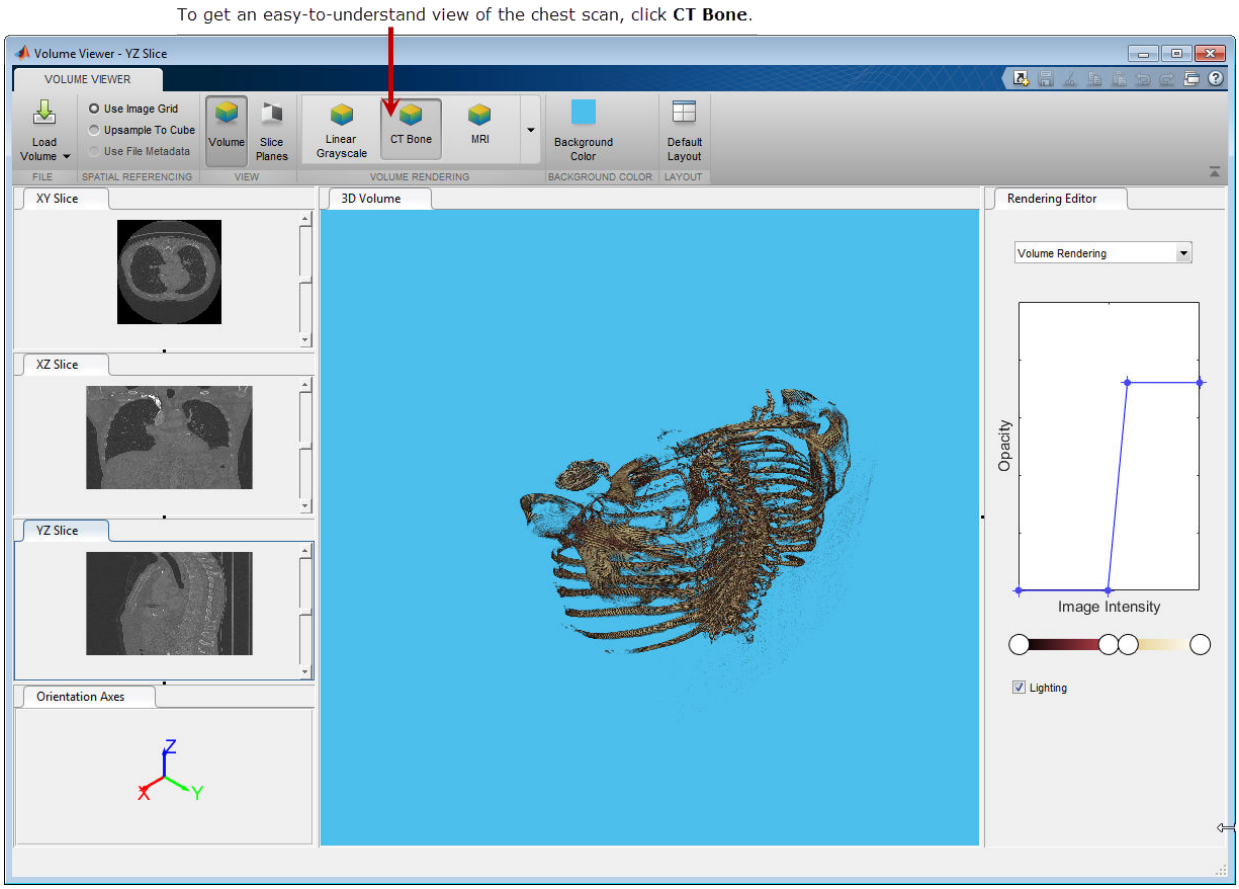
```
V = im2single(V);
```

```
whos
```

```
 Name        Size                     Bytes  Class     Attributes

  V       512x512x318               333447168  single
```

Visualize the chest scans using the Volume Viewer app.

```
volumeViewer(V)
```



Step 1: Segment the Lungs

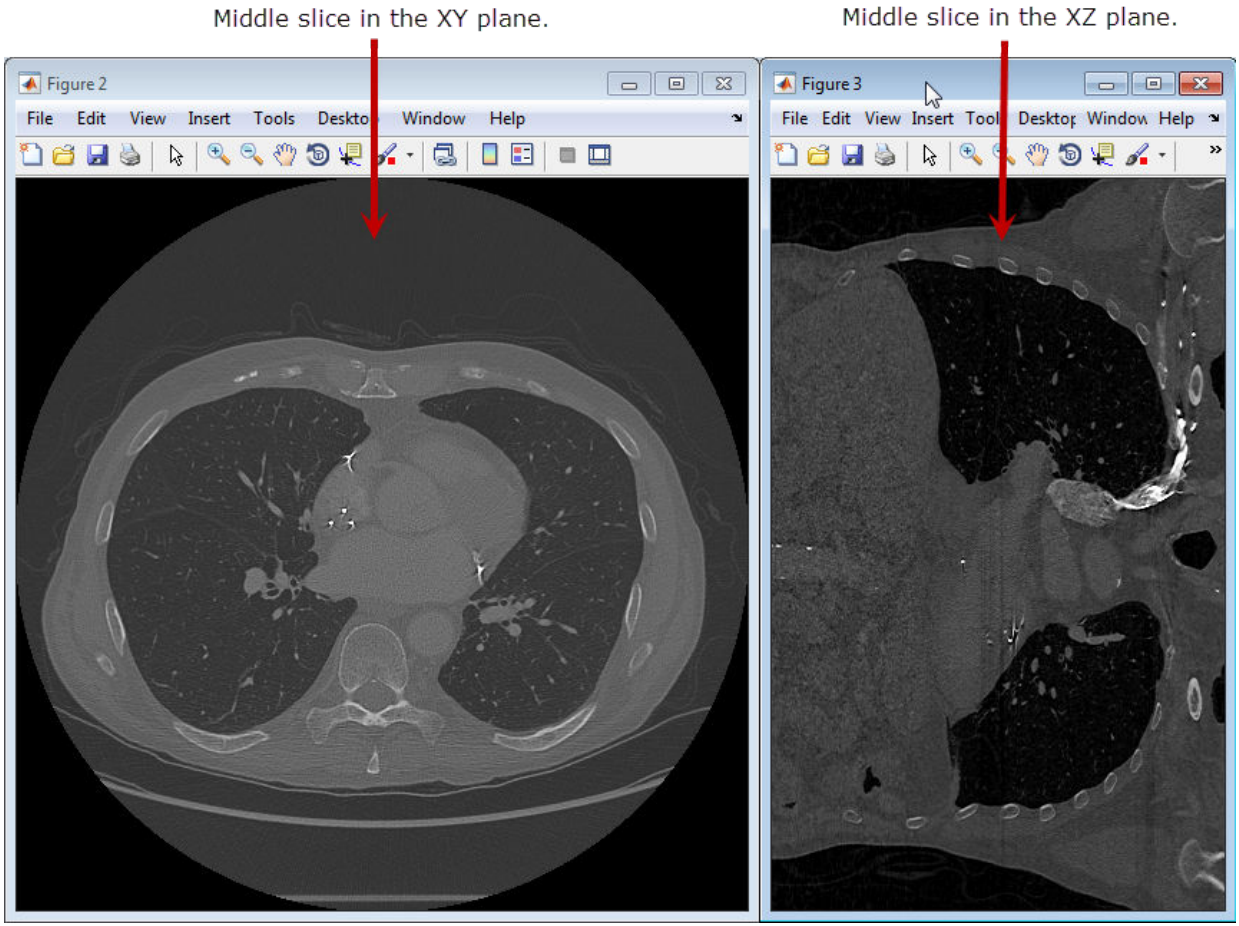
This part of the example shows how to segment the lungs in the CT scan data using the active contour technique. This is a region growing algorithm which requires initial seed points. The example uses the Image Segmenter app to create this seed mask by segmenting two, orthogonal 2-D slices, one in the XY plane and the other in the XZ plane. The example then inserts these two segmentations into a 3-D mask. The example passes this mask to the `activecontour` function to create a 3-D segmentation of the lungs in the chest cavity. (This example uses the active contour method but you could use other segmentation techniques to accomplish the same goal, such as, flood-fill.)

Extract the center slice in both the XY and XZ dimensions.

```
XY = V(:,:,160);
XZ = squeeze(V(256,:,:));
```

Visualize the 2-D slices using the `imshow` function.

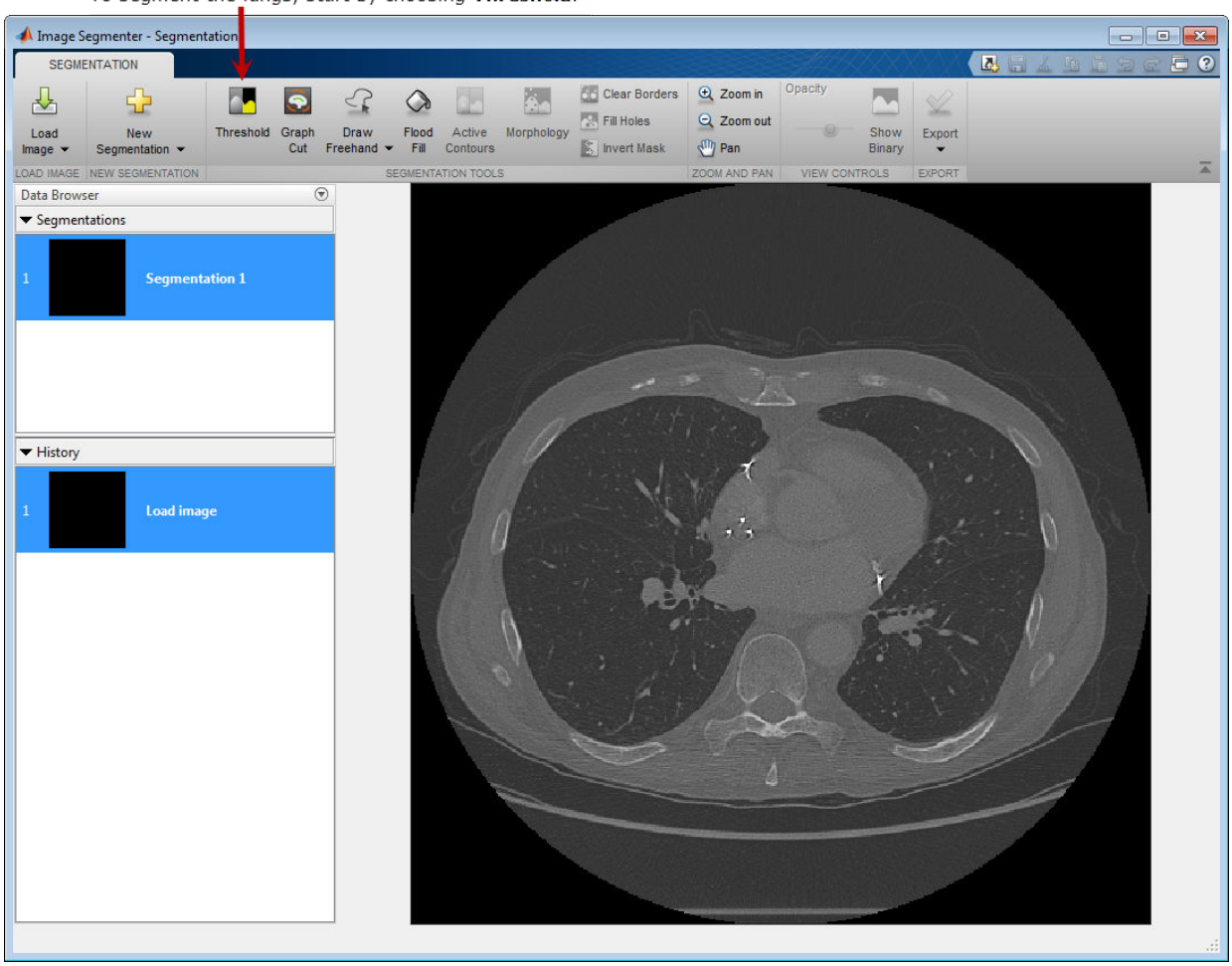
```
figure, imshow(XY, []);
figure, imshow(XZ, []);
```



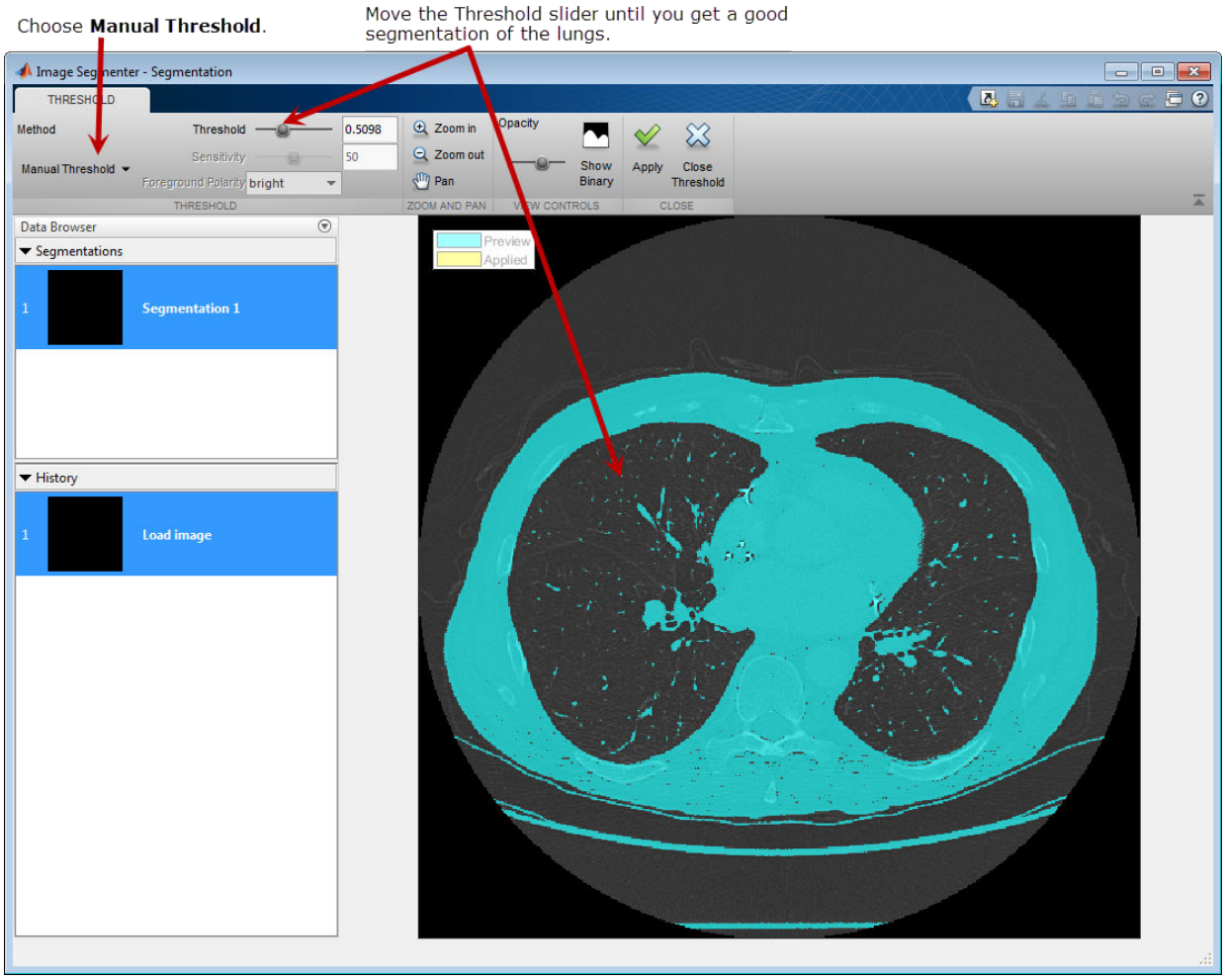
Open the XY slice in the Image Segmenter app and click **Threshold** to start the segmentation process.

```
imageSegmenter(XY)
```

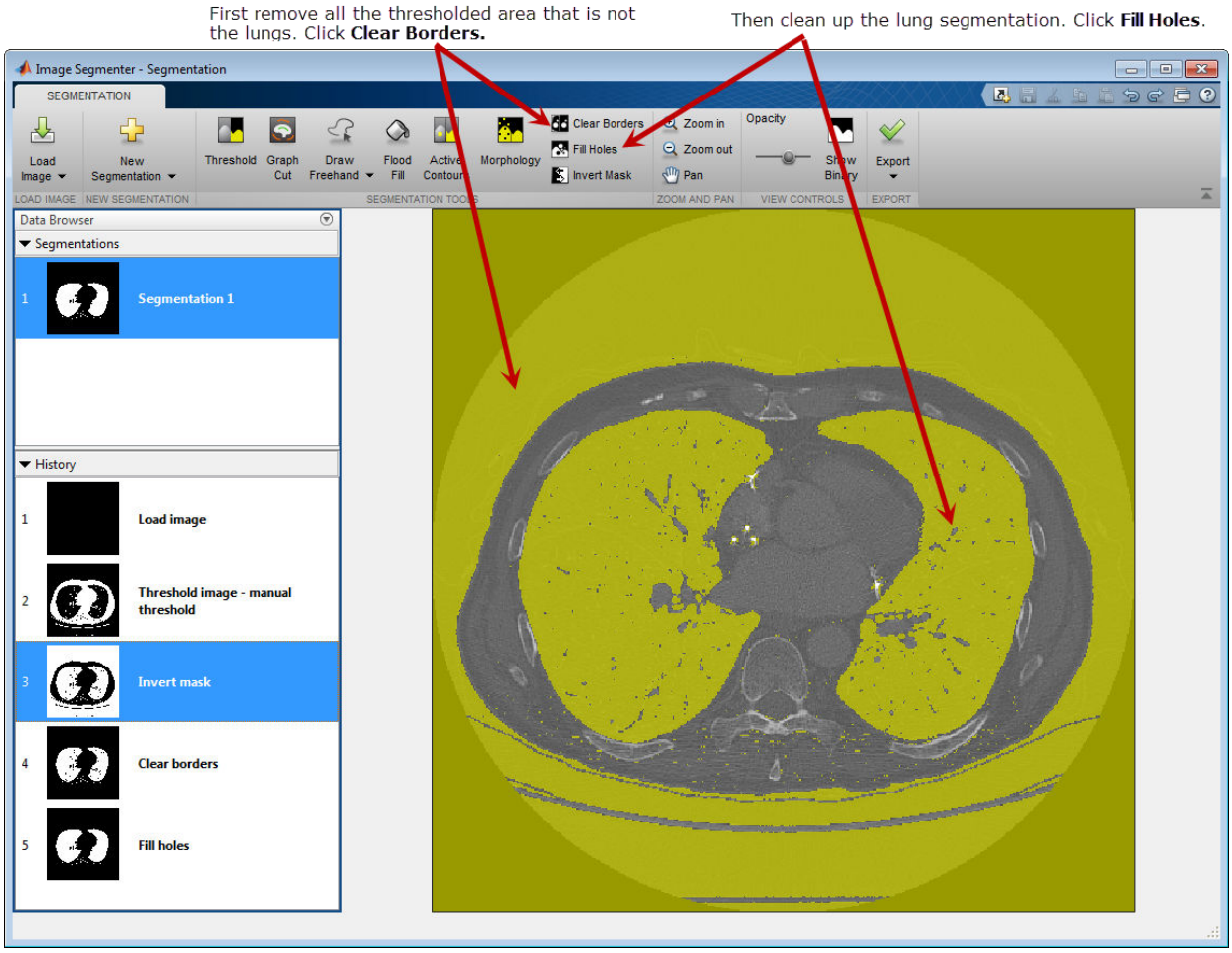
To segment the lungs, start by choosing **Threshold**.



On the Threshold tab, choose **Manual Threshold** and move the **Threshold** slider to adjust the segmentation until the lungs appear well-delineated. It is OK if other objects appear segmented in the thresholded region. Click **Create Mask** to save the result.

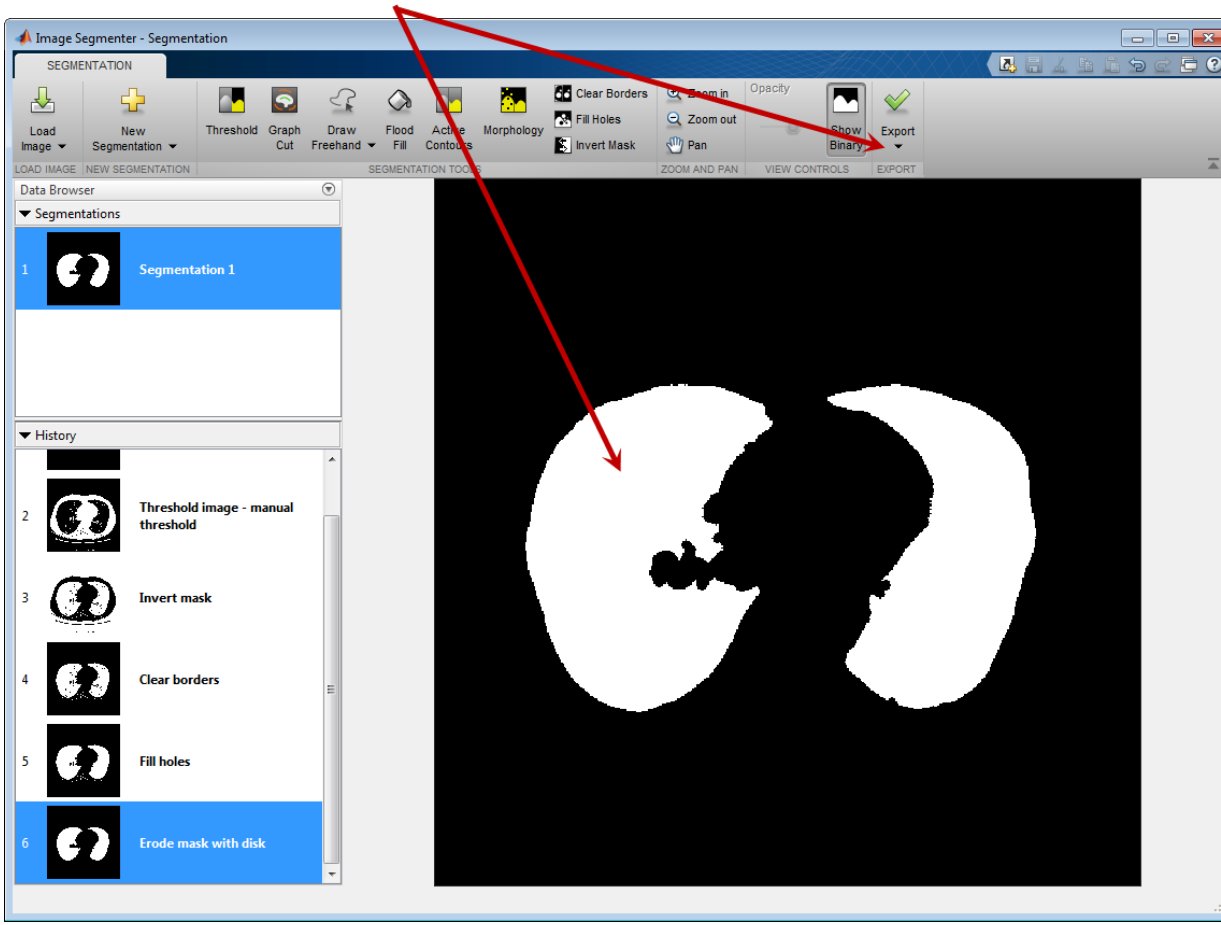


On the Segmentation tab, click **Invert Mask** to make the segmented lungs the foreground. Remove all the segmented parts that are not the lungs. Since these all touch the edges, use the **Clear Borders** option to remove them. Then fill the small holes that appear in the lung areas using the **Fill Holes** option.



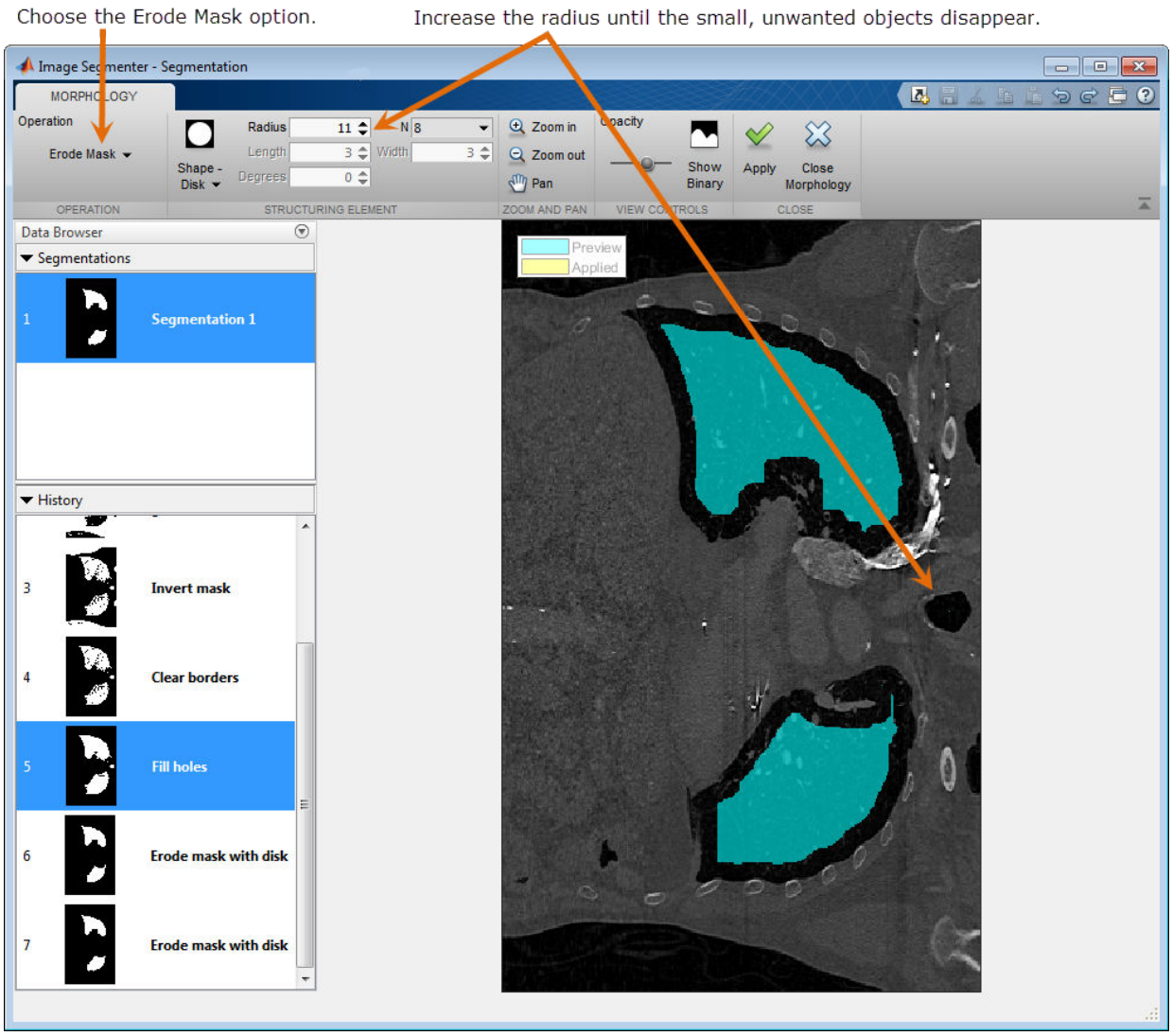
Remove small extraneous areas of the segmentation using erosion. Click **Morphology** and on the Morphology tab, click **Erode Mask** from the **Operation** selections. After this processing, click **Apply**. The segmented parts of the image turn from blue to yellow. Then close the Morphology tab. On the Segmentation tab, click **Show Binary** to view the mask you created. To save the mask, click **Export** and save the final segmentation under the name **mask_XY**.

To save the binary mask of the 2-D lung slice, click **Export.**



Segment the XZ slice using the same procedure. Open the XZ slice in the Image Segmenter app. Choose **Threshold**. With this image, you can use the Global Thresholding option. Click **Create Mask**. On the Segmentation tab, click **Invert Mask** to make the lungs the foreground. Click **Clear Borders** option to remove extraneous regions that were included in the thresholding. Fill small holes in the lung areas using the **Fill Holes** option. To remove the small segmented regions that remain that are not the lungs, click **Morphology** and select **Erode Mask**. Increase the radius until these extraneous regions are gone. Click **Apply** and close the Morphology tab. On the Segmentation tab, click

Show Binary to view the mask you created. Click **Export** and save the final segmentation under the name **mask_XZ**.



Create a 3-D seed mask to be used with the `activecontour` function. First, create a logical 3-D volume the same size as the input volume and insert `mask_XY` and `mask_XY` at the appropriate spatial locations.

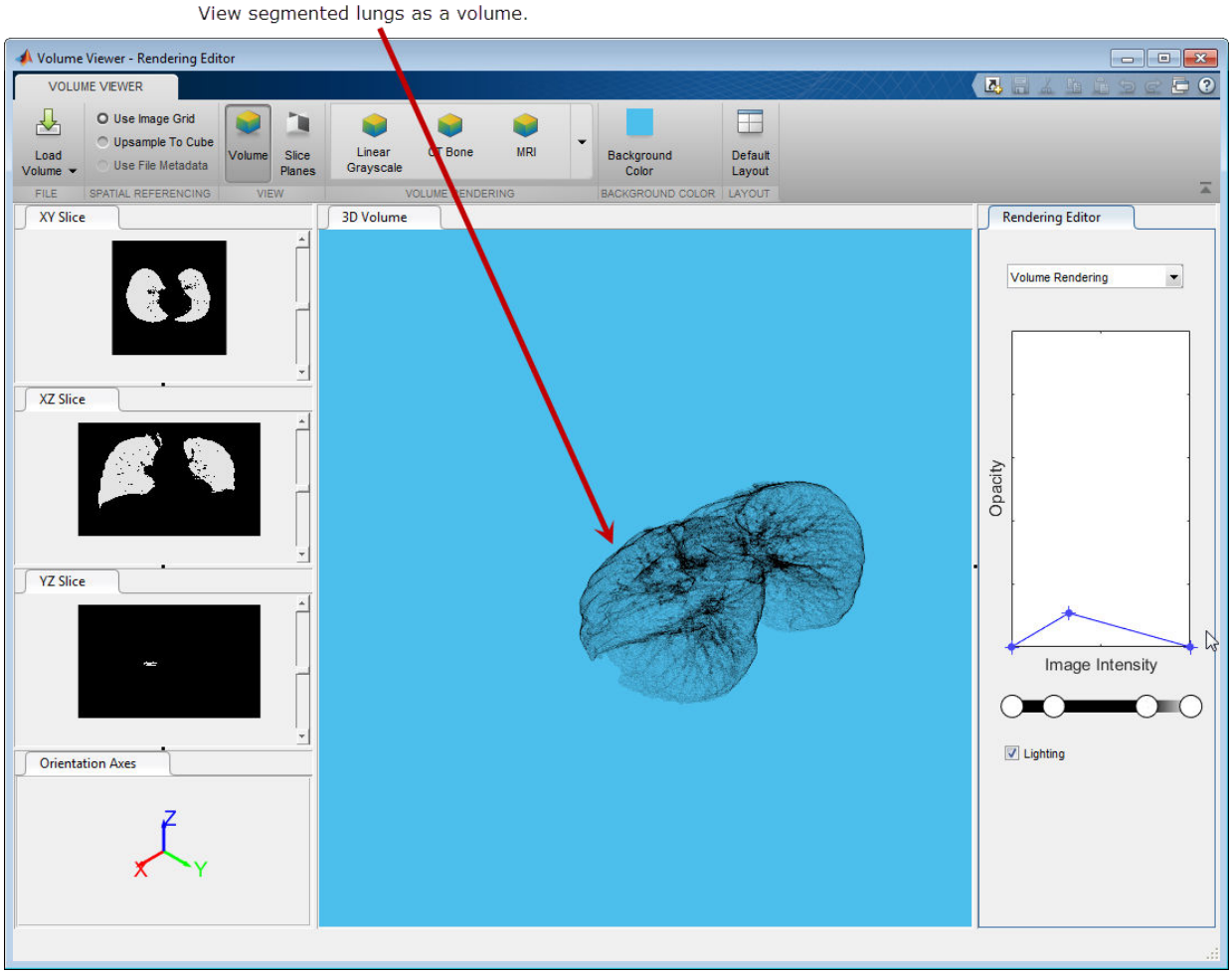
```
mask = false(size(V));
mask(:,:, 160) = mask_XY;
mask(256, :, :) = mask(256, :, :)|reshape(mask_XZ, [1, 512, 318]);
```

Perform a 3-D segmentation of the lungs using the active contour method. This can take a few minutes. To get a quality segmentation, use `histeq` to spread voxel values over the available range.

```
V = histeq(V);

BW  = activecontour(V,mask,100,'Chan-Vese');

segmentedImage = V.*single(BW);
```

View the segmented lungs in the Volume Viewer app. Use the alphamap plot in the Rendering Editor to manipulate the opacity mapping until you get a good view of the lungs.

```
volumeViewer(segmentedImage);
```



Step 2: Compute the Volume of the Segmented Lungs

In this part of the example, use the `regionprops` function to calculate the volume (area) of the lungs.

Calculate the area of the lungs using the `regionprops` function with the `'area'` option.

```
volLungsPixels = regionprops3(logical(BW), 'volume');
```

Specify the spacing of the voxels in the *x*, *y*, and *z* dimensions. (Gathered from the file metadata previously.)

```
xSpacing = 0.76; %(mm)
ySpacing = 0.76; %(mm)
zSpacing = 1.25; %(mm)
```

Calculate lung capacity, in liters.

```
volLungs1 = volLungsPixels.Volume(1)*0.76*0.76*1.25*1e-6;

volLungs2 = volLungsPixels.Volume(2)*0.76*0.76*1.25*1e-6;

volLungsLiters = volLungs1 + volLungs2;
```

```
volLungsLiters =

    6.7157
```

See Also

`activecontour` | `histeq` | `regionprops`

Image Deblurring

This chapter describes how to deblur an image using the toolbox deblurring functions.

- “Image Deblurring” on page 14-2
- “Deblur with the Wiener Filter” on page 14-6
- “Deblurring Images Using a Wiener Filter” on page 14-7
- “Deblur with a Regularized Filter” on page 14-15
- “Deblurring Images Using a Regularized Filter” on page 14-18
- “Deblur with the Lucy-Richardson Algorithm” on page 14-30
- “Deblurring Images Using the Lucy-Richardson Algorithm” on page 14-35
- “Deblur with the Blind Deconvolution Algorithm” on page 14-50
- “Deblurring Images Using the Blind Deconvolution Algorithm” on page 14-60
- “Create Your Own Deblurring Functions” on page 14-72
- “Avoid Ringing in Deblurred Images” on page 14-73

Image Deblurring

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

A blurred or degraded image can be approximately described by this equation **g = Hf + 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** does not really exist. This image represents what you would have if you had perfect image acquisition conditions.

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.

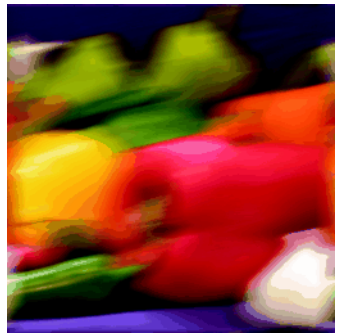
Note The quality of the deblurred image is mainly determined by knowledge of the PSF.

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 “Deblur with the Wiener Filter” on page 14-6.)

```
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.

`deconvwnr`	Implements a least squares solution. You should provide some information about the noise to reduce possible noise amplification during deblurring. See “Deblur with the Wiener Filter” on page 14-6 for more information.
`deconvreg`	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 “Deblur with a Regularized Filter” on page 14-15 for more information.
`deconvlucy`	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 “Deblur with the Lucy-Richardson Algorithm” on page 14-30 for more information.
`deconvblind`	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 `deconvblind` function returns a restored PSF in addition to the restored image. The implementation uses the same damping and iterative model as the `deconvlucy` function. See “Deblur with the Blind Deconvolution Algorithm” on page 14-50 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 “Avoid Ringing in Deblurred Images” on page 14-73 for more information.

- For information about creating your own deblurring functions, see “Create Your Own Deblurring Functions” on page 14-72.

Deblur 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, see the example “Deblurring Images Using a Wiener Filter” on page 14-7.

See Also

`deconvblind` | `deconvlucy` | `deconvreg` | `deconvwnr`

Related Examples

- “Deblurring Images Using a Wiener Filter” on page 14-7

More About

- “Image Deblurring” on page 14-2

Deblurring Images Using a Wiener Filter

This example shows how to use Wiener deconvolution to deblur images. Wiener deconvolution can be useful when the point-spread function and noise level are either known or estimated.

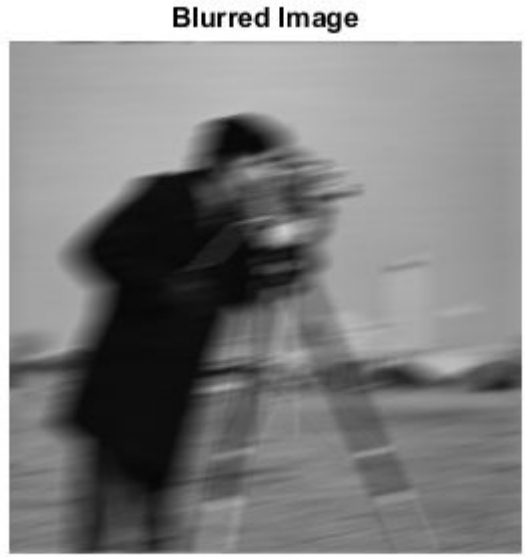
Read Image

```
I = im2double(imread('cameraman.tif'));
imshow(I);
title('Original Image (courtesy of MIT)');
```

Simulate a Motion Blur

Simulate a blurred image that you might get from camera motion. Create a point-spread function, `PSF`, corresponding to the linear motion across 21 pixels (`LEN=21`), at an angle of 11 degrees (`THETA=11`). To simulate the blur, convolve the filter with the image using `imfilter`.

```
LEN = 21;
THETA = 11;
PSF = fspecial('motion', LEN, THETA);
blurred = imfilter(I, PSF, 'conv', 'circular');
imshow(blurred);
title('Blurred Image');
```



Restore the Blurred Image

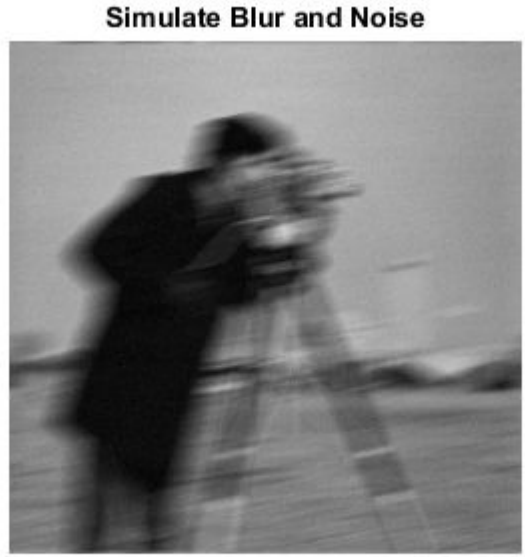
The simplest syntax for `deconvwnr` is `deconvwnr(A, PSF, NSR)`, where `A` is the blurred image, `PSF` is the point-spread function, and `NSR` is the noise-power-to-signal-power ratio. The blurred image formed in Step 2 has no noise, so we'll use 0 for `NSR`.

```
wnr1 = deconvwnr(blurred, PSF, 0);
imshow(wnr1);
title('Restored Image');
```

Simulate Blur and Noise

Now let's try adding noise.

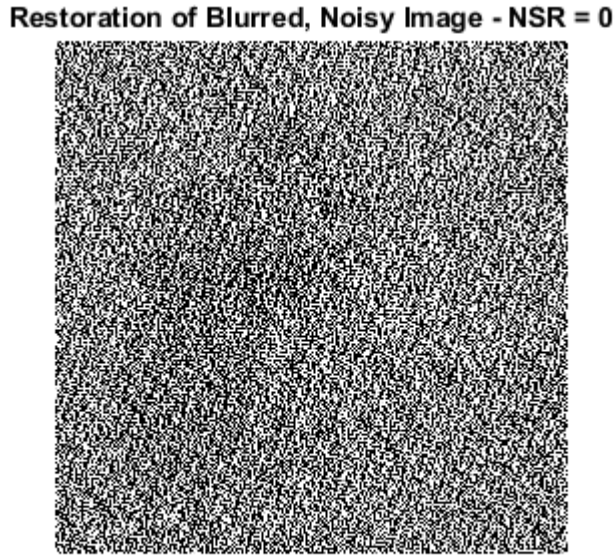
```
noise_mean = 0;
noise_var = 0.0001;
blurred_noisy = imnoise(blurred, 'gaussian', ...
                        noise_mean, noise_var);
imshow(blurred_noisy)
title('Simulate Blur and Noise')
```



Restore the Blurred and Noisy Image: First Attempt

In our first restoration attempt, we'll tell `deconvwnr` that there is no noise (NSR = 0). When NSR = 0, the Wiener restoration filter is equivalent to an ideal inverse filter. The ideal inverse filter can be extremely sensitive to noise in the input image, as the next image shows:

```
wnr2 = deconvwnr(blurred_noisy, PSF, 0);
imshow(wnr2)
title('Restoration of Blurred, Noisy Image - NSR = 0')
```

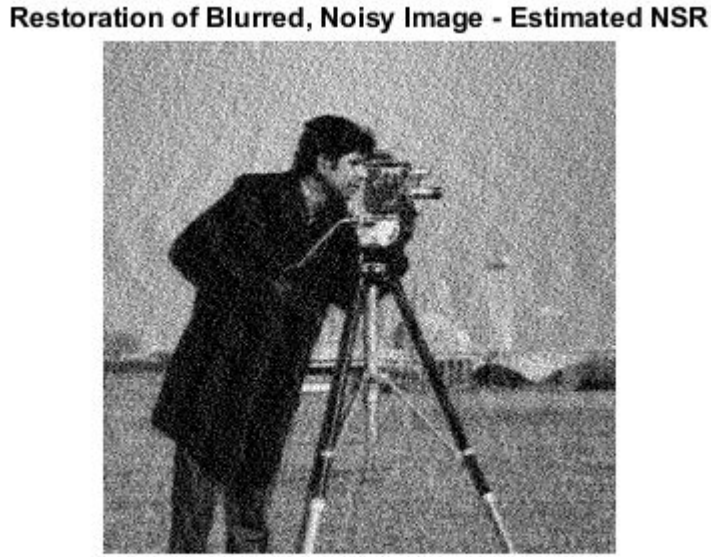


The noise was amplified by the inverse filter to such a degree that only the barest hint of the man's shape is visible.

Restore the Blurred and Noisy Image: Second Attempt

In our second attempt we supply an estimate of the noise-power-to-signal-power ratio.

```
signal_var = var(I(:));
wnr3 = deconvwnr(blurred_noisy, PSF, noise_var / signal_var);
imshow(wnr3)
title('Restoration of Blurred, Noisy Image - Estimated NSR');
```



Simulate Blur and 8-Bit Quantization Noise

Even a visually imperceptible amount of noise can affect the result. Let's try keeping the input image in `uint8` representation instead of converting it to `double`.

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

```
ans =
'uint8'
```

If you pass a `uint8` image to `imfilter`, it will quantize the output in order to return another `uint8` image.

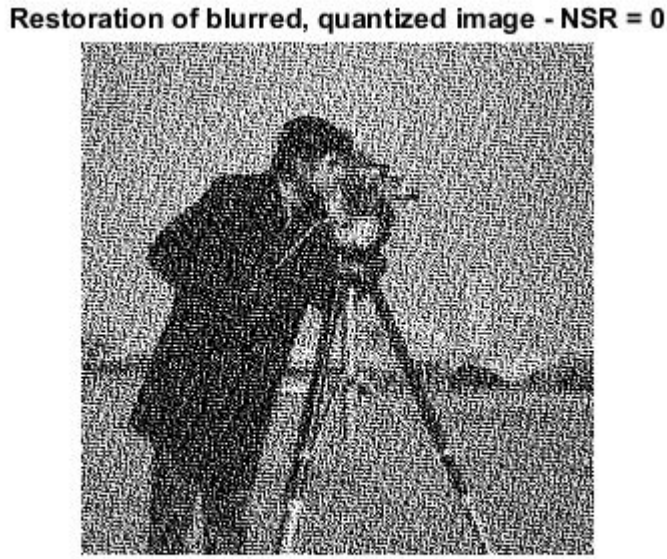
```
blurred_quantized = imfilter(I, PSF, 'conv', 'circular');
class(blurred_quantized)
```

```
ans =
'uint8'
```

Restore the Blurred, Quantized Image: First Attempt

Again, we'll try first telling `deconvwnr` that there is no noise.

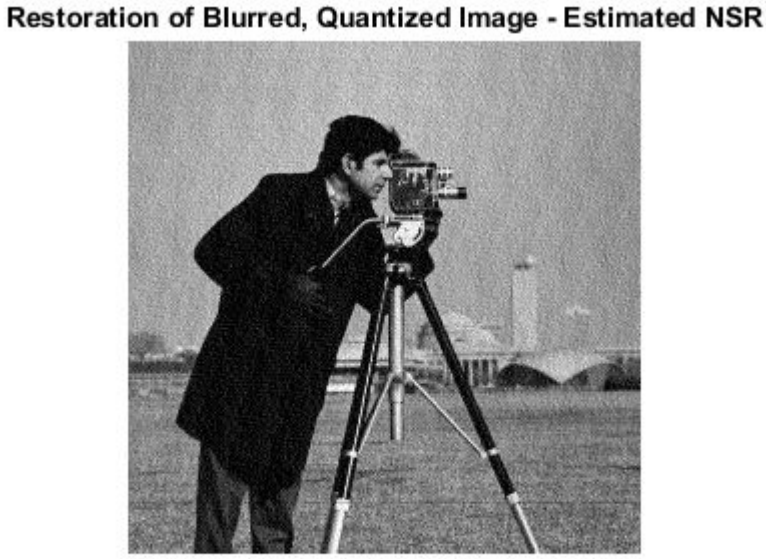
```
wnr4 = deconvwnr(blurred_quantized, PSF, 0);
imshow(wnr4)
title('Restoration of blurred, quantized image - NSR = 0');
```



Restore the Blurred, Quantized Image: Second Attempt

Next, we supply an NSR estimate to `deconvwnr`.

```
uniform_quantization_var = (1/256)^2 / 12;
signal_var = var(im2double(I(:)));
wnr5 = deconvwnr(blurred_quantized, PSF, ...
    uniform_quantization_var / signal_var);
imshow(wnr5)
title('Restoration of Blurred, Quantized Image - Estimated NSR');
```



See Also

`deconvblind` | `deconvlucy` | `deconvreg` | `deconvwnr`

More About

- “Image Deblurring” on page 14-2
- “Deblur with the Wiener Filter” on page 14-6

Deblur with a Regularized Filter

Use the `deconvreg` function to deblur an image using a regularized filter. A regularized filter can be used effectively when limited information is known about the additive noise.

To illustrate, this example simulates a blurred image by convolving a Gaussian filter PSF with an image (using `imfilter`). Additive noise in the image is simulated by adding Gaussian noise of variance `V` to the blurred image (using `imnoise`):

1. Read an image into the MATLAB workspace. The example uses cropping to reduce the size of the image to be deblurred. This is not a required step in deblurring operations.

   ```
   I = imread('tissue.png');
   I = I(125+[1:256],1:256,:);
   figure, imshow(I)
   title('Original Image')
   ```

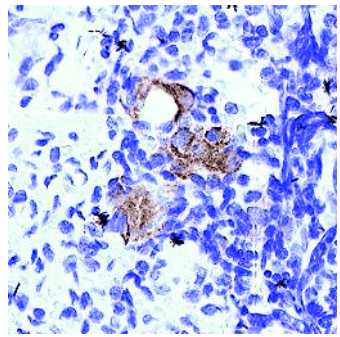
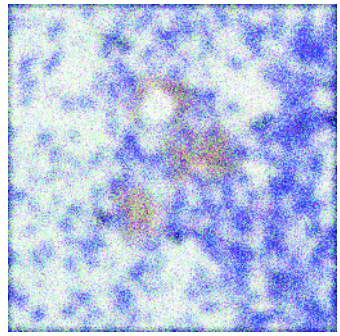
   

   Image Courtesy Alan W. Partin
2. Create the PSF.

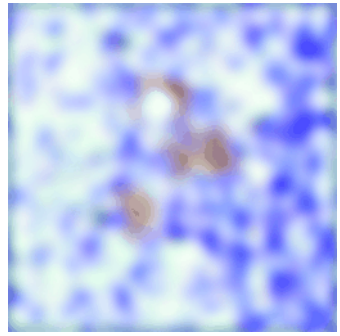
   ```
   PSF = fspecial('gaussian',11,5);
   ```
3. Create a simulated blur in the image and add noise.

   ```
   Blurred = imfilter(I,PSF,'conv');

   V = .02;
   BlurredNoisy = imnoise(Blurred,'gaussian',0,V);
   figure, imshow(BlurredNoisy)
   title('Blurred and Noisy Image')
   ```



4 Use `deconvreg` to deblur the image, specifying the PSF used to create the blur and the noise power, `NP`.

```
NP = V*prod(size(I));
[reg1 LAGRA] = deconvreg(BlurredNoisy,PSF,NP);
figure,imshow(reg1)
title('Restored Image')
```



Refining the Result

You can affect the deconvolution results by providing values for the optional arguments supported by the `deconvreg` function. Using these arguments you can specify the noise power value, the range over which `deconvreg` should iterate as it converges on the optimal solution, and the regularization operator to constrain the deconvolution. To see the impact of these optional arguments, see the example “Deblurring Images Using a Regularized Filter” on page 14-18.

See Also

`deconvblind` | `deconvlucy` | `deconvreg` | `deconvwnr`

Related Examples

- “Deblurring Images Using a Regularized Filter” on page 14-18

More About

- “Image Deblurring” on page 14-2

Deblurring Images Using a Regularized Filter

This example shows how to use regularized deconvolution to deblur images. Regularized deconvolution can be used effectively when constraints are applied on the recovered image (e.g., smoothness) and limited information is known about the additive noise. The blurred and noisy image is restored by a constrained least square restoration algorithm that uses a regularized filter.

Step 1: Read Image

The example reads in an RGB image and crops it to be 256-by-256-by-3. The deconvreg function can handle arrays of any dimension.

```
I = imread('tissue.png');
I = I(125+(1:256),1:256,:);
f1 = figure;
imshow(I);
figure(f1);
title('Original Image');
text(size(I,2),size(I,1)+15, ...
    'Image courtesy of Alan Partin, Johns Hopkins University', ...
    'FontSize',7,'HorizontalAlignment','right');
```

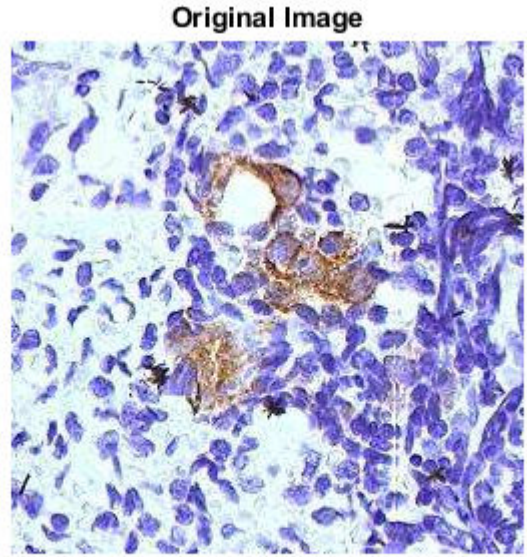
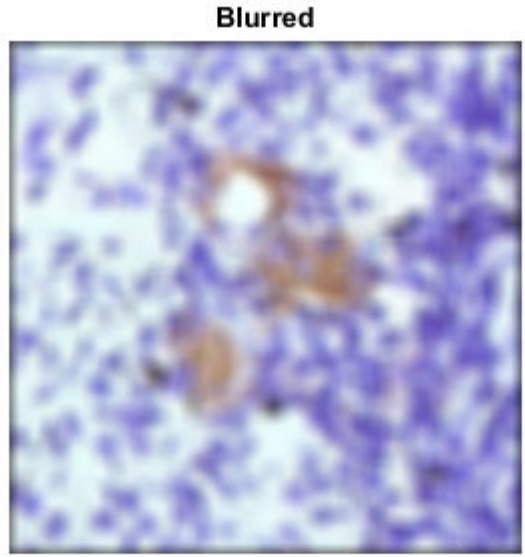


Image courtesy of Alan Partin, Johns Hopkins University

Step 2: Simulate a Blur and Noise

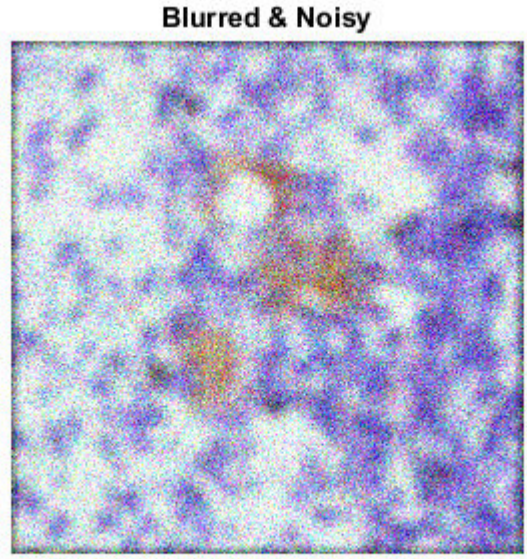
Simulate a real-life image that could be blurred (e.g., due to camera motion or lack of focus) and noisy (e.g., due to random disturbances). The example simulates the blur by convolving a Gaussian filter with the true image (using imfilter). The Gaussian filter represents a point-spread function, PSF.

```
PSF = fspecial('gaussian',11,5);
Blurred = imfilter(I,PSF,'conv');
f2 = figure;
imshow(Blurred);
figure(f2);
title('Blurred');
```



We simulate the noise by adding a Gaussian noise of variance V to the blurred image (using imnoise).

```
V = .02;
BlurredNoisy = imnoise(Blurred,'gaussian',0,V);
f3 = figure;
imshow(BlurredNoisy);
figure(f3);
title('Blurred & Noisy');
```

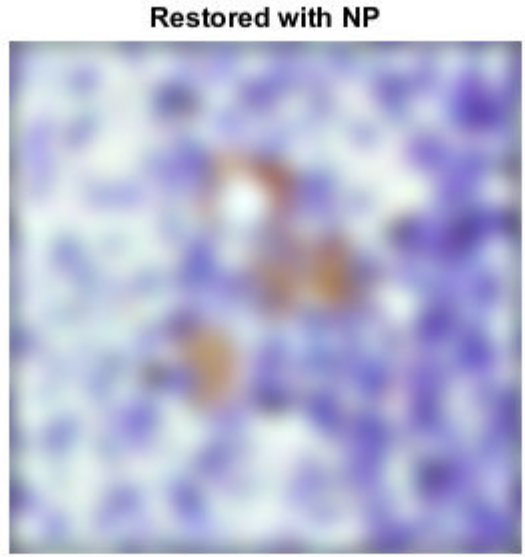


Step 3: Restore the Blurred and Noisy Image

Restore the blurred and noisy image supplying noise power, NP, as the third input parameter. To illustrate how sensitive the algorithm is to the value of noise power, NP, the example performs three restorations.

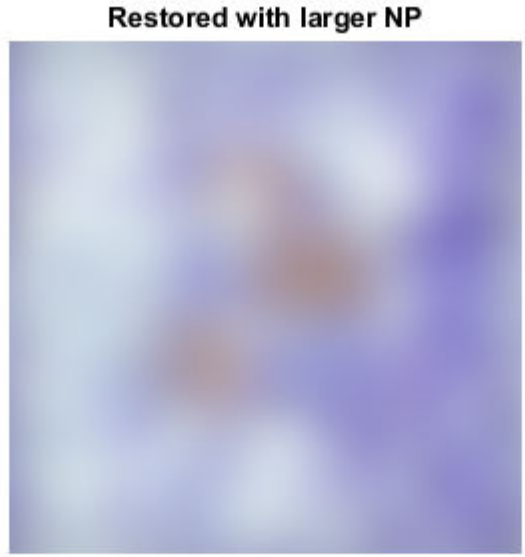
The first restoration, reg1, uses the true NP. Note that the example outputs two parameters here. The first return value, reg1, is the restored image. The second return value, LAGRA, is a scalar, Lagrange multiplier, on which the deconvreg has converged. This value is used later in the example.

```
NP = V*numel(I); % noise power
[reg1, LAGRA] = deconvreg(BlurredNoisy,PSF,NP);
f4 = figure;
imshow(reg1);
figure(f4);
title('Restored with NP');
```



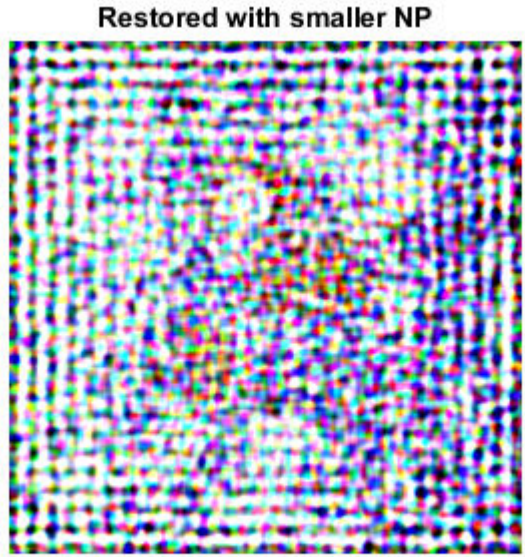
The second restoration, reg2, uses a slightly over-estimated noise power, which leads to a poor resolution.

```
reg2 = deconvreg(BlurredNoisy,PSF,NP*1.3);
f5 = figure;
imshow(reg2);
figure(f5);
title('Restored with larger NP');
```



The third restoration, reg3, is given an under-estimated NP value. This leads to an overwhelming noise amplification and "ringing" from the image borders.

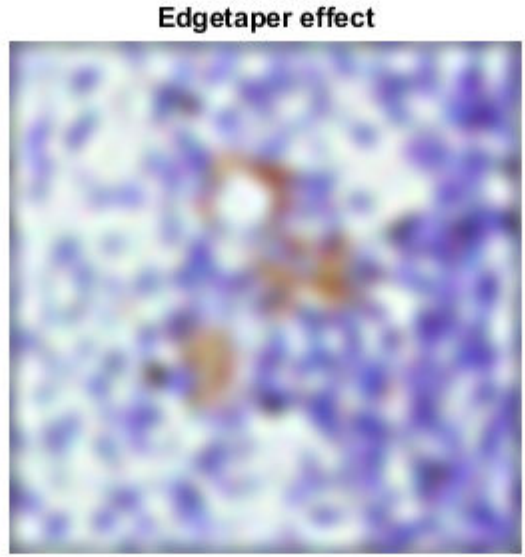
```
reg3 = deconvreg(BlurredNoisy,PSF,NP/1.3);
f6 = figure;
imshow(reg3);
figure(f6);
title('Restored with smaller NP');
```



Step 4: Reduce Noise Amplification and Ringing

Reduce the noise amplification and "ringing" along the boundary of the image by calling the edgetaper function prior to deconvolution. Note how the image restoration becomes less sensitive to the noise power parameter. Use the noise power value NP from the previous example.

```
Edged = edgetaper(BlurredNoisy,PSF);
reg4 = deconvreg(Edged,PSF,NP/1.3);
f7 = figure;
imshow(reg4);
figure(f7);
title('Edgetaper effect');
```

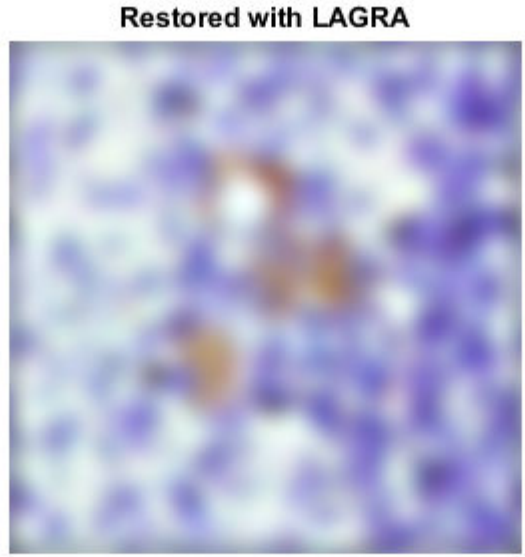


Step 5: Use the Lagrange Multiplier

Restore the blurred and noisy image, assuming that the optimal solution is already found and the corresponding Lagrange multiplier, LAGRA, is given. In this case, any value passed for noise power, NP, is ignored.

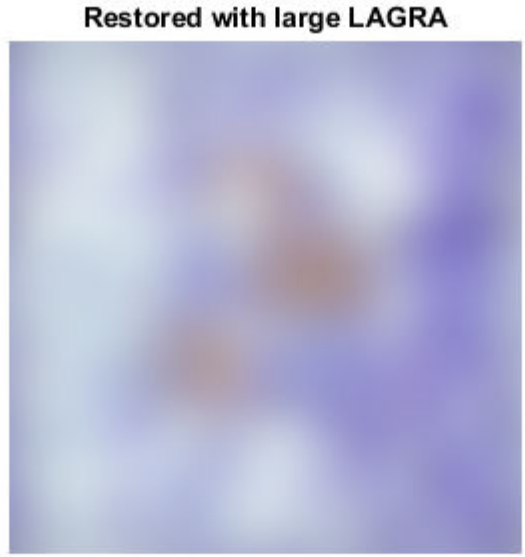
To illustrate how sensitive the algorithm is to the LAGRA value, the example performs three restorations. The first restoration (reg5) uses the LAGRA output from the earlier solution (LAGRA output from first solution in Step 3).

```
reg5 = deconvreg(Edged,PSF,[],LAGRA);
f8 = figure;
imshow(reg5);
figure(f8);
title('Restored with LAGRA');
```



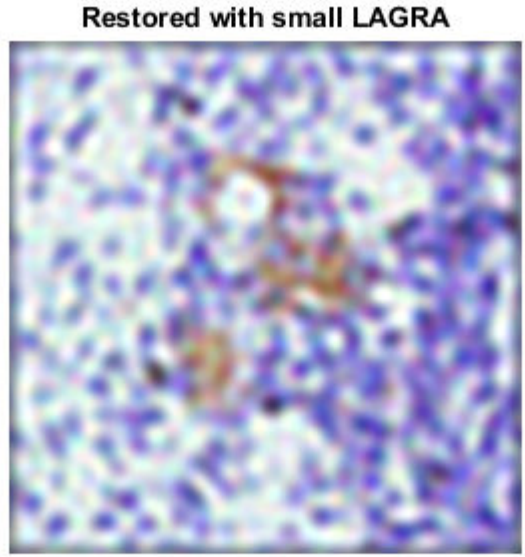
The second restoration (reg6) uses 100*LAGRA which increases the significance of the constraint. By default, this leads to over-smoothing of the image.

```
reg6 = deconvreg(Edged,PSF,[],LAGRA*100);
f9 = figure;
imshow(reg6);
figure(f9);
title('Restored with large LAGRA');
```



The third restoration uses LAGRA/100 which weakens the constraint (the smoothness requirement set for the image). It amplifies the noise and eventually leads to a pure inverse filtering for LAGRA = 0.

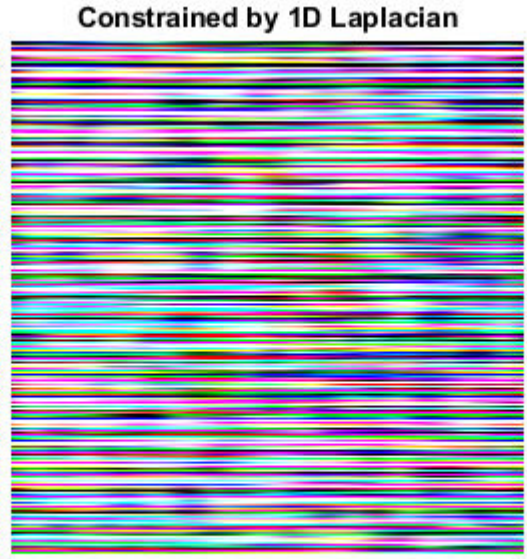
```
reg7 = deconvreg(Edged,PSF,[],LAGRA/100);
f10 = figure;
imshow(reg7);
figure(f10);
title('Restored with small LAGRA');
```



Step 6: Use a Different Constraint

Restore the blurred and noisy image using a different constraint (REGOP) in the search for the optimal solution. Instead of constraining the image smoothness (REGOP is Laplacian by default), constrain the image smoothness only in one dimension (1-D Laplacian).

```
REGOP = [1 -2 1];
reg8 = deconvreg(BlurredNoisy,PSF,[],LAGRA,REGOP);
f11 = figure;
imshow(reg8);
figure(f11);
title('Constrained by 1D Laplacian');
```



See Also

`deconvblind` | `deconvlucy` | `deconvreg` | `deconvwnr`

More About

- “Image Deblurring” on page 14-2
- “Deblur with a Regularized Filter” on page 14-15

Deblur with the Lucy-Richardson Algorithm

In this section...

"Overview" on page 14-30
"Reducing the Effect of Noise Amplification" on page 14-30
"Accounting for Nonuniform Image Quality" on page 14-31
"Handling Camera Read-Out Noise" on page 14-31
"Handling Undersampled Images" on page 14-31
"Example: Using the deconvlucy Function to Deblur an Image" on page 14-32
"Refining the Result" on page 14-34

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 “Avoid Ringing in Deblurred Images” on page 14-73 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.

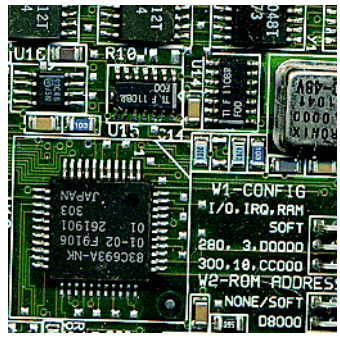
If the undersampled data is the result of camera pixel binning during image acquisition, the PSF observed at each pixel rate can serve as a finer grid PSF. Otherwise, the PSF can be obtained via observations taken at subpixel offsets or via optical modeling techniques. This method is especially effective 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')
   ```

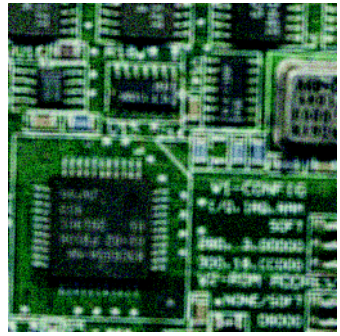
2. Create the PSF.

   ```
   PSF = fspecial('gaussian',5,5);
   ```

3. Create a simulated blur in the image and add noise.

   ```
   Blurred = imfilter(I,PSF,'symmetric','conv');

   V = .002;
   ```

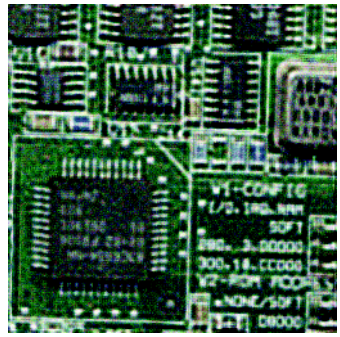
```
BlurredNoisy = imnoise(Blurred,'gaussian',0,V);
figure, imshow(BlurredNoisy)
title('Blurred and Noisy Image')
```



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
`output{1}`	Original input image
`output{2}`	Image produced by the last iteration
`output{3}`	Image produced by the next to last iteration
`output{4}`	Internal information used by `deconvlucy` 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, see the example “Deblurring Images Using the Lucy-Richardson Algorithm” on page 14-35.

See Also

`deconvblind` | `deconvlucy` | `deconvreg` | `deconvwnr`

Related Examples

- “Deblurring Images Using the Lucy-Richardson Algorithm” on page 14-35

More About

- “Image Deblurring” on page 14-2

Deblurring Images Using the Lucy-Richardson Algorithm

This example shows how to use the Lucy-Richardson algorithm to deblur images. It can be used effectively when the point-spread function PSF (blurring operator) is known, but little or no information is available for the noise. The blurred and noisy image is restored by the iterative, accelerated, damped Lucy-Richardson algorithm. You can use characteristics of the optical system as input parameters to improve the quality of the image restoration.

Step 1: Read Image

The example reads in an RGB image and crops it to be 256-by-256-by-3. The `deconvlucy` function can handle arrays of any dimension.

```
I = imread('board.tif');
I = I(50+(1:256),2+(1:256),:);
figure;
imshow(I);
title('Original Image');
text(size(I,2),size(I,1)+15, ...
    'Image courtesy of courtesy of Alexander V. Panasyuk, Ph.D.', ...
    'FontSize',7,'HorizontalAlignment','right');
text(size(I,2),size(I,1)+25, ...
    'Harvard-Smithsonian Center for Astrophysics', ...
    'FontSize',7,'HorizontalAlignment','right');
```

Original Image

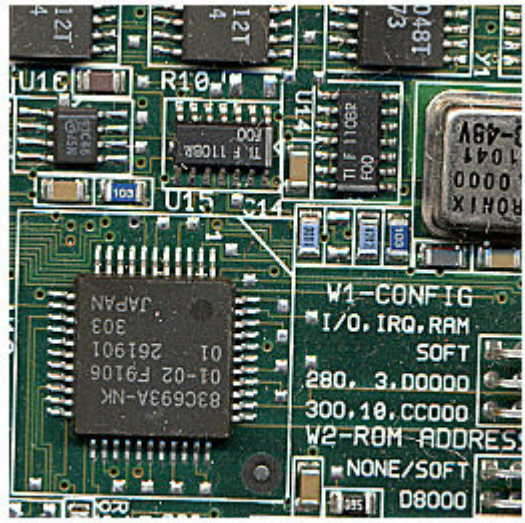
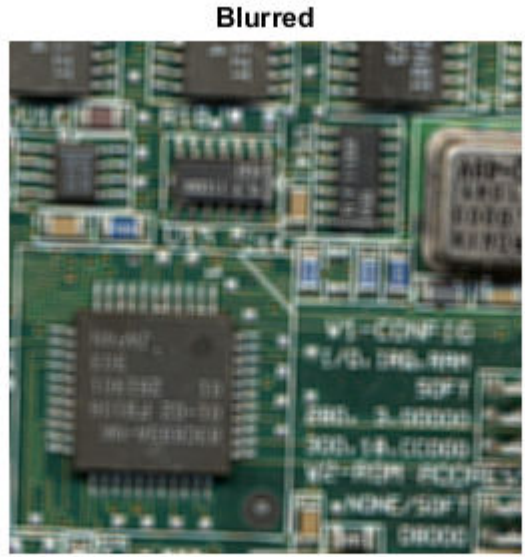


Image courtesy of courtesy of Alexander V. Panasyuk, Ph.D.
Harvard-Smithsonian Center for Astrophysics

Step 2: Simulate a Blur and Noise

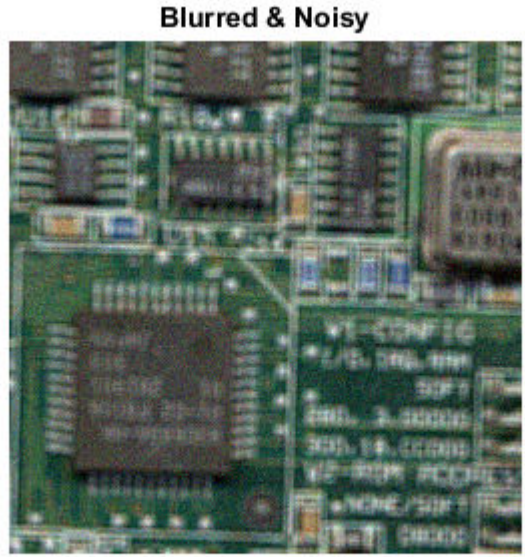
Simulate a real-life image that could be blurred due to camera motion or lack of focus. The image could also be noisy due to random disturbances. The example simulates the blur by convolving a Gaussian filter with the true image (using `imfilter`). The Gaussian filter then represents a point-spread function, `PSF`.

```
PSF = fspecial('gaussian',5,5);
Blurred = imfilter(I,PSF,'symmetric','conv');
figure;
imshow(Blurred);
title('Blurred');
```



The example simulates the noise by adding a Gaussian noise of variance `V` to the blurred image (using `imnoise`). The noise variance `V` is used later to define a damping parameter of the algorithm.

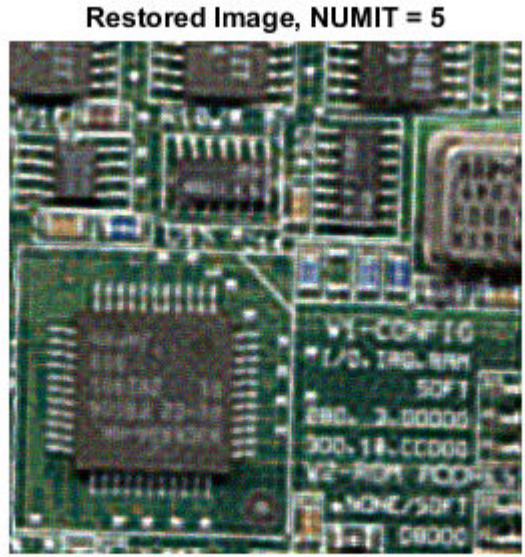
```
V = .002;
BlurredNoisy = imnoise(Blurred,'gaussian',0,V);
figure;
imshow(BlurredNoisy);
title('Blurred & Noisy');
```



Step 3: Restore the Blurred and Noisy Image

Restore the blurred and noisy image providing the PSF and using only 5 iterations (default is 10). The output is an array of the same type as the input image.

```
luc1 = deconvlucy(BlurredNoisy,PSF,5);
figure;
imshow(luc1);
title('Restored Image, NUMIT = 5');
```



Step 4: Iterate to Explore the Restoration

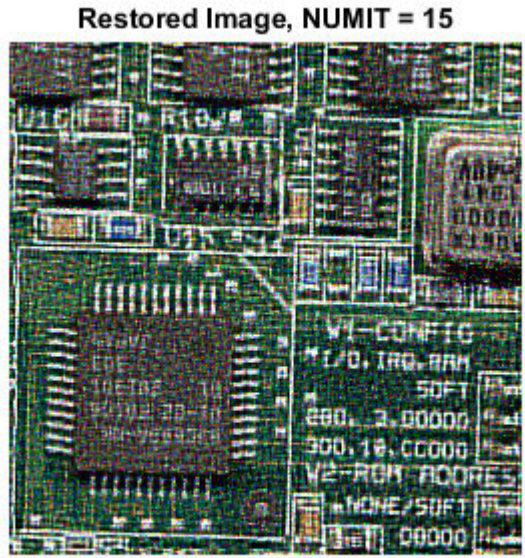
The resulting image changes with each iteration. To investigate the evolution of the image restoration, you can do the deconvolution in steps: do a set of iterations, see the result, and then resume the iterations from where they were stopped. To do so, the input image has to be passed as a part of a cell array. For example, start the first set of iterations by passing in `{BlurredNoisy}` instead of `BlurredNoisy` as input image parameter.

```
luc1_cell = deconvlucy({BlurredNoisy},PSF,5);
```

In that case the output, `luc1_cell`, becomes a cell array. The cell output consists of four numeric arrays, where the first is the `BlurredNoisy` image, the second is the restored image of class double, the third array is the result of the one-before-last iteration, and the fourth array is an internal parameter of the iterated set. The second numeric array of the output cell-array, image `luc1_cell{2}`, is identical to the output array of the Step 3, image `luc1`, with a possible exception of their class (the cell output always gives the restored image of class double).

To resume the iterations, take the output from the previous function call, the cell-array `luc1_cell`, and pass it into the `deconvlucy` function. Use the default number of iterations (`NUMIT` = 10). The restored image is the result of a total of 15 iterations.

```
luc2_cell = deconvlucy(luc1_cell,PSF);
luc2 = im2uint8(luc2_cell{2});
figure;
imshow(luc2);
title('Restored Image, NUMIT = 15');
```



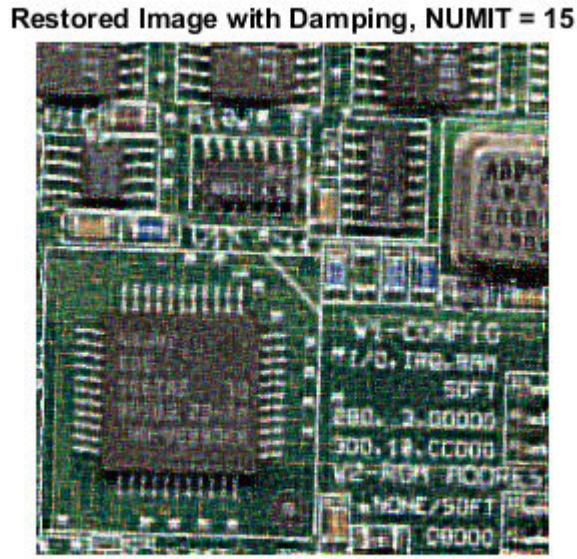
Step 5: Control Noise Amplification by Damping

The latest image, `luc2`, is the result of 15 iterations. Although it is sharper than the earlier result from 5 iterations, the image develops a "speckled" appearance. The speckles do not correspond to any real structures (compare it to the true image), but instead are the result of fitting the noise in the data too closely.

To control the noise amplification, use the damping option by specifying the `DAMPAR` parameter. `DAMPAR` has to be of the same class as the input image. The algorithm

dampens changes in the model in regions where the differences are small compared with the noise. The DAMPAR used here equals 3 standard deviations of the noise. Notice that the image is smoother.

```
DAMPAR = im2uint8(3*sqrt(V));
luc3 = deconvlucy(BlurredNoisy,PSF,15,DAMPAR);
figure;
imshow(luc3);
title('Restored Image with Damping, NUMIT = 15');
```



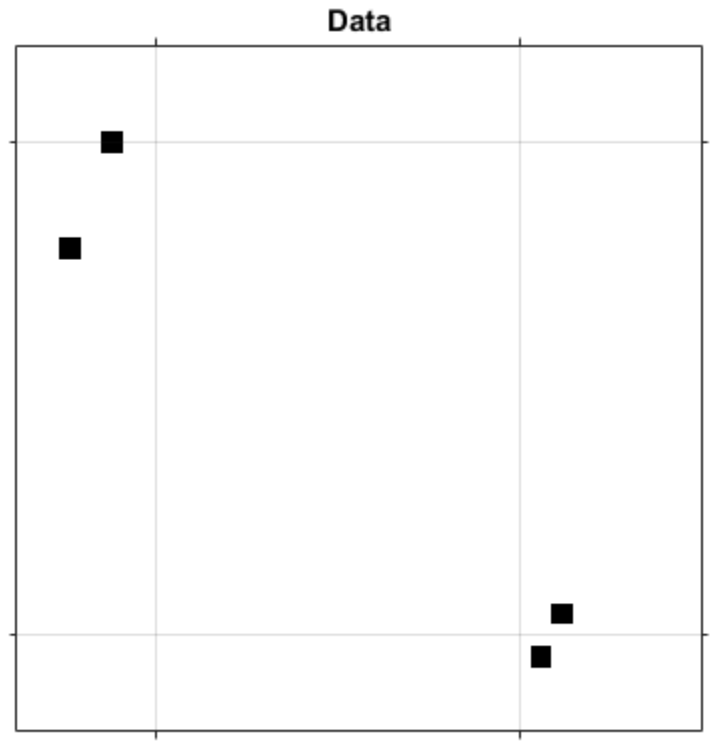
The next part of this example explores the WEIGHT and SUBSMPL input parameters of the deconvlucy function, using a simulated star image (for simplicity & speed).

Step 6: Create Sample Image

The example creates a black/white image of four stars.

```
I = zeros(32);
I(5,5) = 1;
```

```
I(10,3) = 1;
I(27,26) = 1;
I(29,25) = 1;
figure;
imshow(1-I,[],'InitialMagnification','fit');
ax = gca;
ax.Visible = 'on';
ax.XTickLabel = [];
ax.YTickLabel = [];
ax.XTick = [7 24];
ax.XGrid = 'on';
ax.YTick = [5 28];
ax.YGrid = 'on';
title('Data');
```



Step 7: Simulate a Blur

The example simulates a blur of the image of the stars by creating a Gaussian filter, PSF, and convolving it with the true image.

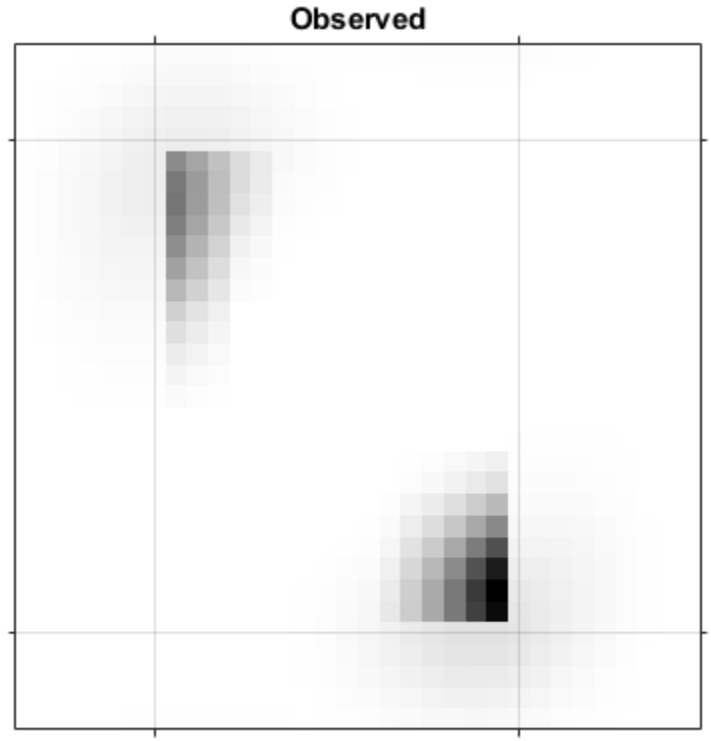
```
PSF = fspecial('gaussian',15,3);
Blurred = imfilter(I,PSF,'conv','sym');
```

Now simulate a camera that can only observe part of the stars' images (only the blur is seen). Create a weighting function array, WEIGHT, that consists of ones in the central part of the Blurred image ("good" pixels, located within the dashed lines) and zeros at the edges ("bad" pixels - those that do not receive the signal).

```
WT = zeros(32);
WT(6:27,8:23) = 1;
CutImage = Blurred.*WT;
```

To reduce the ringing associated with borders, apply the edgetaper function with the given PSF.

```
CutEdged = edgetaper(CutImage,PSF);
figure;
imshow(1-CutEdged,[],'InitialMagnification','fit');
ax = gca;
ax.Visible = 'on';
ax.XTickLabel = [];
ax.YTickLabel = [];
ax.XTick = [7 24];
ax.XGrid = 'on';
ax.YTick = [5 28];
ax.YGrid = 'on';
title('Observed');
```

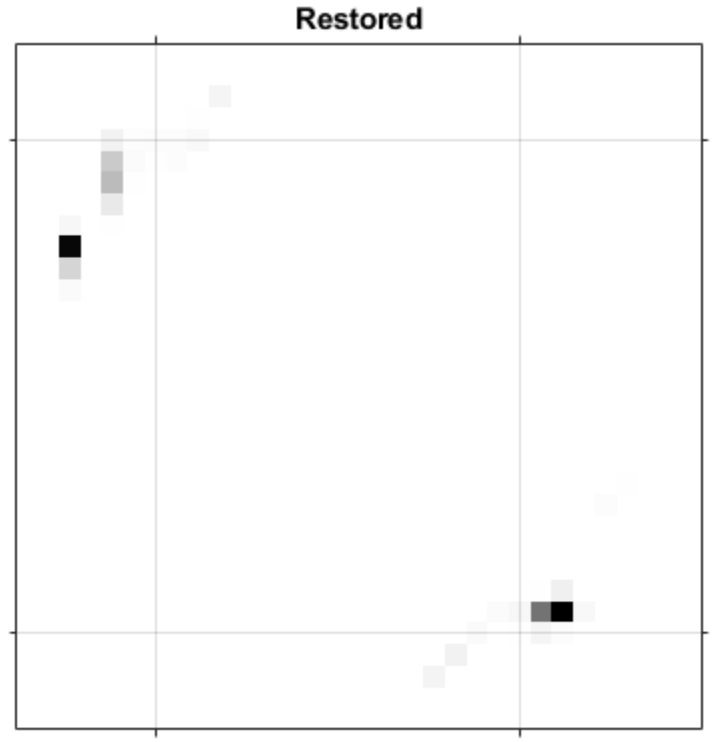


Step 8: Provide the WEIGHT Array

The algorithm weights each pixel value according to the WEIGHT array while restoring the image. In our example, only the values of the central pixels are used (where WEIGHT = 1), while the "bad" pixel values are excluded from the optimization. However, the algorithm can place the signal power into the location of these "bad" pixels, beyond the edge of the camera's view. Notice the accuracy of the resolved star positions.

```
luc4 = deconvlucy(CutEdged,PSF,300,0,WT);
figure;
imshow(1-luc4,[],'InitialMagnification','fit');
ax = gca;
ax.Visible = 'on';
ax.XTickLabel = [];
```

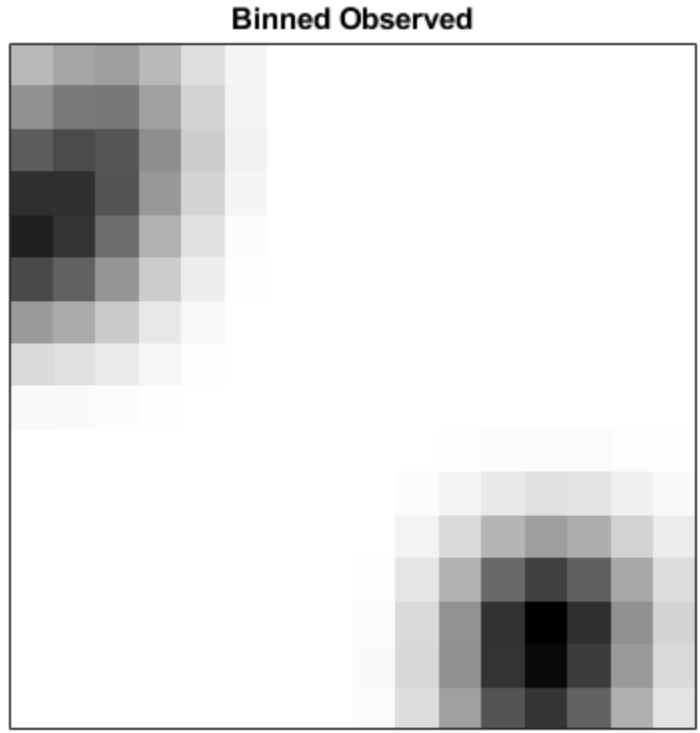
```
ax.YTickLabel = [];
ax.XTick = [7 24];
ax.XGrid = 'on';
ax.YTick = [5 28];
ax.YGrid = 'on';
title('Restored');
```



Step 9: Provide a finer-sampled PSF

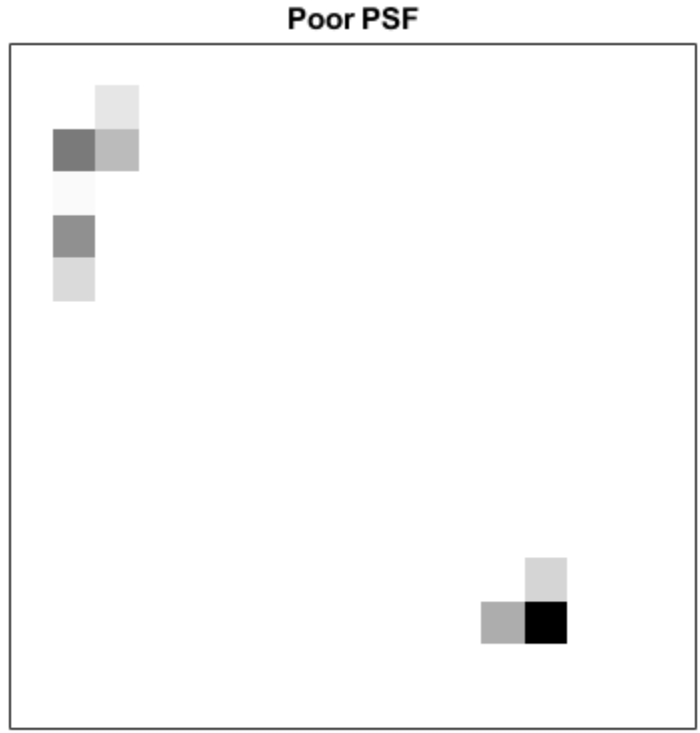
deconvlucy can restore undersampled image given a finer sampled PSF (finer by SUBSMPL times). To simulate the poorly resolved image and PSF, the example bins the `Blurred` image and the original PSF, two pixels in one, in each dimension.

```
Binned = squeeze(sum(reshape(Blurred,[2 16 2 16])));
BinnedImage = squeeze(sum(Binned,2));
Binned = squeeze(sum(reshape(PSF(1:14,1:14),[2 7 2 7])));
BinnedPSF = squeeze(sum(Binned,2));
figure;
imshow(1-BinnedImage,[],'InitialMagnification','fit');
ax = gca;
ax.Visible = 'on';
ax.XTick = [];
ax.YTick = [];
title('Binned Observed');
```



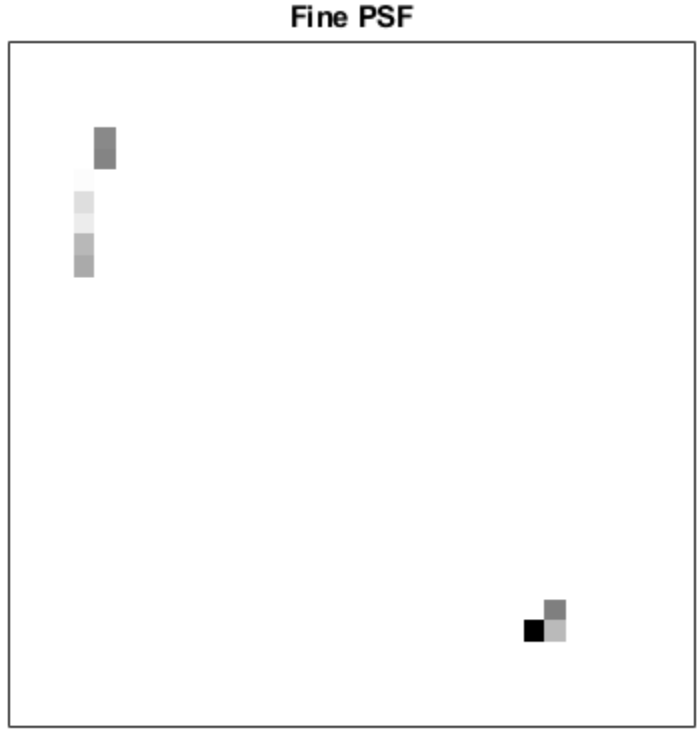
Restore the undersampled image, `BinnedImage`, using the undersampled PSF, `BinnedPSF`. Notice that the `luc5` image distinguishes only 3 stars.

```
luc5 = deconvlucy(BinnedImage,BinnedPSF,100);
figure;
imshow(1-luc5,[],'InitialMagnification','fit');
ax = gca;
ax.Visible = 'on';
ax.XTick = [];
ax.YTick = [];
title('Poor PSF');
```



The next example restores the undersampled image (`BinnedImage`), this time using the finer PSF (defined on a SUBSMPL-times finer grid). The reconstructed image (`luc6`) resolves the position of the stars more accurately. Note how it distributes power between the two stars in the lower right corner of the image. This hints at the existence of two bright objects, instead of one, as in the previous restoration.

```
luc6 = deconvlucy(BinnedImage,PSF,100,[],[],[],2);
figure;
imshow(1-luc6,[],'InitialMagnification','fit');
ax = gca;
ax.Visible = 'on';
ax.XTick = [];
ax.YTick = [];
title('Fine PSF');
```



See Also

deconvblind | deconvlucy | deconvreg | deconvwnr

More About

- “Image Deblurring” on page 14-2
- “Deblur with the Lucy-Richardson Algorithm” on page 14-30

Deblur 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 “Deblur with the Lucy-Richardson Algorithm” on page 14-30. In addition, the `deconvblind` function supports PSF constraints that can be passed in through a user-specified function.

Deblur images using blind deconvolution

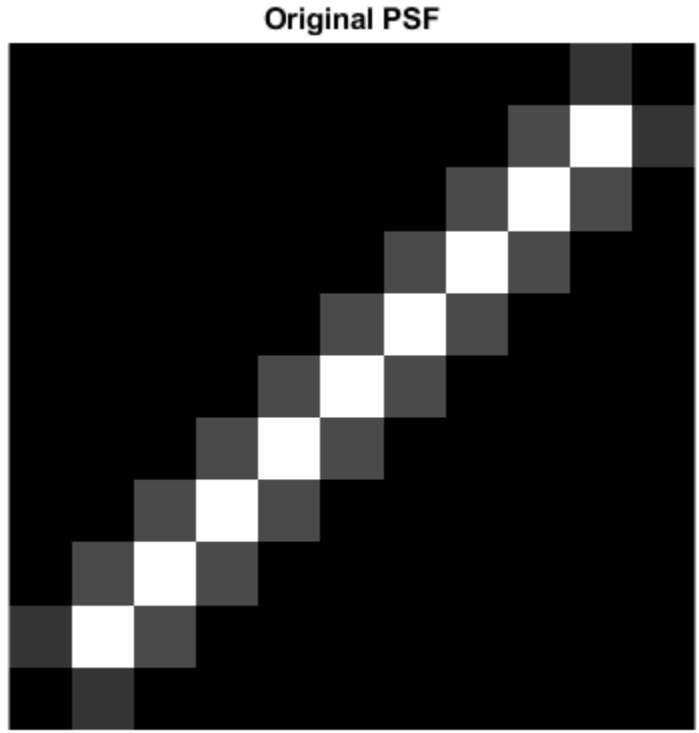
This example shows how to deblur an image using blind deconvolution. The example illustrates the iterative nature of this operation, making two passes at deblurring the image using optional parameters.

Read an image into the workspace and display it.

```
I = imread('cameraman.tif');
figure
imshow(I)
title('Original Image')
```

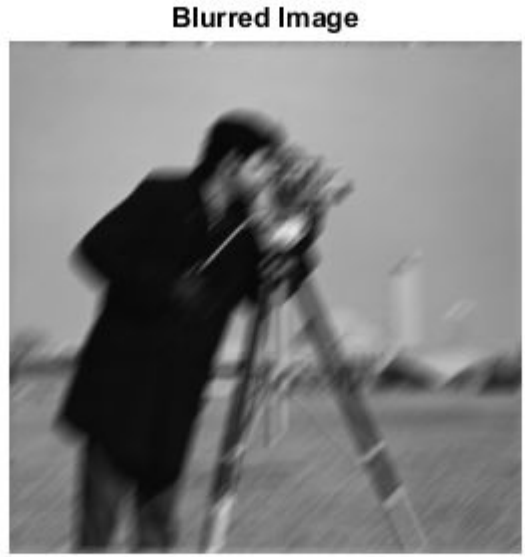
Create a point spread function (PSF). A PSF describes the degree to which an optical system blurs (spreads) a point of light.

```
PSF = fspecial('motion',13,45);
figure
imshow(PSF,[],'InitialMagnification','fit')
title('Original PSF')
```



Create a simulated blur in the image, using the PSF, and display the blurred image.

```
Blurred = imfilter(I,PSF,'circ','conv');
figure
imshow(Blurred)
title('Blurred Image')
```

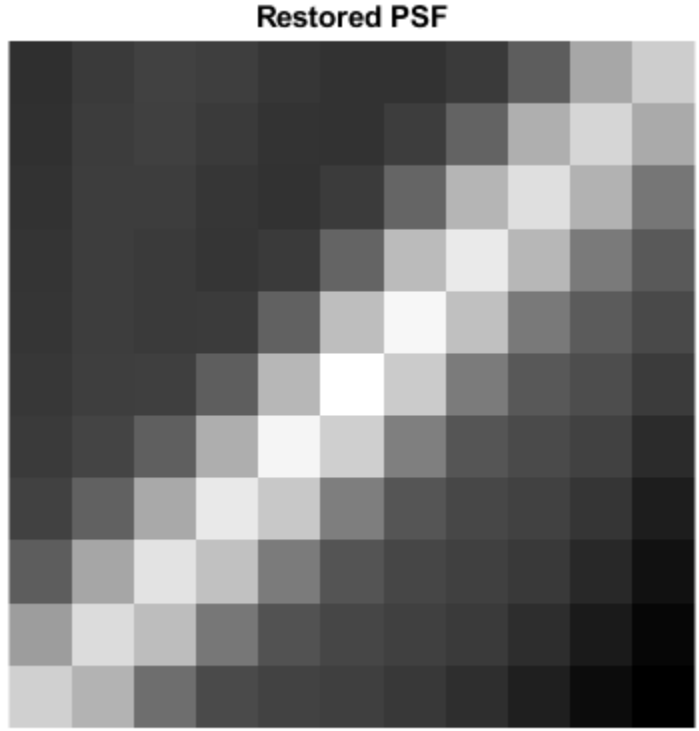


Deblur the image using the `deconvblind` function. You must make an initial guess at the PSF. To determine the size of the PSF, examine the blurred image and measure the width of a blur (in pixels) around an obviously sharp object. Because the size of the PSF is more important than the values it contains, you can typically specify an array of 1's as the initial PSF.

In this initial restoration, `deconvblind` was able to deblur the image to a great extent. Note, however, the ringing around the sharp intensity contrast areas in the restored image. (The example eliminated edge-related ringing by using the 'circular' option with `imfilter` when creating the simulated blurred image.) To achieve a more satisfactory result, rerun the operation, experimenting with PSFs of different sizes. The restored PSF returned by each deconvolution can also provide valuable hints at the optimal PSF size.

```
INITPSF = ones(size(PSF));
[J P] = deconvblind(Blurred,INITPSF,30);
figure
imshow(J)
title('Restored Image')
```

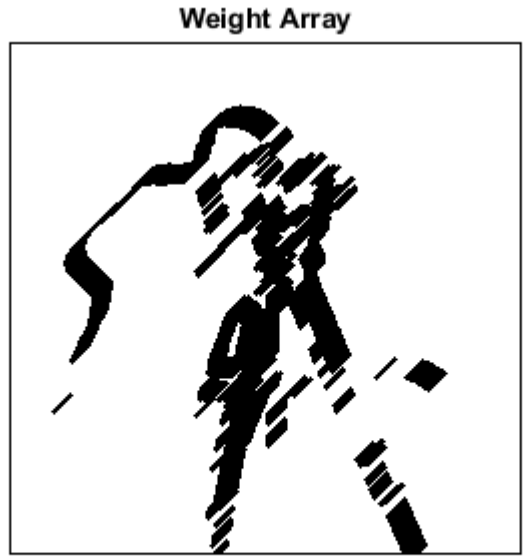
```
figure
imshow(P,[],'InitialMagnification','fit')
title('Restored PSF')
```



One way to improve the result is to create a weight array to exclude areas of high contrast from the deblurring operation. This can reduce contrast-related ringing in the result.

To create a weight array, create an array the same size as the 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. 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. 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),[1 1]);
figure
imshow(WEIGHT)
title('Weight Array')
```

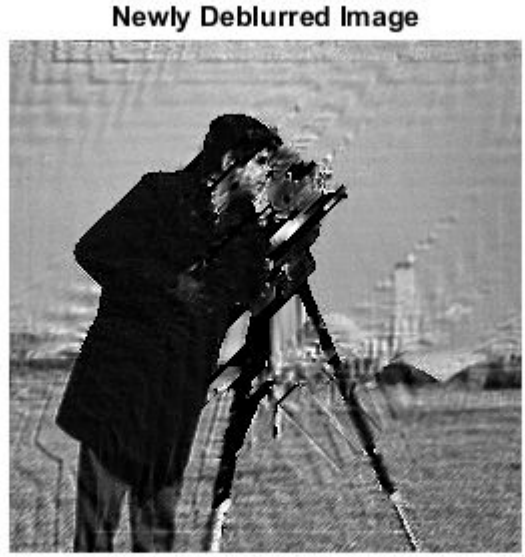


Refine the guess at the PSF. The reconstructed PSF returned by the first pass at deconvolution, P , shows a clear linearity. For this second pass, the example uses a new PSF which is the same as the returned PSF but with the small amplitude pixels set to 0.

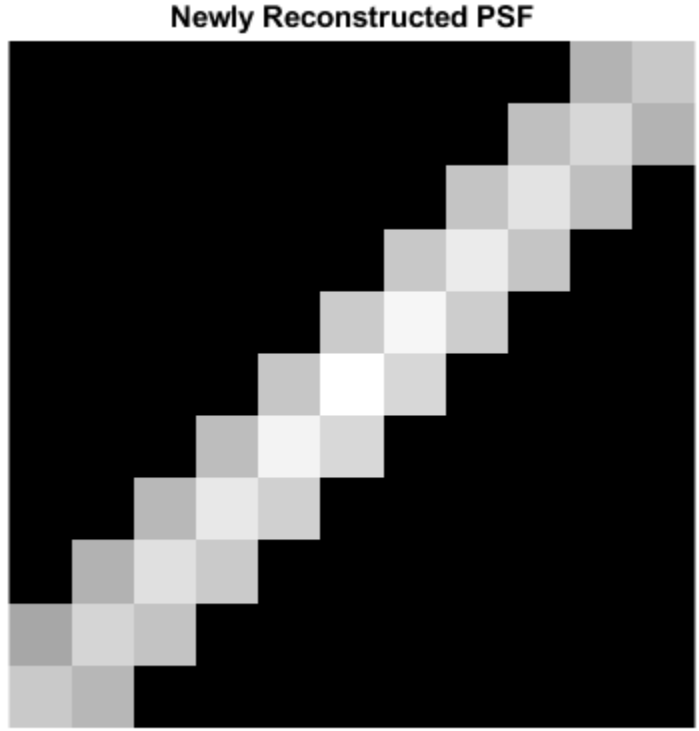
```
P1 = P;
P1(find(P1 < 0.01))= 0;
```

Run the deconvolution again, this time specifying the weight array and the modified PSF. Note how the restored image has much less ringing around the sharp intensity areas than the result of the first pass.

```
[J2 P2] = deconvblind(Blurred,P1,50,[],double(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
`output{1}`	Original input image
`output{2}`	Image produced by the last iteration
`output{3}`	Image produced by the next to last iteration
`output{4}`	Internal information used by `deconvlucy` 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 example “Deblurring Images Using the Blind Deconvolution Algorithm” on page 14-60.

See Also

`deconvblind` | `deconvlucy` | `deconvreg` | `deconvwnr`

Related Examples

- “Deblurring Images Using the Blind Deconvolution Algorithm” on page 14-60

More About

- “Image Deblurring” on page 14-2

Deblurring Images Using the Blind Deconvolution Algorithm

This example shows how to use blind deconvolution to deblur images. The blind deconvolution algorithm can be used effectively when no information about the distortion (blurring and noise) is known. The algorithm restores the image and the point-spread function (PSF) simultaneously. The accelerated, damped Richardson-Lucy algorithm is used in each iteration. Additional optical system (e.g. camera) characteristics can be used as input parameters that could help to improve the quality of the image restoration. PSF constraints can be passed in through a user-specified function.

Step 1: Read Image

Read a grayscale image into the workspace. The `deconvblind` function can handle arrays of any dimension.

```
I = imread('cameraman.tif');
figure;imshow(I);title('Original Image');
text(size(I,2),size(I,1)+15, ...
    'Image courtesy of Massachusetts Institute of Technology', ...
    'FontSize',7,'HorizontalAlignment','right');
```

Image courtesy of Massachusetts Institute of Technology

Step 2: Simulate a Blur

Simulate a real-life image that could be blurred (e.g., due to camera motion or lack of focus). The example simulates the blur by convolving a Gaussian filter with the true image (using `imfilter`). The Gaussian filter then represents a point-spread function, `PSF`.

```
PSF = fspecial('gaussian',7,10);
Blurred = imfilter(I,PSF,'symmetric','conv');
imshow(Blurred)
title('Blurred Image')
```

Step 3: Restore the Blurred Image Using PSFs of Various Sizes

To illustrate the importance of knowing the size of the true PSF, this example performs three restorations. Each time the PSF reconstruction starts from a uniform array (an array of ones).

The first restoration, `J1` and `P1`, uses an undersized array, `UNDERPSF`, for an initial guess of the PSF. The size of the UNDERPSF array is 4 pixels shorter in each dimension than the true PSF.

```
UNDERPSF = ones(size(PSF)-4);
[J1,P1] = deconvblind(Blurred,UNDERPSF);
imshow(J1)
title('Deblurring with Undersized PSF')
```

The second restoration, J2 and P2, uses an array of ones, OVERPSF, for an initial PSF that is 4 pixels longer in each dimension than the true PSF.

```
OVERPSF = padarray(UNDERPSF,[4 4],'replicate','both');
[J2,P2] = deconvblind(Blurred,OVERPSF);
imshow(J2)
title('Deblurring with Oversized PSF')
```

The third restoration, J3 and P3, uses an array of ones, INITPSF, for an initial PSF that is exactly of the same size as the true PSF.

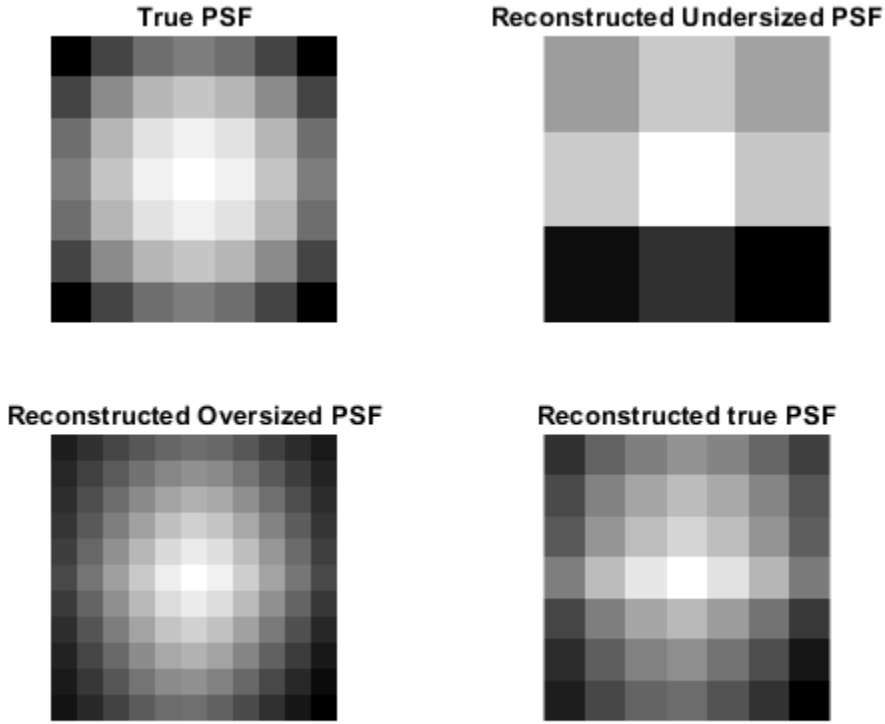
```
INITPSF = padarray(UNDERPSF,[2 2],'replicate','both');
[J3,P3] = deconvblind(Blurred,INITPSF);
imshow(J3)
title('Deblurring with INITPSF')
```

Step 4: Analyzing the Restored PSF

All three restorations also produce a PSF. The following pictures show how the analysis of the reconstructed PSF might help in guessing the right size for the initial PSF. In the true PSF, a Gaussian filter, the maximum values are at the center (white) and diminish at the borders (black).

```
figure;
subplot(2,2,1)
imshow(PSF,[],'InitialMagnification','fit')
title('True PSF')
subplot(222)
imshow(P1,[],'InitialMagnification','fit')
title('Reconstructed Undersized PSF')
subplot(2,2,3)
imshow(P2,[],'InitialMagnification','fit')
title('Reconstructed Oversized PSF')
subplot(2,2,4)
```

```
imshow(P3,[],'InitialMagnification','fit')
title('Reconstructed true PSF')
```



The PSF reconstructed in the first restoration, `P1`, obviously does not fit into the constrained size. It has a strong signal variation at the borders. The corresponding image, `J1`, does not show any improved clarity vs. the blurred image, `Blurred`.

The PSF reconstructed in the second restoration, `P2`, becomes very smooth at the edges. This implies that the restoration can handle a PSF of a smaller size. The corresponding image, `J2`, shows some deblurring but it is strongly corrupted by the ringing.

Finally, the PSF reconstructed in the third restoration, `P3`, is somewhat intermediate between `P1` and `P2`. The array, `P3`, resembles the true PSF very well. The corresponding image, `J3`, shows significant improvement; however it is still corrupted by the ringing.

Step 5: Improving the Restoration

The ringing in the restored image, `J3`, occurs along the areas of sharp intensity contrast in the image and along the image borders. This example shows how to reduce the ringing effect by specifying a weighting function. The algorithm weights each pixel according to the `WEIGHT` array while restoring the image and the PSF. In our example, we start by finding the "sharp" pixels using the edge function. By trial and error, we determine that a desirable threshold level is 0.08.

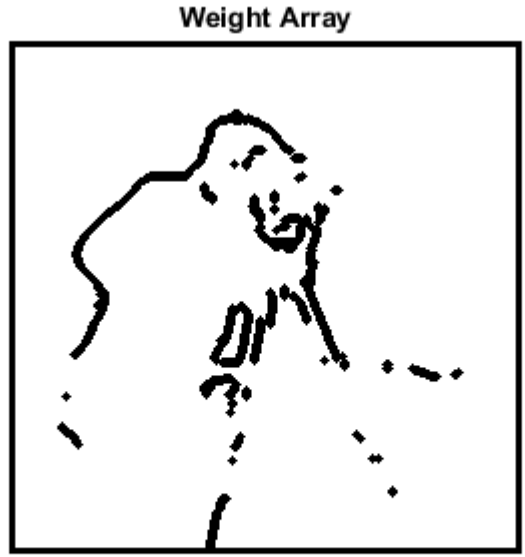
```
WEIGHT = edge(Blurred,'sobel',.08);
```

To widen the area, we use `imdilate` and pass in a structuring element, `se`.

```
se = strel('disk',2);
WEIGHT = 1-double(imdilate(WEIGHT,se));
```

The pixels close to the borders are also assigned the value 0.

```
WEIGHT([1:3 end-(0:2)],:) = 0;
WEIGHT(:,[1:3 end-(0:2)]) = 0;
figure
imshow(WEIGHT)
title('Weight Array')
```



The image is restored by calling deconvblind with the `WEIGHT` array and an increased number of iterations (30). Almost all the ringing is suppressed.

```
[J,P] = deconvblind(Blurred,INITPSF,30,[],WEIGHT);
imshow(J)
title('Deblurred Image')
```

Step 6: Using Additional Constraints on the PSF Restoration

The example shows how you can specify additional constraints on the PSF. The function, `FUN`, below returns a modified PSF array which deconvblind uses for the next iteration.

In this example, `FUN` modifies the PSF by cropping it by `P1` and `P2` number of pixels in each dimension, and then padding the array back to its original size with zeros. This operation does not change the values in the center of the PSF, but effectively reduces the PSF size by `2*P1` and `2*P2` pixels.

```
P1 = 2;
P2 = 2;
FUN = @(PSF) padarray(PSF(P1+1:end-P1,P2+1:end-P2),[P1 P2]);
```

The anonymous function, `FUN`, is passed into `deconvblind` last. See the section Parameterizing Functions, in the MATLAB Mathematics documentation, for information about providing additional parameters to the function `FUN`.

In this example, the size of the initial PSF, OVERPSF, is 4 pixels larger than the true PSF. Setting P1 = 2 and P2 = 2 as parameters in FUN effectively makes the valuable space in OVERPSF the same size as the true PSF. Therefore, the outcome, JF and PF, is similar to the result of deconvolution with the right sized PSF and no FUN call, J and P, from step 4.

```
[JF,PF] = deconvblind(Blurred,OVERPSF,30,[],WEIGHT,FUN);
imshow(JF)
title('Deblurred Image')
```

If we had used the oversized initial PSF, OVERPSF, without the constraining function, FUN, the resulting image would be similar to the unsatisfactory result, J2, achieved in Step 3.

Note, that any unspecified parameters before FUN can be omitted, such as DAMPAR and READOUT in this example, without requiring a place holder, ([]).

See Also

deconvblind | deconvlucy | deconvreg | deconvwnr

More About

- “Image Deblurring” on page 14-2
- “Deblur with the Blind Deconvolution Algorithm” on page 14-50

Create 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.)

See Also

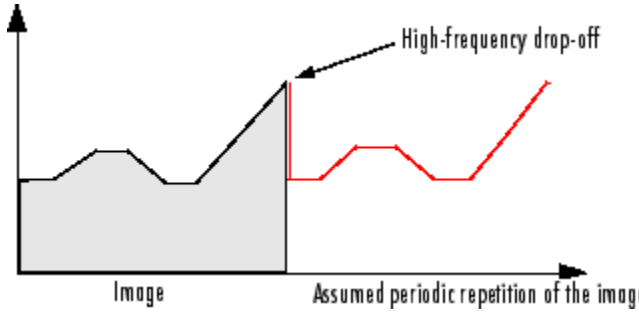
`deconvblind` | `deconvlucy` | `deconvreg` | `deconvwnr`

More About

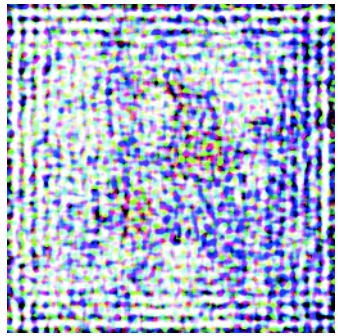
- "Image Deblurring" on page 14-2

Avoid 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.

See Also

`deconvblind` | `deconvlucy` | `deconvreg` | `deconvwnr`

More About

- “Image Deblurring” on page 14-2
- “Fourier Transform” on page 9-2

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.

- “Display Colors” on page 15-2
- “Reduce the Number of Colors in an Image” on page 15-4
- “Profile-Based Color Space Conversions” on page 15-13
- “Device-Independent Color Spaces” on page 15-17
- “Understanding Color Spaces and Color Space Conversion” on page 15-20
- “Convert from HSV to RGB Color Space” on page 15-21
- “Convert from YIQ to RGB Color Space” on page 15-26
- “Convert from YCbCr to RGB Color Space” on page 15-27
- “Determine If L*a*b* Value Is in RGB Gamut” on page 15-28
- “Comparison of Auto White Balance Algorithms” on page 15-29

Display 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 “Add Transparency to Graphics Objects” (MATLAB).

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 “Reduce the Number of Colors in an Image” on page 15-4.

Reduce the Number of Colors in an Image

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.

The methods to reduce the number of colors in an image include:

In this section...
“Reduce Colors of Truecolor Image Using Color Approximation” on page 15-4 “Reduce Colors of Indexed Image Using imapprox” on page 15-9 “Reduce Colors Using Dithering” on page 15-10

Reduce Colors of Truecolor Image Using Color Approximation

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

- Quantization (described in “Quantization” on page 15-5)
    - Uniform quantization

  - Minimum variance quantization
- Colormap mapping (described in “Colormap Mapping” on page 15-9)

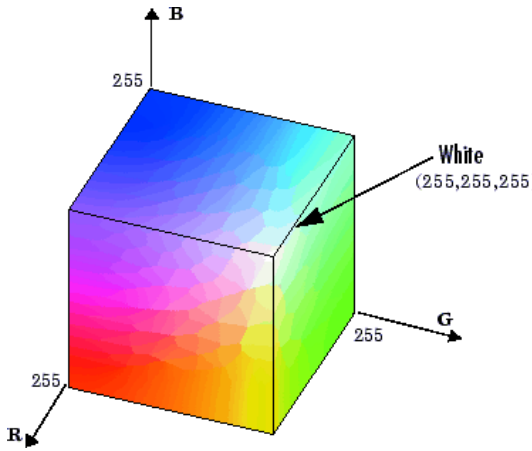
The quality of the resulting image depends on the approximation method you use, the range of colors in the input image, and whether or not you use dithering. Note that different methods work better for different images. See “Reduce Colors Using Dithering” on page 15-10 for a description of dithering 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

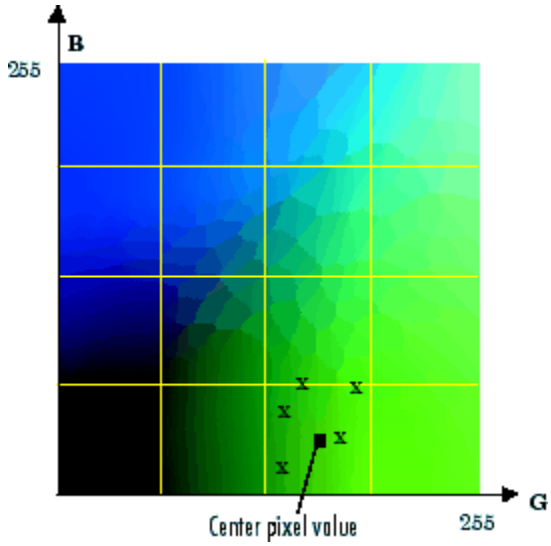
To perform uniform quantization, call `rgb2ind` and specify a *tolerance*. The tolerance determines the size of the cube-shaped boxes into which the RGB color cube is divided. The allowable range for a tolerance setting is [0,1]. For example, if you specify a tolerance of `0.1`, the edges of the boxes 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);
```

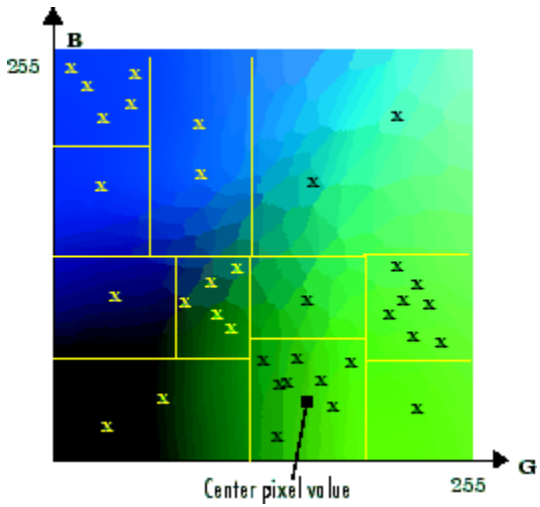
Minimum variance quantization works by associating pixels into groups based on the variance between their pixel values. For example, a set of blue pixels might be grouped together because they have a small variance from the center pixel of the group.

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 `imshow` is also helpful for displaying multiple indexed images. For more information, see “Display Images Individually in the Same Figure” on page 4-12 or the reference page for `imshow`.

Reduce Colors of Indexed Image 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 “Reduce Colors Using Dithering” on page 15-10 for a description of dithering and how to enable or disable it.

Reduce Colors Using 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.

Read image and display it.

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

Create an indexed image with eight colors and without dithering.

```
[X_no_dither,map] = rgb2ind(rgb,8,'nodither');
imshow(X_no_dither,map)
```

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');
imshow(X_dither,map)
```

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, http://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 TIFF. If the format is TIFF and the data is an m-by-n-by-4 array, `imwrite` writes CMYK data to the file.

```
imwrite(I_cmyk,'pep_cmyk.tif','tif')
```

To verify that the CMYK data was written to the file, use `imfinfo` to get information about the file and look at the `PhotometricInterpretation` field.

```
info = imfinfo('pep_cmyk.tif');
info.PhotometricInterpretation
```

```
ans =
   'CMYK'
```

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*, *uvl*, *u′v′L*, and *L*a*b**
xyY	CIE specification that provides normalized chromaticity values. The capital *Y* value represents luminance and is the same as in *XYZ*.	*XYZ*
uvL	CIE specification that attempts to make the chromaticity plane more visually uniform. *L* is luminance and is the same as *Y* in *XYZ*.	*XYZ*
u′v′L	CIE specification in which *u* and *v* are rescaled to improve uniformity.	*XYZ*

Color Space	Description	Supported Conversions
*L*a*b**	CIE specification that attempts to make the luminance scale more perceptually uniform. *L** is a nonlinear scaling of *L*, normalized to a reference white point.	*XYZ*
*L*ch*	CIE specification where *c* is chroma and *h* is hue. These values are a polar coordinate conversion of *a** and *b** in *L*a*b**.	*L*a*b**
sRGB	Standard adopted by major manufacturers that characterizes the average PC monitor.	*XYZ* and *L*a*b**

Color Space Data Encodings

When you convert between two device-independent color spaces, the data type used to encode the color data can sometimes change, depending on what encodings the color spaces support. In the preceding example, the original image is `uint8` data. The XYZ conversion is `uint16` data. The XYZ color space does not define a `uint8` encoding. The following table lists the data types that can be used to represent values in all the device-independent color spaces.

Color Space	Encodings
`XYZ`	`uint16` or `double`
`xyY`	`double`
`uvL`	`double`
`u'v'L`	`double`
`L*a*b*`	`uint8`, `uint16`, or `double`
`L*ch`	`double`
`RGB`	`double uint8 uint16`

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.

See Also

Related Examples

- “Convert from HSV to RGB Color Space” on page 15-21
- “Convert from YIQ to RGB Color Space” on page 15-26
- “Convert from YCbCr to RGB Color Space” on page 15-27
- “Determine If L*a*b* Value Is in RGB Gamut” on page 15-28

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 15-17.

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.

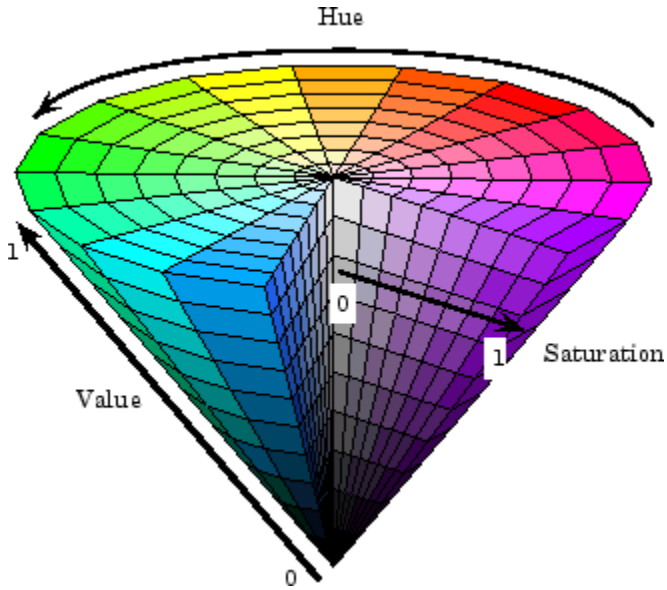


Illustration of the HSV Color Space

Convert RGB image to HSV and Explore HSV Color Space

This example shows how to convert an RGB image to the HSV color space using the `rgb2hsv` function. The example then displays the separate HSV color planes (hue, saturation, and value) of a synthetic image.

Convert an RGB Image into an HSV Image

Read an RGB image into the workspace.

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

Convert it to the HSV color space.

```
HSV = rgb2hsv(RGB);
```

Closer Look at the HSV Color Space

For closer inspection of the HSV color space, create a synthetic RGB image.

```
RGB=reshape(ones(64,1)*reshape(jet(64),1,192),[64,64,3]);
```

Convert the synthetic RGB image to the HSV colorspace.

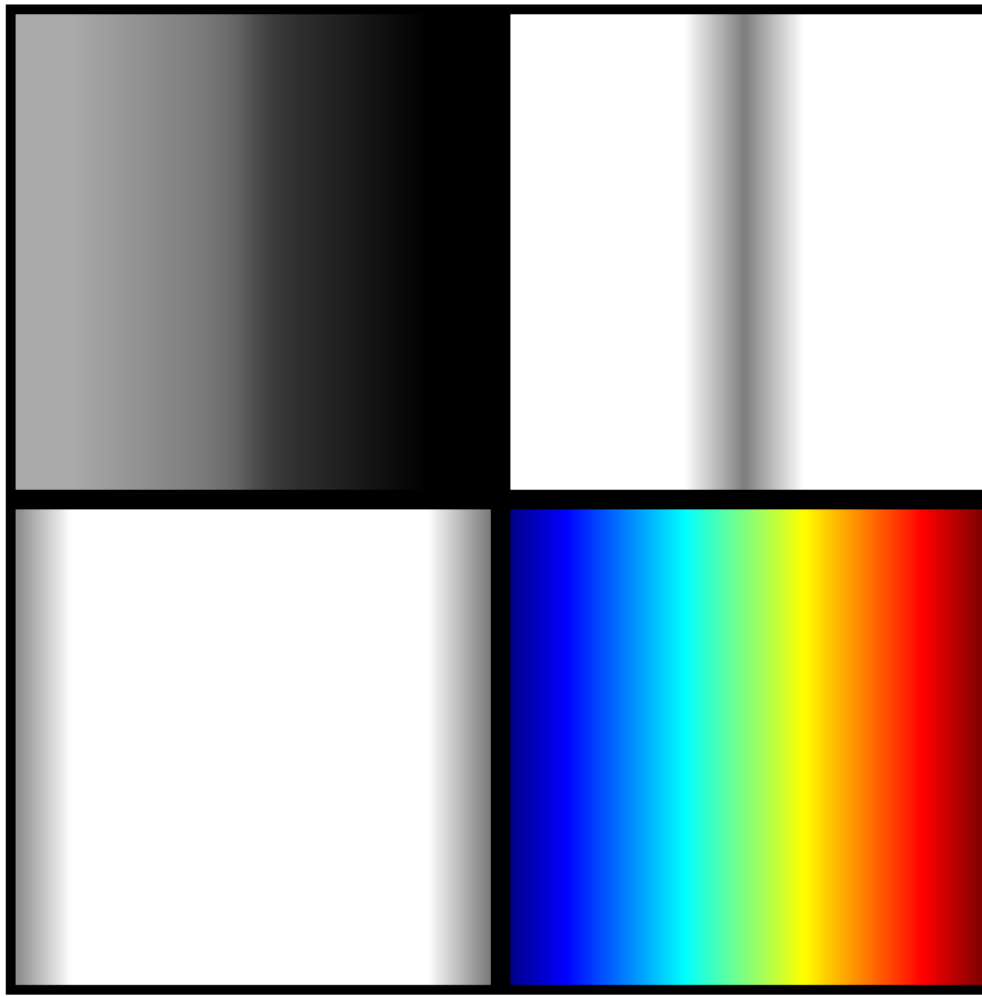
```
HSV=rgb2hsv(RGB);
```

Split the HSV version of the synthetic image into its component planes: hue, saturation, and value.

```
[h s v] = imsplit(HSV);
```

Display the individual HSV color planes with the original image.

```
montage({h, s, v, RGB}, 'BorderSize', 10);
```



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.

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 (non-image) information can be included in a video stream.

The function `rgb2ycbcr` converts colormaps or RGB images to the YCbCr color space. `ycbcr2rgb` performs the reverse operation.

For example, these commands convert an RGB image to YCbCr format.

```
RGB = imread('peppers.png');
YCBCR = rgb2ycbcr(RGB);
```

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 = 1×3

   -0.6209    0.9537   -0.1927
```

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 = 1×3

    0.1236    0.9522    0.1072
```

Comparison of Auto White Balance Algorithms

Our eyes are very good at judging what is white under different lighting conditions. Digital cameras, however, without some kind of adjustment, can easily capture unrealistic images with a strong color cast. Automatic white balance (AWB) algorithms try to correct for the ambient light with minimum input from the user, so that the resulting image looks like what our eyes would see. But which is the best AWB algorithm to use? In this example, we explain the process behind auto white balance and show how to compare and select the best algorithm.

Automatic white balancing is done in two steps:

- Step 1: Estimate the scene illumination.
- Step 2: Correct the color balance of the image.

The hardest step, and the one we will focus on optimizing, is Step 1 -- the estimation of the ambient light. Once the ambient light is known, then correcting the colors in the image (Step 2) is an easy and fixed process.

In the following, we judge the quality of three algorithms for illuminant estimation by comparing them to the ground truth scene illumination:

- White Patch Retinex [1]
- Gray World [2]
- Cheng's Principal Component Analysis (PCA) method [3]

Read 16-bit Image of Linear RGB Values

Auto white balance algorithms are usually applied on the raw image data, before the image is compressed and saved to the memory card. `foosballraw.tiff` is an image file that contains such raw sensor data after correcting the black level and scaling the intensities to 16 bits per pixel. This image is free of the white balancing done by the camera, as well as demosaicking, denoising, chromatic aberration compensation, tone adjustments, gamma correction and other processing done before the image is saved to the memory card.

```
A = imread('foosballraw.tiff');
```

The image `A` contains linear RGB values. Most illuminant estimation algorithms assume a linear relationship between the response of the imaging sensor and pixel intensities. However, because of the nonlinear nature of display devices, image files meant to be

displayed such as JPEG files incorporate a gamma correction. Without gamma correction images would look very dim on a computer screen. If you are working with gamma-corrected images, make sure to linearize them with the `rgb2lin` function. This is not the case with `foosballraw.tiff`, so skip this step.

Interpolate to Recover Missing Color Information

Digital cameras use a color filter array superimposed on the imaging sensor to simulate color vision, so that each pixel is sensitive to either red, green or blue. To recover the missing color information at every pixel, you must interpolate. The Bayer pattern used by the camera with which the photo was captured (Canon EOS 30D) is RGGB.

```
A = demosaic(A,'rggb');
```

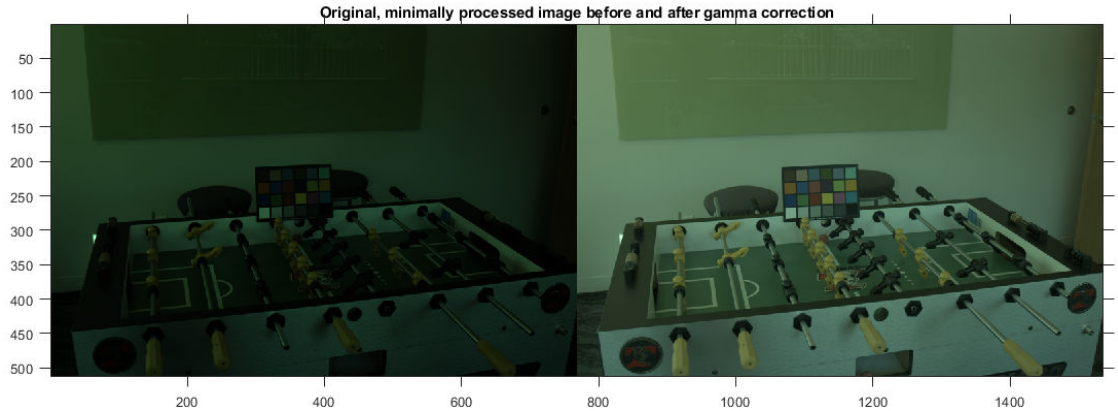
Gamma-Correct the Image for Display

If you try to display the linear image as-is, it will appear very dim, because of the non-linear characteristic of display devices. Therefore, for display purposes, gamma-correct the image to use the sRGB color space.

```
A_sRGB = lin2rgb(A);
```

Display the original, minimally processed image before and after gamma correction

```
warning('off','images:initSize:adjustingMag')
montage({A,A_sRGB})
title('Original, minimally processed image before and after gamma correction')
```



Locate the ColorChecker Chart

A ColorChecker Chart has been included in the scene. This chart is made of 24 neutral and color patches with known spectral reflectances. We will use the 6 neutral (achromatic) patches on the bottom row to estimate the ground truth illumination in the scene, against which the algorithms will be compared. However, when testing the algorithms, the chart must be excluded to prevent the algorithms to unfairly take advantage of it – there is no ColorChecker Chart in real-life situations.

Specify the location of the ColorChecker Chart. By default, the example provides the cooridinates of a bounding quadrilateral. If you want to select the polygon coordinates interactively, change the value of `select_polygon` to `true`.

```
select_polygon = false;

if select_polygon

    % Use roipoly to create a mask from a polygon drawn manually.
    % Click to add vertices, then right-click and select "Create Mask" to return.
    imshow(A_sRGB)
    title('Draw a polygon around the chart')
    mask_chart = roipoly;

else

    % Use the provided coordinates of the bounding rectangle.
    c = [930 1280 1316 953];
    r = [1877 1890 1382 1370];
    mask_chart = roipoly(A_sRGB,r,c);

end
```

Dilate the mask slightly to ensure we do not include any pixels belonging to the chart.

```
mask_chart = imdilate(mask_chart,ones(7));
```

Measure the Ground Truth Illumination Using the ColorChecker Chart

For the purpose of white balance we will only use the 6 neutral patches on the bottom row of the chart. These neutral patches reflect light equally across the visible spectrum. They reflect the illumination of the scene. The ground truth illuminant is computed as the average color of the neutral patches, excluding under- and over-exposed pixels.

Specify the center of each neutral patch. By default, the example provides an estimate of the center coordinate of each patch. If you want to select the ROI centers interactively, change the value of `estimate_roi_centers` to `true`.

```
estimate_roi_centers = false;

if estimate_roi_centers

    %  Zoom in and click the center of each of the 6 neutral patches.
    xlim([1350 1930])
    ylim([900 1350])
    title('Click the center of each of the 6 neutral patches')
    [x,y] = ginput(6);

else

    % Use the provided estimate of ROI center coordinates.
    x = [1424 1514 1598 1676 1757 1835];
    y = [1268 1250 1247 1250 1235 1229];
end
```

We derive a square area from the coordinates of the center of each patch by considering the side of a square to be 80% of the distance separating the centers of two consecutive patches.

```
x = round(x);
y = round(y);
r = mean(diff(x)) / 2 * 0.80;
r = floor(r);
```

Create a binary mask covering the neutral patches.

```
mask = false(size(A,1), size(A,2));
for k = 1:6
    mask(y(k)-r:y(k)+r,x(k)-r:x(k)+r) = true;
end
```

Erode the mask to avoid including pixels outside of the patches or on the border, which are prone to chromatic aberration, whose colors can skew the measurement of the ground truth.

```
mask_eroded = imerode(mask, strel('disk',5));
```

Identify saturated RGB values in the input image. These values should be excluded from the computation of the ground truth as they will skew the measurement as well.

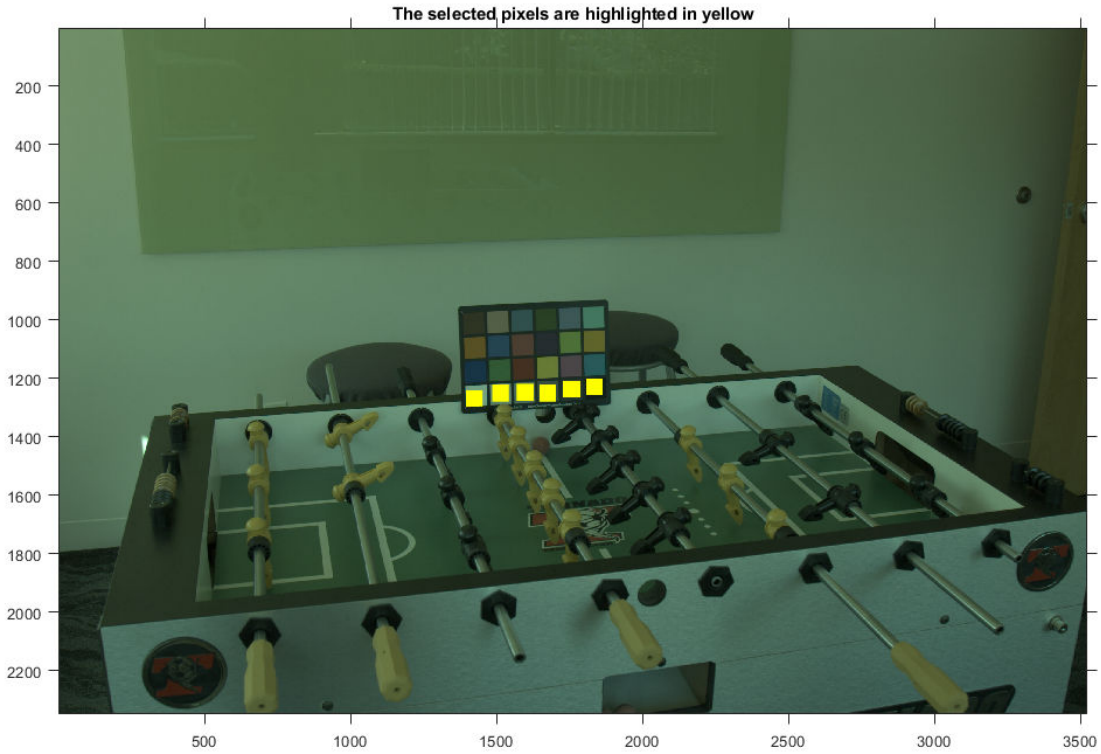
```
mask_clipped = (A == intmax(class(A))) | (A == intmin(class(A)));
mask_clipped = mask_clipped(:,:,1) | mask_clipped(:,:,2) | mask_clipped(:,:,3);
```

Exclude these clipped pixels from the mask corresponding to the neutral patches.

```
mask_patches = mask_eroded & ~mask_clipped;
```

Visualize the selected pixels. The highlighted pixels should all be within the neutral patches. If not, try clicking the centers of the patches again and repeat the steps above.

```
A_patches = imoverlay(A_sRGB,mask_patches);

imshow(A_patches)
title('The selected pixels are highlighted in yellow')
```



Obtain the red, green and red values for the neutral patches.

```
patches_R = A(:,:,1);
patches_G = A(:,:,2);
patches_B = A(:,:,3);
patches_R = patches_R(mask_patches);
patches_G = patches_G(mask_patches);
patches_B = patches_B(mask_patches);
```

For simplicity and because most illuminant estimation algorithms work in floating point, we convert the RGB values of the patches to double and scale the values to be in [0 1] before computing the mean.

```
patches_R = im2double(patches_R);
patches_G = im2double(patches_G);
patches_B = im2double(patches_B);
```

Compute the ground truth RGB illuminant as the mean RGB value across the neutral patches.

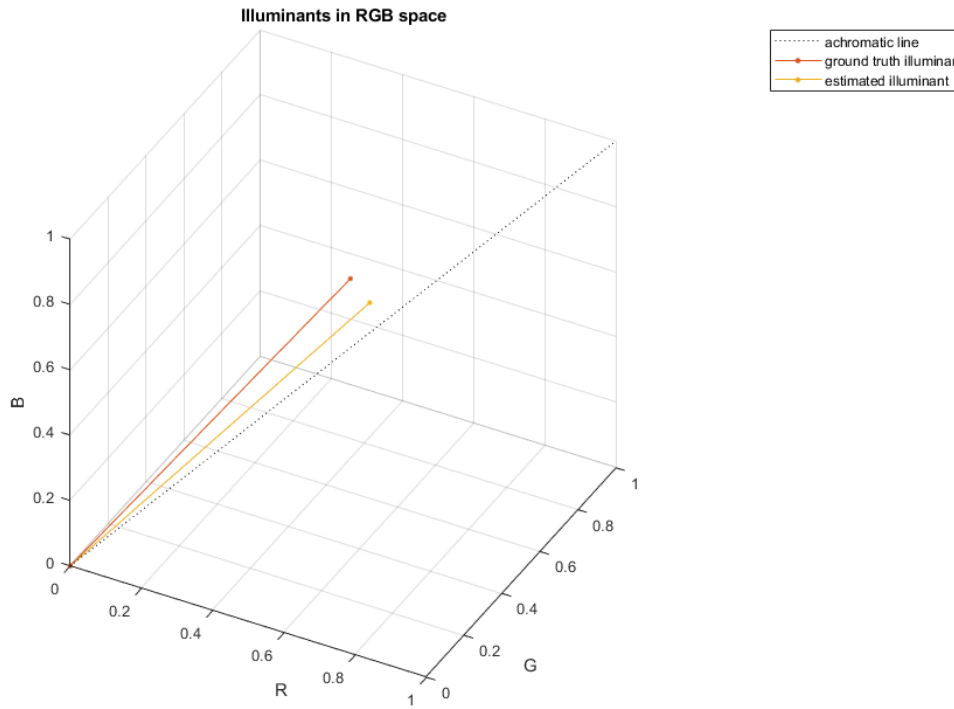
```
illuminant_groundtruth = [mean(patches_R) mean(patches_G) mean(patches_B)];
```

Angular Error

To compare an estimated illuminant against the ground truth, compute the **angular error** between the two colors. To better understand the concept of angular error, consider the following visualization of an arbitrary illuminant and the ground truth we just measured. Each illuminant represents a vector in RGB space.

```
illuminant = [0.066 0.1262 0.0691];

plot3([0 1],[0 1],[0,1],'LineStyle',':','Color','k')
hold on
plot3(...
    [0 illuminant_groundtruth(1)/norm(illuminant_groundtruth)], ... % Red
    [0 illuminant_groundtruth(2)/norm(illuminant_groundtruth)], ... % Green
    [0 illuminant_groundtruth(3)/norm(illuminant_groundtruth)], ... % Blue
    'Marker','.', 'MarkerSize',10)
hold on
plot3( ...
    [0 illuminant(1)/norm(illuminant)], ... % Red
    [0 illuminant(2)/norm(illuminant)], ... % Green
    [0 illuminant(3)/norm(illuminant)], ... % Blue
    'Marker','.', 'MarkerSize',10)
xlabel('R')
ylabel('G')
```

```
zlabel('B')
title('Illuminants in RGB space')
xlim([0 1])
ylim([0 1])
zlim([0 1])
view(28, 36)
legend('achromatic line', 'ground truth illuminant', 'estimated illuminant')
grid on
axis equal
```



The exact value of the estimated illuminant does not matter as much as its direction, since the direction of the illuminant is what is used to white balance an image. Ideally, the estimated illuminant should be aligned with the ground truth illuminant. The angular error between the estimated illuminant and the ground truth is the angle (in degrees) formed by the two vectors. *The smaller the angular error, the better the estimation is.* We

will evaluate the quality of an estimation according to the angular error against the ground truth.

White Patch Retinex

The White Patch Retinex method [1] for illuminant estimation assumes that the scene contains a bright patch. This patch reflects the maximum light possible for each color band, which is the color of the illumination of the scene.

The `illumwhite` function implements the White Patch Retinex method while also providing the ability to exclude part of the brightest pixels from the computation, in order to avoid taking over-exposed pixels into consideration.

First, estimate the illumination of the scene using all the pixels in the image, minus the ColorChecker Chart. This is done by specifying the top percentile to exclude to be 0. The `'Mask'` name-value pair is used to specify which pixels to use for the computation, which in our case are pixels not belonging to the ColorChecker Chart.

```
illuminant_wp1 = illumwhite(A, 0, 'Mask', ~mask_chart);
```

Compute the angular error for the illuminant estimated with White Patch Retinex.

```
err_wp1 = colorangle(illuminant_wp1, illuminant_groundtruth);
disp(['Angular error for White Patch with percentile=0: ' num2str(err_wp1)])
```

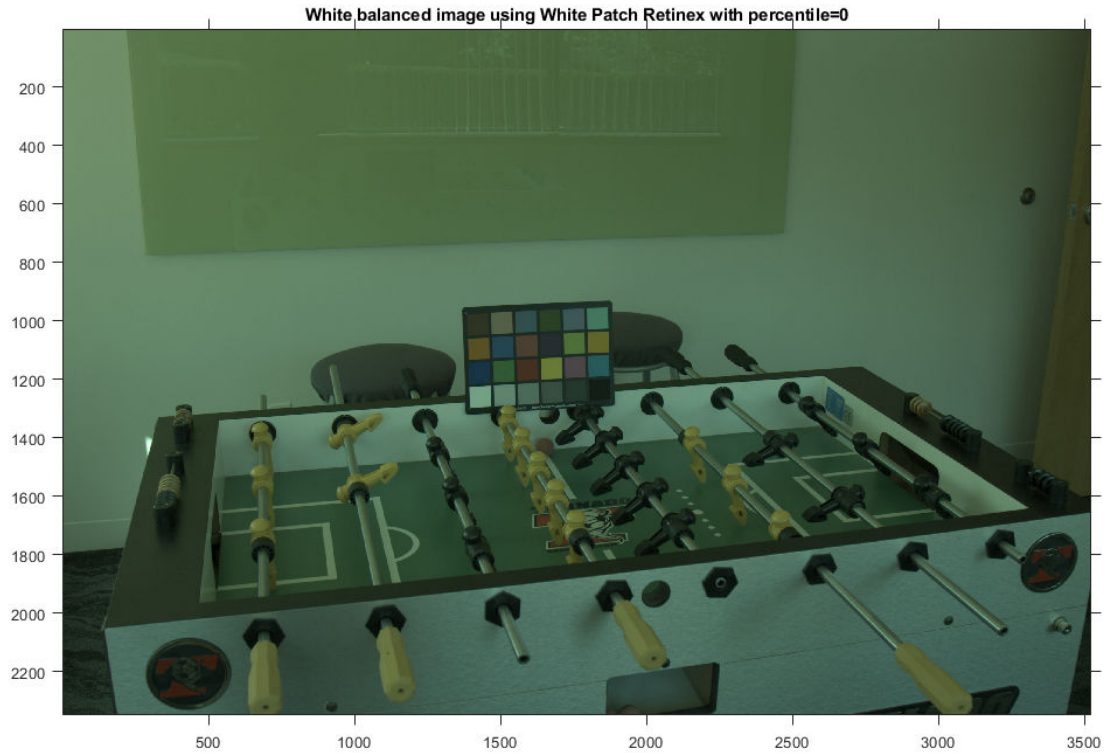
```
Angular error for White Patch with percentile=0: 16.5163
```

To white balance the image (Step 2 of AWB) using this estimated illuminant, use the `chromadapt` function, which by default scales the colors in the Bradford cone response model. Make sure to specify that we are working with linear RGB color values.

```
B_wp1 = chromadapt(A, illuminant_wp1, 'ColorSpace', 'linear-rgb');
```

Display the gamma-corrected white balanced image

```
B_wp1_sRGB = lin2rgb(B_wp1);

figure
imshow(B_wp1_sRGB)
title('White balanced image using White Patch Retinex with percentile=0')
```



Second, since selecting the maximum RGB value is sensitive to over- exposed pixels, the White Patch Retinex algorithm can be made more robust by excluding a certain percentage of the brightest pixels from the computation. This is achieved through the percentile parameter of the illumwhite function. Choose a percentile value of 1. (The default is 1.)

```
illuminant_wp2 = illumwhite(A, 1, 'Mask', ~mask_chart);
```

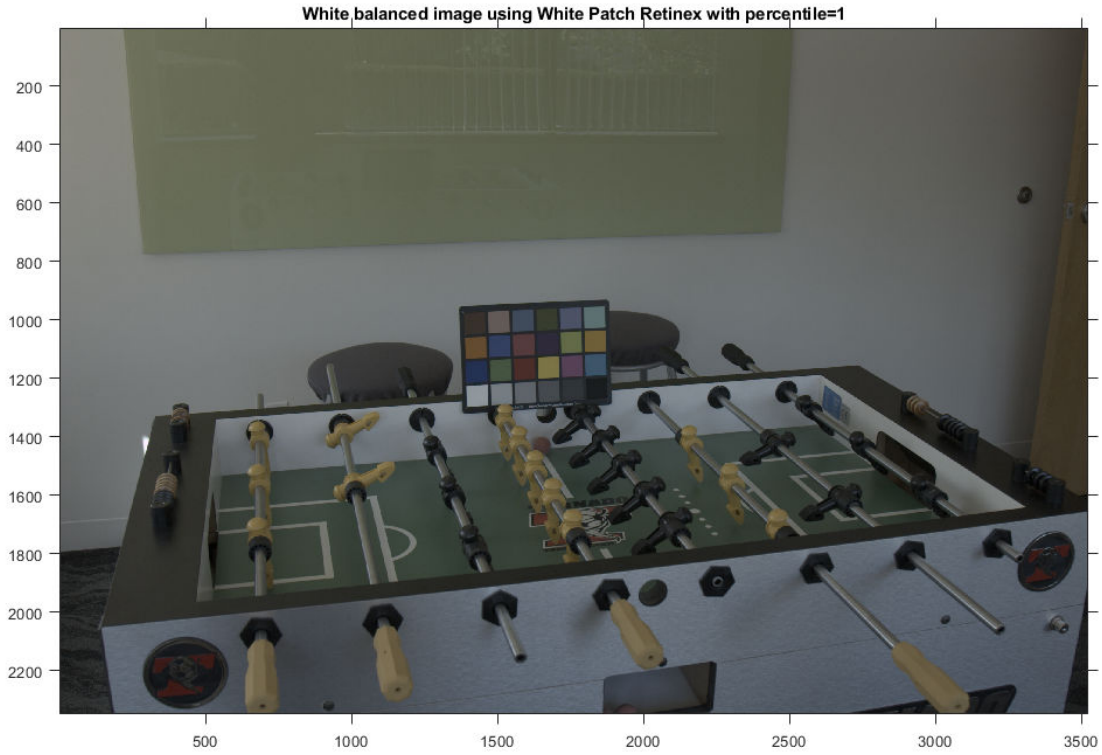
Compute the angular error for the illuminant estimated with the more robust version of White Patch Retinex.

```
err_wp2 = colorangle(illuminant_wp2, illuminant_groundtruth);
disp(['Angular error for White Patch with percentile=1: ' num2str(err_wp2)])
```

```
Angular error for White Patch with percentile=1: 5.0323
```

Display the gamma-corrected white balanced image with the new illuminant

```
B_wp2 = chromadapt(A, illuminant_wp2, 'ColorSpace', 'linear-rgb');
B_wp2_sRGB = lin2rgb(B_wp2);

imshow(B_wp2_sRGB)
title('White balanced image using White Patch Retinex with percentile=1')
```



Gray World

Gray World [2] is most certainly the best known illuminant estimation method. It assumes that the average color of the world is gray, i.e. achromatic. Therefore, it calculates the scene illuminant as the *average* RGB value in the image.

The `illumgray` function implements the Gray World algorithm with one addition: It provides the ability to exclude the darkest and brightest pixels from the computation, which may skew the estimation of the illuminant, making the algorithm more robust.

First, estimate the scene illumination using all pixels of the image, excluding those corresponding to the ColorChecker Chart. The `illumgray` function provides a parameter to specify the percentiles of bottom and top values (ordered by brightness) to exclude. Here, we specify the percentiles as [0 0].

```
illuminant_gw1 = illumgray(A, 0, 'Mask', ~mask_chart);
```

Compare the estimated illuminant to the ground truth by computing the angular error between the two, similarly to what we did for White Patch Retinex.

```
err_gw1 = colorangle(illuminant_gw1, illuminant_groundtruth);
disp(['Angular error for Gray World with percentiles=[0 0]: ' num2str(err_gw1)])
```

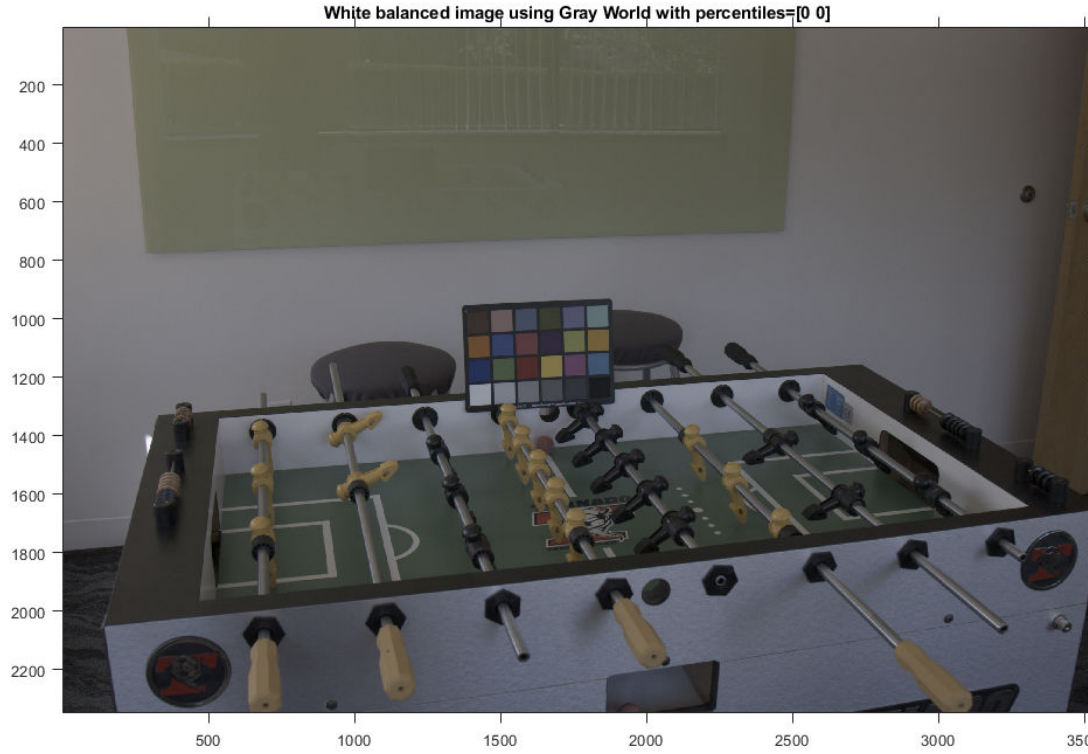
```
Angular error for Gray World with percentiles=[0 0]: 5.063
```

Apply chromatic adaptation to white balance the image with the estimated illuminant.

```
B_gw1 = chromadapt(A, illuminant_gw1, 'ColorSpace', 'linear-rgb');
```

Display the gamma-corrected white balanced image

```
B_gw1_sRGB = lin2rgb(B_gw1);

imshow(B_gw1_sRGB)
title('White balanced image using Gray World with percentiles=[0 0]')
```



Second, since under- and over-exposed pixels can negatively influence the estimation of the illuminant as the mean RGB value in the image, let's exclude the bottom and top 1% pixels. (The default value for `percentiles` is [1 1].)

```
illuminant_gw2 = illumgray(A, 1, 'Mask', ~mask_chart);
```

Compute the angular error for the second illuminant estimated with Gray World.

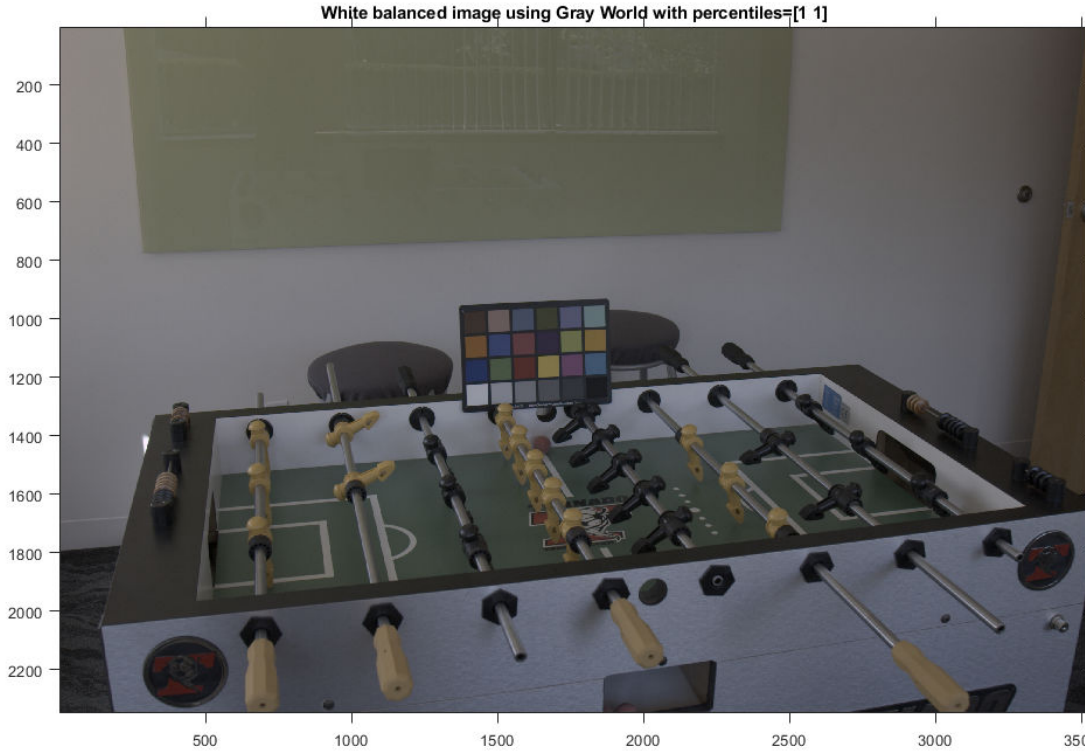
```
err_gw2 = colorangle(illuminant_gw2, illuminant_groundtruth);
disp(['Angular error for Gray World with percentiles=[1 1]: ' num2str(err_gw2)])
```

```
Angular error for Gray World with percentiles=[1 1]: 5.1314
```

Display the gamma-corrected white balanced image with the new illuminant.

```
B_gw2 = chromadapt(A, illuminant_gw2, 'ColorSpace', 'linear-rgb');
B_gw2_sRGB = lin2rgb(B_gw2);
```

```
imshow(B_gw2_sRGB)
title('White balanced image using Gray World with percentiles=[1 1]')
```



Cheng's Principal Component Analysis (PCA) method

Cheng's illuminant estimation method [3] draws inspiration from spatial domain methods such as Grey Edge [4], which assumes that the gradients of an image are achromatic. They show that Grey Edge can be improved by artificially introducing strong gradients by shuffling image blocks, and conclude that the strongest gradients follow the direction of the illuminant. Their method consists in ordering pixels according to the norm of their projection along the direction of the mean image color, and retaining the bottom and top p%. These two groups correspond to strong gradients in the image. Finally, they perform a *principal component analysis* (PCA) on the retained pixels and return the first component as the estimated illumination.

Cheng's method is implemented by the `illumpca` function. We use Cheng's method to estimate the illuminant in the scene using the bottom and top 5% of pixels along the direction of the mean color. (The default is 3.5.)

```
illuminant_ch1 = illumpca(A, 5, 'Mask', ~mask_chart);
```

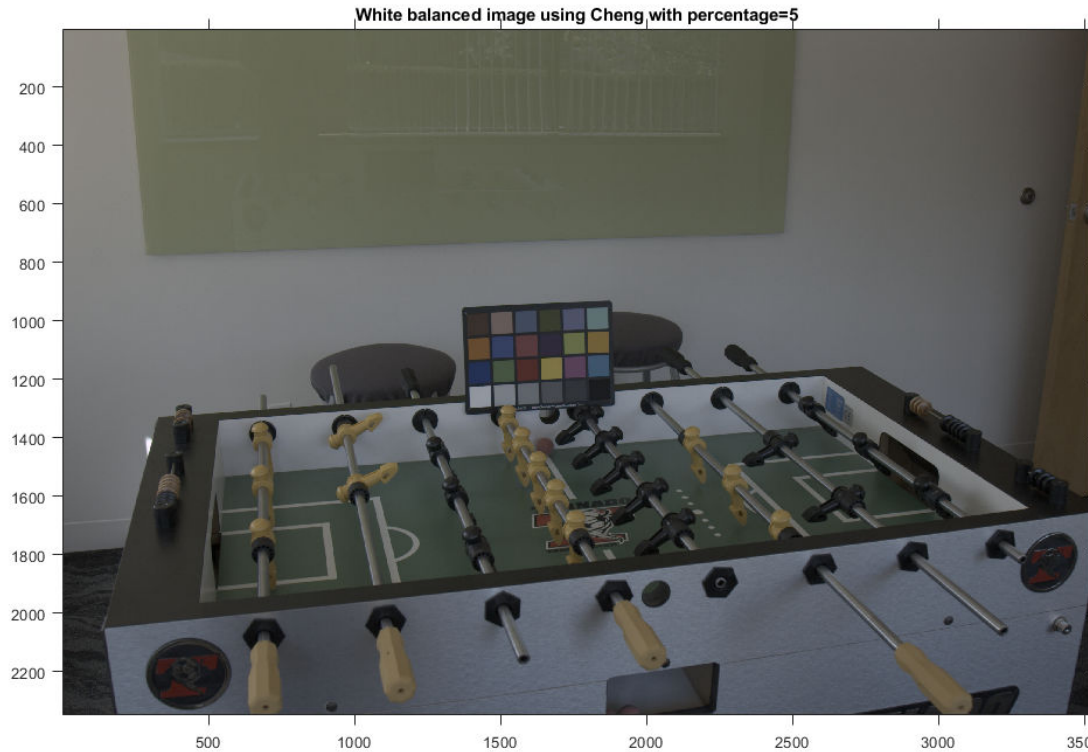
Compare this estimation to the ground truth.

```
err_ch1 = colorangle(illuminant_ch1, illuminant_groundtruth);
disp(['Angular error for Cheng with percentage=5: ' num2str(err_ch1)])
```

```
Angular error for Cheng with percentage=5: 4.7595
```

Display the gamma-corrected white balanced image

```
B_ch1 = chromadapt(A, illuminant_ch1, 'ColorSpace', 'linear-rgb');
B_ch1_sRGB = lin2rgb(B_ch1);

imshow(B_ch1_sRGB)
title('White balanced image using Cheng with percentage=5')
```



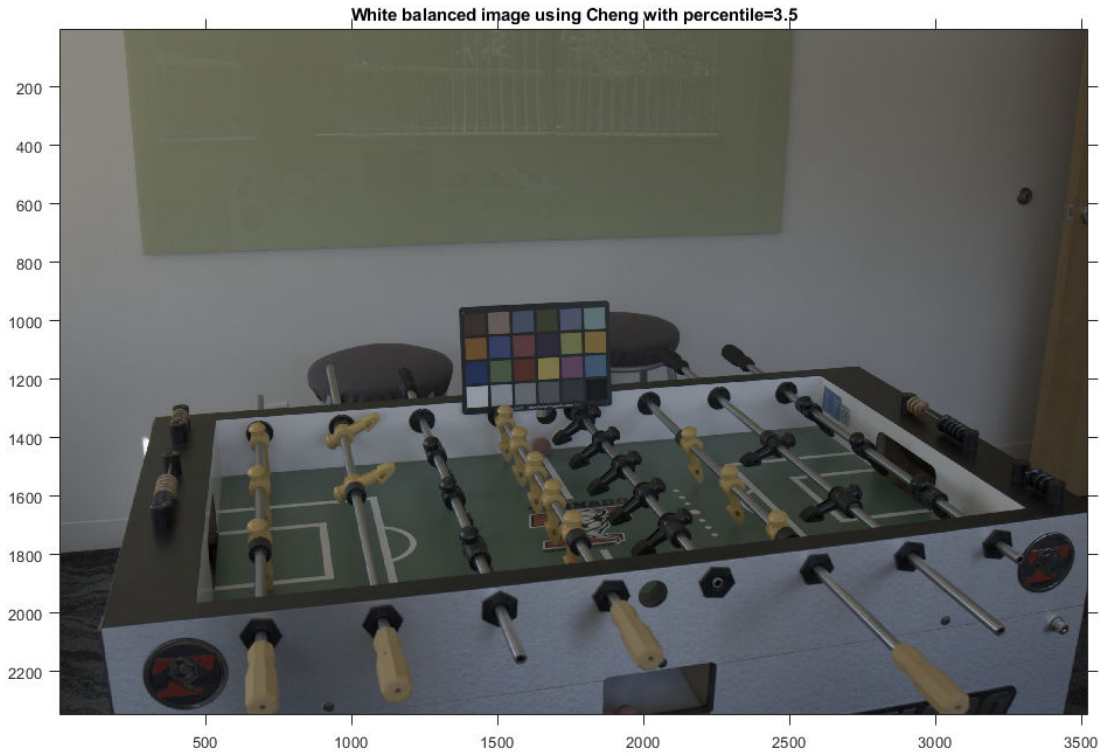
Let's now use the default percentage value and see how it compares.

```
illuminant_ch2 = illumpca(A, 'Mask', ~mask_chart);
err_ch2 = colorangle(illuminant_ch2, illuminant_groundtruth);
disp(['Angular error for Cheng with percentage=3.5: ' num2str(err_ch2)])
```

```
Angular error for Cheng with percentage=3.5: 5.0283
```

Display the corrected image in sRGB.

```
B_ch2 = chromadapt(A, illuminant_ch2, 'ColorSpace', 'linear-rgb');
B_ch2_sRGB = lin2rgb(B_ch2);

imshow(B_ch2_sRGB)
title('White balanced image using Cheng with percentile=3.5')
```



Parameter Sweep

To find the best parameter to use for each method we can sweep through a range and calculate the angular error for each of them. The parameters of the three algorithms have different meanings, but the similar ranges of these parameters makes it easy to programmatically search for the best one for each algorithm.

```
param_range = 0:0.25:5;
err = zeros(numel(param_range),3);
for k = 1:numel(param_range)
    % White Patch
    illuminant_wp = illumwhite(A, param_range(k), 'Mask', ~mask_chart);
    err(k,1) = colorangle(illuminant_wp, illuminant_groundtruth);
    % Gray World
    illuminant_gw = illumgray(A, param_range(k), 'Mask', ~mask_chart);
```

```
    err(k,2) = colorangle(illuminant_gw, illuminant_groundtruth);
    % Cheng
    if (param_range(k) ~= 0)
        illuminant_ch = illumpca(A, param_range(k), 'Mask', ~mask_chart);
        err(k,3) = colorangle(illuminant_ch, illuminant_groundtruth);
    else
        % Cheng's algorithm is undefined for percentage=0.
        err(k,3) = NaN;
    end
end
```

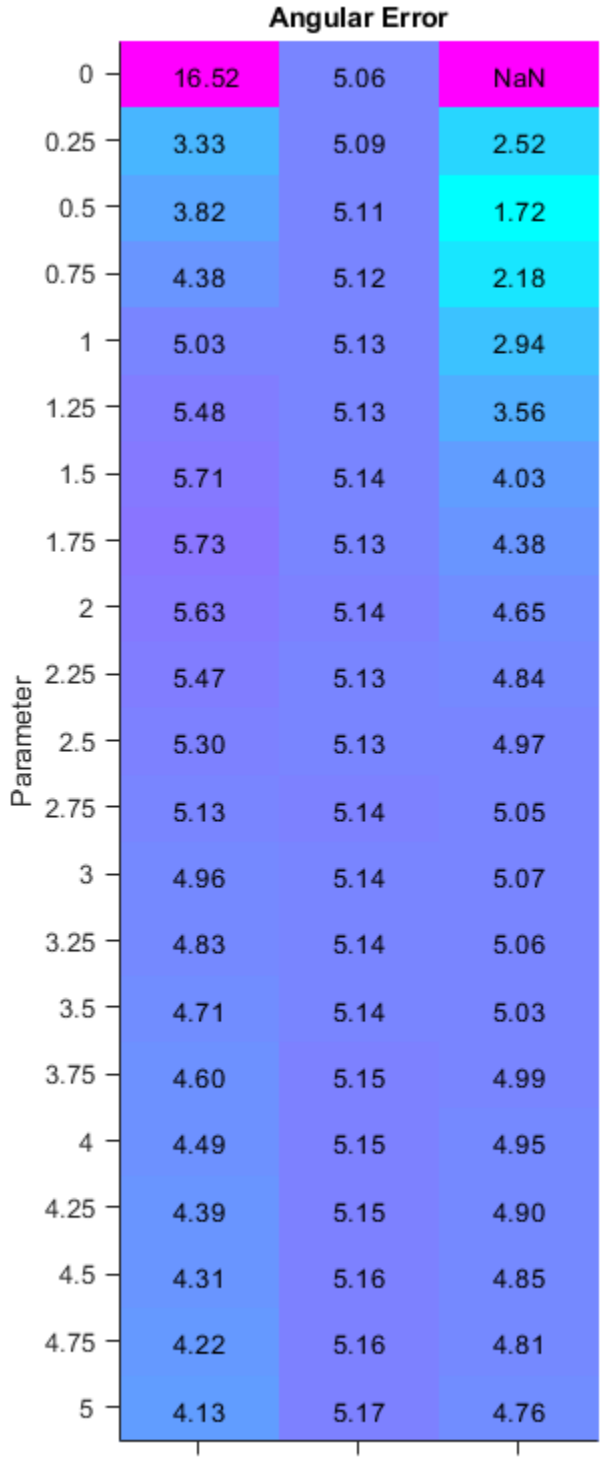
Create a visualization of the angular error as heat map.

```
err_normalized = mat2gray(log(err));
block_size_x = 120;
block_size_y = 50;
err_image = ones(size(err,1) * block_size_y, size(err,2) * block_size_x);
for i = 0:size(err,1)-1
    for j = 0:size(err,2)-1
        err_image(block_size_y*i+1:block_size_y*(i+1), block_size_x*j+1:block_size_x*(
    end
end
```

Display the heat map of the angular error. Light blue colors indicate a low angular error (good), while red colors indicate a high angular error (bad).

```
old_pref = iptgetpref('ImshowAxesVisible');
iptsetpref('ImshowAxesVisible','on')
imshow(err_image, 'Colormap', cool)
iptsetpref('ImshowAxesVisible',old_pref)
for i = 0:size(err,1)-1
    for j = 0:size(err,2)-1
        y = block_size_y*i + 1 + block_size_y/2;
        x = block_size_x*j + 1 + block_size_x/3;
        text(x,y,sprintf('%1.2f',err(i+1,j+1)))
    end
end
box off
title('Angular Error')
ylabel('Parameter')

yticks(linspace(block_size_y/2, size(err_image,1) - block_size_y/2, numel(param_range))
yticklabels(arrayfun(@(x) {num2str(x)}, param_range))
```

```
xticks(block_size_x/2 + [0 block_size_x 2*block_size_x])
xticklabels({'White Patch','Gray World','Cheng'})
```


Angular Error
Parameter
0
0.25
0.5
0.75
1
1.25
1.5
1.75
2
2.25
2.5
2.75
3
3.25
3.5
3.75
4
4.25
4.5
4.75
5
16.52 5.06 NaN
3.33 5.09 2.52
3.82 5.11 1.72
4.38 5.12 2.18
5.03 5.13 2.94
5.48 5.13 3.56
5.71 5.14 4.03
5.73 5.13 4.38
5.63 5.14 4.65
5.47 5.13 4.84
5.30 5.13 4.97
5.13 5.14 5.05
4.96 5.14 5.07
4.83 5.14 5.06
4.71 5.14 5.03
4.60 5.15 4.99
4.49 5.15 4.95
4.39 5.15 4.90
4.31 5.16 4.85
4.22 5.16 4.81
4.13 5.17 4.76

Find the best parameter for each algorithm.

```
[~,idx_best] = min(err);
best_param_wp = param_range(idx_best(1));
best_param_gw = param_range(idx_best(2));
best_param_ch = param_range(idx_best(3));

fprintf('The best parameter for White Patch is %1.2f with angular error %1.2f degrees\n
    best_param_wp, err(idx_best(1),1));
```

```
The best parameter for White Patch is 0.25 with angular error 3.33 degrees
```

```
fprintf('The best parameter for Gray World is %1.2f with angular error %1.2f degrees\n
    best_param_gw, err(idx_best(2),2));
```

```
The best parameter for Gray World is 0.00 with angular error 5.06 degrees
```

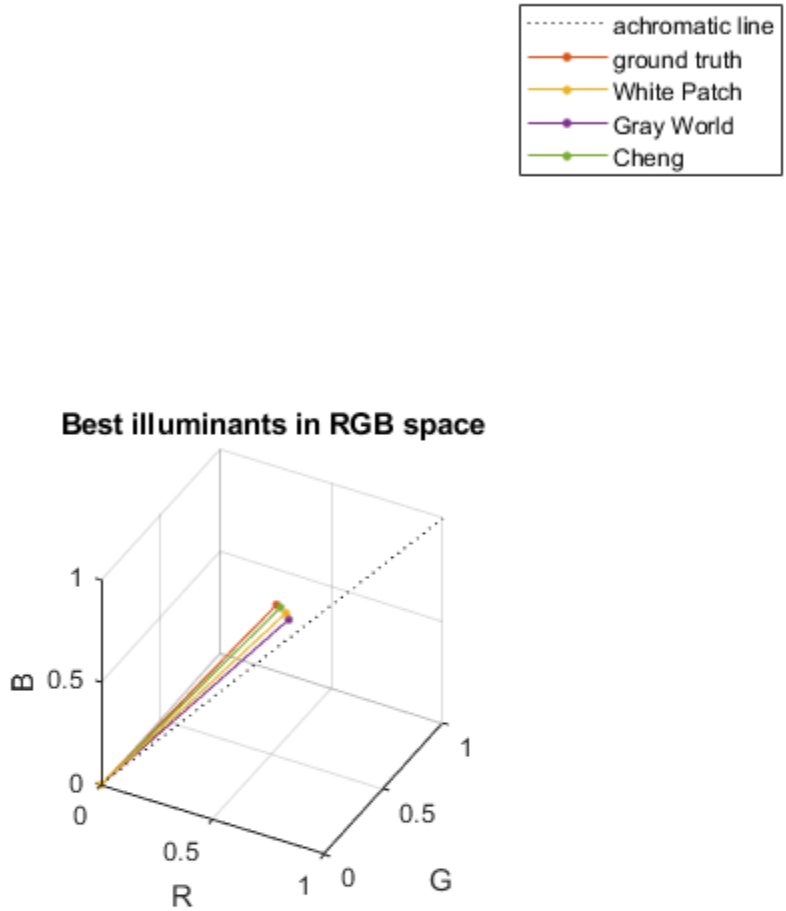
```
fprintf('The best parameter for Cheng is %1.2f with angular error %1.2f degrees\n', ...
    best_param_ch, err(idx_best(3),3));
```

```
The best parameter for Cheng is 0.50 with angular error 1.72 degrees
```

Compute and display the best estimated illuminant for each algorithm in RGB space.

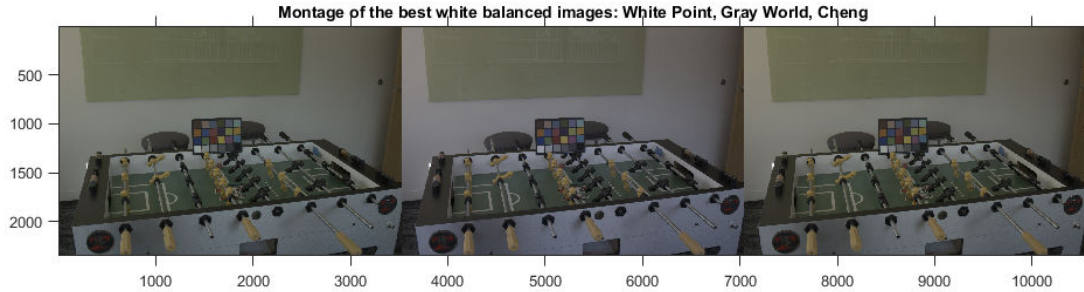
```
best_illum_wp = illumwhite(A, best_param_wp, 'Mask', ~mask_chart);
best_illum_gw = illumgray(A, best_param_gw, 'Mask', ~mask_chart);
best_illum_ch = illumpca(A, best_param_ch, 'Mask', ~mask_chart);

plot3([0 1],[0 1],[0,1],'LineStyle',':','Color','k')
hold on
plot3(...
    [0 illuminant_groundtruth(1)/norm(illuminant_groundtruth)], ... % Red
    [0 illuminant_groundtruth(2)/norm(illuminant_groundtruth)], ... % Green
    [0 illuminant_groundtruth(3)/norm(illuminant_groundtruth)], ... % Blue
    'Marker','.', 'MarkerSize',10)
plot3( ...
    [0 best_illum_wp(1)/norm(best_illum_wp)], ... % Red
    [0 best_illum_wp(2)/norm(best_illum_wp)], ... % Green
    [0 best_illum_wp(3)/norm(best_illum_wp)], ... % Blue
    'Marker','.', 'MarkerSize',10)
plot3( ...
    [0 best_illum_gw(1)/norm(best_illum_gw)], ... % Red
    [0 best_illum_gw(2)/norm(best_illum_gw)], ... % Green
    [0 best_illum_gw(3)/norm(best_illum_gw)], ... % Blue
    'Marker','.', 'MarkerSize',10)
plot3( ...
```

```
    [0 best_illum_ch(1)/norm(best_illum_ch)], ... % Red
    [0 best_illum_ch(2)/norm(best_illum_ch)], ... % Green
    [0 best_illum_ch(3)/norm(best_illum_ch)], ... % Blue
    'Marker','.', 'MarkerSize',10)
xlabel('R')
ylabel('G')
zlabel('B')
title('Best illuminants in RGB space')
xlim([0 1])
ylim([0 1])
zlim([0 1])
view(28, 36)
legend('achromatic line', 'ground truth', 'White Patch', 'Gray World', 'Cheng')
grid on
axis equal
```


achromatic line
ground truth
White Patch
Gray World
Cheng
Best illuminants in RGB space
B
1
0.5
0
0
0.5
1
R
0
0.5
1
G

Display the white balanced images for each method using the best illuminant side by side.

```
B_wp_best = chromadapt(A, best_illum_wp, 'ColorSpace', 'linear-rgb');
B_wp_best_sRGB = lin2rgb(B_wp_best);
B_gw_best = chromadapt(A, best_illum_gw, 'ColorSpace', 'linear-rgb');
B_gw_best_sRGB = lin2rgb(B_gw_best);
B_ch_best = chromadapt(A, best_illum_ch, 'ColorSpace', 'linear-rgb');
B_ch_best_sRGB = lin2rgb(B_ch_best);

M = zeros(size(A,1), 3*size(A,2), size(A,3), 'like', A);
M(:,1:size(A,2),:) = B_wp_best_sRGB;
M(:,size(A,2)+1:2*size(A,2),:) = B_gw_best_sRGB;
M(:,2*size(A,2)+1:end,:) = B_ch_best_sRGB;

figure
imshow(M)
title('Montage of the best white balanced images: White Point, Gray World, Cheng')
```



Conclusion

This quick shootout between two classic illuminant estimation methods and a more recent one shows that Cheng's method, using the top and bottom 0.75% darkest and brightest pixels, wins for that particular image. However, this result should be taken with a grain of salt.

First, the ground truth illuminant was measured using a ColorChecker Chart and is sensitive to shot and sensor noise. The ground truth illuminant of a scene can be better estimated using a spectrophotometer.

Second, we estimated the ground truth illuminant as the mean color of the neutral patches. It is also common to use the median instead of the mean. Doing so could shift the

ground truth by a significant amount. For example, for the image in this study, using the same pixels, the median color and the mean color of the neutral patches are 0.5 degrees apart, which in some cases can be more than the angular error of the illuminants estimated by different methods.

Third, a full comparison of illuminant estimation methods should use a variety of images taken under different conditions. One method might work better than the others for a particular image, but might perform poorly over the entire data set.

References

[1] Ebner, Marc. White Patch Retinex, Color Constancy. John Wiley & Sons, 2007. ISBN 978-0-470-05829-9.

[2] Ebner, Marc. The Gray World Assumption, Color Constancy. John Wiley & Sons, 2007. ISBN 978-0-470-05829-9.

[3] Cheng, Dongliang, Dilip K. Prasad, and Michael S. Brown. "Illuminant estimation for color constancy: why spatial-domain methods work and the role of the color distribution." JOSA A 31.5 (2014): 1049-1058.

[4] Van De Weijer, Joost, Theo Gevers, and Arjan Gijsenij. "Edge-based color constancy." IEEE Transactions on image processing 16.9 (2007): 2207-2214.

See Also

`chromadapt` | `colorangle` | `illumgray` | `illumpca` | `illumwhite` | `imdilate` | `imerode` | `imoverlay` | `lin2rgb` | `rgb2lin`

More About

- “Gamma Correction” on page 11-90

Neighborhood and Block Operations

This chapter discusses these generic block processing functions. Topics covered include

- “Neighborhood or Block Processing: An Overview” on page 16-2
- “Sliding Neighborhood Operations” on page 16-3
- “Distinct Block Processing” on page 16-7
- “Block Size and Performance” on page 16-11
- “Parallel Block Processing on Large Image Files” on page 16-15
- “Perform Block Processing on Image Files in Unsupported Formats” on page 16-17
- “Use Column-wise Processing to Speed Up Sliding Neighborhood or Distinct Block Operations” on page 16-25
- “Block Processing Large Images” on page 16-29
- “Computing Statistics for Large Images” on page 16-36

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
`nlfilter`	Implements sliding neighborhood operations that you can use to process an input image in a pixel-wise 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 “Sliding Neighborhood Operations” on page 16-3
`blockproc`	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 “Distinct Block Processing” on page 16-7
`colfilt`	Implements *column-wise processing operations* which provide a way of speeding up neighborhood or block operations by rearranging blocks into matrix columns. For more information, see “Use Column-wise Processing to Speed Up Sliding Neighborhood or Distinct Block Operations” on page 16-25.

Sliding Neighborhood Operations

In this section...

"Determine the Center Pixel" on page 16-4

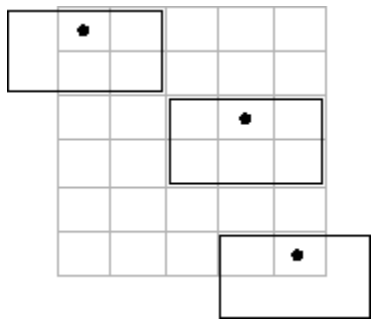
"General Algorithm of Sliding Neighborhood Operations" on page 16-4

"Border Padding Behavior in Sliding Neighborhood Operations" on page 16-4

"Implementing Linear and Nonlinear Filtering as Sliding Neighborhood Operations" on page 16-5

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 "Distinct Block Processing" on page 16-7 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 "Determine the Center Pixel" on page 16-4.



Neighborhood Blocks in a 6-by-5 Matrix

Determine 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

```
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.

Border Padding Behavior 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 neighbor 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?” on page 8-2 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 “Use Column-wise Processing to Speed Up Sliding Neighborhood or Distinct Block Operations” on page 16-25.

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 = nlfilter(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` data type.

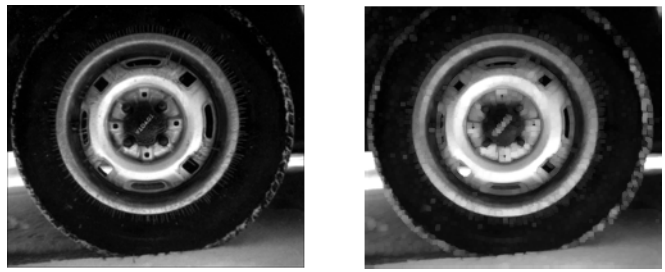
```
I = im2double(imread('tire.tif'));
f = @(x) sqrt(min(x(:)));
I2 = nlfilter(I,[2 2],f);
```

(For more information on function handles, see “Create Function Handle” (MATLAB). For more information about anonymous functions, see “Anonymous Functions” (MATLAB).)

The following example uses `nlfilter` 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 `nlfilter`. For a faster way to perform this local maximum operation, use `imdilate`.

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



Each Output Pixel Set to Maximum Input Neighborhood Value

Distinct Block Processing

In this section...

"Implement Block Processing Using the blockproc Function" on page 16-7
"Apply Padding" on page 16-9

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 "Apply Padding" on page 16-9. (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 "Sliding Neighborhood Operations" on page 16-3.)

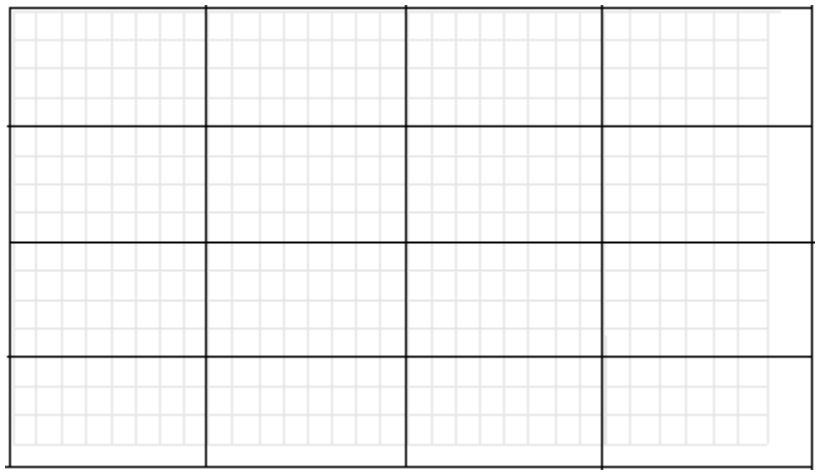


Image Divided into Distinct Blocks

Implement 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 function handles, see “Create Function Handle” (MATLAB). For more information about anonymous functions, see “Anonymous Functions” (MATLAB).)

```
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

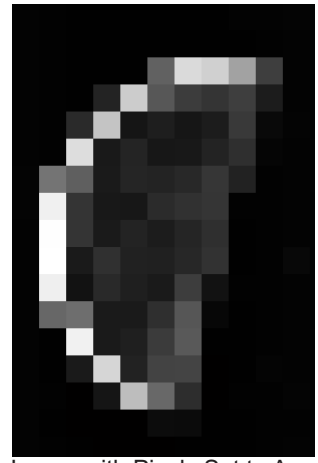


Image with Pixels Set to Average Value

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 “Use Column-wise Processing to Speed Up Sliding Neighborhood or Distinct Block Operations” on page 16-25.

Apply 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 “Distinct Block Processing” on page 16-7, 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.

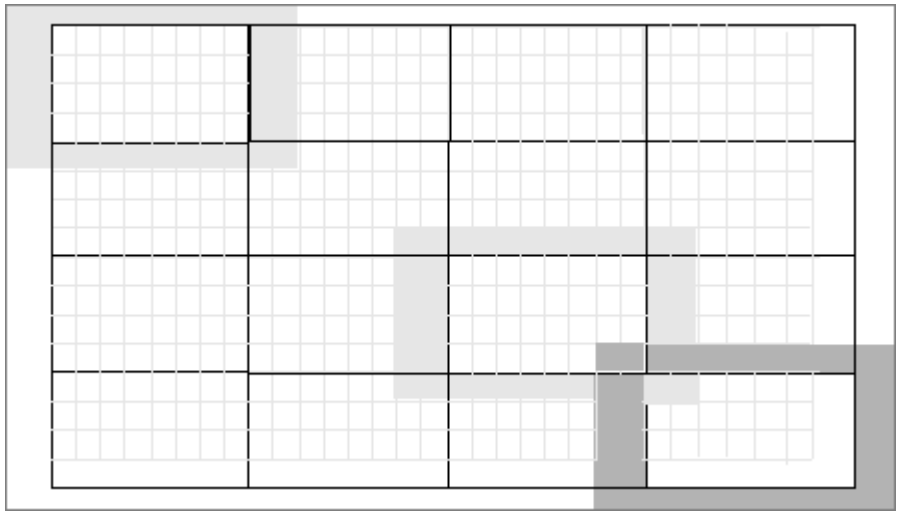


Image Divided into Distinct Blocks with Specified Borders

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:

```
ans =

  struct with fields:

                     Filename: '\\fs-21-ah\home$\jholohan\Documents\MATLAB\concordortho
                  FileModDate: '10-Nov-2016 17:34:44'
                     FileSize: 6586702
                       Format: 'tif'
                FormatVersion: []
                        Width: 2956
                       Height: 2215
                     BitDepth: 8
                    ColorType: 'grayscale'
              FormatSignature: [73 73 42 0]
                    ByteOrder: 'little-endian'
                NewSubFileType: 0
                 BitsPerSample: 8
                   Compression: 'PackBits'
    PhotometricInterpretation: 'BlackIsZero'
                  StripOffsets: [1×66 double]
              SamplesPerPixel: 1
                 RowsPerStrip: 34
              StripByteCounts: [1×66 double]
                  XResolution: 72
                  YResolution: 72
                ResolutionUnit: 'Inch'
                     Colormap: []
          PlanarConfiguration: 'Chunky'
                    TileWidth: []
                   TileLength: []
                  TileOffsets: []
                TileByteCounts: []
                  Orientation: 1
                    FillOrder: 1
             GrayResponseUnit: 0.0100
```

```
                MaxSampleValue: 255
                MinSampleValue: 0
                  Thresholding: 1
                        Offset: 6585984
```

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.
```

Parallel Block Processing on Large Image Files

If you have a Parallel Computing Toolbox license, then the `blockproc` function can take advantage of multiple processor cores on your machine to perform parallel block processing.

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 *parallel pool*. Then, the client divides the input image into blocks and sends blocks to the worker MATLAB sessions. Each worker processes a subset of blocks and sends the results back to the client. The client aggregates the results into an output variable.

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, then you can invoke parallel processing in `blockproc` by specifying the `'UseParallel'` argument as `true`. When you do so, MATLAB automatically opens a parallel pool of workers on your local machine and uses all available workers to process the input image.

In the following example, compute a discrete cosine transform for each 8-by-8 block of an image in parallel:

```
blockFun = @(block_struct) dct2(block_struct.data);
result = blockproc(input_image,[8 8],blockFun, ...
   'UseParallel',true);
```

Control parallel behavior with the parallel preferences, including scaling up to a cluster. See `parpool` for information on configuring your parallel environment.

See Also

`blockproc`

More About

- “What Is Parallel Computing?” (Parallel Computing Toolbox)
- “Choose a Parallel Computing Solution” (Parallel Computing Toolbox)
- “Run Batch Parallel Jobs” (Parallel Computing Toolbox)

Perform Block Processing on Image Files 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 (MATLAB) 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 array of characters that identify the version of the file format	`'HEADER'` or `'HEAD74'` (Pre-7.4 files say `'HEADER'`.)
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 first six bytes of the header:

```
headword = fread(fid,6,'uint8=>char')';
fprintf('Version ID: %s\n',headword);
```

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:

```
Version ID: 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 (MATLAB) for general information on writing classes.

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 headword. 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 headword 'HEADER' or
      % 'HEAD74', as per the Erdas LAN file specification.
      headword = fread(fid,6,'uint8=>char');
      if ~(strcmp(headword','HEADER') || strcmp(headword',...
            '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. See the “Computing Statistics for Large Images” on page 16-36 example to see how the `blockproc` function works with the `LanAdapter` class.

Use Column-wise Processing to Speed Up Sliding Neighborhood or Distinct Block Operations

In this section...

"Using Column Processing with Sliding Neighborhood Operations" on page 16-25
"Using Column Processing with Distinct Block Operations" on page 16-26

Performing sliding neighborhood and distinct block operations column-wise, 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.

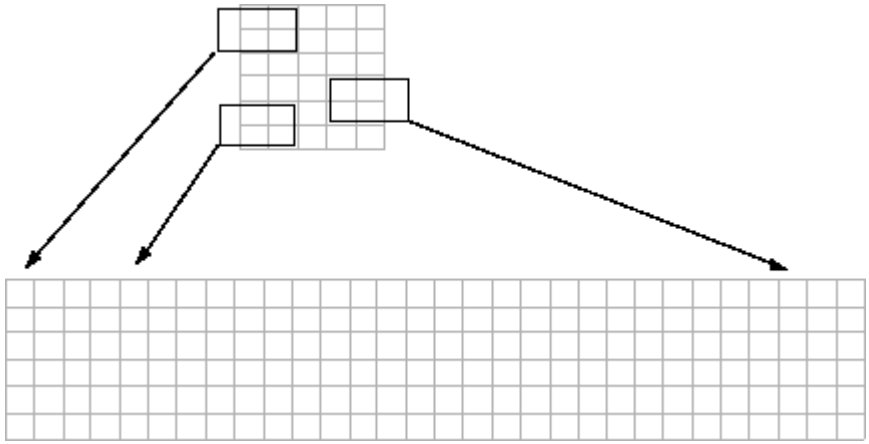
To use column processing, use the `colfilt` function . This function

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.

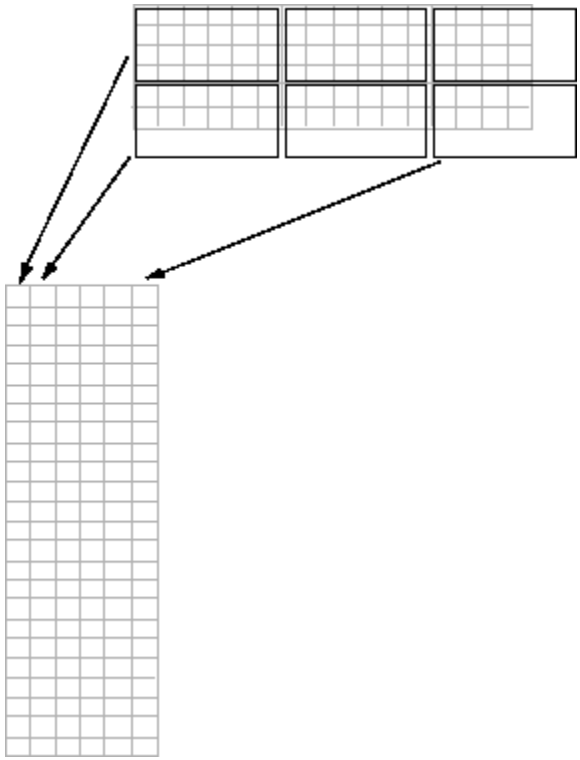
`colfilt` can produce the same results as `nlfilter` with faster execution time; however, it might use more memory. The example below sets each output pixel to the maximum value in the input pixel's neighborhood, producing the same result as the `nlfilter` example shown in “Implementing Linear and Nonlinear Filtering as Sliding Neighborhood Operations” on page 16-5.

```
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-`(ceil(mm/m)*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`.

Block Processing Large Images

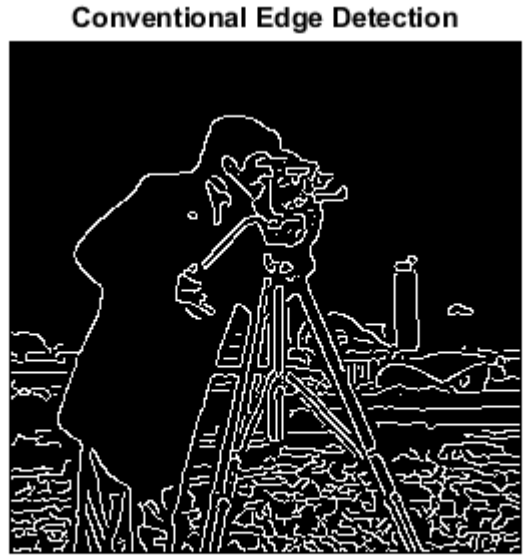
This example shows how to perform edge detection on a TIFF image by dividing the image into blocks. When working with large images, normal image processing techniques can sometimes break down. The images can either be too large to load into memory, or else they can be loaded into memory but then be too large to process.

To avoid these problems, you can process large images incrementally: reading, processing, and finally writing the results back to disk, one region at a time. The `blockproc` function helps you with this process. Using `blockproc`, specify an image, a block size, and a function handle. `blockproc` then divides the input image into blocks of the specified size, processes them using the function handle one block at a time, and then assembles the results into an output image. `blockproc` returns the output to memory or to a new file on disk.

First, consider the results of performing edge detection without block processing. This example uses a small image, cameraman.tif, to illustrate the concepts, but block processing is often more useful for large images.

```
file_name = 'cameraman.tif';
I = imread(file_name);
normal_edges = edge(I,'canny');

imshow(I)
title('Original Image')
```

```
figure
imshow(normal_edges)
title('Conventional Edge Detection')
```



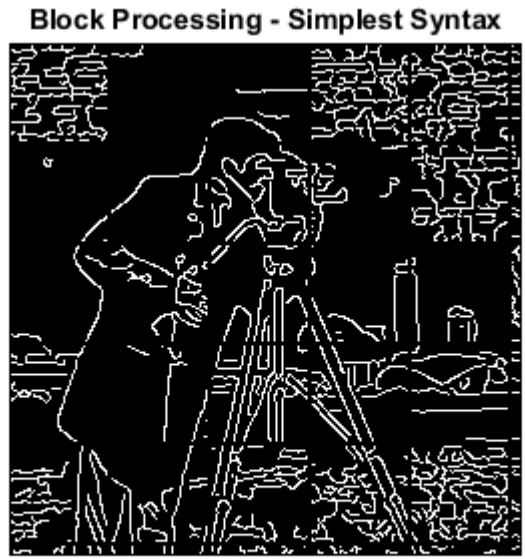
Now try the same task using block processing. The `blockproc` function has built-in support for TIFF images, so you do not have to read the file completely into memory using `imread`. Instead, call the function using the string filename as input. `blockproc` reads in one block at a time, making this workflow ideal for very large images.

When working with large images you will often use the 'Destination' parameter to specify a file into which `blockproc` will write the output image. However, in this example you will return the results to a variable, in memory.

This example uses a block size of [50 50]. In general, choosing larger block sizes yields better performance for `blockproc`. This is particularly true for file-to-file workflows where accessing the disk will incur a significant performance cost. Appropriate block sizes vary based on the machine resources available, but should likely be in the range of thousands of pixels per dimension.

```
% You can use an anonymous function to define the function handle. The
% function is passed a structure as input, a "block struct", with several
% fields containing the block data as well as other relevant information.
```

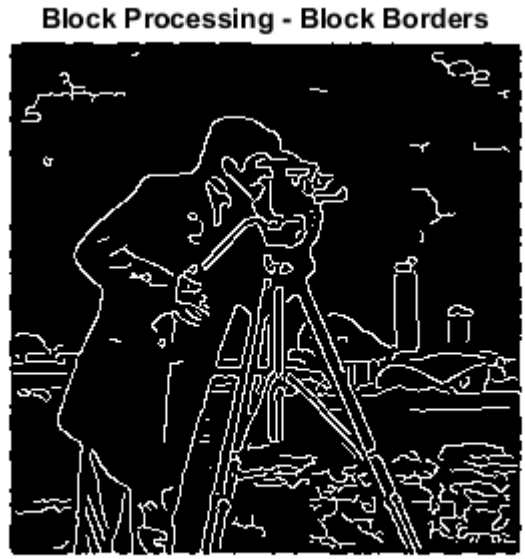
```
% The function should return the processed block data.
edgeFun = @(block_struct) edge(block_struct.data,'canny');

block_size = [50 50];
block_edges = blockproc(file_name,block_size,edgeFun);

figure
imshow(block_edges)
title('Block Processing - Simplest Syntax')
```



Notice the significant artifacts from the block processing. Determining whether a pixel is an edge pixel or not requires information from the neighboring pixels. This means that each block cannot be processed completely separately from its surrounding pixels. To remedy this, use the `blockproc` parameter 'BorderSize' to specify vertical and horizontal borders around each block. The necessary 'BorderSize' varies depending on the task being performed.

```
border_size = [10 10];
block_edges = blockproc(file_name,block_size,edgeFun,'BorderSize',border_size);
```

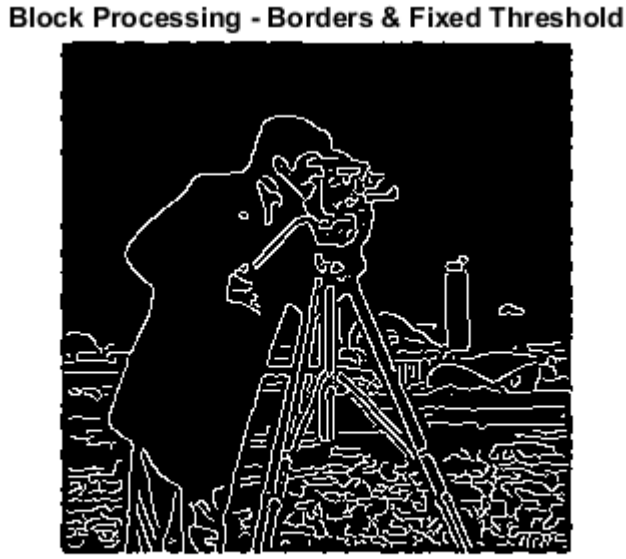
```
figure
imshow(block_edges)
title('Block Processing - Block Borders')
```



The blocks are now being processed with an additional 10 pixels of image data on each side. This looks better, but the result is still significantly different from the original in-memory result. The reason for this is that the Canny edge detector uses a threshold that is computed based on the complete image histogram. Since the `blockproc` function calls the `edge` function for each block, the Canny algorithm is working with incomplete histograms and therefore using varying thresholds across the image.

When block processing images, it is important to understand these types of algorithm constraints. Some functions will not directly translate to block processing for all syntaxes. In this case, the `edge` function allows you to pass in a fixed threshold as an input argument instead of computing it. Modify your function handle to use the three-argument syntax of `edge`, and thus remove one of the "global" constraints of the function. Some trial and error finds that a threshold of 0.09 gives good results.

```
thresh = 0.09;
edgeFun = @(block_struct) edge(block_struct.data,'canny',thresh);
block_edges = blockproc(file_name,block_size,edgeFun,'BorderSize',border_size);

figure
imshow(block_edges)
title('Block Processing - Borders & Fixed Threshold')
```



The result now closely matches the original in-memory result. You can see some additional artifacts along the boundaries. These are due to the different methods of padding used by the Canny edge detector. Currently, `blockproc` only supports zero-padding along the image boundaries.

See Also

`blockproc` | `edge`

Related Examples

- “Computing Statistics for Large Images” on page 16-36

More About

- “Distinct Block Processing” on page 16-7
- “Block Size and Performance” on page 16-11
- “Create Function Handle” (MATLAB)

Computing Statistics for Large Images

This example shows how to use `blockproc` to compute statistics from large images and then use that information to more accurately process the images blockwise. The `blockproc` function is well suited for applying an operation to an image blockwise, assembling the results, and returning them as a new image. Many image processing algorithms, however, require "global" information about the image, which is not available when you are only considering one block of image data at a time. These constraints can prove to be problematic when working with images that are too large to load completely into memory.

This example performs a task similar to that found in the Enhancing Multispectral Color Composite Images example, but adapted for large images using `blockproc`. You will be enhancing the visible bands of the Erdas LAN file `rio.lan`. These types of block processing techniques are typically more useful for large images, but a small image will work for the purpose of this example.

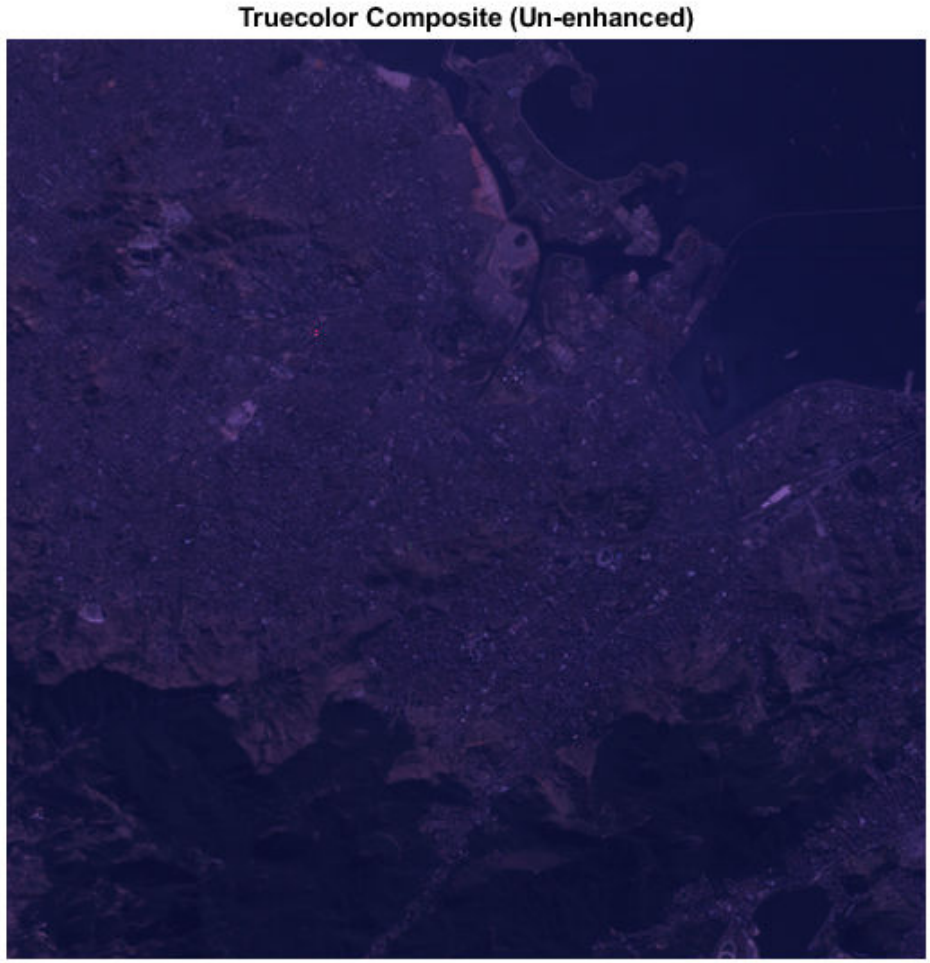
Step 1: Construct a Truecolor Composite

Using `blockproc`, read the data from `rio.lan`, a file containing Landsat thematic mapper imagery in the Erdas LAN file format. `blockproc` has built-in support for reading TIFF and JPEG2000 files only. To read other types of files you must write an Image Adapter class to support I/O for your particular file format. This example uses a pre-built Image Adapter class, the `LanAdapter`, which supports reading LAN files. For more information on writing Image Adapter classes see the tutorial in the Users' Guide describing how the `LanAdapter` class was built.

The Erdas LAN format contains the visible red, green, and blue spectrum in bands 3, 2, and 1, respectively. Use `blockproc` to extract the visible bands into an RGB image.

```
% Create the LanAdapter object associated with rio.lan.
input_adapter = LanAdapter('rio.lan');

% Select the visible R, G, and B bands.
input_adapter.SelectedBands = [3 2 1];

% Create a block function to simply return the block data unchanged.
identityFcn = @(block_struct) block_struct.data;

% Create the initial truecolor image.
truecolor = blockproc(input_adapter,[100 100],identityFcn);
```

```
% Display the un-enhanced results.
figure;
imshow(truecolor);
title('Truecolor Composite (Un-enhanced)');
```

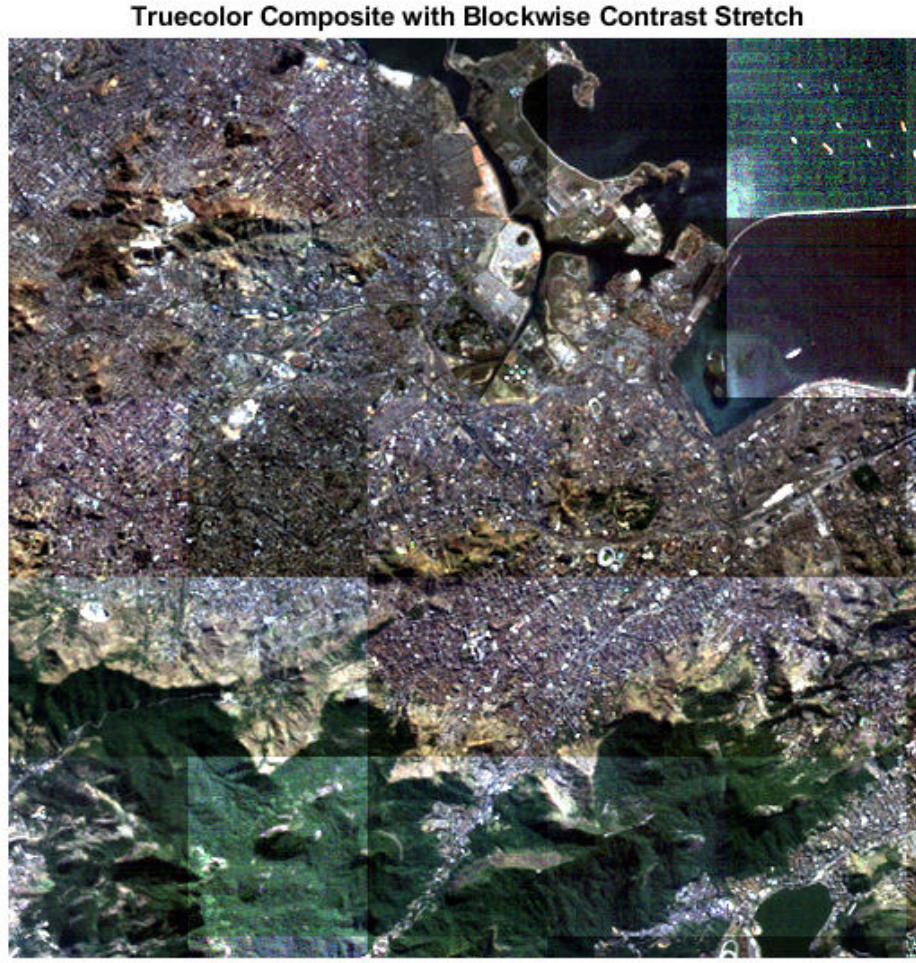


The resulting truecolor image is similar to that of `paris.lan` in the Enhancing Multispectral Color Composite Images example. The RGB image appears dull, with little contrast.

Step 2: Enhance the Image - First Attempt

First, try to stretch the data across the dynamic range using `blockproc`. This first attempt simply defines a new function handle that calls `stretchlim` and `imadjust` on each block of data individually.

```
adjustFcn = @(block_struct) imadjust(block_struct.data,...
    stretchlim(block_struct.data));
truecolor_enhanced = blockproc(input_adapter,[100 100],adjustFcn);
figure
imshow(truecolor_enhanced)
title('Truecolor Composite with Blockwise Contrast Stretch')
```



You can see immediately that the results are incorrect. The problem is that the `stretchlim` function computes the histogram on the input image and uses this information to compute the stretch limits. Since each block is adjusted in isolation from its neighbors, each block is computing different limits from its local histogram.

Step 3: Examine the Histogram Accumulator Class

To examine the distribution of data across the dynamic range of the image, you can compute the histogram for each of the three visible bands.

When working with sufficiently large images, you cannot simply call `imhist` to create an image histogram. One way to incrementally build the histogram is to use `blockproc` with a class that will sum the histograms of each block as you move over the image.

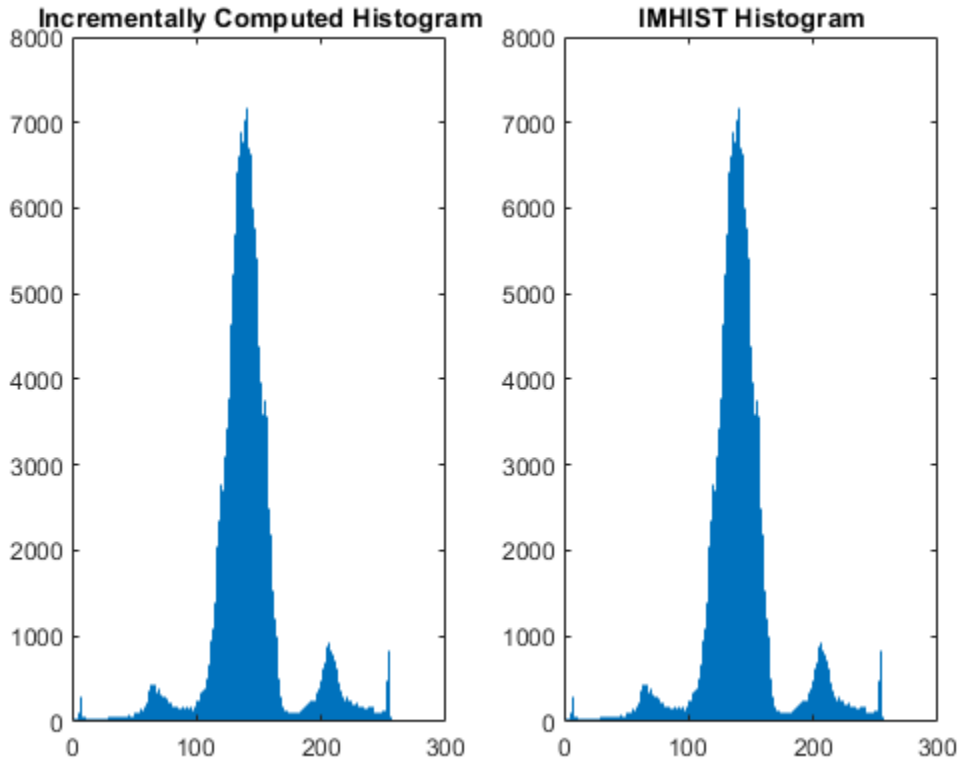
Examine the `HistogramAccumulator` class.

```
type HistogramAccumulator
```

```
% HistogramAccumulator Accumulate incremental histogram.
%   HistogramAccumulator is a class that incrementally builds up a
%   histogram for an image.  This class is appropriate for use with 8-bit
%   or 16-bit integer images and is for educational purposes ONLY.

%   Copyright 2009 The MathWorks, Inc.

classdef HistogramAccumulator < handle

    properties
        Histogram
        Range
    end

    methods

        function obj = HistogramAccumulator()
            obj.Range = [];
            obj.Histogram = [];
        end

        function addToHistogram(obj,new_data)
            if isempty(obj.Histogram)
                obj.Range = double(0:intmax(class(new_data)));
                obj.Histogram = hist(double(new_data(:)),obj.Range);
            else
                new_hist = hist(double(new_data(:)),obj.Range);
                obj.Histogram = obj.Histogram + new_hist;
            end
        end
    end
end
```

The class is a simple wrapper around the `hist` function, allowing you to add data to a histogram incrementally. It is not specific to `blockproc`. Observe the following simple use of the `HistogramAccumulator` class.

```
% Create the HistogramAccumulator object.
hist_obj = HistogramAccumulator();

% Split a sample image into 2 halves.
full_image = imread('liftingbody.png');
top_half = full_image(1:256,:);
bottom_half = full_image(257:end,:);

% Compute the histogram incrementally.
addToHistogram(hist_obj, top_half);
addToHistogram(hist_obj, bottom_half);
computed_histogram = hist_obj.Histogram;

% Compare against the results of IMHIST.
normal_histogram = imhist(full_image);

% Examine the results.  The histograms are numerically identical.
figure
subplot(1,2,1);
stem(computed_histogram,'Marker','none');
title('Incrementally Computed Histogram');
subplot(1,2,2);
stem(normal_histogram','Marker','none');
title('IMHIST Histogram');
```

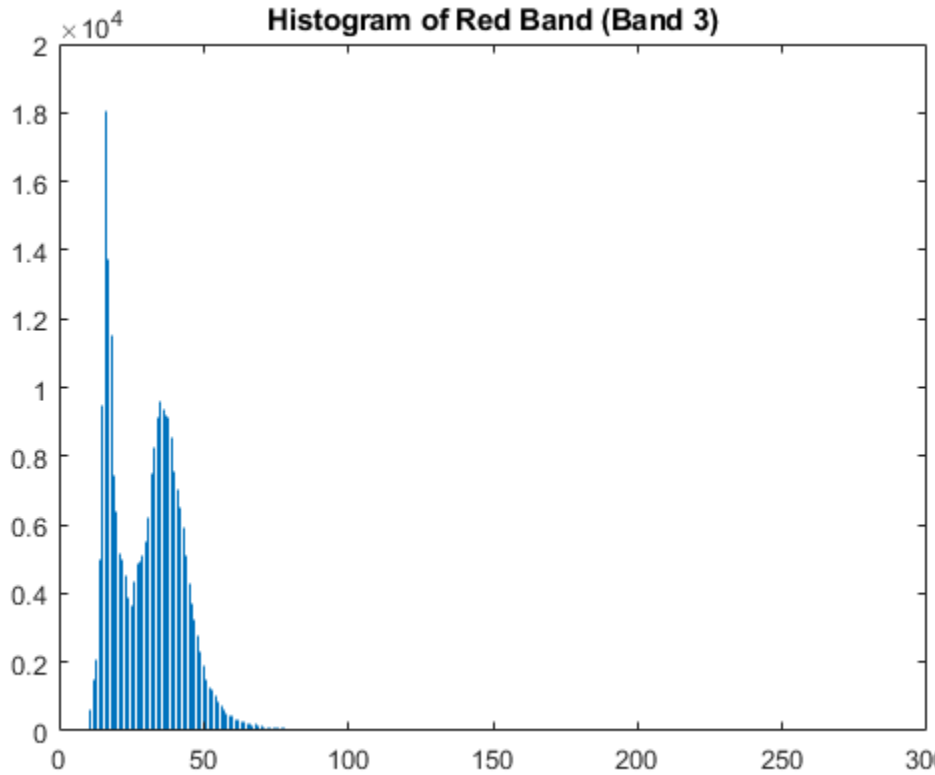


Step 4: Use the HistogramAccumulator Class with BLOCKPROC

Now use the `HistogramAccumulator` class with `blockproc` to build the histogram of the red band of data in `rio.lan`. You can define a function handle for `blockproc` that will invoke the `addToHistogram` method on each block of data. By viewing this histogram, you can see that the data is concentrated within a small part of the available dynamic range. The other visible bands have similar distributions. This is one reason why the original truecolor composite appears dull.

```
% Create the HistogramAccumulator object.
hist_obj = HistogramAccumulator();

% Setup blockproc function handle
addToHistFcn = @(block_struct) addToHistogram(hist_obj, block_struct.data);
```

```
% Compute histogram of the red channel.  Notice that the addToHistFcn
% function handle does generate any output.  Since the function handle we
% are passing to blockproc does not return anything, blockproc will not
% return anything either.
input_adapter.SelectedBands = 3;
blockproc(input_adapter,[100 100],addToHistFcn);
red_hist = hist_obj.Histogram;

% Display results.
figure
stem(red_hist,'Marker','none');
title('Histogram of Red Band (Band 3)');
```



Step 5: Enhance the Truecolor Composite with a Contrast Stretch

You can now perform a proper contrast stretch on the image. For conventional, in-memory workflows, you can simply use the `stretchlim` function to compute the arguments to `imadjust` (like the `MultispectralImageEnhancementExample` example does). When working with large images, as we have seen, `stretchlim` is not easily adapted for use with `blockproc` since it relies on the full image histogram.

Once you have computed the image histograms for each of the visible bands, compute the proper arguments to `imadjust` by hand (similar to how `stretchlim` does).

First compute the histograms for the green and blue bands.

```
% Compute histogram for green channel.
hist_obj = HistogramAccumulator();
addToHistFcn = @(block_struct) addToHistogram(hist_obj, block_struct.data);
input_adapter.SelectedBands = 2;
blockproc(input_adapter,[100 100],addToHistFcn);
green_hist = hist_obj.Histogram;

% Compute histogram for blue channel.
hist_obj = HistogramAccumulator();
addToHistFcn = @(block_struct) addToHistogram(hist_obj, block_struct.data);
input_adapter.SelectedBands = 1;
blockproc(input_adapter,[100 100],addToHistFcn);
blue_hist = hist_obj.Histogram;
```

Now compute the CDF of each histogram and prepare to call `imadjust`.

```
computeCDF = @(histogram) cumsum(histogram) / sum(histogram);
findLowerLimit = @(cdf) find(cdf > 0.01, 1, 'first');
findUpperLimit = @(cdf) find(cdf >= 0.99, 1, 'first');

red_cdf = computeCDF(red_hist);
red_limits(1) = findLowerLimit(red_cdf);
red_limits(2) = findUpperLimit(red_cdf);

green_cdf = computeCDF(green_hist);
green_limits(1) = findLowerLimit(green_cdf);
green_limits(2) = findUpperLimit(green_cdf);

blue_cdf = computeCDF(blue_hist);
blue_limits(1) = findLowerLimit(blue_cdf);
blue_limits(2) = findUpperLimit(blue_cdf);
```

```
% Prepare argument for IMADJUST.
rgb_limits = [red_limits' green_limits' blue_limits'];

% Scale to [0,1] range.
rgb_limits = (rgb_limits - 1) / (255);
```

Create a new `adjustFcn` that applies the global stretch limits and use `blockproc` to adjust the truecolor image.

```
adjustFcn = @(block_struct) imadjust(block_struct.data,rgb_limits);

% Select full RGB data.
input_adapter.SelectedBands = [3 2 1];
truecolor_enhanced = blockproc(input_adapter,[100 100],adjustFcn);
figure;
imshow(truecolor_enhanced)
title('Truecolor Composite with Corrected Contrast Stretch')
```

Truecolor Composite with Corrected Contrast Stretch

The resulting image is much improved, with the data covering more of the dynamic range, and by using `blockproc` you avoid loading the whole image into memory.

See Also

Classes
`ImageAdapter`

Functions
`blockproc` | `imadjust` | `imhist` | `stretchlim`

Related Examples

- “Enhancing Multispectral Color Composite Images” on page 11-115
- “Block Processing Large Images” on page 16-29

More About

- “Distinct Block Processing” on page 16-7
- “Perform Block Processing on Image Files in Unsupported Formats” on page 16-17
- “Create Function Handle” (MATLAB)

Deep Learning

This topic describes functions that enable image denoising using convolutional neural networks, and provides examples of other image processing applications using deep learning techniques.

- “Train and Apply Denoising Neural Networks” on page 17-2
- “Remove Noise from Color Image Using Pretrained Neural Network” on page 17-6
- “Single Image Super-Resolution Using Deep Learning” on page 17-13
- “JPEG Image Deblocking Using Deep Learning” on page 17-32
- “Semantic Segmentation of Multispectral Images Using Deep Learning” on page 17-50
- “Image Processing Operator Approximation Using Deep Learning” on page 17-72

Train and Apply Denoising Neural Networks

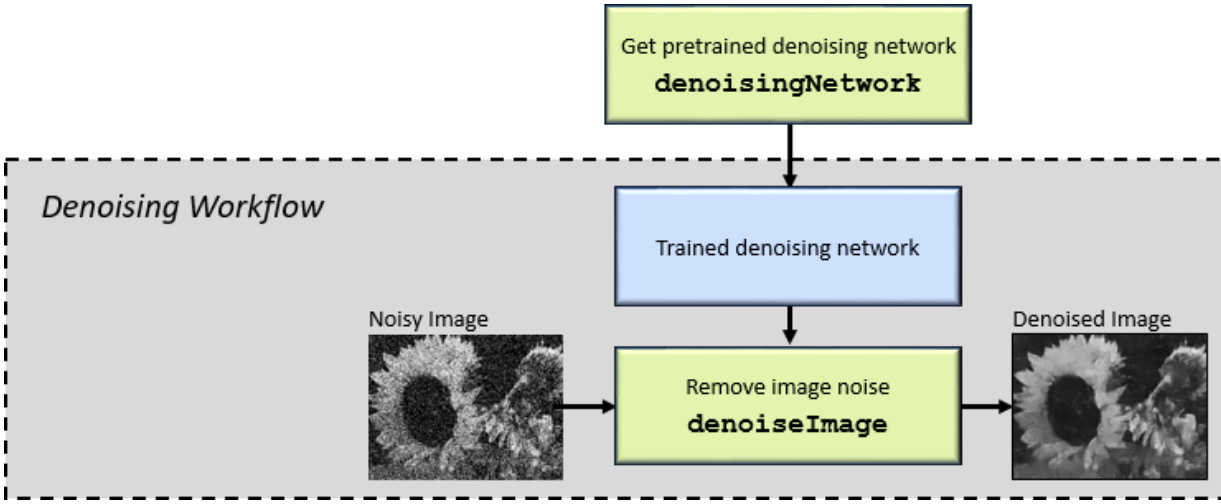
Image Processing Toolbox and Deep Learning Toolbox™ provide many options to remove noise from images. The simplest and fastest solution is to use the built-in pretrained denoising neural network, called DnCNN. However, the pretrained network does not offer much flexibility in the type of noise recognized. For more flexibility, train your own network using predefined layers, or train a fully custom denoising neural network.

Denoise Images Using Pretrained Network

You can use the built-in pretrained DnCNN network to remove Gaussian noise without the challenges of training a network. Removing noise with the pretrained network has these limitations:

- Noise removal works only with 2-D single-channel images. If you have multiple color channels, or if you are working with 3-D images, remove noise by treating each channel or plane separately. For an example, see “Remove Noise from Color Image Using Pretrained Neural Network” on page 17-6.
- The network recognizes only Gaussian noise, with a limited range of standard deviation.

To load the pretrained DnCNN network, use the `denoisingNetwork` function. Then, pass the DnCNN network and a noisy 2-D single-channel image to `denoiseImage`. The image shows the workflow to denoise an image using the pretrained DnCNN network.

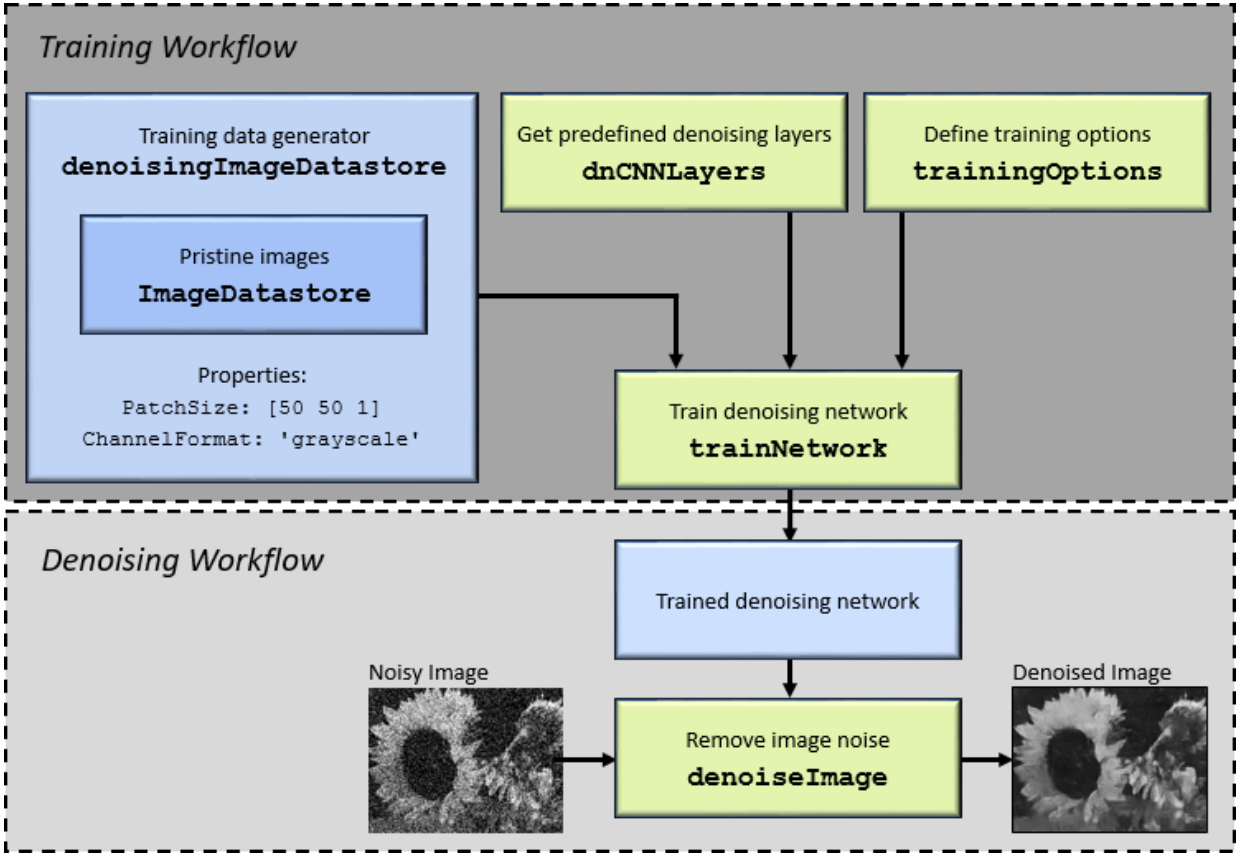


Train a Denoising Network Using Predefined Layers

You can train a network to detect a larger range of Gaussian noise standard deviations from grayscale images, starting with predefined layers provided by Image Processing Toolbox. To train a denoising network using predefined layers, follow these steps. The diagram shows the training workflow in the dark gray box.

- Create an `ImageDatastore` object that stores pristine images.
- Create a `denoisingImageDatastore` object that generates noisy training data from the pristine images in the `ImageDatastore`. To specify the range of the Gaussian noise standard deviations, set the `GaussianNoiseLevel` property. You must use the default value of `PatchSize` (`50`) and `ChannelFormat` (`'grayscale'`) so that the size of the training data matches the input size of the network.
- Get the predefined denoising layers using the `dnCNNLayers` function.
- Define training options using the `trainingOptions` function.
- Train the network, specifying the denoising image datastore as the data source for `trainNetwork`. For each iteration of training, the denoising image datastore generates one mini-batch of training data by randomly cropping pristine images from the `ImageDatastore`, then adding randomly generated zero-mean Gaussian white noise to each image patch. The standard deviation of the added noise is unique for each image patch, and has a value within the range specified by the `GaussianNoiseLevel` property of the denoising image datastore.

After you have trained the network, pass the network and a noisy grayscale image to `denoiseImage`. The diagram shows the denoising workflow in the light gray box.



Train Fully Customized Denoising Neural Network

To train a denoising neural network with maximum flexibility, you can use a custom datastore to generate training data or define your own network architecture. For example, you can:

- Train a network that detects a range of Gaussian noise distributions for color images. To generate training images for this network, you can use a `denoisingImageDatastore` and set the `ChannelFormat` property to `'rgb'`. You

must define a custom convolutional neural network architecture that supports RGB images.

- Train a network that detects a larger variety of noise, such as non-Gaussian noise distributions, in single-channel images. To define the network architecture, you can use the layers returned by the `dnCNNLayers` function. You must define a custom datastore that generates batches of noisy image patches and corresponding noise patches. For more information, see “Develop Custom Mini-Batch Datastore” (Deep Learning Toolbox).

  Although the DnCNN network was primarily designed to remove noise from images, it can also detect high-frequency image artifacts. For example, you can train the DnCNN network to increase image resolution or remove JPEG compression artifacts. The example “JPEG Image Deblocking Using Deep Learning” on page 17-32 shows how to train a DnCNN network with a mini-batch datastore that generates pairs of distorted image patches and corresponding noise patches.

If you train a denoising network using the DnCNN network architecture, then you can use the `denoiseImage` function to remove image noise or high-frequency artifacts. If you train a denoising network using a custom network architecture, then you can use the `activations` function to isolate the noise or high-frequency artifacts in a distorted image. Then, subtract the noise from the distorted image to obtain a denoised image.

See Also

`activations` | `denoiseImage` | `denoisingImageDatastore` | `denoisingNetwork` | `dnCNNLayers` | `trainNetwork` | `trainingOptions`

Related Examples

- “Remove Noise from Color Image Using Pretrained Neural Network” on page 17-6
- “JPEG Image Deblocking Using Deep Learning” on page 17-32

More About

- “Develop Custom Mini-Batch Datastore” (Deep Learning Toolbox)
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Remove Noise from Color Image Using Pretrained Neural Network

This example shows how to remove Gaussian noise from an RGB image. Split the image into separate color channels, then denoise each channel using a pretrained denoising neural network, DnCNN.

Read a color image into the workspace and convert the data to `double`. Display the pristine color image.

```
pristineRGB = imread('lighthouse.png');
pristineRGB = im2double(pristineRGB);
imshow(pristineRGB)
title('Pristine Image')
```

Pristine Image

Add zero-mean Gaussian white noise with a variance of 0.01 to the image. `imnoise` adds noise to each color channel independently. Display the noisy color image.

```
noisyRGB = imnoise(pristineRGB,'gaussian',0,0.01);
imshow(noisyRGB)
title('Noisy Image')
```

Noisy Image

Split the noisy RGB image into its individual color channels.

```
noisyR = noisyRGB(:,:,1);
noisyG = noisyRGB(:,:,2);
noisyB = noisyRGB(:,:,3);
```

Load the pretrained DnCNN network.

```
net = denoisingNetwork('dncnn');
```

Use the DnCNN network to remove noise from each color channel.

```
denoisedR = denoiseImage(noisyR,net);
denoisedG = denoiseImage(noisyG,net);
denoisedB = denoiseImage(noisyB,net);
```

Recombine the denoised color channels to form the denoised RGB image. Display the denoised color image.

```
denoisedRGB = cat(3,denoisedR,denoisedG,denoisedB);
imshow(denoisedRGB)
title('Denoised Image')
```

Denoised Image

Calculate the peak signal-to-noise ratio (PSNR) for the noisy and denoised images. A larger PSNR indicates that noise has a smaller relative signal, and is associated with higher image quality.

```
noisyPSNR = psnr(noisyRGB,pristineRGB);
fprintf('\n The PSNR value of the noisy image is %0.4f.',noisyPSNR);
```

```
 The PSNR value of the noisy image is 20.6395.
```

```
denoisedPSNR = psnr(denoisedRGB,pristineRGB);
fprintf('\n The PSNR value of the denoised image is %0.4f.',denoisedPSNR);
```

```
 The PSNR value of the denoised image is 29.6857.
```

Calculate the structural similarity (SSIM) index for the noisy and denoised images. An SSIM index close to 1 indicates good agreement with the reference image, and higher image quality.

```
noisySSIM = ssim(noisyRGB,pristineRGB);
fprintf('\n The SSIM value of the noisy image is %0.4f.',noisySSIM);
```

```
 The SSIM value of the noisy image is 0.7393.
```

```
denoisedSSIM = ssim(denoisedRGB,pristineRGB);
fprintf('\n The SSIM value of the denoised image is %0.4f.',denoisedSSIM);
```

```
 The SSIM value of the denoised image is 0.9507.
```

In practice, image color channels frequently have correlated noise. To remove correlated image noise, first convert the RGB image to a color space with a luminance channel, such as the L*a*b* color space. Remove noise on the luminance channel only, then convert the denoised image back to the RGB color space.

See Also

`denoiseImage` | `denoisingNetwork` | `imnoise` | `lab2rgb` | `psnr` | `rgb2lab` | `ssim`

More About

- “Train and Apply Denoising Neural Networks” on page 17-2

Single Image Super-Resolution Using Deep Learning

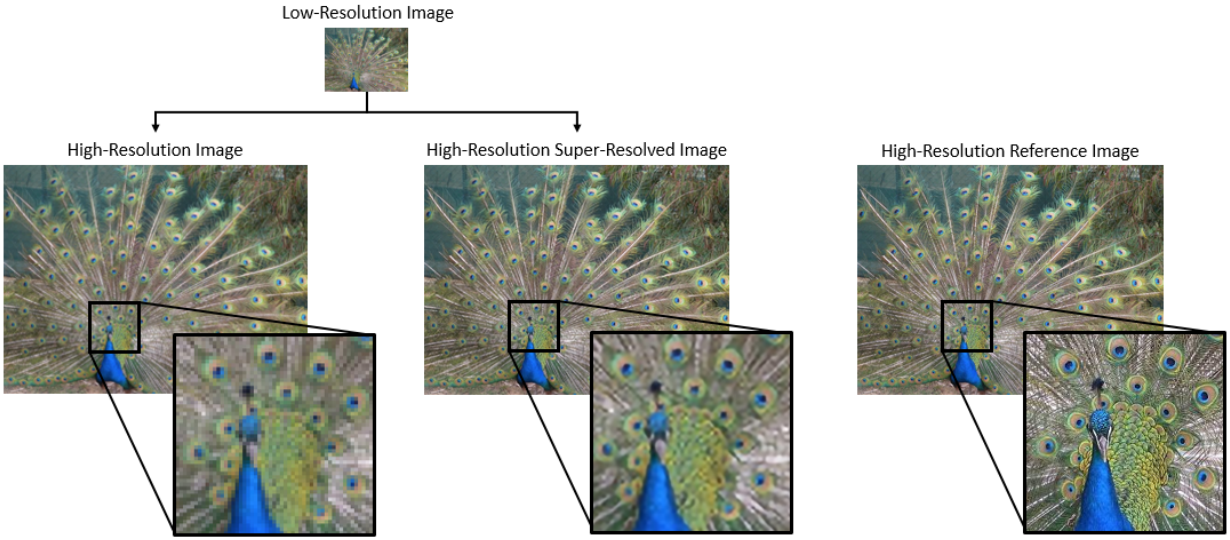
This example shows how to train a Very-Deep Super-Resolution (VDSR) neural network, then use a VDSR network to estimate a high-resolution image from a single low-resolution image.

The example shows how to train a VDSR network and also provides a pretrained VDSR network. If you choose to train the VDSR network, use of a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended. Use of a GPU requires the Parallel Computing Toolbox™.

If you do not want to download the training data set or train the network, then you can load the pretrained VDSR network by typing `load('trainedVDSR-Epoch-100-ScaleFactors-234.mat');` at the command line. Then, go directly to the *Perform Single Image Super-Resolution Using VDSR Network* section in this example.

Introduction

Super-resolution is the process of creating high-resolution images from low-resolution images. This example considers single image super-resolution (SISR), where the goal is to recover one high-resolution image from one low-resolution image. SISR is challenging because high-frequency image content typically cannot be recovered from the low-resolution image. Without high-frequency information, the quality of the high-resolution image is limited. Further, SISR is an ill-posed problem because a low-resolution image can yield several possible high-resolution images.



Several techniques, including deep learning algorithms, have been proposed to perform SISR. This example explores one deep learning algorithm for SISR, called very-deep super-resolution (VDSR) [1].

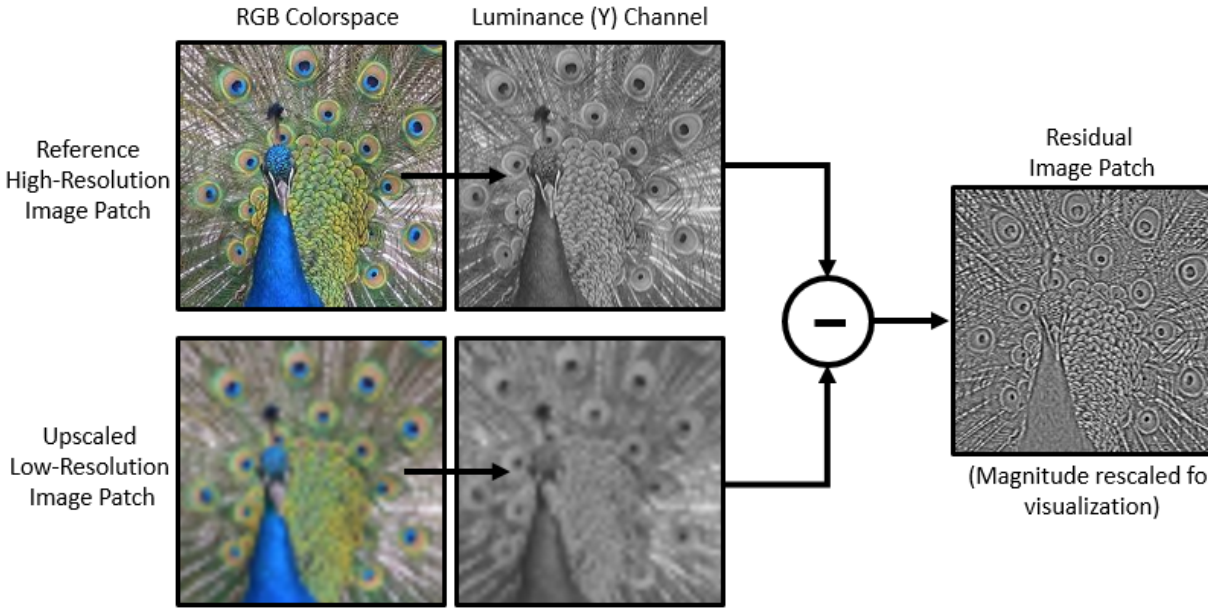
The VDSR Network

VDSR is a convolutional neural network architecture designed to perform single image super-resolution [1]. The VDSR network learns the mapping between low- and high-resolution images. This mapping is possible because low-resolution and high-resolution images have similar image content and differ primarily in high-frequency details.

VDSR employs a residual learning strategy, meaning that the network learns to estimate a residual image. In the context of super-resolution, a residual image is the difference between a high-resolution reference image and a low-resolution image that has been upscaled using bicubic interpolation to match the size of the reference image. A residual image contains information about the high-frequency details of an image.

The VDSR network detects the residual image from the luminance of a color image. The luminance channel of an image, Y, represents the brightness of each pixel through a linear combination of the red, green, and blue pixel values. In contrast, the two chrominance channels of an image, *Cb* and *Cr*, are different linear combinations of the red, green, and blue pixel values that represent color-difference information. VDSR is

trained using only the luminance channel because human perception is more sensitive to changes in brightness than to changes in color.



If Y_{highres} is the luminance of the high-resolution image and Y_{lowres} is the luminance a low-resolution image that has been upscaled using bicubic interpolation, then the input to the VDSR network is Y_{lowres} and the network learns to predict $Y_{\text{residual}} = Y_{\text{highres}} - Y_{\text{lowres}}$ from the training data.

After the VDSR network learns to estimate the residual image, you can reconstruct high-resolution images by adding the estimated residual image to the upsampled low-resolution image, then converting the image back to the RGB color space.

A scale factor relates the size of the reference image to the size of the low-resolution image. As the scale factor increases, SISR becomes more ill-posed because the low-resolution image loses more information about the high-frequency image content. VDSR solves this problem by using a large receptive field. This example trains a VDSR network with multiple scale factors using scale augmentation. Scale augmentation improves the results at larger scale factors because the network can take advantage of the image

context from smaller scale factors. Additionally, the VDSR network can generalize to accept images with interscale factors.

Download Training and Test Data

Download the IAPR TC-12 Benchmark, which consists of 20,000 still natural images [2]. The data set includes photos of people, animals, cities and more. You can use the helper function, `downloadIAPRTC12Data`, to download the data. The size of the data file is ~1.8 GB.

```
imagesDir = tempdir;
url = 'http://www-i6.informatik.rwth-aachen.de/imageclef/resources/iaprtc12.tgz';
downloadIAPRTC12Data(url,imagesDir);
```

This example will train the network with a small subset of the IAPR TC-12 Benchmark data. Load the imageCLEF training data. All images are 32-bit JPEG color images.

```
trainImagesDir = fullfile(imagesDir,'iaprtc12','images','02');
exts = {'.jpg','.bmp','.png'};
trainImages = imageDatastore(trainImagesDir,'FileExtensions',exts);
```

List the number of training images.

```
numel(trainImages.Files)
```

```
ans = 616
```

Define Random Patch Extraction Datastore for Training

Use a random patch extraction datastore to feed the training data to the network. This datastore extracts random corresponding patches from two image datastores that contain the network inputs and desired network responses.

In this example, the network inputs are low-resolution images that have been upscaled using bicubic interpolation. The desired network responses are the residual images. Use the helper function `createTrainingSet` to preprocess the training data, such as converting the images to the YCbCr color space, resizing the luminance (Y) channel by different scale factors using bicubic interpolation, and calculating the difference between the pristine and resized images. The processed input images are stored on disk as `.mat` files in the directory `upsampledDirName`. The computed residual images are stored on disk as `.mat` files in the directory `residualDirName`.

```
scaleFactors = [2 3 4];
[upsampledDirName,residualDirName] = createTrainingSet(trainImages,scaleFactors);
```

Create an image datastore called `upsampledImages` from the collection of input image files. Create an image datastore called `residualImages` from the collection of computed residual image files. Both datastores require a helper function, `matRead`, to read the image data from the image files.

```
upsampledImages = imageDatastore(upsampledDirName,'FileExtensions','.mat','ReadFcn',@ma
residualImages = imageDatastore(residualDirName,'FileExtensions','.mat','ReadFcn',@matR
```

Create an `imageDataAugmenter` that specifies the parameters of data augmentation. Use data augmentation during training to vary the training data, which effectively increases the amount of available training data. Here, the augmenter specifies random rotation by 90 degrees and random reflections in the *x*-direction.

```
augmenter = imageDataAugmenter( ...
    'RandRotation',@()randi([0,1],1)*90, ...
    'RandXReflection',true);
```

Create the `randomPatchExtractionDatastore` from the two image datastores. Specify a patch size of 41-by-41 pixels (the choice of patch size is later explained when setting up the VDSR layers). Specify `'PatchesPerImage'` to extract 64 randomly-positioned patches from each pair of images during training. Specify a mini-batch size of 64.

```
miniBatchSize = 64;
patchSize = [41 41];
patchds = randomPatchExtractionDatastore(upsampledImages,residualImages,patchSize, ...
    'PatchesPerImage',64, ...
    'DataAugmentation',augmenter);
patchds.MiniBatchSize = miniBatchSize;
```

The `randomPatchExtractionDatastore` provides mini-batches of data to the network at each iteration of the epoch. Perform a read operation on the datastore to explore the data.

```
inputBatch = read(patchds);
summary(inputBatch)
```

```
Variables:

    InputImage: 64×1 cell

    ResponseImage: 64×1 cell
```

Set Up VDSR Layers

This example defines the VDSR network using 41 individual layers from Deep Learning Toolbox™, including:

- `imageInputLayer` - Image input layer
- `convolution2dLayer` - 2-D convolution layer for convolutional neural networks
- `reluLayer` - Rectified linear unit (ReLU) layer
- `regressionLayer` - Regression output layer for a neural network

The first layer, `imageInputLayer`, operates on image patches. The patch size is based on the network receptive field which is the spatial image region that affects the response of the top-most layer in the network. Ideally, the network receptive field is the same as the image size so that it can see all the high-level features in the image. In this case, for a network of depth D, the receptive field is $(2D+1)$-by-$(2D+1)$. Because VDSR is a 20-layer network, the receptive field and the image patch size are 41-by-41. The image input layer accepts images with one channel because VDSR is trained using only the luminance channel.

```
networkDepth = 20;
firstLayer = imageInputLayer([41 41 1],'Name','InputLayer','Normalization','none');
```

The image input layer is followed by a 2-D convolutional layer that contains 64 filters of size 3-by-3. The mini-batch size determines the number of filters. Zero-pad the inputs to each convolutional layer so that the feature maps remain the same size as the input after each convolution. He's method [3] initializes the weights to random values so that there is asymmetry in neuron learning. Each convolutional layer is followed by a ReLU layer, which introduces nonlinearity in the network.

```
convolutionLayer = convolution2dLayer(3,64,'Padding',1, ...
    'Name','Conv1');
convolutionLayer.Weights = sqrt(2/(9*64))*randn(3,3,1,64);
convolutionLayer.Bias = zeros(1,1,64);
```

Specify a ReLU layer.

```
relLayer = reluLayer('Name','ReLU1');
```

The middle layers contain 18 alternating convolutional and rectified linear unit layers. Every convolutional layer contains 64 filters of size 3-by-3-by-64, where a filter operates on a 3-by-3 spatial region across 64 channels. He's method [3] initializes the weights to random values. As before, a ReLU layer follows every convolutional layer.

```
middleLayers = [convolutionLayer relLayer];
for layerNumber = 2:networkDepth-1
    conv2dLayer = convolution2dLayer(3,64, ...
        'Padding',[1 1], ...
        'Name',['Conv' num2str(layerNumber)]);

    % He initialization
    conv2dLayer.Weights = sqrt(2/(9*64))*randn(3,3,64,64);
    conv2dLayer.Bias = zeros(1,1,64);

    relLayer = reluLayer('Name',['ReLU' num2str(layerNumber)]);
    middleLayers = [middleLayers conv2dLayer relLayer];
end
```

The penultimate layer is a convolutional layer with a single filter of size 3x3x64 that reconstructs the image.

```
conv2dLayer = convolution2dLayer(3,1,...
    'NumChannels',64,...
    'Padding',[1 1],...
    'Name',['Conv' num2str(networkDepth)]);
conv2dLayer.Weights = sqrt(2/(9*64))*randn(3,3,64,1);
conv2dLayer.Bias = zeros(1,1,1);
```

The last layer is a regression layer instead of a ReLU layer. The regression layer computes the mean-squared error between the residual image and network prediction.

```
finalLayers = [conv2dLayer regressionLayer('Name','FinalRegressionLayer')];
```

Concatenate all the layers to form the VDSR network.

```
layers = [firstLayer middleLayers finalLayers];
```

Alternatively, you can use this helper function to create VDSR layers.

```
layers = vdsrLayers();
```

Specify Training Options

Train the network using stochastic gradient descent with momentum (SGDM) optimization. Specify the hyperparameter settings for SDGM by using the `trainingOptions` function. The learning rate is initially set to 0.1 and then decreased by a factor of 10 every 20 epochs. Train for 100 epochs.

Training a deep network is time-consuming. Accelerate the training by specifying a high learning rate. However, this can cause the gradients of the network to explode or grow

uncontrollably, preventing the network from training successfully. To keep the gradients in a meaningful range, enable gradient clipping by setting `'GradientThreshold'` to a value of 0.01, and specify `'GradientThresholdMethod'` to use the L2-norm of the gradients.

```
maxEpochs = 100;
epochIntervals = 1;
initLearningRate = 0.1;
learningRateFactor = 0.1;
l2reg = 0.0001;
options = trainingOptions('sgdm', ...
    'Momentum',0.9, ...
    'InitialLearnRate',initLearningRate, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',10, ...
    'LearnRateDropFactor',learningRateFactor, ...
    'L2Regularization',l2reg, ...
    'MaxEpochs',maxEpochs ,...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThresholdMethod','l2norm', ...
    'Plots','training-progress', ...
    'GradientThreshold',0.01);
```

Train the Network

After configuring the training options and the random patch extraction datastore, train the VDSR network using the `trainNetwork` function. To train the network, set the `doTraining` parameter in the following code to `true`. A CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

If you keep the `doTraining` parameter in the following code as `false`, then the example returns a pretrained VDSR network that has been trained to super-resolve images for scale factors 2, 3 and 4.

Note: Training takes about 6 hours on an NVIDIA™ Titan X and can take even longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    modelDateTime = datestr(now,'dd-mmm-yyyy-HH-MM-SS');
    net = trainNetwork(patchds,layers,options);
    save(['trainedVDSR-' modelDateTime '-Epoch-' num2str(maxEpochs*epochIntervals) 'Sca
else
```

```
    load('trainedVDSR-Epoch-100-ScaleFactors-234.mat');
end
```

Perform Single Image Super-Resolution Using VDSR Network

To perform single image super-resolution (SISR) using the VDSR network, follow the remaining steps of this example. The remainder of the example shows how to:

- Create a sample low-resolution image from a high-resolution reference image.
- Perform SISR on the low-resolution image using bicubic interpolation, a traditional image processing solution that does not rely on deep learning.
- Perform SISR on the low-resolution image using the VDSR neural network.
- Visually compare the reconstructed high-resolution images using bicubic interpolation and VDSR.
- Evaluate the quality of the super-resolved images by quantifying the similarity of the images to the high-resolution reference image.

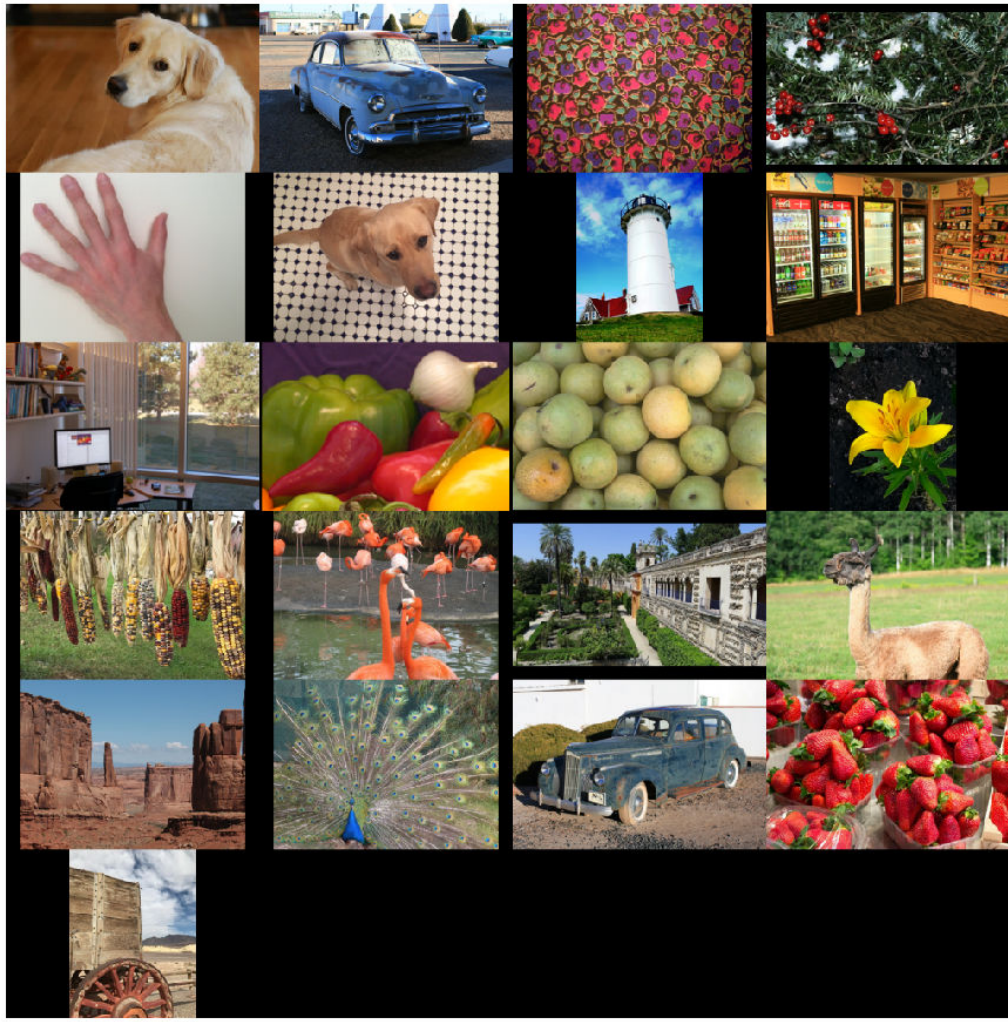
Create Sample Low-Resolution Image

Create a low-resolution image that will be used to compare the results of super-resolution using deep-learning to the result using traditional image processing techniques such as bicubic interpolation. The test data set, `testImages`, contains 21 undistorted images shipped in Image Processing Toolbox™. Load the images into an `imageDatastore`.

```
exts = {'.jpg','.png'};
fileNames = {'sherlock.jpg','car2.jpg','fabric.png','greens.jpg','hands1.jpg','kobi.png
    'lighthouse.png','micromarket.jpg','office_4.jpg','onion.png','pears.png','yellowli
    'indiancorn.jpg','flamingos.jpg','sevilla.jpg','llama.jpg','parkavenue.jpg', ...
    'peacock.jpg','car1.jpg','strawberries.jpg','wagon.jpg'};
filePath = [fullfile(matlabroot,'toolbox','images','imdata') filesep];
filePathNames = strcat(filePath,fileNames);
testImages = imageDatastore(filePathNames,'FileExtensions',exts);
```

Display the testing images as a montage.

```
montage(testImages)
```



Select one of the images to use as the reference image for super-resolution. You can optionally use your own high-resolution image as the reference image.

```
indx = 1; % Index of image to read from the test image datastore
Ireference = readimage(testImages,indx);
Ireference = im2double(Ireference);
imshow(Ireference)
title('High-Resolution Reference Image')
```

Create a low-resolution version of the high-resolution reference image by using `imresize` with a scaling factor of 0.25. The high-frequency components of the image are lost during the downscaling.

```
scaleFactor = 0.25;
Ilowres = imresize(Ireference,scaleFactor,'bicubic');
imshow(Ilowres)
title('Low-Resolution Image')
```

Improve Image Resolution Using Bicubic Interpolation

A standard way to increase image resolution without deep learning is to use bicubic interpolation. Upscale the low-resolution image using bicubic interpolation so that the resulting high-resolution image is the same size as the reference image.

```
[nrows,ncols,np] = size(Ireference);
Ibicubic = imresize(Ilowres,[nrows ncols],'bicubic');
imshow(Ibicubic)
title('High-Resolution Image Obtained Using Bicubic Interpolation')
```

High-Resolution Image Obtained Using Bicubic Interpolation

Improve Image Resolution Using Pretrained VDSR Network

Recall that VDSR is trained using only the luminance channel of an image because human perception is more sensitive to changes in brightness than to changes in color.

Convert the low-resolution image from the RGB color space to luminance (`Iy`) and chrominance (`Icb` and `Icr`) channels by using the `rgb2ycbcr` function.

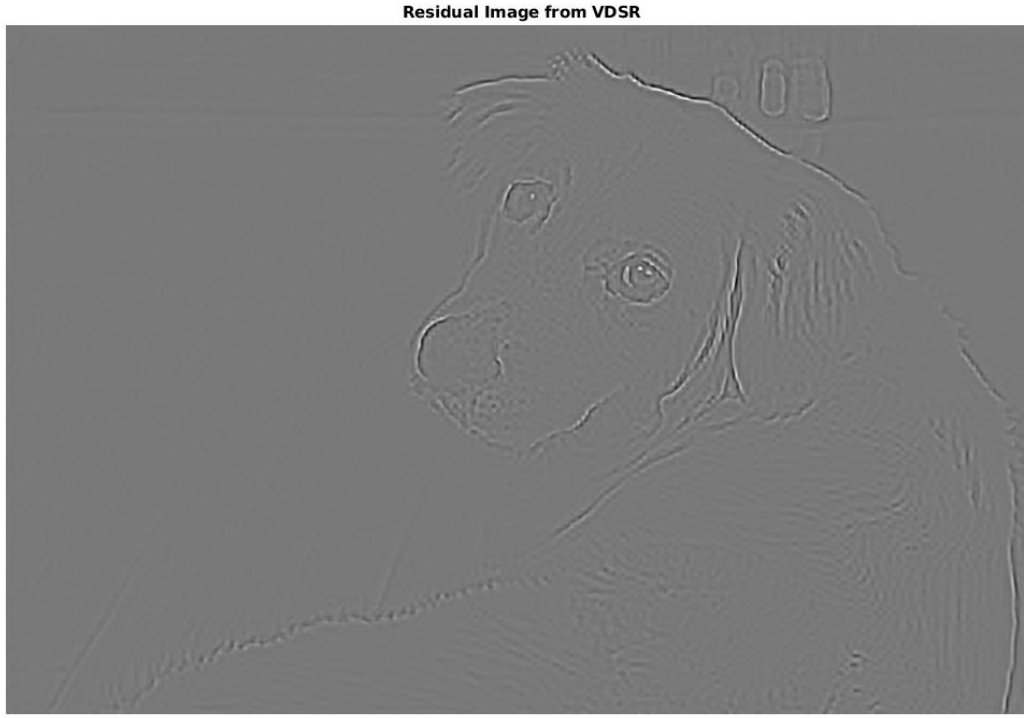
```
Iycbcr = rgb2ycbcr(Ilowres);
Iy = Iycbcr(:,:,1);
Icb = Iycbcr(:,:,2);
Icr = Iycbcr(:,:,3);
```

Upscale the luminance and two chrominance channels using bicubic interpolation. The upsampled chrominance channels, `Icb_bicubic` and `Icr_bicubic`, require no further processing.

```
Iy_bicubic = imresize(Iy,[nrows ncols],'bicubic');
Icb_bicubic = imresize(Icb,[nrows ncols],'bicubic');
Icr_bicubic = imresize(Icr,[nrows ncols],'bicubic');
```

Pass the upscaled luminance component, `Iy_bicubic`, through the trained VDSR network. Observe the `activations` from the final layer (a regression layer). The output of the network is the desired residual image.

```
Iresidual = activations(net,Iy_bicubic,41);
Iresidual = double(Iresidual);
imshow(Iresidual,[])
title('Residual Image from VDSR')
```



Add the residual image to the upscaled luminance component to get the high-resolution VDSR luminance component.

```
Isr = Iy_bicubic + Iresidual;
```

Concatenate the high-resolution VDSR luminance component with the upscaled color components. Convert the image to the RGB color space by using the `ycbcr2rgb` function. The result is the final high-resolution color image using VDSR.

```
Ivdsr = ycbcr2rgb(cat(3,Isr,Icb_bicubic,Icr_bicubic));
imshow(Ivdsr)
title('High-Resolution Image Obtained Using VDSR')
```

Visual and Quantitative Comparison

To get a better visual understanding of the high-resolution images, examine a small region inside each image. Specify a region of interest (ROI) using vector `roi` in the format [*x y width height*]. The elements define the x- and y-coordinate of the top left corner, and the width and height of the ROI.

```
roi = [320 30 480 400];
```

Crop the high-resolution images to this ROI, and display the result as a montage.

```
montage({imcrop(Ibicubic,roi),imcrop(Ivdsr,roi)})
title('High-Resolution Results Using Bicubic Interpolation (Left) vs. VDSR (Right)');
```

The VDSR image has clearer details and sharper edges.

Use image quality metrics to quantitatively compare the high-resolution image using bicubic interpolation to the VDSR image. The reference image is the original high-resolution image, `Ireference`, before preparing the sample low-resolution image.

Measure the peak signal-to-noise ratio (PSNR) of each image against the reference image. Larger PNSR values generally indicate better image quality. See `psnr` for more information about this metric.

```
bicubicPSNR = psnr(Ibicubic,Ireference)
```

```
bicubicPSNR = 38.4747
```

```
vdsrPSNR = psnr(Ivdsr,Ireference)
```

```
vdsrPSNR = 39.2491
```

Measure the structural similarity index (SSIM) of each image. SSIM assesses the visual impact of three characteristics of an image: luminance, contrast and structure, against a reference image. The closer the SSIM value is to 1, the better the test image agrees with the reference image. See `ssim` for more information about this metric.

```
bicubicSSIM = ssim(Ibicubic,Ireference)
```

```
bicubicSSIM = 0.9861
```

```
vdsrSSIM = ssim(Ivdsr,Ireference)
```

```
vdsrSSIM = 0.9875
```

Measure perceptual image quality using the Naturalness Image Quality Evaluator (NIQE). Smaller NIQE scores indicate better perceptual quality. See `niqe` for more information about this metric.

```
bicubicNIQE = niqe(Ibicubic)
```

```
bicubicNIQE = 5.1719
```

```
vdsrNIQE = niqe(Ivdsr)
```

```
vdsrNIQE = 4.7015
```

Calculate the average PSNR and SSIM of the entire set of test images for the scale factors 2, 3, and 4. These are the same scale factors that were used to define the randomPatchExtraction`Datastore`. For simplicity, you can use the helper function, `superResolutionMetrics`, to compute the average metrics.

```
superResolutionMetrics(net,testImages,scaleFactors);
```

```
Results for Scale factor 2

Average PSNR for Bicubic = 31.809683
Average PSNR for VDSR = 31.926088
Average SSIM for Bicubic = 0.938194
Average SSIM for VDSR = 0.949278

Results for Scale factor 3

Average PSNR for Bicubic = 28.170441
Average PSNR for VDSR = 28.515179
Average SSIM for Bicubic = 0.884381
Average SSIM for VDSR = 0.895356
```

```
Results for Scale factor 4

Average PSNR for Bicubic = 27.010839
Average PSNR for VDSR = 27.849618
Average SSIM for Bicubic = 0.861604
Average SSIM for VDSR = 0.877360
```

For each scale factor, compared to bicubic interpolation, VDSR has better metric scores.

Summary

This example shows how to create and train a VDSR network for multiple scale factors, then use the network to improve image resolution through super-resolution. These are the steps to training the network:

- Download the training data.
- Define a `RandomPatchExtractionDatastore` that feeds training data to the network.
- Define the layers of the VDSR network.
- Specify training options.
- Train the network using the `trainNetwork` function.

After training the VDSR network or loading a pretrained VDSR network, the example performs super-resolution on a low-resolution image. The example compares the VDSR result against super-resolution using bicubic interpolation, which does not use deep learning. VDSR outperforms bicubic interpolation with respect to both perceptual image quality and quantitative quality measurements.

References

[1] Kim, J., J. K. Lee, and K. M. Lee. "Accurate Image Super-Resolution Using Very Deep Convolutional Networks." *Proceedings of the IEEE® Conference on Computer Vision and Pattern Recognition*. 2016, pp. 1646-1654.

[2] Grubinger, M., P. Clough, H. Müller, and T. Deselaers. "The IAPR TC-12 Benchmark: A New Evaluation Resource for Visual Information Systems." *Proceedings of the OntoImage 2006 Language Resources For Content-Based Image Retrieval*. Genoa, Italy. Vol. 5, May 2006, p. 10.

[3] He, K., X. Zhang, S. Ren, and J. Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 1026-1034.

See Also

`randomPatchExtractionDatastore` | `rgb2ycbcr` | `trainNetwork` | `trainingOptions` | `ycbcr2rgb`

More About

- “Preprocess Images for Deep Learning” (Deep Learning Toolbox)
- “List of Deep Learning Layers” (Deep Learning Toolbox)

JPEG Image Deblocking Using Deep Learning

This example shows how to train a denoising convolutional neural network (DnCNN), then use the network to reduce JPEG compression artifacts in an image.

The example shows how to train a DnCNN network and also provides a pretrained DnCNN network. If you choose to train the DnCNN network, use of a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended (requires Parallel Computing Toolbox™).

If you do not want to download the training data set or train the network, then you can load the pretrained DnCNN network by typing `load('pretrainedJPEGDnCNN.mat')` at the command line. Then, go directly to the *Perform JPEG Deblocking Using DnCNN Network* section in this example.

Introduction

Image compression is used to reduce the memory footprint of an image. One popular and powerful compression method is employed by the JPEG image format, which uses a quality factor to specify the amount of compression. Reducing the quality value results in higher compression and a smaller memory footprint, at the expense of visual quality of the image.

JPEG compression is *lossy*, meaning that the compression process causes the image to lose information. For JPEG images, this information loss appears as blocking artifacts in the image. As shown in the figure, more compression results in more information loss and stronger artifacts. Textured regions with high-frequency content, such as the grass and clouds, look blurry. Sharp edges, such as the roof of the house and the guardrails atop the lighthouse, exhibit ringing.

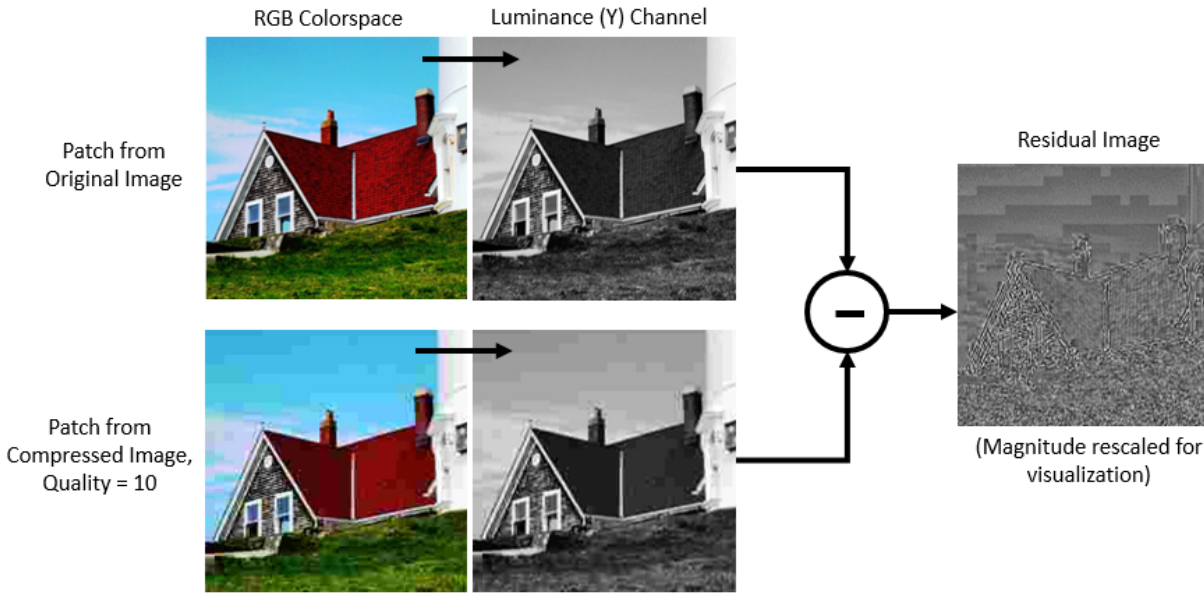
JPEG deblocking is the process of reducing the effects of compression artifacts in JPEG images. Several JPEG deblocking methods exist, including more effective methods that use deep learning. This example implements one such deep learning-based method that attempts to minimize the effect of JPEG compression artifacts.

The DnCNN Network

This example uses a built-in deep feed-forward convolutional neural network, called `DnCNN`. The network was primarily designed to remove noise from images. However, the DnCNN architecture can also be trained to remove JPEG compression artifacts or increase image resolution.

The reference paper [1] employs a residual learning strategy, meaning that the DnCNN network learns to estimate the residual image. A residual image is the difference between a pristine image and a distorted copy of the image. The residual image contains information about the image distortion. For this example, distortion appears as JPEG blocking artifacts.

The DnCNN network is trained to detect the residual image from the luminance of a color image. The luminance channel of an image, *Y*, represents the brightness of each pixel through a linear combination of the red, green, and blue pixel values. In contrast, the two chrominance channels of an image, *Cb* and *Cr*, are different linear combinations of the red, green, and blue pixel values that represent color-difference information. DnCNN is trained using only the luminance channel because human perception is more sensitive to changes in brightness than changes in color.



If Y_{Original} is the luminance of the pristine image and $Y_{\text{Compressed}}$ is the luminance of the image containing JPEG compression artifacts, then the input to the DnCNN network is $Y_{\text{Compressed}}$ and the network learns to predict $Y_{\text{Residual}} = Y_{\text{Compressed}} - Y_{\text{Original}}$ from the training data.

Once the DnCNN network learns how to estimate a residual image, it can reconstruct an undistorted version of a compressed JPEG image by adding the residual image to the compressed luminance channel, then converting the image back to RGB color space.

Download Training Data

Download the IAPR TC-12 Benchmark, which consists of 20,000 still natural images [2]. The data set includes photos of people, animals, cities and more. You can use the helper function, `downloadIAPRTC12Data`, to download the data. The size of the data file is ~1.8 GB.

```
imagesDir = tempdir;
url = "http://www-i6.informatik.rwth-aachen.de/imageclef/resources/iaprtc12.tgz";
downloadIAPRTC12Data(url,imagesDir);
```

This example will train the network with a small subset of the IAPR TC-12 Benchmark data. Load the imageCLEF training data. All images are 32-bit JPEG color images.

```
trainImagesDir = fullfile(imagesDir,'iaprtc12','images','00');
exts = {'.jpg','.bmp','.png'};
trainImages = imageDatastore(trainImagesDir,'FileExtensions',exts);
```

List the number of training images.

```
numel(trainImages.Files)
```

```
ans = 251
```

Prepare Training Data

To create a training data set, read in pristine images and write out images in the JPEG file format with various levels of compression.

Create the folder structure to properly organize the training data.

```
originalFileLocation = fullfile(imagesDir,'iaprtc12','images','00');

% Make a folder structure for the training data
if ~exist(fullfile(imagesDir,'iaprtc12','JPEGDeblockingData','Original'),'dir')
    mkdir(fullfile(imagesDir,'iaprtc12','JPEGDeblockingData','Original'));
end
if ~exist(fullfile(imagesDir,'iaprtc12','JPEGDeblockingData','Compressed'),'dir')
    mkdir(fullfile(imagesDir,'iaprtc12','JPEGDeblockingData','Compressed'));
end

uncompressedFileLocation = fullfile(imagesDir,'iaprtc12','JPEGDeblockingData','Original
compressedFileLocation = fullfile(imagesDir,'iaprtc12','JPEGDeblockingData','Compressed
```

Specify the JPEG image quality values used to render image compression artifacts. Quality values must be in the range [0, 100]. Small quality values result in more compression and stronger compression artifacts. Use a denser sampling of small quality values so the training data has a broad range of compression artifacts.

```
JPEGQuality = [5:5:40 50 60 70 80];
```

Write pristine and compressed training images from the original data.

```
files = dir([originalFileLocation filesep '*.jpg']);
imNumber = 1;
for fileIndex = 1:size(files,1)
```

```
    fname = [originalFileLocation filesep files(fileIndex).name];
    im = imread(fname);
    if size(im,3) == 3
        im = rgb2gray(im);
    end
    for index = 1:length(JPEGQuality)
        imwrite(im,[uncompressedFileLocation filesep num2str(imNumber) '.jpg'],'JPEG',
        imwrite(im,[compressedFileLocation filesep num2str(imNumber) '.jpg'],'JPEG','Qu
        imNumber = imNumber + 1;
    end
end
```

Define Mini-Batch Datastore for Training

A mini-batch datastore is used to feed the training data to the network. This example defines a custom implementation of a mini-batch datastore, called `JPEGimagePatchDatastore`, as a convenient way to generate augmented image patches for training a JPEG deblocking network.

The `JPEGimagePatchDatastore` extracts patches from the distorted input images and computes the target residuals from the corresponding patches in the pristine images. The distorted image patches act as the network input. The residual patches are the desired network output. Each mini-batch contains 128 patches of size 50-by-50 pixels. Only one mini-batch will be extracted from each image during training and all patches will be extracted from random positions in the images.

```
batchSize = 128;
patchSize = 50;
batchesPerImage = 1;

exts = {'.jpg'};
imdsUncompressed = imageDatastore(uncompressedFileLocation,'FileExtensions',exts);
imdsCompressed = imageDatastore(compressedFileLocation,'FileExtensions',exts);

ds = JPEGimagePatchDatastore(imdsUncompressed,imdsCompressed,...
    'MiniBatchSize',batchSize,...
    'PatchSize',patchSize,...
    'BatchesPerImage',batchesPerImage);
```

Perform a read operation on the mini-batch datastore to explore the data.

```
inputBatch = read(ds);
summary(inputBatch)
```

```
Variables:

    jpegPatches: 128×1 cell

    residuals: 128×1 cell
```

Set up DnCNN Layers

Create the layers of the built-in DnCNN network by using the `dnCNNLayers` function. By default, the network depth (the number of convolution layers) is 20.

```
layers = dnCNNLayers()
```

```
layers = 
  1x59 Layer array with layers:

     1   'InputLayer'             Image Input           50x50x1 images
     2   'Conv1'                  Convolution           64 3x3x1 convolutions with stri
     3   'ReLU1'                  ReLU                  ReLU
     4   'Conv2'                  Convolution           64 3x3x64 convolutions with str
     5   'BNorm2'                 Batch Normalization   Batch normalization with 64 cha
     6   'ReLU2'                  ReLU                  ReLU
     7   'Conv3'                  Convolution           64 3x3x64 convolutions with str
     8   'BNorm3'                 Batch Normalization   Batch normalization with 64 cha
     9   'ReLU3'                  ReLU                  ReLU
    10   'Conv4'                  Convolution           64 3x3x64 convolutions with str
    11   'BNorm4'                 Batch Normalization   Batch normalization with 64 cha
    12   'ReLU4'                  ReLU                  ReLU
    13   'Conv5'                  Convolution           64 3x3x64 convolutions with str
    14   'BNorm5'                 Batch Normalization   Batch normalization with 64 cha
    15   'ReLU5'                  ReLU                  ReLU
    16   'Conv6'                  Convolution           64 3x3x64 convolutions with str
    17   'BNorm6'                 Batch Normalization   Batch normalization with 64 cha
    18   'ReLU6'                  ReLU                  ReLU
    19   'Conv7'                  Convolution           64 3x3x64 convolutions with str
    20   'BNorm7'                 Batch Normalization   Batch normalization with 64 cha
    21   'ReLU7'                  ReLU                  ReLU
    22   'Conv8'                  Convolution           64 3x3x64 convolutions with str
    23   'BNorm8'                 Batch Normalization   Batch normalization with 64 cha
    24   'ReLU8'                  ReLU                  ReLU
    25   'Conv9'                  Convolution           64 3x3x64 convolutions with str
    26   'BNorm9'                 Batch Normalization   Batch normalization with 64 cha
    27   'ReLU9'                  ReLU                  ReLU
    28   'Conv10'                 Convolution           64 3x3x64 convolutions with str
    29   'BNorm10'                Batch Normalization   Batch normalization with 64 cha
```

```
 30   'ReLU10'                 ReLU                   ReLU
 31   'Conv11'                 Convolution            64 3x3x64 convolutions with str
 32   'BNorm11'                Batch Normalization    Batch normalization with 64 cha
 33   'ReLU11'                 ReLU                   ReLU
 34   'Conv12'                 Convolution            64 3x3x64 convolutions with str
 35   'BNorm12'                Batch Normalization    Batch normalization with 64 cha
 36   'ReLU12'                 ReLU                   ReLU
 37   'Conv13'                 Convolution            64 3x3x64 convolutions with str
 38   'BNorm13'                Batch Normalization    Batch normalization with 64 cha
 39   'ReLU13'                 ReLU                   ReLU
 40   'Conv14'                 Convolution            64 3x3x64 convolutions with str
 41   'BNorm14'                Batch Normalization    Batch normalization with 64 cha
 42   'ReLU14'                 ReLU                   ReLU
 43   'Conv15'                 Convolution            64 3x3x64 convolutions with str
 44   'BNorm15'                Batch Normalization    Batch normalization with 64 cha
 45   'ReLU15'                 ReLU                   ReLU
 46   'Conv16'                 Convolution            64 3x3x64 convolutions with str
 47   'BNorm16'                Batch Normalization    Batch normalization with 64 cha
 48   'ReLU16'                 ReLU                   ReLU
 49   'Conv17'                 Convolution            64 3x3x64 convolutions with str
 50   'BNorm17'                Batch Normalization    Batch normalization with 64 cha
 51   'ReLU17'                 ReLU                   ReLU
 52   'Conv18'                 Convolution            64 3x3x64 convolutions with str
 53   'BNorm18'                Batch Normalization    Batch normalization with 64 cha
 54   'ReLU18'                 ReLU                   ReLU
 55   'Conv19'                 Convolution            64 3x3x64 convolutions with str
 56   'BNorm19'                Batch Normalization    Batch normalization with 64 cha
 57   'ReLU19'                 ReLU                   ReLU
 58   'Conv20'                 Convolution            1 3x3x64 convolutions with stri
 59   'FinalRegressionLayer'   Regression Output      mean-squared-error
```

Select Training Options

Train the network using stochastic gradient descent with momentum (SGDM) optimization. Specify the hyperparameter settings for SDGM by using the `trainingOptions` function.

Training a deep network is time-consuming. Accelerate the training by specifying a high learning rate. However, this can cause the gradients of the network to explode or grow uncontrollably, preventing the network from training successfully. To keep the gradients in a meaningful range, enable gradient clipping by setting `'GradientThreshold'` to a value of 0.01, and specify `'GradientThresholdMethod'` to use the absolute value of the gradients.

```
maxEpochs = 30;
initLearningRate = 0.1;
l2reg = 0.0001;
batchSize = 128;

options = trainingOptions('sgdm',...
    'Momentum',0.9,...
    'InitialLearnRate',initLearningRate,...
    'LearnRateSchedule','piecewise',...
    'GradientThresholdMethod','absolute-value',...
    'GradientThreshold',0.005,...
    'L2Regularization',l2reg,...
    'MiniBatchSize',batchSize,...
    'MaxEpochs',maxEpochs,...
    'Plots','training-progress');
```

Train the Network

After configuring the training options and the mini-batch datastore, train the DnCNN network using the `trainNetwork` function. To train the network, set the `doTraining` parameter in the following code to `true`. A CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

If you keep the `doTraining` parameter in the following code as `false`, then the example returns a pretrained DnCNN network.

Note: Training takes about 40 hours on an NVIDIA(TM) Titan X and can take even longer depending on your GPU hardware.

```
% Training runs when doTraining is true
doTraining = false;
if doTraining
    [net, info] = trainNetwork(ds,layers,options);
else
    load('pretrainedJPEGDnCNN.mat');
end
```

You can now use the DnCNN network to remove JPEG compression artifacts from new images.

Perform JPEG Deblocking Using DnCNN Network

To perform JPEG deblocking using DnCNN, follow the remaining steps of this example. The remainder of the example shows how to:

- Create sample test images with JPEG compression artifacts at three different quality levels.
- Remove the compression artifacts using the DnCNN network.
- Visually compare the images before and after deblocking.
- Evaluate the quality of the compressed and deblocked images by quantifying their similarity to the undistorted reference image.

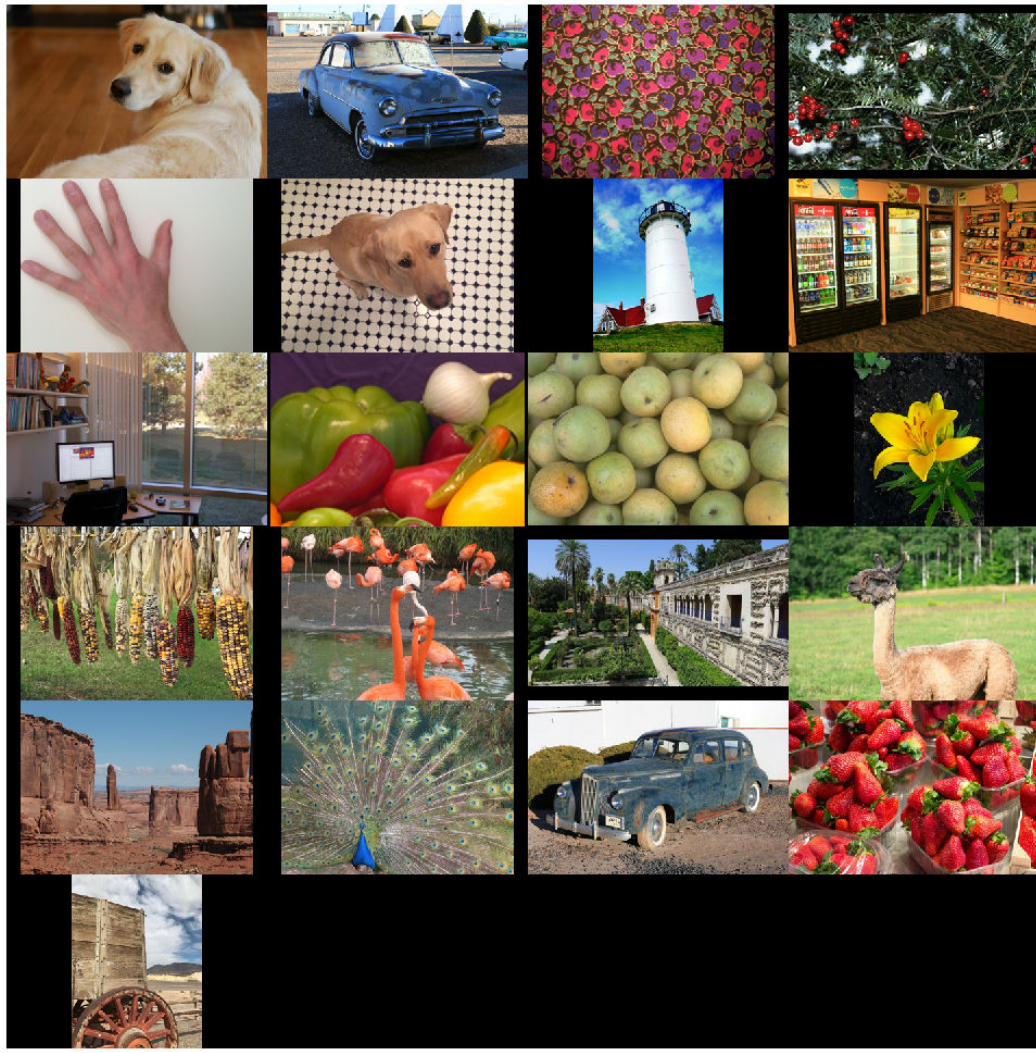
Create Sample Images with Blocking Artifacts

Create sample images to evaluate the result of JPEG image deblocking using the DnCNN network. The test data set, `testImages`, contains 21 undistorted images shipped in Image Processing Toolbox™. Load the images into an `imageDatastore`.

```
exts = {'.jpg','.png'};
fileNames = {'sherlock.jpg','car2.jpg','fabric.png','greens.jpg','hands1.jpg','kobi.png
    'lighthouse.png','micromarket.jpg','office_4.jpg','onion.png','pears.png','yellowli
    'indiancorn.jpg','flamingos.jpg','sevilla.jpg','llama.jpg','parkavenue.jpg',...
    'peacock.jpg','car1.jpg','strawberries.jpg','wagon.jpg'};
filePath = [fullfile(matlabroot,'toolbox','images','imdata') filesep];
filePathNames = strcat(filePath,fileNames);
testImages = imageDatastore(filePathNames,'FileExtensions',exts);
```

Display the testing images as a montage.

```
montage(testImages)
```



Select one of the images to use as the reference image for JPEG deblocking. You can optionally use your own uncompressed image as the reference image.

```
indx = 7; % Index of image to read from the test image datastore
Ireference = readimage(testImages,indx);
imshow(Ireference)
title('Uncompressed Reference Image')
```

Uncompressed Reference Image

Create three compressed test images with the JPEG `Quality` values of 10, 20, and 50.

```
imwrite(Ireference,fullfile(tempdir,'testQuality10.jpg'),'Quality',10);
imwrite(Ireference,fullfile(tempdir,'testQuality20.jpg'),'Quality',20);
imwrite(Ireference,fullfile(tempdir,'testQuality50.jpg'),'Quality',50);
```

Preprocess Compressed Images

Read the compressed versions of the image into the workspace.

```
I10 = imread(fullfile(tempdir,'testQuality10.jpg'));
I20 = imread(fullfile(tempdir,'testQuality20.jpg'));
I50 = imread(fullfile(tempdir,'testQuality50.jpg'));
```

Display the compressed images as a montage.

```
montage({I50,I20,I10},'Size',[1 3])
title('JPEG-Compressed Images with Quality Factor: 50, 20 and 10 (left to right)')
```

Recall that DnCNN is trained using only the luminance channel of an image because human perception is more sensitive to changes in brightness than changes in color. Convert the JPEG-compressed images from the RGB color space to the YCbCr color space using the `rgb2ycbcr` function.

```
I10ycbcr = rgb2ycbcr(I10);
I20ycbcr = rgb2ycbcr(I20);
I50ycbcr = rgb2ycbcr(I50);
```

Apply the DnCNN Network

In order to perform the forward pass of the network, use the `denoiseImage` function. This function uses exactly the same training and testing procedures for denoising an image. You can think of the JPEG compression artifacts as a type of image noise.

```
I10y_predicted = denoiseImage(I10ycbcr(:,:,1),net);
I20y_predicted = denoiseImage(I20ycbcr(:,:,1),net);
I50y_predicted = denoiseImage(I50ycbcr(:,:,1),net);
```

The chrominance channels do not need processing. Concatenate the deblocked luminance channel with the original chrominance channels to obtain the deblocked image in the YCbCr color space.

```
I10ycbcr_predicted = cat(3,I10y_predicted,I10ycbcr(:,:,2:3));
I20ycbcr_predicted = cat(3,I20y_predicted,I20ycbcr(:,:,2:3));
I50ycbcr_predicted = cat(3,I50y_predicted,I50ycbcr(:,:,2:3));
```

Convert the deblocked YCbCr image to the RGB color space by using the `ycbcr2rgb` function.

```
I10_predicted = ycbcr2rgb(I10ycbcr_predicted);
I20_predicted = ycbcr2rgb(I20ycbcr_predicted);
I50_predicted = ycbcr2rgb(I50ycbcr_predicted);
```

Display the deblocked images as a montage.

```
montage({I50_predicted,I20_predicted,I10_predicted},'Size',[1 3])
title('Deblocked Images with Quality Factor: 50, 20 and 10 (left to right)')
```

To get a better visual understanding of the improvements, examine a smaller region inside each image. Specify a region of interest (ROI) using vector `roi` in the format [*x y width height*]. The elements define the x- and y-coordinate of the top left corner, and the width and height of the ROI.

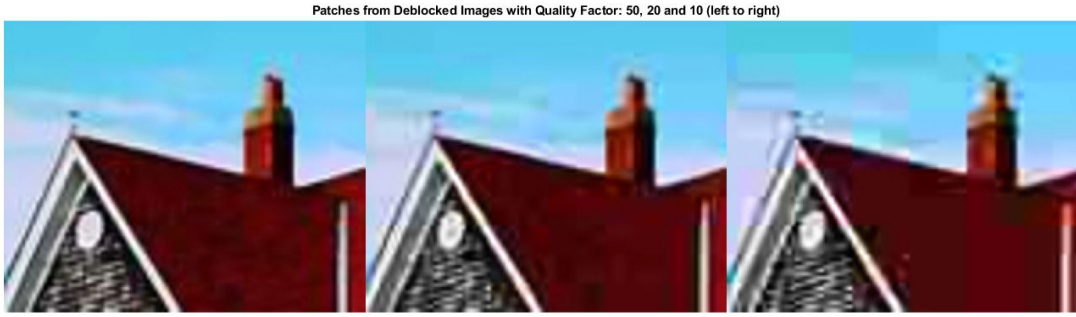
```
roi = [30 440 100 80];
```

Crop the compressed images to this ROI, and display the result as a montage.

```
i10 = imcrop(I10,roi);
i20 = imcrop(I20,roi);
i50 = imcrop(I50,roi);
montage({i50 i20 i10},'Size',[1 3])
title('Patches from JPEG-Compressed Images with Quality Factor: 50, 20 and 10 (left to
```

Crop the deblocked images to this ROI, and display the result as a montage.

```
i10predicted = imcrop(I10_predicted,roi);
i20predicted = imcrop(I20_predicted,roi);
i50predicted = imcrop(I50_predicted,roi);
montage({i50predicted,i20predicted,i10predicted},'Size',[1 3])
title('Patches from Deblocked Images with Quality Factor: 50, 20 and 10 (left to right)
```



Quantitative Comparison

Quantify the quality of the deblocked images through four metrics. You can use the `displayJPEGResults` helper function to compute these metrics for compressed and deblocked images at the quality factors 10, 20, and 50.

- Structural Similarity Index (SSIM). SSIM assesses the visual impact of three characteristics of an image: luminance, contrast and structure, against a reference

image. The closer the SSIM value is to 1, the better the test image agrees with the reference image. Here, the reference image is the undistorted original image, `Ireference`, before JPEG compression. See `ssim` for more information about this metric.

- Peak signal-to-noise ratio (PSNR). The larger the PNSR value, the stronger the signal compared to the distortion. See `psnr` for more information about this metric.
- Naturalness Image Quality Evaluator (NIQE). NIQE measures perceptual image quality using a model trained from natural scenes. Smaller NIQE scores indicate better perceptual quality. See `niqe` for more information about this metric.
- Blind/Referenceless Image Spatial Quality Evaluator (BRISQUE). BRISQUE measures perceptual image quality using a model trained from natural scenes with image distortion. Smaller BRISQUE scores indicate better perceptual quality. See `brisque` for more information about this metric.

```
displayJPEGResults(Ireference,I10,I20,I50,I10_predicted,I20_predicted,I50_predicted)
```

```
-----------------------------------------
SSIM Comparison
================
I10: 0.90624    I10_predicted: 0.91286
I20: 0.94904    I20_predicted: 0.95444
I50: 0.97238    I50_predicted: 0.97482
-----------------------------------------
PSNR Comparison
================
I10: 26.6046    I10_predicted: 27.0793
I20: 28.8015    I20_predicted: 29.3378
I50: 31.4512    I50_predicted: 31.8584
-----------------------------------------
NIQE Comparison
================
I10: 7.2194    I10_predicted: 3.9473
I20: 4.5158    I20_predicted: 3.0685
I50: 2.8874    I50_predicted: 2.4101
NOTE: Smaller NIQE score signifies better perceptual quality
-----------------------------------------
BRISQUE Comparison
===================
I10: 52.372    I10_predicted: 38.9271
I20: 45.3772    I20_predicted: 30.8994
I50: 27.7093    I50_predicted: 24.3846
NOTE: Smaller BRISQUE score signifies better perceptual quality
```

Summary

This example shows how to create and train a DnCNN network, then use the network to reduce JPEG compression artifacts in images. These were the steps to train the network:

- Download the training data.
- Create training images by writing pristine images in the JPEG file format with various levels of compression.
- Define a custom mini-batch datastore, called a `JPEGimagePatchDatastore`, to extract patches from the input compressed image and compute the target residuals from the corresponding patches in the pristine images. This datastore was used to feed training data to the network.
- Create the layers of the DnCNN network by using the `dnCNNLayers` function.
- Specify training options.
- Train the network using the `trainNetwork` function.

After training the DnCNN network or loading a pretrained DnCNN network, the example compresses a test image at three quality values, then uses the network to remove the compression artifacts.

References

[1] Zhang, K., W. Zuo, Y. Chen, D. Meng, and L. Zhang, "Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising." *IEEE® Transactions on Image Processing*. Feb 2017.

[2] Grubinger, M., P. Clough, H. Müller, and T. Deselaers. "The IAPR TC-12 Benchmark: A New Evaluation Resource for Visual Information Systems." *Proceedings of the OntoImage 2006 Language Resources For Content-Based Image Retrieval*. Genoa, Italy. Vol. 5, May 2006, p. 10.

See Also

`denoiseImage` | `dnCNNLayers` | `rgb2ycbcr` | `trainNetwork` | `trainingOptions` | `ycbcr2rgb`

More About

- “Develop Custom Mini-Batch Datastore” (Deep Learning Toolbox)

Semantic Segmentation of Multispectral Images Using Deep Learning

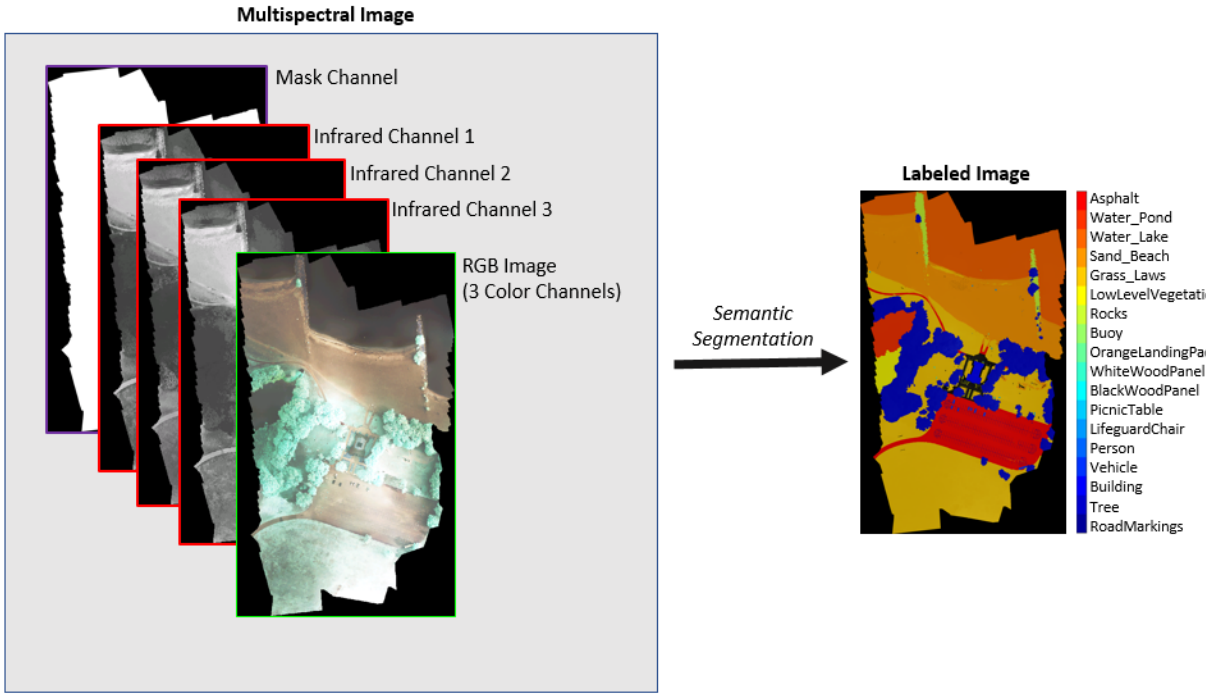
This example shows how to train a U-Net convolutional neural network to perform semantic segmentation of a multispectral image with seven channels: three color channels, three near-infrared channels, and a mask.

The example shows how to train a U-Net network and also provides a pretrained U-Net network. If you choose to train the U-Net network, use of a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended (requires Parallel Computing Toolbox™).

Introduction

Semantic segmentation involves labeling each pixel in an image with a class. One application of semantic segmentation is tracking deforestation, which is the change in forest cover over time. Environmental agencies track deforestation to assess and quantify the environmental and ecological health of a region.

Deep-learning-based semantic segmentation can yield a precise measurement of vegetation cover from high-resolution aerial photographs. One challenge is differentiating classes with similar visual characteristics, such as trying to classify a green pixel as grass, shrubbery, or tree. To increase classification accuracy, some data sets contain multispectral images that provide additional information about each pixel. For example, the Hamlin Beach State Park data set supplements the color images with near-infrared channels that provide a clearer separation of the classes.



This example shows how to use deep-learning-based semantic segmentation techniques to calculate the percentage vegetation cover in a region from a set of multispectral images.

Download Data

This example uses a high-resolution multispectral data set to train the network [1]. The image set was captured using a drone over the Hamlin Beach State Park, NY. The data contains labeled training, validation, and test sets, with 18 object class labels.

Download the MAT-file version of the data set. You can use the helper function, `downloadHamlinBeachMSIData`, to download the data. The size of the data file is ~3.0 GB.

```
imageDir = tempdir;
url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';
downloadHamlinBeachMSIData(url,imageDir);
```

```
Downloading Hamlin Beach dataset...
This will take several minutes to download...
done.
```

In addition, download a pretrained version of U-Net for this dataset. The pretrained model enables you to run the entire example without having to wait for training to complete.

```
trainedUnet_url = 'https://www.mathworks.com/supportfiles/vision/data/multispectralUnet
downloadTrainedUnet(trainedUnet_url,imageDir);
```

```
Downloading Pre-trained U-net for Hamlin Beach dataset...
This will take several minutes to download...
done.
```

Inspect Training Data

Load the data set into the MATLAB® workspace.

```
load(fullfile(imageDir,'rit18_data','rit18_data.mat'));
```

Examine the structure of the data.

```
whos train_data val_data test_data
```

```
  Name            Size                          Bytes  Class     Attributes

  test_data       7x12446x7654             1333663576  uint16
  train_data      7x9393x5642               741934284  uint16
  val_data        7x8833x6918               855493716  uint16
```

The multispectral image data is arranged as *numChannels*-by-*width*-by-*height* arrays. However, in MATLAB, multichannel images are arranged as *width*-by-*height*-by-*numChannels* arrays. To reshape the data so that the channels are in the third dimension, use the helper function, `switchChannelsToThirdPlane`.

```
train_data = switchChannelsToThirdPlane(train_data);
val_data   = switchChannelsToThirdPlane(val_data);
test_data  = switchChannelsToThirdPlane(test_data);
```

Confirm that the data has the correct structure.

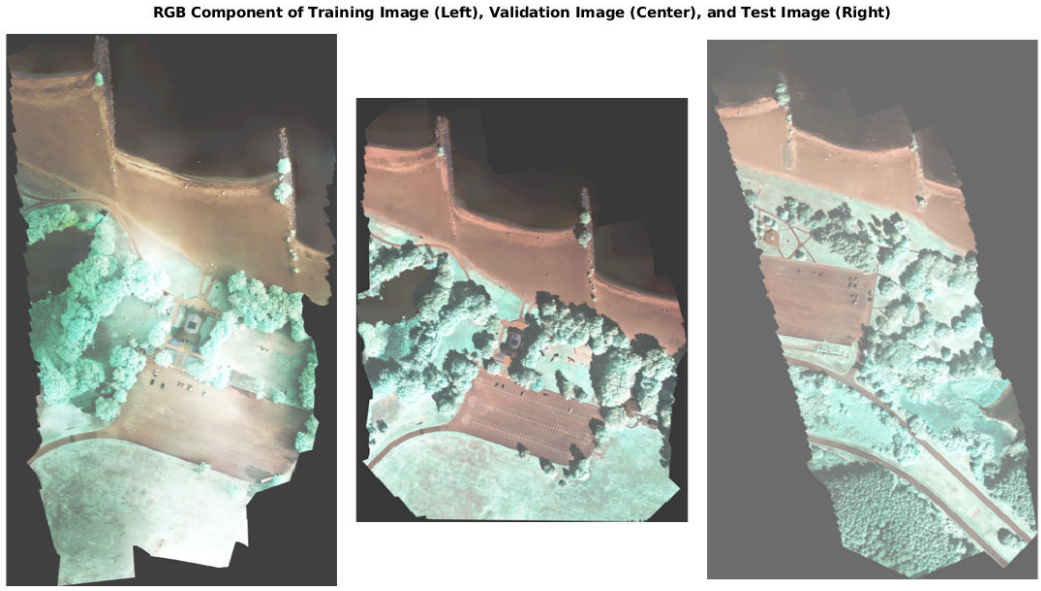
```
whos train_data val_data test_data
```

```
  Name                Size                      Bytes  Class     Attributes
```

```
  test_data        12446x7654x7            1333663576  uint16
  train_data        9393x5642x7             741934284  uint16
  val_data          8833x6918x7             855493716  uint16
```

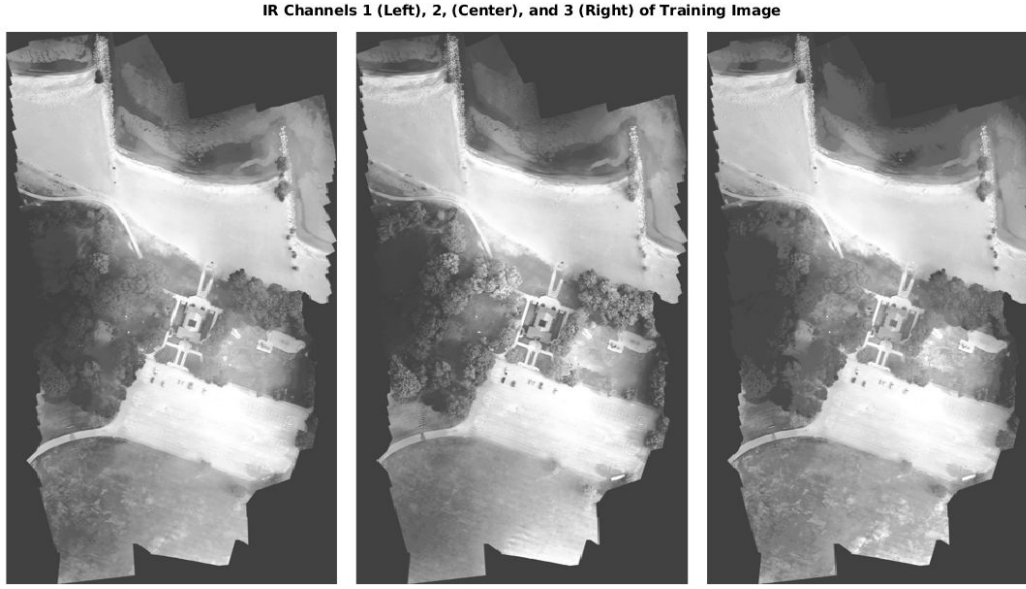
The RGB color channels are the 4th, 5th, and 6th image channels. Display the color component of the training, validation, and test images as a montage. To make the images appear brighter on the screen, equalize their histograms by using the `histeq` function.

```
figure
montage(...
    {histeq(train_data(:,:,4:6)), ...
    histeq(val_data(:,:,4:6)), ...
    histeq(test_data(:,:,4:6))}, ...
    'BorderSize',10,'BackgroundColor','white')
title('RGB Component of Training Image (Left), Validation Image (Center), and Test Imag
```



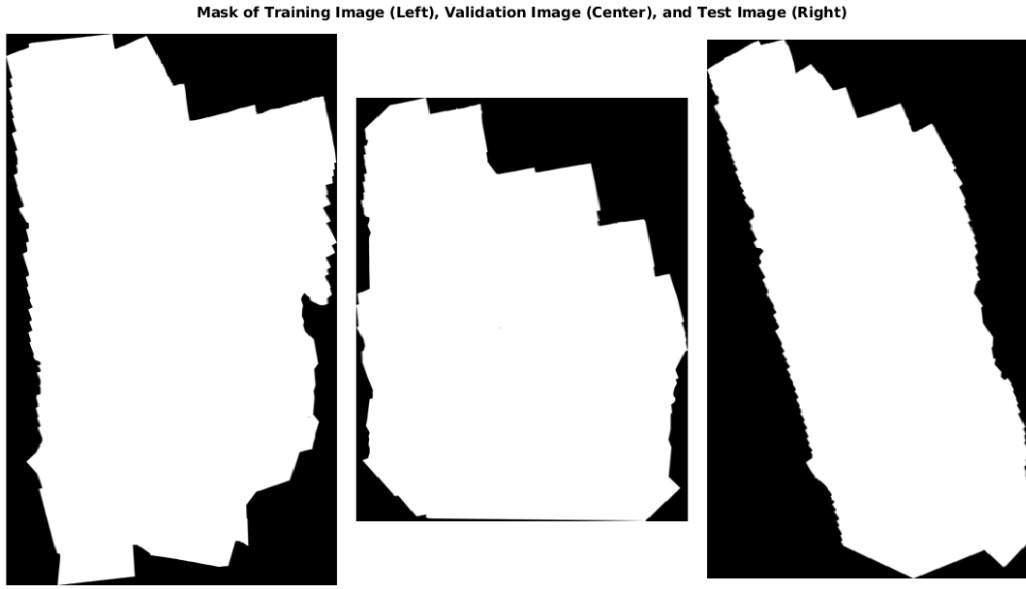
Display the first three histogram-equalized channels of the training data as a montage. These channels correspond to the near-infrared bands and highlight different components of the image based on their heat signatures. For example, the trees near the center of the 2nd channel image show more detail than the trees in the other two channels.

```
figure
montage(...
    {histeq(train_data(:,:,1)), ...
    histeq(train_data(:,:,2)), ...
    histeq(train_data(:,:,3))}, ...
    'BorderSize',10,'BackgroundColor','white')
title('IR Channels 1 (Left), 2, (Center), and 3 (Right) of Training Image')
```



Channel 7 is a mask that indicates the valid segmentation region. Display the mask for the training, validation, and test images.

```
figure
montage(...
    {train_data(:,:,7), ...
    val_data(:,:,7), ...
    test_data(:,:,7)}, ...
    'BorderSize',10,'BackgroundColor','white')
title('Mask of Training Image (Left), Validation Image (Center), and Test Image (Right
```



The labeled images contain the ground truth data for the segmentation, with each pixel assigned to one of the 18 classes. Get a list of the classes with their corresponding IDs.

```
disp(classes)
```

```
0. Other Class/Image Border
1. Road Markings
2. Tree
3. Building
4. Vehicle (Car, Truck, or Bus)
5. Person
6. Lifeguard Chair
7. Picnic Table
8. Black Wood Panel
9. White Wood Panel
10. Orange Landing Pad
11. Water Buoy
12. Rocks
13. Other Vegetation
14. Grass
15. Sand
```

```
16. Water (Lake)
17. Water (Pond)
18. Asphalt (Parking Lot/Walkway)
```

Create a vector of class names.

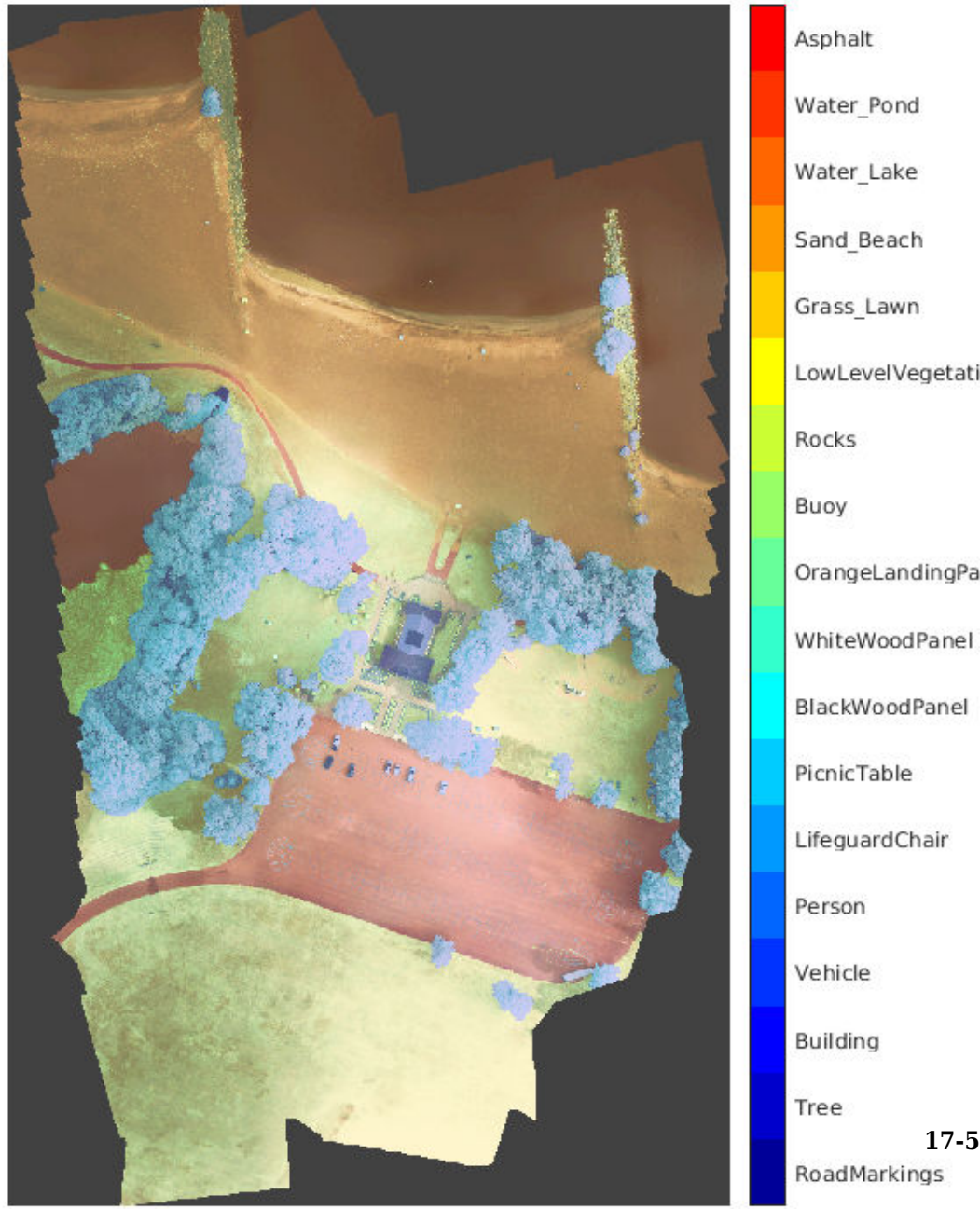
```
classNames = [ "RoadMarkings","Tree","Building","Vehicle","Person", ...
               "LifeguardChair","PicnicTable","BlackWoodPanel",...
               "WhiteWoodPanel","OrangeLandingPad","Buoy","Rocks",...
               "LowLevelVegetation","Grass_Lawn","Sand_Beach",...
               "Water_Lake","Water_Pond","Asphalt"];
```

Overlay the labels on the histogram-equalized RGB training image. Add a colorbar to the image.

```
cmap = jet(numel(classNames));
B = labeloverlay(histeq(train_data(:,:,4:6)),train_labels,'Transparency',0.8,'Colormap

figure
title('Training Labels')
imshow(B)
```

```
Warning: Image is too big to fit on screen; displaying at 8%
```

```
N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterp
colormap(cmap)
```


Asphalt
Water_Pond
Water_Lake
Sand_Beach
Grass_Lawn
LowLevelVegetatio
Rocks
Buoy
OrangeLandingPad
WhiteWoodPanel
BlackWoodPanel
PicnicTable
LifeguardChair
Person
Vehicle
Building
Tree
RoadMarkings

Save the training data as a .MAT file and the training labels as a .PNG file.

```
save('train_data.mat','train_data');
imwrite(train_labels,'train_labels.png');
```

Define Mini-Batch Datastore for Training

A mini-batch datastore is used to feed the training data to the network. This example defines a custom implementation of a mini-bach datastore, called `PixelLabelImagePatchDatastore`, as a convenient way to generate augmented image and label patches for training the semantic segmentation network.

Begin by storing the training images from `'train_data.mat'` in an `imageDatastore`. Because the MAT file format is a nonstandard image format, you must use a MAT file reader to enable reading the image data. You can use the helper MAT file reader, `matReader`, that extracts the first six channels from the training data and omits the last channel containing the mask.

```
imds = imageDatastore('train_data.mat','FileExtensions','.mat','ReadFcn',@matReader);
```

Create a `pixelLabelDatastore` to store the label patches containing the 18 labeled regions.

```
pixelLabelIds = 1:18;
pxds = pixelLabelDatastore('train_labels.png',classNames,pixelLabelIds);
```

Create the `PixelLabelImagePatchDatastore` from the image datastore and the pixel label datastore. The `PixelLabelImagePatchDatastore` extracts pairs of patches from the training image and labels. It also augments the training data to increase the size of the training data set and to make the training more robust. To mimic aerial landscape data collected by a drone, the `PixelLabelImagePatchDatastore` augments the pairs of patches by adding random rotation, reflection, and scaling.

Each mini-batch contains 16 patches of size 256-by-256 pixels. One thousand mini-batches are extracted at each iteration of the epoch. All patches are extracted from random positions in the image.

```
ds = PixelLabelImagePatchDatastore(imds,pxds,'PatchSize',256,...
                                             'MiniBatchSize',16,...
                                             'BatchesPerImage',1000)
```

```
ds 
  PixelLabelImagePatchDatastore with properties:
```

```
      MiniBatchSize: 16
    NumObservations: 16000
```

To understand what gets passed into the network during training, read the first mini-batch from the `PixelLabelImagePatchDatastore`.

```
sampleBatch = read(ds)
```

```
sampleBatch=16×2 table
        imPatches              labelPatches
    ________________    ____________________

    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
    [256×256×6 uint16]    [256×256 categorical]
```

Reset the datastore to its original state after reading the sample batch.

```
reset(ds)
```

Create U-Net Network Layers

This example uses a variation of the U-Net network. In U-Net, the initial series of convolutional layers are interspersed with max pooling layers, successively decreasing the resolution of the input image. These layers are followed by a series of convolutional layers interspersed with upsampling operators, successively increasing the resolution of the input image [2]. The name U-Net comes from the fact that the network can be drawn with a symmetric shape like the letter U.

This example modifies the U-Net to use zero-padding in the convolutions, so that the input and the output to the convolutions have the same size. Use the helper function, `createUnet`, to create a U-Net with a few preselected hyperparameters.

```
inputTileSize = [256,256,6];
lgraph = createUnet(inputTileSize);
disp(lgraph.Layers)
```

```
  58x1 Layer array with layers:

     1   'ImageInputLayer'                        Image Input                 256x256x
     2   'Encoder-Section-1-Conv-1'               Convolution                 64 3x3x6
     3   'Encoder-Section-1-ReLU-1'               ReLU                        ReLU
     4   'Encoder-Section-1-Conv-2'               Convolution                 64 3x3x6
     5   'Encoder-Section-1-ReLU-2'               ReLU                        ReLU
     6   'Encoder-Section-1-MaxPool'              Max Pooling                 2x2 max
     7   'Encoder-Section-2-Conv-1'               Convolution                 128 3x3x
     8   'Encoder-Section-2-ReLU-1'               ReLU                        ReLU
     9   'Encoder-Section-2-Conv-2'               Convolution                 128 3x3x
    10   'Encoder-Section-2-ReLU-2'               ReLU                        ReLU
    11   'Encoder-Section-2-MaxPool'              Max Pooling                 2x2 max
    12   'Encoder-Section-3-Conv-1'               Convolution                 256 3x3x
    13   'Encoder-Section-3-ReLU-1'               ReLU                        ReLU
    14   'Encoder-Section-3-Conv-2'               Convolution                 256 3x3x
    15   'Encoder-Section-3-ReLU-2'               ReLU                        ReLU
    16   'Encoder-Section-3-MaxPool'              Max Pooling                 2x2 max
    17   'Encoder-Section-4-Conv-1'               Convolution                 512 3x3x
    18   'Encoder-Section-4-ReLU-1'               ReLU                        ReLU
    19   'Encoder-Section-4-Conv-2'               Convolution                 512 3x3x
    20   'Encoder-Section-4-ReLU-2'               ReLU                        ReLU
    21   'Encoder-Section-4-DropOut'              Dropout                     50% drop
    22   'Encoder-Section-4-MaxPool'              Max Pooling                 2x2 max
    23   'Mid-Conv-1'                             Convolution                 1024 3x3
    24   'Mid-ReLU-1'                             ReLU                        ReLU
    25   'Mid-Conv-2'                             Convolution                 1024 3x3
    26   'Mid-ReLU-2'                             ReLU                        ReLU
    27   'Mid-DropOut'                            Dropout                     50% drop
    28   'Decoder-Section-1-UpConv'               Transposed Convolution      512 2x2x
    29   'Decoder-Section-1-UpReLU'               ReLU                        ReLU
    30   'Decoder-Section-1-DepthConcatenation'   Depth concatenation         Depth co
    31   'Decoder-Section-1-Conv-1'               Convolution                 512 3x3x
    32   'Decoder-Section-1-ReLU-1'               ReLU                        ReLU
    33   'Decoder-Section-1-Conv-2'               Convolution                 512 3x3x
    34   'Decoder-Section-1-ReLU-2'               ReLU                        ReLU
    35   'Decoder-Section-2-UpConv'               Transposed Convolution      256 2x2x
```

```
    36   'Decoder-Section-2-UpReLU'                ReLU                          ReLU
    37   'Decoder-Section-2-DepthConcatenation'    Depth concatenation           Depth co
    38   'Decoder-Section-2-Conv-1'                Convolution                   256 3x3x
    39   'Decoder-Section-2-ReLU-1'                ReLU                          ReLU
    40   'Decoder-Section-2-Conv-2'                Convolution                   256 3x3x
    41   'Decoder-Section-2-ReLU-2'                ReLU                          ReLU
    42   'Decoder-Section-3-UpConv'                Transposed Convolution        128 2x2x
    43   'Decoder-Section-3-UpReLU'                ReLU                          ReLU
    44   'Decoder-Section-3-DepthConcatenation'    Depth concatenation           Depth co
    45   'Decoder-Section-3-Conv-1'                Convolution                   128 3x3x
    46   'Decoder-Section-3-ReLU-1'                ReLU                          ReLU
    47   'Decoder-Section-3-Conv-2'                Convolution                   128 3x3x
    48   'Decoder-Section-3-ReLU-2'                ReLU                          ReLU
    49   'Decoder-Section-4-UpConv'                Transposed Convolution        64 2x2x1
    50   'Decoder-Section-4-UpReLU'                ReLU                          ReLU
    51   'Decoder-Section-4-DepthConcatenation'    Depth concatenation           Depth co
    52   'Decoder-Section-4-Conv-1'                Convolution                   64 3x3x1
    53   'Decoder-Section-4-ReLU-1'                ReLU                          ReLU
    54   'Decoder-Section-4-Conv-2'                Convolution                   64 3x3x6
    55   'Decoder-Section-4-ReLU-2'                ReLU                          ReLU
    56   'Final-ConvolutionLayer'                  Convolution                   18 1x1x6
    57   'Softmax-Layer'                           Softmax                       softmax
    58   'Segmentation-Layer'                      Pixel Classification Layer    Cross-en
```

Select Training Options

Train the network using stochastic gradient descent with momentum (SGDM) optimization. Specify the hyperparameter settings for SDGM by using the `trainingOptions` function.

Training a deep network is time-consuming. Accelerate the training by specifying a high learning rate. However, this can cause the gradients of the network to explode or grow uncontrollably, preventing the network from training successfully. To keep the gradients in a meaningful range, enable gradient clipping by setting `'GradientThreshold'` to a value of 0.05, and specify `'GradientThresholdMethod'` to use the L2-norm of the gradients.

```
initialLearningRate = 0.05;
maxEpochs = 150;
minibatchSize = 16;
l2reg = 0.0001;

options = trainingOptions('sgdm',...
    'InitialLearnRate', initialLearningRate, ...
```

```
    'Momentum',0.9,...
    'L2Regularization',l2reg,...
    'MaxEpochs',maxEpochs,...
    'MiniBatchSize',minibatchSize,...
    'VerboseFrequency',20,...
    'LearnRateSchedule','piecewise',...
    'Shuffle','every-epoch',...
    'Plots','training-progress',...
    'GradientThresholdMethod','l2norm',...
    'GradientThreshold',0.05);
```

Train the Network

After configuring the training options and the mini-batch datastore, train the U-Net network by using the `trainNetwork` function. To train the network, set the `doTraining` parameter in the following code to `true`. A CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

If you keep the `doTraining` parameter in the following code as `false`, then the example returns a pretrained U-Net network.

Note: Training takes about 20 hours on an NVIDIA(TM) Titan X and can take even longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    [net,info] = trainNetwork(ds,lgraph,options);
else
    load(fullfile(imageDir,'trainedUnet','multispectralUnet.mat'));
end
```

You can now use the U-Net to semantically segment the multispectral image.

Predict Results on Test Data

To perform the forward pass on the trained network, use the helper function, `segmentImage`, with the validation data set. `segmentImage` performs segmentation on image patches using the `semanticseg` function.

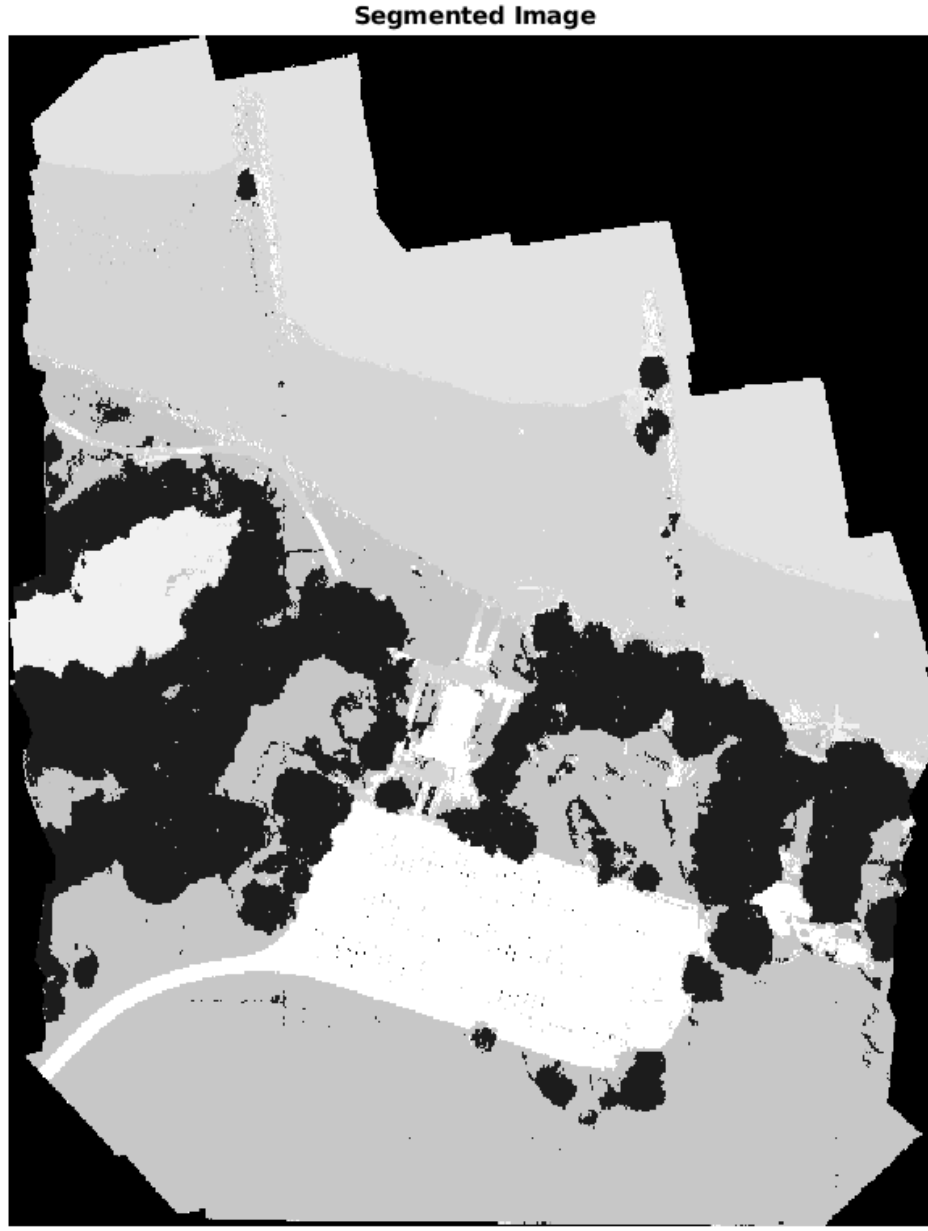
```
predictPatchSize = [1024 1024];
segmentedImage = segmentImage(val_data,net,predictPatchSize);
```

To extract only the valid portion of the segmentation, multiply the segmented image by the mask channel of the validation data.

```
segmentedImage = uint8(val_data(:,:,7)~=0) .* segmentedImage;

figure
imshow(segmentedImage,[])
```

```
Warning: Image is too big to fit on screen; displaying at 8%
```

```
title('Segmented Image')
```

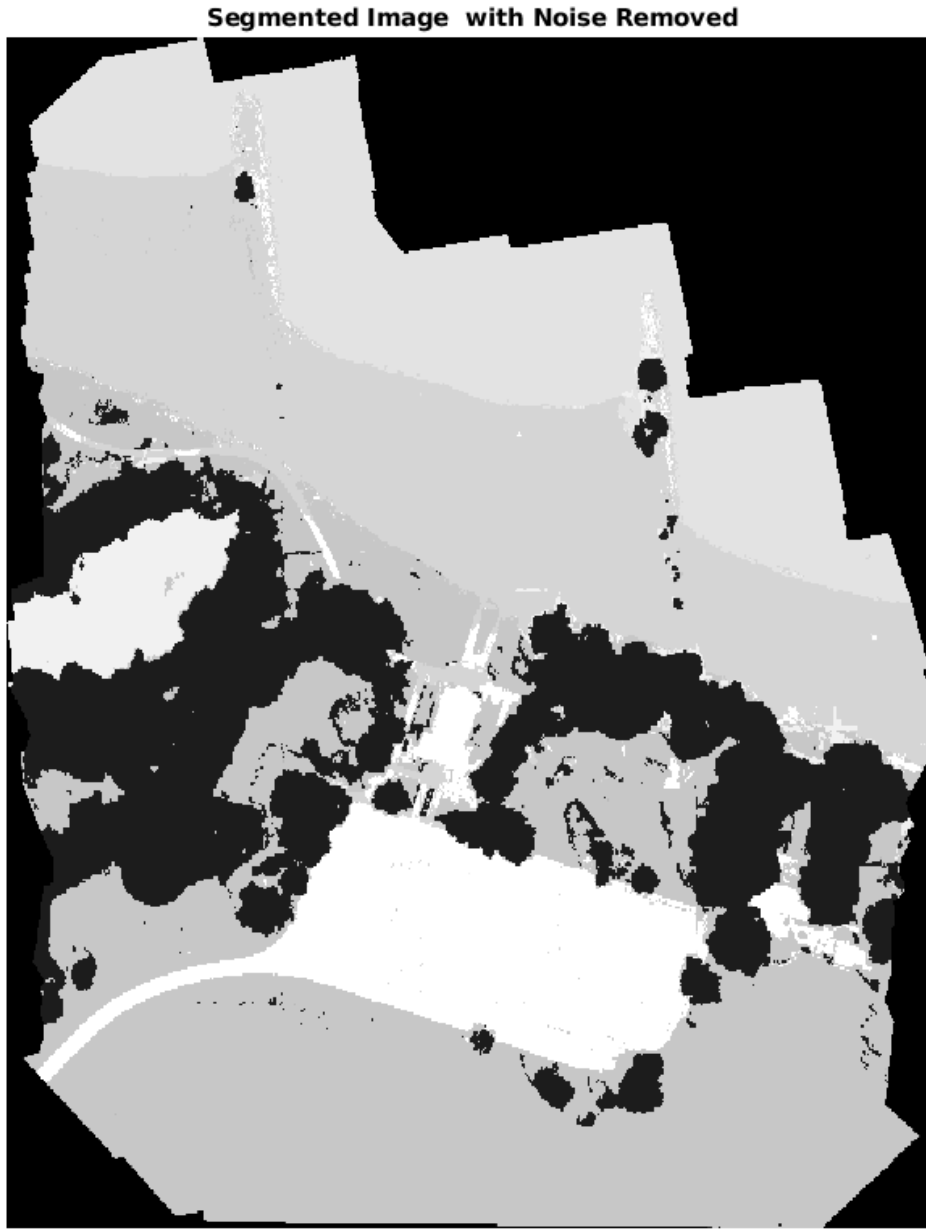

Segmented Image

The output of semantic segmentation is noisy. Perform post image processing to remove noise and stray pixels. Use the `medfilt2` function to remove salt-and-pepper noise from the segmentation. Visualize the segmented image with the noise removed.

```
segmentedImage = medfilt2(segmentedImage,[7,7]);
imshow(segmentedImage,[]);
```

```
Warning: Image is too big to fit on screen; displaying at 8%
```

```
title('Segmented Image  with Noise Removed')
```

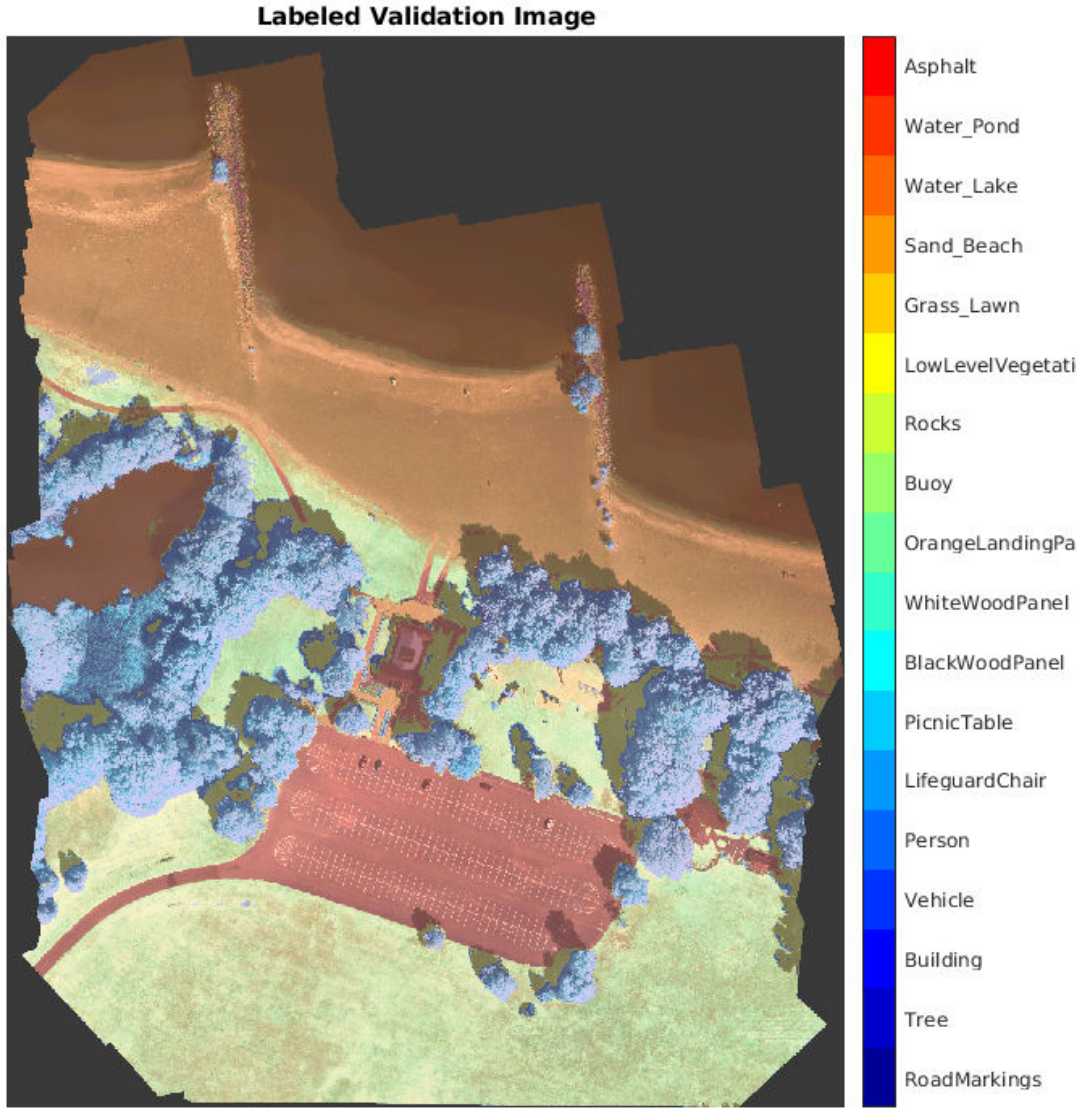

Segmented Image with Noise Removed

Overlay the segmented image on the histogram-equalized RGB validation image.

```
B = labeloverlay(histeq(val_data(:,:,4:6)),segmentedImage,'Transparency',0.8,'Colormap

figure
imshow(B)
```

```
Warning: Image is too big to fit on screen; displaying at 8%
```

```
title('Labeled Validation Image')
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterp
colormap(cmap)
```


Labeled Validation Image
Asphalt
Water_Pond
Water_Lake
Sand_Beach
Grass_Lawn
LowLevelVegetatio
Rocks
Buoy
OrangeLandingPad
WhiteWoodPanel
BlackWoodPanel
PicnicTable
LifeguardChair
Person
Vehicle
Building
Tree
RoadMarkings

Save the segmented image and ground truth labels as .PNG files. These will be used to compute accuracy metrics.

```
imwrite(segmentedImage,'results.png');
imwrite(val_labels,'gtruth.png');
```

Quantify Segmentation Accuracy

Create a `pixelLabelDatastore` for the segmentation results and the ground truth labels.

```
pxdsResults = pixelLabelDatastore('results.png',classNames,pixelLabelIds);
pxdsTruth = pixelLabelDatastore('gtruth.png',classNames,pixelLabelIds);
```

Measure the global accuracy of the semantic segmentation by using the `evaluateSemanticSegmentation` function.

```
ssm = evaluateSemanticSegmentation(pxdsResults,pxdsTruth,'Metrics','global-accuracy');
```

```
Evaluating semantic segmentation results
----------------------------------------
* Selected metrics: global accuracy.
* Processing 1 images...
[==================================================] 100%
Elapsed time: 00:00:28
Estimated time remaining: 00:00:00
* Finalizing... Done.
* Data set metrics:

    GlobalAccuracy
    ______________

       0.90698
```

The global accuracy score indicates that just over 90% of the pixels are classified correctly.

Calculate Extent of Vegetation Cover

The final goal of this example is to calculate the extent of vegetation cover in the multispectral image.

Find the number of pixels labeled vegetation. The label IDs 2 ("Trees"), 13 ("LowLevelVegetation"), and 14 ("Grass_Lawn") are the vegetation classes. Also find the total number of valid pixels by summing the pixels in the ROI of the mask image.

```
vegetationClassIds = uint8([2,13,14]);
vegetationPixels = ismember(segmentedImage(:),vegetationClassIds);
validPixels = (segmentedImage~=0);

numVegetationPixels = sum(vegetationPixels(:));
numValidPixels = sum(validPixels(:));
```

Calculate the percentage of vegetation cover by dividing the number of vegetation pixels by the number of valid pixels.

```
percentVegetationCover = (numVegetationPixels/numValidPixels)*100;
fprintf('The percentage of vegetation cover is %3.2f%%.',percentVegetationCover);
```

```
The percentage of vegetation cover is 51.72%.
```

Summary

This example shows how to create and train a U-Net network for semantic segmentation of a seven-channel multispectral image. These were the steps to train the network:

- Download and reshape the training data.
- Define a custom mini-batch datastore, called a `PixelLabelImagePatchDatastore` from the image datastore and the pixel label datastore. The `PixelLabelImagePatchDatastore` extracts patches from the input compressed image and compute the target residuals from the corresponding patches in the pristine images. This datastore was used to feed training data to the network.
- Define the layers of the U-Net network.
- Specify training options.
- Train the network using the `trainNetwork` function.

After training the U-Net network or loading a pretrained U-Net network, the example performs semantic segmentation of the validation data and measures the segmentation accuracy.

References

[1] Kemker, R., C. Salvaggio, and C. Kanan. "High-Resolution Multispectral Dataset for Semantic Segmentation." CoRR, abs/1703.01918. 2017.

[2] Ronneberger, O., P. Fischer, and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." CoRR, abs/1505.04597. 2015.

See Also

`evaluateSemanticSegmentation` | `histeq` | `imageDatastore` | `pixelLabelDatastore` | `pixelLabelImageDatastore` | `semanticseg` | `trainNetwork` | `trainingOptions` | `unetLayers`

More About

- “Semantic Segmentation Basics” (Computer Vision System Toolbox)
- “Semantic Segmentation Using Deep Learning” (Computer Vision System Toolbox)
- “Develop Custom Mini-Batch Datastore” (Deep Learning Toolbox)

External Websites

- https://github.com/rmkemker/RIT-18

Image Processing Operator Approximation Using Deep Learning

This example shows how to train a multiscale context aggregation network (CAN) that is used to approximate an image filtering operation.

The example shows how to train a multiscale CAN network to approximate a bilateral filter, and also provides a pretrained network. If you choose to train the network, use of a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended. Use of a GPU requires the Parallel Computing Toolbox™.

If you do not want to download the training data set or train the network, then you can load the pretrained CAN by typing `load('trainedOperatorLearning-Epoch-181.mat');` at the command line. Then, go directly to the *Perform Bilateral Filtering Approximation Using Multiscale CAN* section in this example.

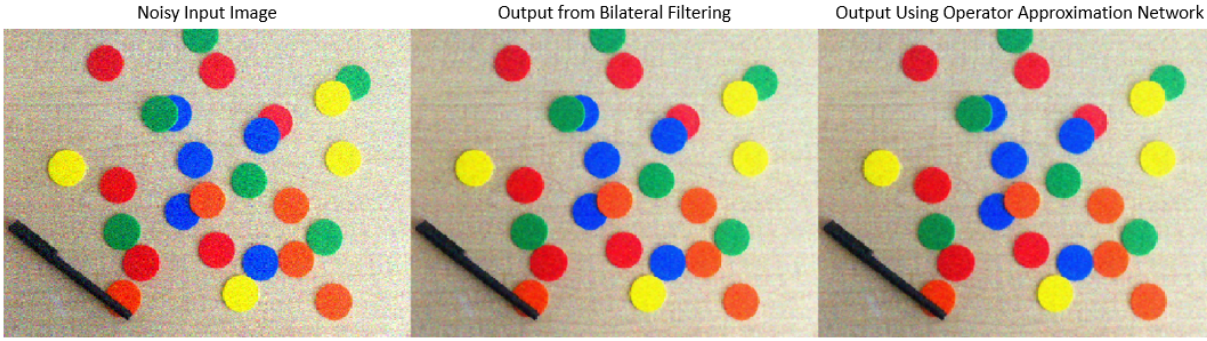
Introduction

Operator approximation finds alternative ways to process images such that the result resembles the output from a conventional image processing operation or pipeline. The goal of operator approximation is often to reduce the time required to process an image.

Several classical and deep learning techniques have been proposed to perform operator approximation. Some classical techniques improve the efficiency of a single algorithm but cannot be generalized to other operations. Another common technique approximates a wide range of operations by applying the operator to a low resolution copy of an image, but the loss of high-frequency content limits the accuracy of the approximation.

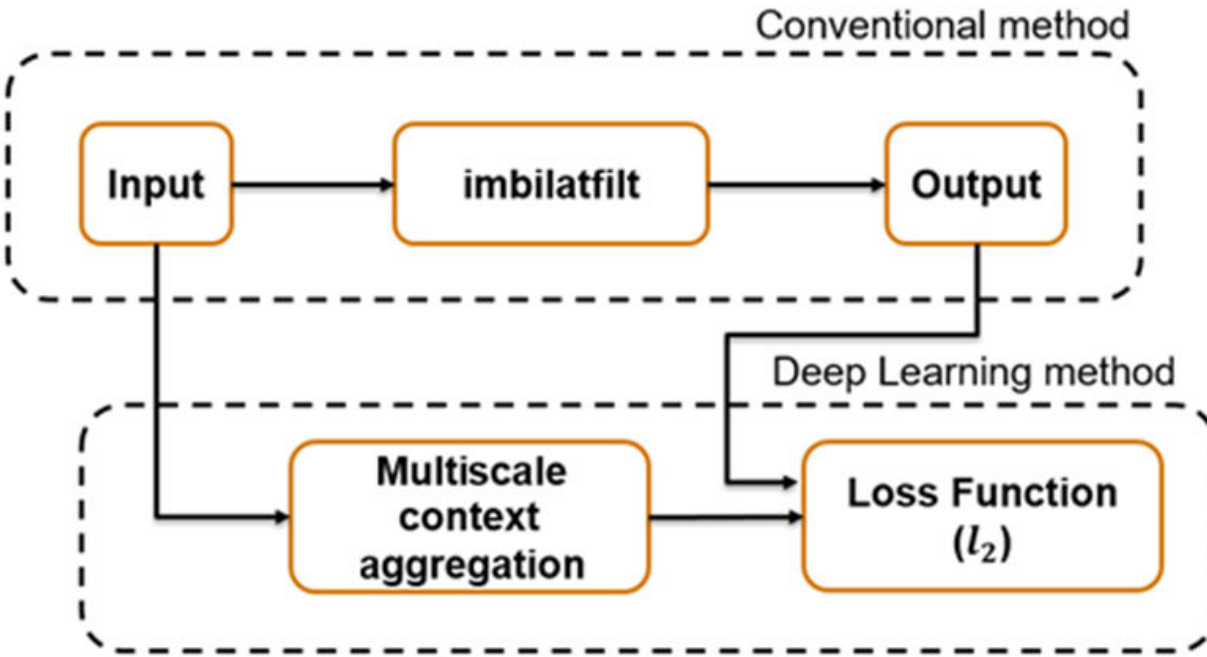
Deep learning solutions enable the approximation of more general and complex operations. For example, the multiscale context aggregation network (CAN) presented by Q. Chen [1] can approximate multiscale tone mapping, photographic style transfer, nonlocal dehazing, and pencil drawing. Multiscale CAN trains on full-resolution images for greater accuracy in processing high-frequency details. After the network is trained, the network can bypass the conventional processing operation and process images directly.

This example explores how to train a multiscale CAN to approximate a bilateral image filtering operation, which reduces image noise while preserving edge sharpness. The example presents the complete training and inference workflow, which includes the process of creating a training datastore, selecting training options, training the network, and using the network to process test images.



The Operator Approximation Network

The multiscale CAN is trained to minimize the l_2 loss between the conventional output of an image processing operation and the network response after processing the input image using multiscale context aggregation. Multiscale context aggregation looks for information about each pixel from across the entire image, rather than limiting the search to a small neighborhood surrounding the pixel.



To help the network learn global image properties, the multiscale CAN architecture has a large receptive field. The first and last layers have the same size because the operator should not change the size of the image. Successive intermediate layers are dilated by exponentially increasing scale factors (hence the "multiscale" nature of the CAN). Dilation enables the network to look for spatially separated features at various spatial frequencies, without reducing the resolution of the image. After each convolution layer, the network uses adaptive normalization to balance the impact of batch normalization and the identity mapping on the approximated operator.

Download Training and Test Data

Download the IAPR TC-12 Benchmark, which consists of 20,000 still natural images [2]. The data set includes photos of people, animals, cities and more. You can use the helper function, `downloadIAPRTC12Data`, to download the data. The size of the data file is ~1.8 GB.

```
imagesDir = tempdir;
url_1 = 'http://www-i6.informatik.rwth-aachen.de/imageclef/resources/iaprtc12.tgz';
downloadIAPRTC12Data(url_1,imagesDir);
```

This example will train the network with small subset of the IAPRTC-12 Benchmark data.

```
trainImagesDir = fullfile(imagesDir,'iaprtc12','images','39');
exts = {'.jpg','.bmp','.png'};
inputImages = imageDatastore(trainImagesDir,'FileExtensions',exts);
```

List the number of training images.

```
numel(inputImages.Files)
```

```
ans = 916
```

Define Random Patch Extraction Datastore for Training

Use a random patch extraction datastore to feed the training data to the network. This datastore extracts random corresponding patches from two image datastores that contain the network inputs and desired network responses.

In this example, the network inputs are the unprocessed training images. The desired network responses are the processed images after bilateral filtering. Create an image datastore containing the unprocessed training images, and specify a custom read function that applies bilateral filtering when images are read from the datastore. The custom read function, `readAndBilateralFilterImage`, is attached as a supporting file to this example.

```
bilatFilteredImages = imageDatastore(trainImagesDir,'FileExtensions',exts);
bilatFilteredImages.ReadFcn = @readAndBilateralFilterImage;
```

Create the `randomPatchExtractionDatastore` from the two image datastores. Specify a patch size of 256-by-256 pixels. Specify '`PatchesPerImage`' to extract one randomly-positioned patch from each pair of images during training. Specify a mini-batch size of one.

```
miniBatchSize = 1;
patchSize = [256 256];
patchds = randomPatchExtractionDatastore(inputImages,bilatFilteredImages,patchSize, ...
    'PatchesPerImage',1);
patchds.MiniBatchSize = miniBatchSize;
```

The `randomPatchExtractionDatastore` provides mini-batches of data to the network at each iteration of the epoch. Perform a read operation on the datastore to explore the data.

```
inputBatch = read(patchds);
summary(inputBatch)
```

```
Variables:

    InputImage: 1×1 cell

    ResponseImage: 1×1 cell
```

Set Up Multiscale CAN Layers

This example defines the multiscale CAN using layers from Deep Learning Toolbox™, including:

- `imageInputLayer` - Image input layer
- `convolution2dLayer` - 2D convolution layer for convolutional neural networks
- `batchNormalizationLayer` - Batch normalization layer
- `leakyReluLayer` - Leaky rectified linear unit layer
- `regressionLayer` - Regression output layer for a neural network

Two custom scale layers are added to implement an adaptive batch normalization lLayer. These layers are attached as supporting files to this example.

- **adaptiveNormalizationMu** - Scale layer that adjusts the strengths of the batch-normalization branch
- **adaptiveNormalizationLambda** - Scale layer that adjusts the strengths of the identity branch

The first layer, `imageInputLayer`, operates on image patches. The patch size is based on the network receptive field, which is the spatial image region that affects the response of top-most layer in the network. Ideally, the network receptive field is the same as the image size so that it can see all the high level features in the image. For a bilateral filter, the approximation image patch size is fixed to 256-by-256.

```
networkDepth = 10;
numberOfFilters = 32;
firstLayer = imageInputLayer([256 256 3],'Name','InputLayer','Normalization','none');
```

The image input layer is followed by a 2-D convolution layer that contains 32 filters of size 3x3. Zero-pad the inputs to each convolution layer so that feature maps remain the same size as the input after each convolution. Initialize the weights to the identity matrix.

```
convolutionLayer = convolution2dLayer(3,numberOfFilters,'Padding',1,'Name','Conv1');
Wgts = zeros(3,3,3,numberOfFilters);
for ii = 1:3
```

```
    Wgts(2,2,ii,ii) = 1;
end
convolutionLayer.Weights = Wgts;
```

Each convolution layer is followed by a batch normalization layer and an adaptive normalization scale layer that adjusts the strengths of the batch-normalization branch. Later, this example will create the corresponding adaptive normalization scale layer that adjusts the strength of the identity branch. For now, follow the `adaptiveNormalizationMu` layer with an addition layer. Finally, specify a leaky ReLU layer with a scalar multiplier of 0.2 for negative inputs.

```
batchNorm = batchNormalizationLayer('Name','BN1');
adaptiveMu = adaptiveNormalizationMu(numberOfFilters,'Mu1');
addLayer = additionLayer(2,'Name','add1');
leakyrelLayer = leakyReluLayer(0.2,'Name','Leaky1');
```

Specify the middle layers of the network following the same pattern. Successive convolution layers have a dilation factor that scales exponentially with the network depth.

```
middleLayers = [convolutionLayer batchNorm adaptiveMu addLayer leakyrelLayer];

Wgts = zeros(3,3,numberOfFilters,numberOfFilters);
for ii = 1:numberOfFilters
    Wgts(2,2,ii,ii) = 1;
end

for layerNumber = 2:networkDepth-2
    dilationFactor = 2^(layerNumber-1);
    padding = dilationFactor;
    conv2dLayer = convolution2dLayer(3,numberOfFilters, ...
        'Padding',padding, ...
        'Name',['Conv' num2str(layerNumber)]);
    conv2dLayer.DilationFactor = dilationFactor;
    conv2dLayer.Weights = Wgts;
    batchNorm = batchNormalizationLayer('Name',['BN' num2str(layerNumber)]);
    adaptiveMu = adaptiveNormalizationMu(numberOfFilters,['Mu' num2str(layerNumber)]);
    addLayer = additionLayer(2,'Name',['add' num2str(layerNumber)]);
    leakyrelLayer = leakyReluLayer(0.2, 'Name', ['Leaky' num2str(layerNumber)]);
    middleLayers = [middleLayers conv2dLayer batchNorm adaptiveMu addLayer leakyrelLaye
end
```

Do not apply a dilation factor to the second-to-last convolution layer.

```
conv2dLayer = convolution2dLayer(3,numberOfFilters, ...
    'Padding', 1,'Name','Conv9');
```

```
conv2dLayer.Weights = Wgts;
batchNorm = batchNormalizationLayer('Name','AN9');
adaptiveMu = adaptiveNormalizationMu(numberOfFilters,'Mu9');
addLayer = additionLayer(2,'Name','add9');
leakyrelLayer = leakyReluLayer(0.2,'Name', 'Leaky9');
middleLayers = [middleLayers conv2dLayer batchNorm adaptiveMu addLayer leakyrelLayer];
```

The last convolution layer has a single filter of size 1x1x32x3 that reconstructs the image.

```
conv2dLayer = convolution2dLayer(1,3, ...
    'NumChannels',numberOfFilters, ...
    'Name','Conv10');
conv2dLayer.Weights = sqrt(2/(9*numberOfFilters))*randn(1,1,numberOfFilters,3);
```

The last layer is a regression layer instead of a leaky ReLU layer. The regression layer computes the mean-squared error between the bilateral-filtered image and the network prediction.

```
finalLayers = [conv2dLayer
    regressionLayer('Name','FinalRegressionLayer')
];
```

Concatenate all the layers.

```
layers = [firstLayer middleLayers finalLayers'];
lgraph = layerGraph(layers);
```

Create skip connections, which act as the identity branch for the adaptive normalization equation. Connect the skip connections to the addition layers.

```
skipConv1 = adaptiveNormalizationLambda(numberOfFilters,'Lambda1');
skipConv2 = adaptiveNormalizationLambda(numberOfFilters,'Lambda2');
skipConv3 = adaptiveNormalizationLambda(numberOfFilters,'Lambda3');
skipConv4 = adaptiveNormalizationLambda(numberOfFilters,'Lambda4');
skipConv5 = adaptiveNormalizationLambda(numberOfFilters,'Lambda5');
skipConv6 = adaptiveNormalizationLambda(numberOfFilters,'Lambda6');
skipConv7 = adaptiveNormalizationLambda(numberOfFilters,'Lambda7');
skipConv8 = adaptiveNormalizationLambda(numberOfFilters,'Lambda8');
skipConv9 = adaptiveNormalizationLambda(numberOfFilters,'Lambda9');

lgraph = addLayers(lgraph,skipConv1);
lgraph = connectLayers(lgraph,'Conv1','Lambda1');
lgraph = connectLayers(lgraph,'Lambda1','add1/in2');
```

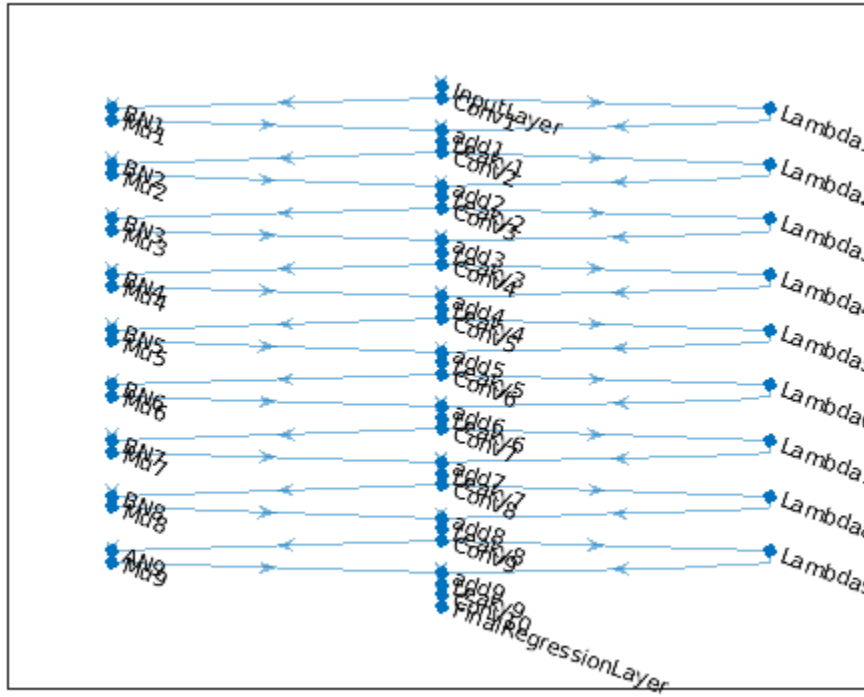
```
lgraph = addLayers(lgraph,skipConv2);
lgraph = connectLayers(lgraph,'Conv2','Lambda2');
lgraph = connectLayers(lgraph,'Lambda2','add2/in2');

lgraph = addLayers(lgraph,skipConv3);
lgraph = connectLayers(lgraph,'Conv3','Lambda3');
lgraph = connectLayers(lgraph,'Lambda3','add3/in2');

lgraph = addLayers(lgraph,skipConv4);
lgraph = connectLayers(lgraph,'Conv4','Lambda4');
lgraph = connectLayers(lgraph,'Lambda4','add4/in2');

lgraph = addLayers(lgraph,skipConv5);
lgraph = connectLayers(lgraph,'Conv5','Lambda5');
lgraph = connectLayers(lgraph,'Lambda5','add5/in2');

lgraph = addLayers(lgraph,skipConv6);
lgraph = connectLayers(lgraph,'Conv6','Lambda6');
lgraph = connectLayers(lgraph,'Lambda6','add6/in2');

lgraph = addLayers(lgraph,skipConv7);
lgraph = connectLayers(lgraph,'Conv7','Lambda7');
lgraph = connectLayers(lgraph,'Lambda7','add7/in2');

lgraph = addLayers(lgraph,skipConv8);
lgraph = connectLayers(lgraph,'Conv8','Lambda8');
lgraph = connectLayers(lgraph,'Lambda8','add8/in2');

lgraph = addLayers(lgraph,skipConv9);
lgraph = connectLayers(lgraph,'Conv9','Lambda9');
lgraph = connectLayers(lgraph,'Lambda9','add9/in2');
```

Alternatively, you can use this helper function to create the multiscale CAN layers.

```
layers = operatorApproximationLayers();
```

Plot the layer graph.

```
plot(lgraph)
```



Specify Training Options

Train the network using the Adam optimizer. Specify the hyperparameter settings by using the `trainingOptions` function. Use the default values of 0.9 for '`Momentum`' and 0.0001 for '`L2Regularization`' (weight decay). Specify a constant learning rate of 0.0001. Train for 181 epochs.

```
maxEpochs = 181;
initLearningRate = 0.0001;
miniBatchSize = 1;

options = trainingOptions('adam', ...
    'InitialLearnRate',initLearningRate, ...
```

```
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize);
```

Train the Network

Now that the data source and training options are configured, train the multiscale CAN using the `trainNetwork function`. To train the network, set the `doTraining` parameter in the following code to `true`. A CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

If you keep the `doTraining` parameter in the following code as `false`, then the example returns a pretrained multiscale CAN that has been trained to approximate a bilateral filter.

Note: Training usually takes about 15 hours on an `NVIDIA™ Titan X` and can take even longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    modelDateTime = datestr(now,'dd-mmm-yyyy-HH-MM-SS');
    net = trainNetwork(patchds,layers,options);
    save(['trainedOperatorLearning-' modelDateTime '-Epoch-' num2str(maxEpochs) '.mat']
else
    load('trainedOperatorLearning-Epoch-181.mat');
end
```

Perform Bilateral Filtering Approximation Using Multiscale CAN

To process an image using a trained multiscale CAN network that approximates a bilateral filter, follow the remaining steps of this example. The remainder of the example shows how to:

- Create a sample noisy input image from a reference image.
- Perform conventional bilateral filtering of the noisy image using the `imbilatfilt` function.
- Perform an approximation to bilateral filtering on the noisy image using the CAN.
- Visually compare the denoised images from operator approximation and conventional bilateral filtering.
- Evaluate the quality of the denoised images by quantifying the similarity of the images to the pristine reference image.

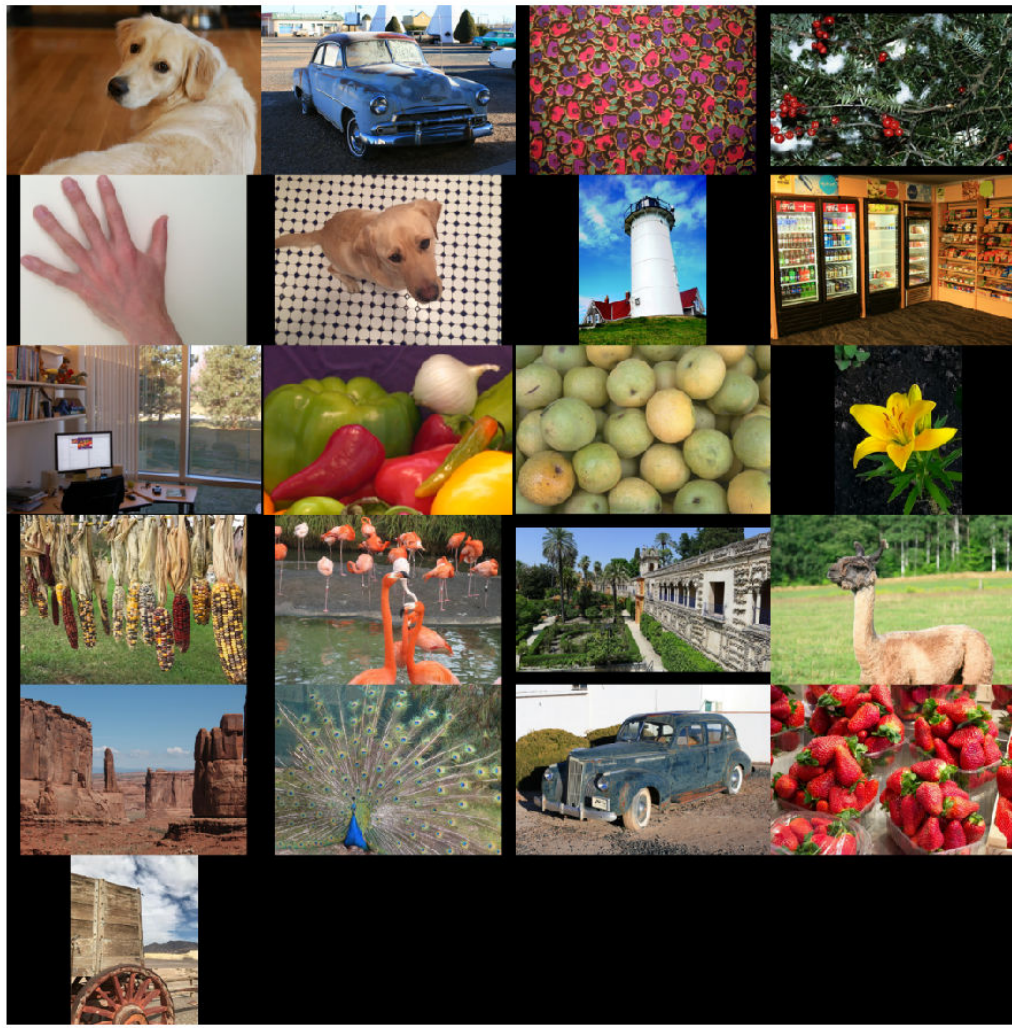
Create Sample Noisy Image

Create a sample noisy image that will be used to compare the results of operator approximation to conventional bilateral filtering. The test data set, `testImages`, contains 21 pristine images shipped in Image Processing Toolbox™. Load the images into an `imageDatastore`.

```
exts = {'.jpg','.png'};
fileNames = {'sherlock.jpg','car2.jpg','fabric.png','greens.jpg','hands1.jpg','kobi.png
    'lighthouse.png','micromarket.jpg','office_4.jpg','onion.png','pears.png','yellowli
    'indiancorn.jpg','flamingos.jpg','sevilla.jpg','llama.jpg','parkavenue.jpg',...
    'peacock.jpg','car1.jpg','strawberries.jpg','wagon.jpg'};
filePath = [fullfile(matlabroot,'toolbox','images','imdata') filesep];
filePathNames = strcat(filePath,fileNames);
testImages = imageDatastore(filePathNames,'FileExtensions',exts);
```

Display the test images as a montage.

```
montage(testImages)
```



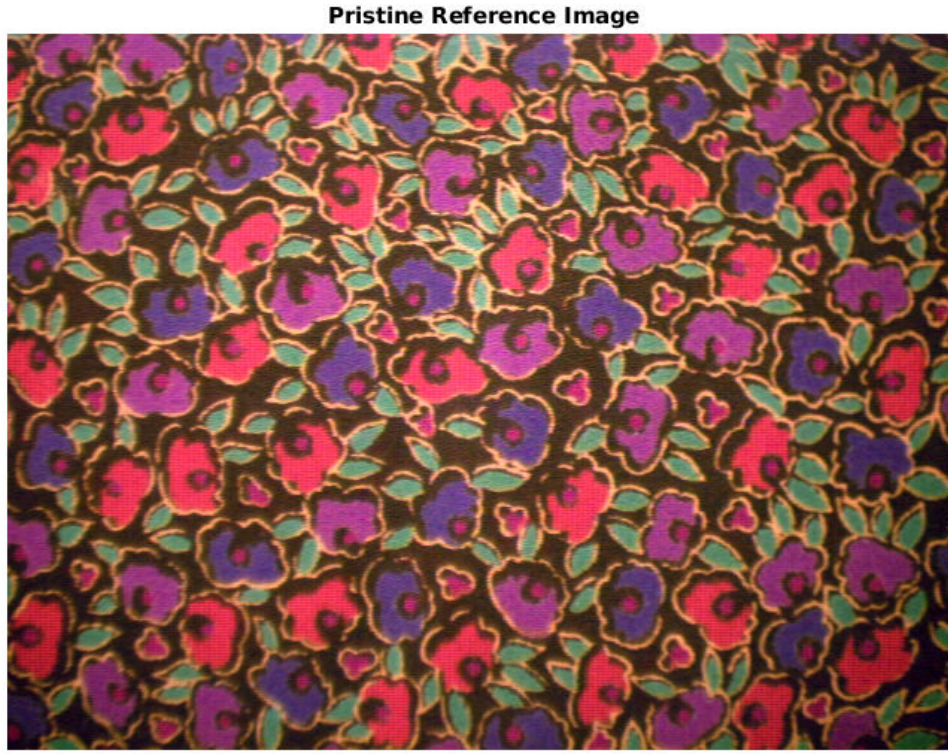
Select one of the images to use as the reference image for bilateral filtering. Convert the image to data type `uint8`.

```
indx = 3; % Index of image to read from the test image datastore
Ireference = readimage(testImages,indx);
Ireference = im2uint8(Ireference);
```

You can optionally use your own image as the reference image. Note that the size of the test image must be at least 256x256. If the test image is smaller than 256x256, then increase the image size by using the `imresize` function. The network also requires an RGB test image. If the test image is grayscale, then convert the image to RGB by using the `cat` function to concatenate three copies of the original image along the third dimension.

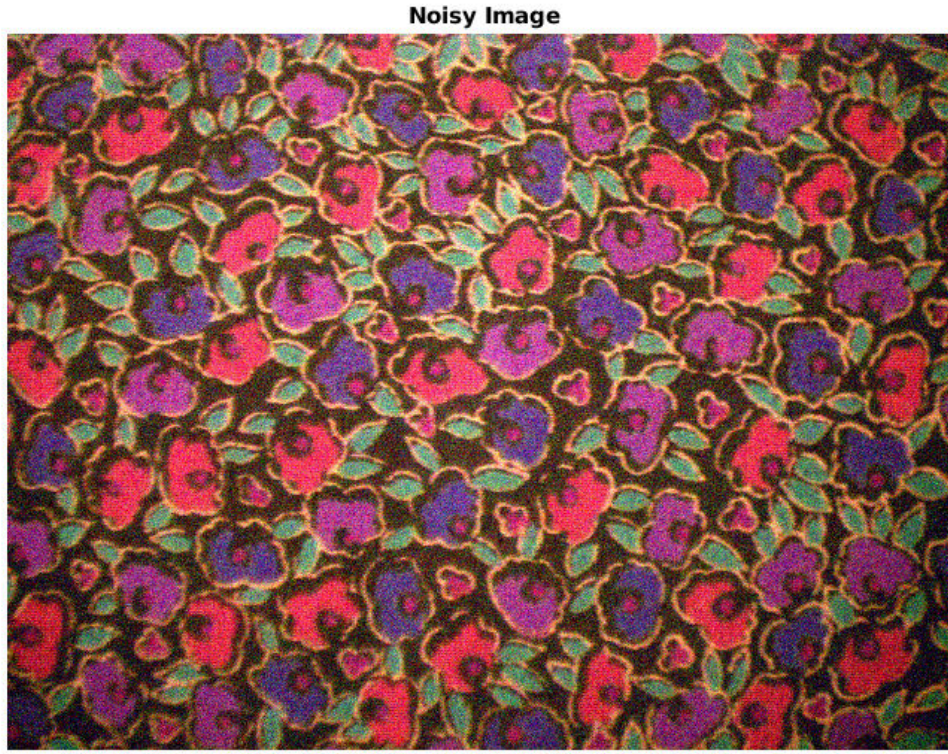
Display the reference image.

```
imshow(Ireference)
title('Pristine Reference Image')
```



Use the `imnoise` function to add zero-mean Gaussian white noise with a variance of 0.00001 to the reference image.

```
Inoisy = imnoise(Ireference,'gaussian',0.00001);
imshow(Inoisy)
title('Noisy Image')
```



Filter Image Using Bilateral Filtering

Conventional bilateral filtering is a standard way to reduce image noise while preserving edge sharpness. Use the `imbilatfilt` function to apply a bilateral filter to the noisy image. Specify a degree of smoothing equal to the variance of pixel values.

```
degreeOfSmoothing = var(double(Inoisy(:)));
Ibilat = imbilatfilt(Inoisy,degreeOfSmoothing);
imshow(Ibilat)
title('Denoised Image Obtained Using Bilateral Filtering')
```

Process Image Using Trained Network

Pass the normalized input image through the trained network and observe the `activations` from the final layer (a regression layer). The output of the network is the desired denoised image.

```
Iapprox = activations(net,Inoisy,'FinalRegressionLayer');
```

Image Processing Toolbox™ requires floating point images to have pixel values in the range [0, 1]. Use the `rescale` function to scale the pixel values to this range, then convert the image to `uint8`.

```
Iapprox = rescale(Iapprox);
Iapprox = im2uint8(Iapprox);
```

```
imshow(Iapprox)
title('Denoised Image Obtained Using Multiscale CAN')
```

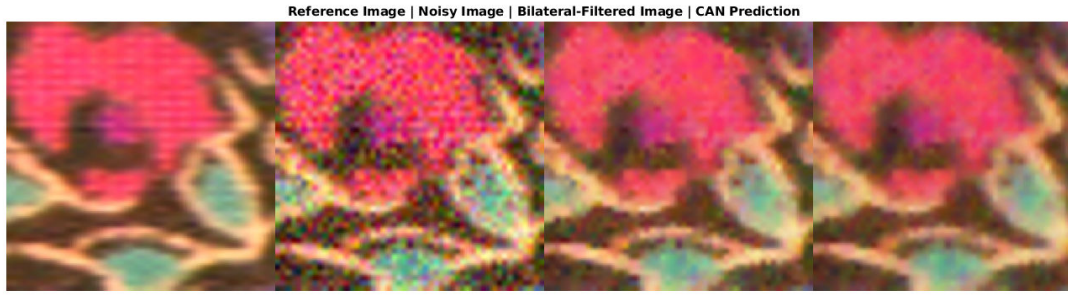
Visual and Quantitative Comparison

To get a better visual understanding of the denoised images, examine a small region inside each image. Specify a region of interest (ROI) using vector `roi` in the format [*x y width height*]. The elements define the x- and y-coordinate of the top left corner, and the width and height of the ROI.

```
roi = [300 30 50 50];
```

Crop the images to this ROI, and display the result as a montage.

```
montage({imcrop(Ireference,roi),imcrop(Inoisy,roi), ...
    imcrop(Ibilat,roi),imcrop(Iapprox,roi)}, ...
    'Size',[1 4]);
title('Reference Image | Noisy Image | Bilateral-Filtered Image | CAN Prediction');
```



The CAN removed more noise than conventional bilateral filtering. Both techniques preserve edge sharpness.

Use image quality metrics to quantitatively compare the noisy input image, the bilateral-filtered image, and the operator-approximated image. The reference image is the original reference image, `Ireference`, before adding noise.

Measure the peak signal-to-noise ratio (PSNR) of each image against the reference image. Larger PNSR values generally indicate better image quality. See `psnr` for more information about this metric.

```
noisyPSNR = psnr(Inoisy,Ireference);
bilatPSNR = psnr(Ibilat,Ireference);
approxPSNR = psnr(Iapprox,Ireference);
disp(['PSNR of: Noisy Image / Bilateral-Filtered Image / Operator Approximated Image =
    num2str([noisyPSNR bilatPSNR approxPSNR])])
```

```
PSNR of: Noisy Image / Bilateral-Filtered Image / Operator Approximated Image = 20.284
```

Measure the structural similarity index (SSIM) of each image. SSIM assesses the visual impact of three characteristics of an image: luminance, contrast and structure, against a reference image. The closer the SSIM value is to 1, the better the test image agrees with the reference image. See `ssim` for more information about this metric.

```
noisySSIM = ssim(Inoisy,Ireference);
bilatSSIM = ssim(Ibilat,Ireference);
```

```
approxSSIM = ssim(Iapprox,Ireference);
disp(['SSIM of: Noisy Image / Bilateral-Filtered Image / Operator Approximated Image =
    num2str([noisySSIM bilatSSIM approxSSIM])])
```

```
SSIM of: Noisy Image / Bilateral-Filtered Image / Operator Approximated Image = 0.76248
```

Measure perceptual image quality using the Naturalness Image Quality Evaluator (NIQE). Smaller NIQE scores indicate better perceptual quality. See `niqe` for more information about this metric.

```
noisyNIQE = niqe(Inoisy);
bilatNIQE = niqe(Ibilat);
approxNIQE = niqe(Iapprox);
disp(['NIQE score of: Noisy Image / Bilateral-Filtered Image / Operator Approximated Im
    num2str([noisyNIQE bilatNIQE approxNIQE])])
```

```
NIQE score of: Noisy Image / Bilateral-Filtered Image / Operator Approximated Image = 1
```

Compared to conventional bilateral filtering, the operator approximation produces better metric scores.

Summary

This example shows how to create and train a multiscale CAN to approximate a bilateral filter. These are the steps to train the network:

- Download the training data.
- Create a `randomPatchExtractionDatastore` that feeds training data to the network.
- Define the layers of the multiscale CAN.
- Specify training options.
- Train the network using the `trainNetwork` function.

After training the multiscale CAN or loading a pretrained network, the example approximates a bilateral filtering operation on a noisy image. The example compares the result against a conventional bilateral filter that does not use deep learning. Operator approximation outperforms bilateral filtering with respect to both perceptual image quality and quantitative quality measurements.

References

[1] Chen, Q. J. Xu, and V. Koltun. "Fast Image Processing with Fully-Convolutional Networks." In *Proceedings of the 2017 IEEE Conference on Computer Vision*. Venice, Italy, Oct. 2017, pp. 2516-2525.

[2] Grubinger, M., P. Clough, H. Müller, and T. Deselaers. "The IAPR TC-12 Benchmark: A New Evaluation Resource for Visual Information Systems." *Proceedings of the OntoImage 2006 Language Resources For Content-Based Image Retrieval*. Genoa, Italy. Vol. 5, May 2006, p. 10.

See Also

`activations` | `imbilatfilt` | `layerGraph` | `randomPatchExtractionDatastore` | `trainNetwork` | `trainingOptions`

More About

- “Preprocess Images for Deep Learning” (Deep Learning Toolbox)
- “List of Deep Learning Layers” (Deep Learning Toolbox)

Code Generation for Image Processing Toolbox Functions

- “Code Generation for Image Processing” on page 18-2
- “List of Supported Functions with Usage Notes” on page 18-3
- “Generate Code from Application Containing Image Processing Functions” on page 18-8
- “Understand Code Generation with Image Processing Toolbox” on page 18-24

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 **Generate** on the Generate Code page of the MATLAB Coder app. You can choose to generate 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 18-8.

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” on page 18-2.
- 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.

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

adaptthresh*
affine2d*
boundarymask*
bwareaopen*
bwboundaries*
bwconncomp*

bwdist*
bweuler*
bwlabel*
bwlookup*
bwmorph*
bwpack*
bwperim*
bwselect*
bwtraceboundary*
bwunpack*
conndef*
demosaic*
edge*
fitgeotrans*
fspecial*
getrangefromclass*
grayconnected*
histeq*
hough*
houghlines*
houghpeaks*
im2int16*
im2uint8*
im2uint16*
im2single*
im2double*
imabsdiff*
imadjust*
imbinarize*

imbothat*
imboxfilt*
imclearborder*
imclose*
imcomplement*
imcrop*
imdilate*
imerode*
imextendedmax*
imextendedmin*
imfill*
imfilter*
imfindcircles*
imgaborfilt*
imgaussfilt*
imgradient3*
imgradientxyz*
imhist*
imhmax*
imhmin*
imlincomb*
immse*
imopen*
imoverlay*
impyramid*
imquantize*
imread*
imreconstruct*
imref2d*

imref3d*
imregionalmax*
imregionalmin*
imresize*
imrotate*
imtophat*
imtranslate*
imwarp*
integralBoxFilter*
integralImage
intlut*
iptcheckconn*
iptcheckmap*
lab2rgb*
label2idx*
label2rgb*
mean2*
medfilt2*
multithresh*
offsetstrel*
ordfilt2*
otsuthresh*
padarray*
projective2d*
psnr*
regionprops*
rgb2gray*
rgb2lab*
rgb2ycbcr*

`strel*`
`stretchlim*`
`superpixels*`
`watershed*`
`ycbcr2rgb*`

See Also

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 18-8 “Generate Code” on page 18-9

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 than 2^32-1 elements. In the near future
     you will be required to update your code to utilize the
     new API. You can find more information about this at:
     https://www.mathworks.com/help/matlab/matlab_external/upgrading-mex-files-to-use-6

To choose a different C compiler, select one from the following:
Microsoft Visual C++ 2012 (C)  mex -setup:C:\matlab\bin\win64\mexopts\msvc2012.xml C
Intel C++ Composer XE 2013 with Microsoft Visual Studio 2012 (C)  mex -setup:C:\matlab\
Intel C++ Composer XE 2011 with Microsoft Visual Studio 2010 (C)  mex -setup:C:\mexopts
Intel C++ Composer XE 2013 with Microsoft Visual Studio 2010 (C)  mex -setup:C:\matlab\
Microsoft Visual C++ 2010 (C)  mex -setup:C:\MATLAB\R2015a\mex_C_win64.xml C

To choose a different language, select one from the following:
```

```
mex -setup C++
mex -setup FORTRAN mex -setup FORTRAN
```

Generate Code

This example shows how to generate C code from MATLAB code that includes Image Processing Toolbox functions using MATLAB Coder. To illustrate the process, the code used by this example includes some readiness issues and build issues that you must overcome before you can generate code.

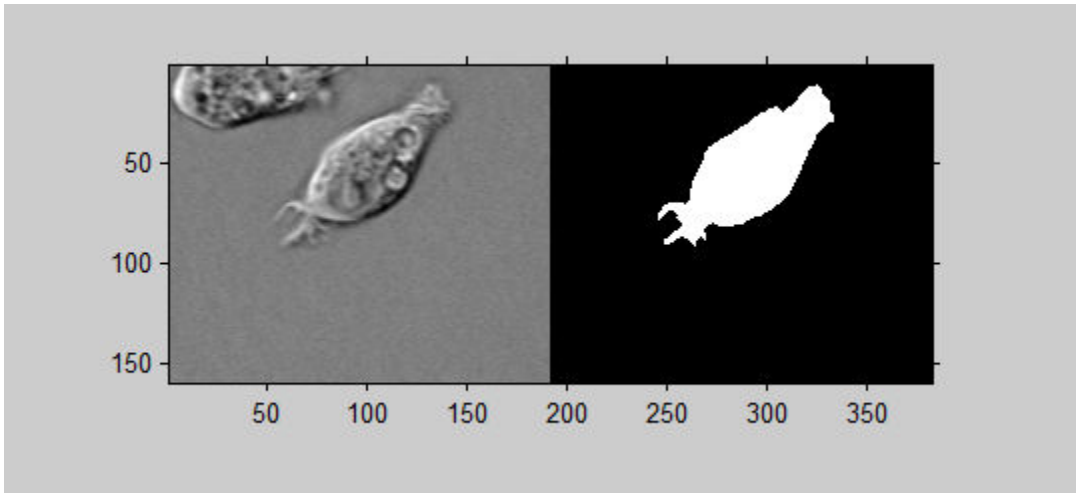
Copy the following code into a file and save it, giving it the name `cellDetectionMATLAB.m`.

```
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. 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 your MATLAB code before generating C code. Since you modify this code for code generation, it is good to work with a copy.

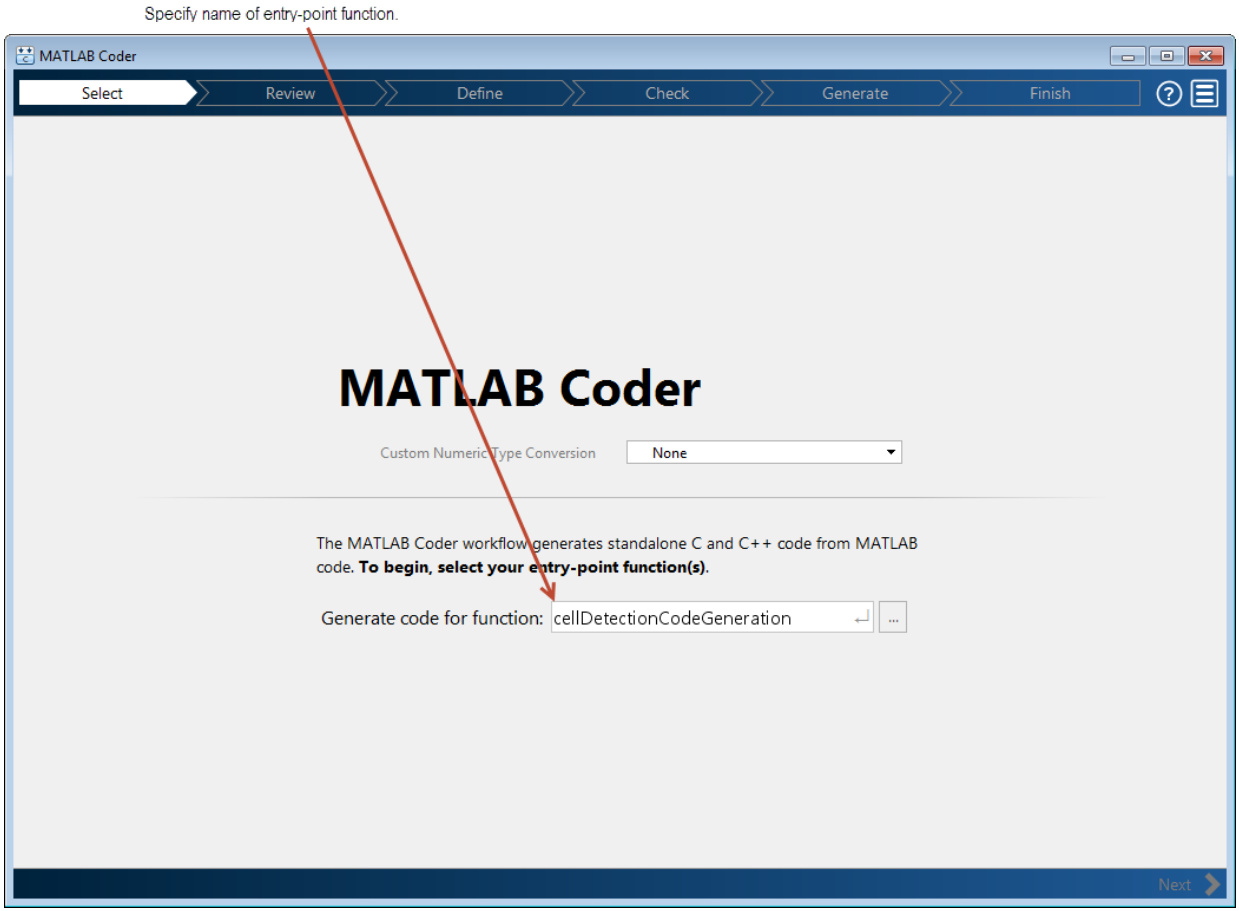
```
copyfile('cellDetectionMATLAB.m','cellDetectionCodeGeneration.m');
```

Open the code generation version of the function in the MATLAB editor. 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.

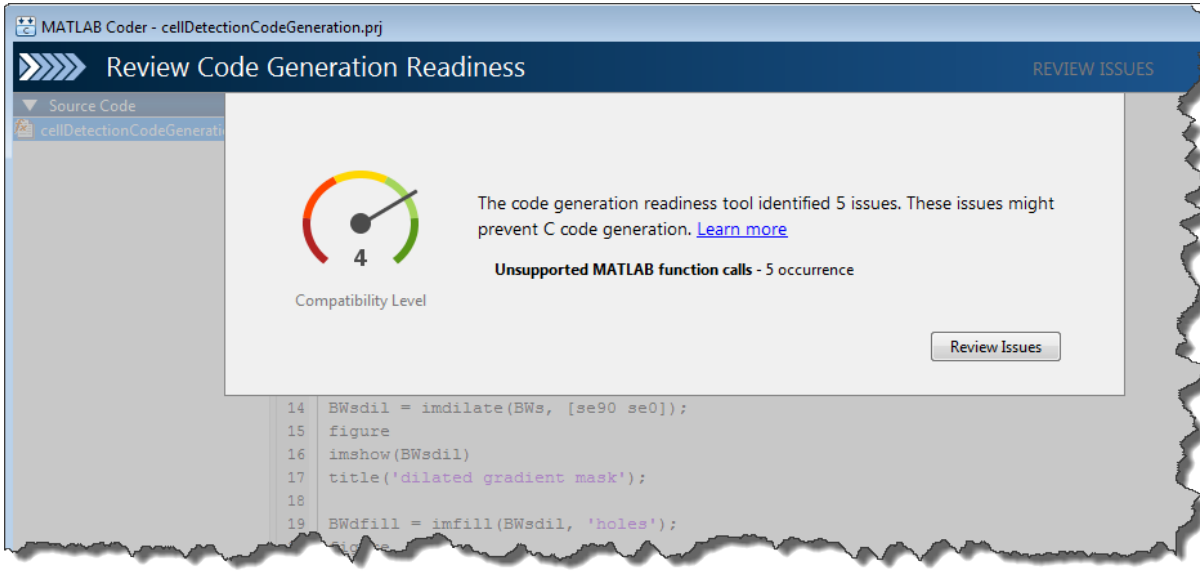
```
function BWfinal = cellDetectionCodeGeneration(I) %#codegen
.
.
.
```

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.)

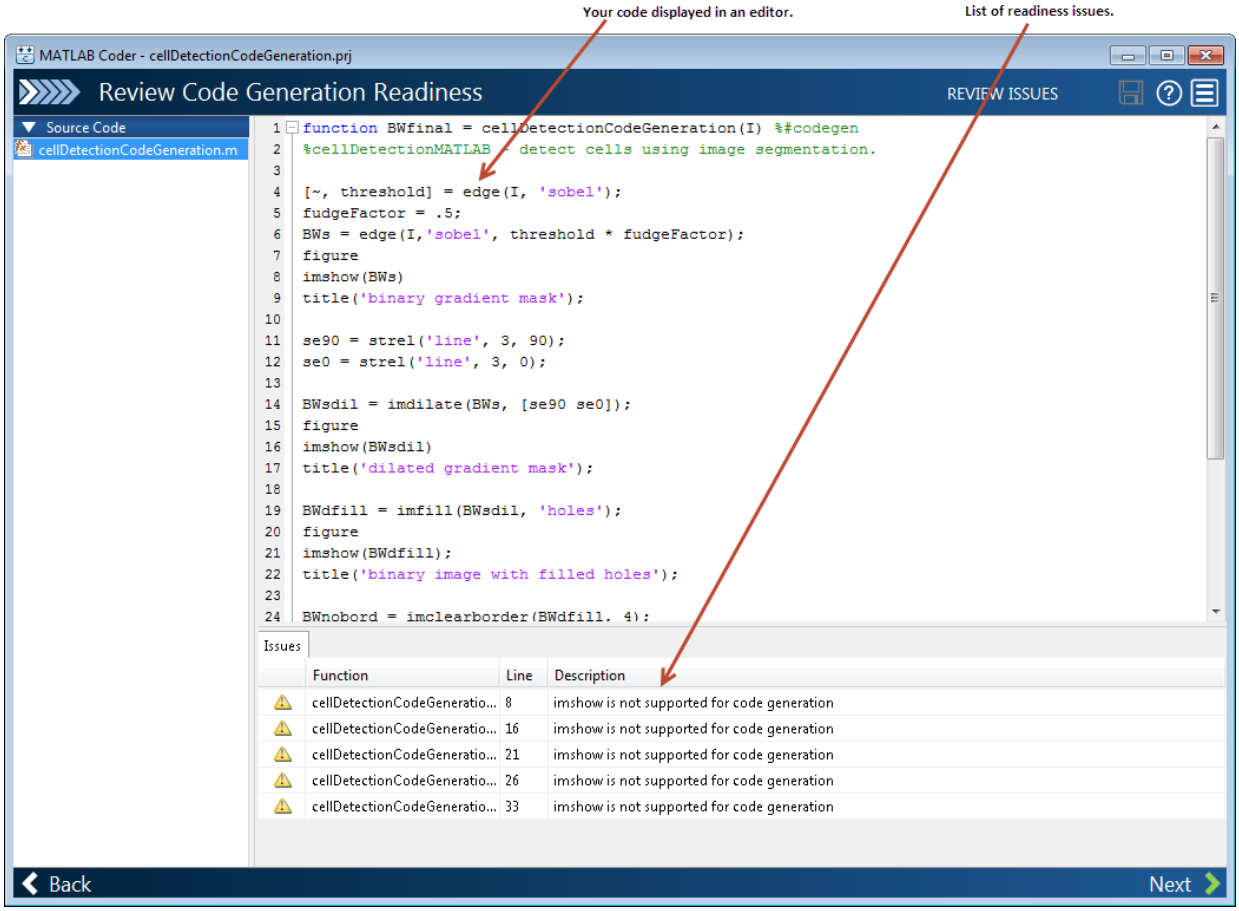
Specify the name of your entry-point function, `cellDetectionCodeGeneration`, and press **Enter**.



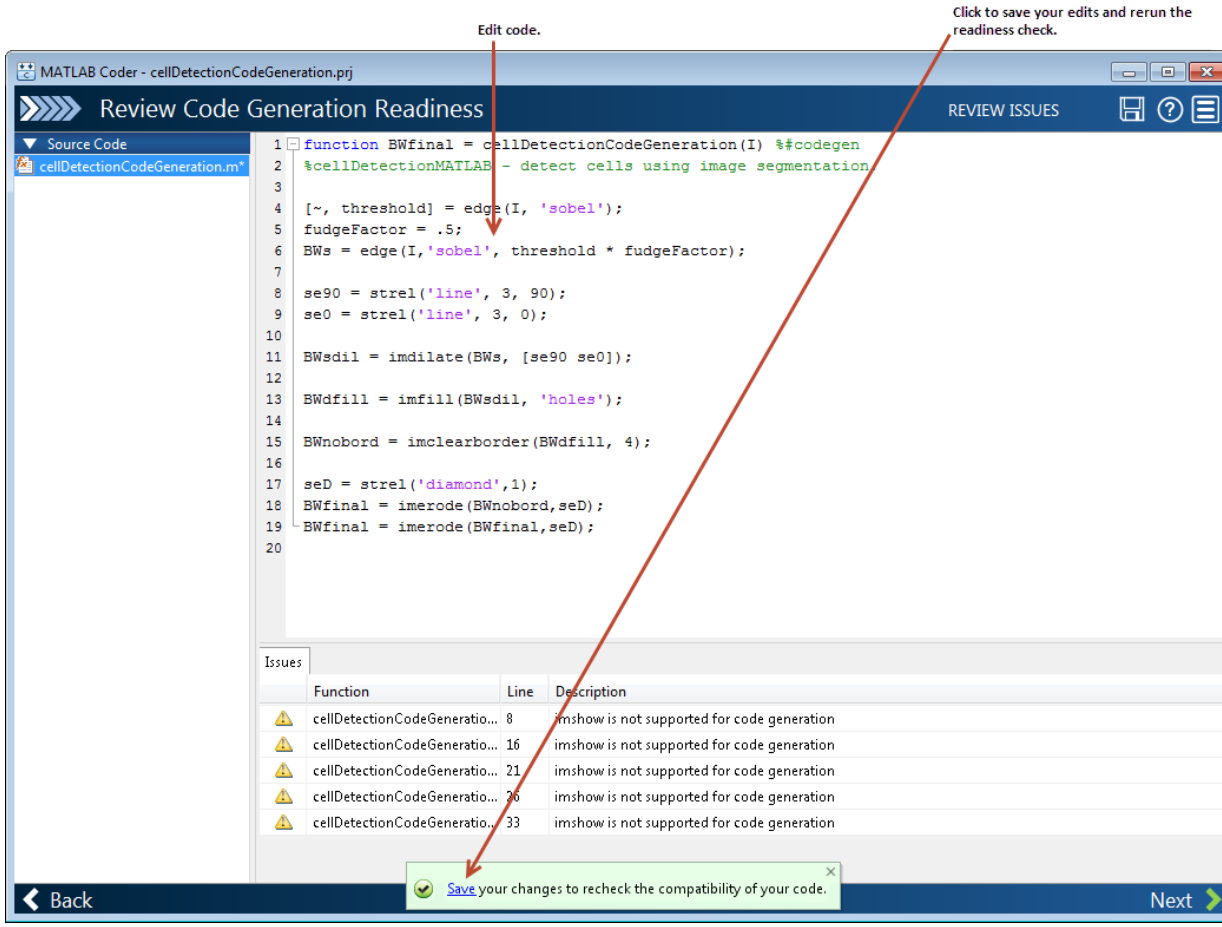
Determine your code's readiness for code generation. Click **Next** (at the bottom right of the coder window). MATLAB Coder identifies any issues that might prevent code generation. The example code contains five unsupported function calls.



Review the readiness issues. Click **Review Issues**. In the report, MATLAB Coder displays your code in an editing window with the readiness issues listed below, flagging uses of the `imshow` function which does not support code generation.



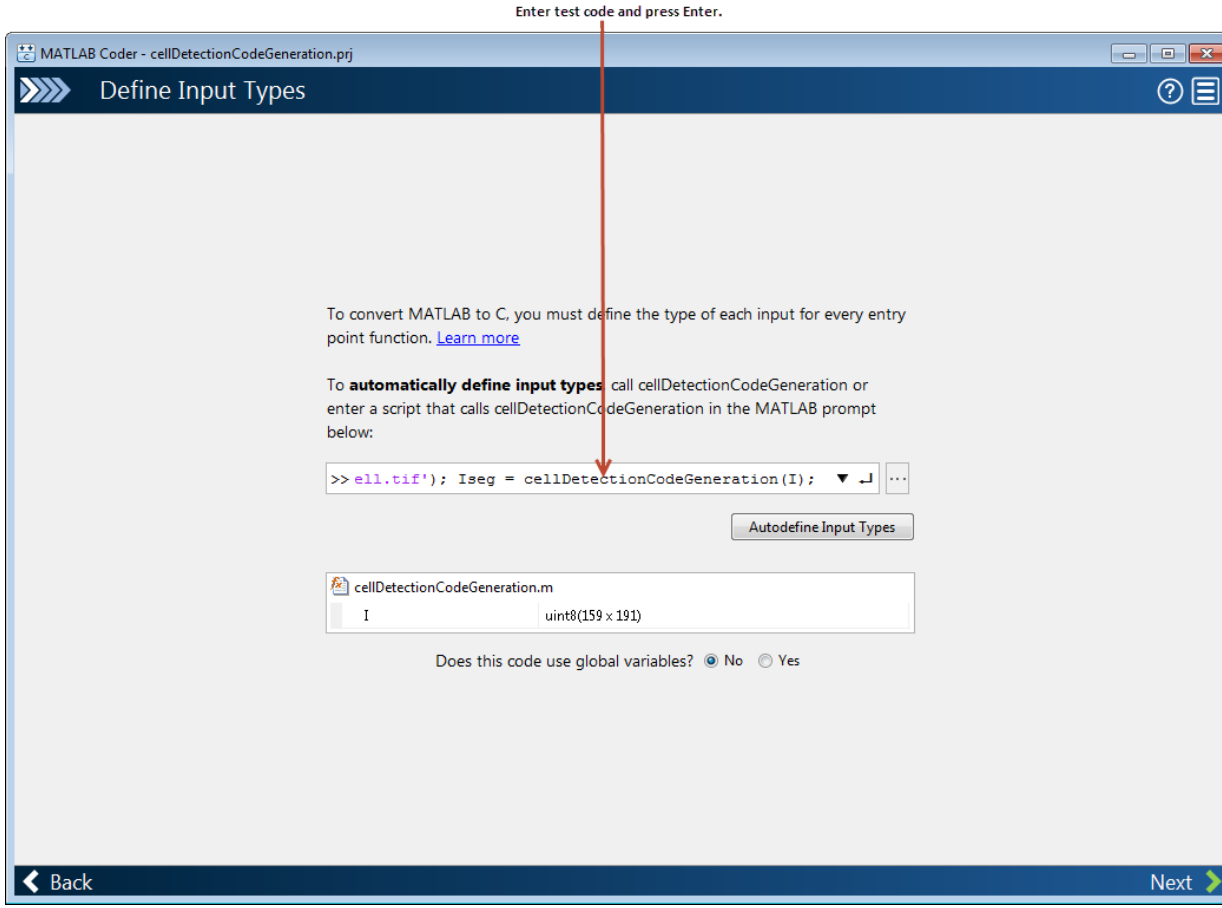
Address the readiness issues. Remove the calls to `imshow` and related display code from your example. The display statements aren't necessary for the segmentation operation. You can edit the example code in the MATLAB Coder. When you have removed the code, click **Save** to save your edits and rerun the readiness check. After rerunning the readiness check, MATLAB Coder displays the **No issues found** message.



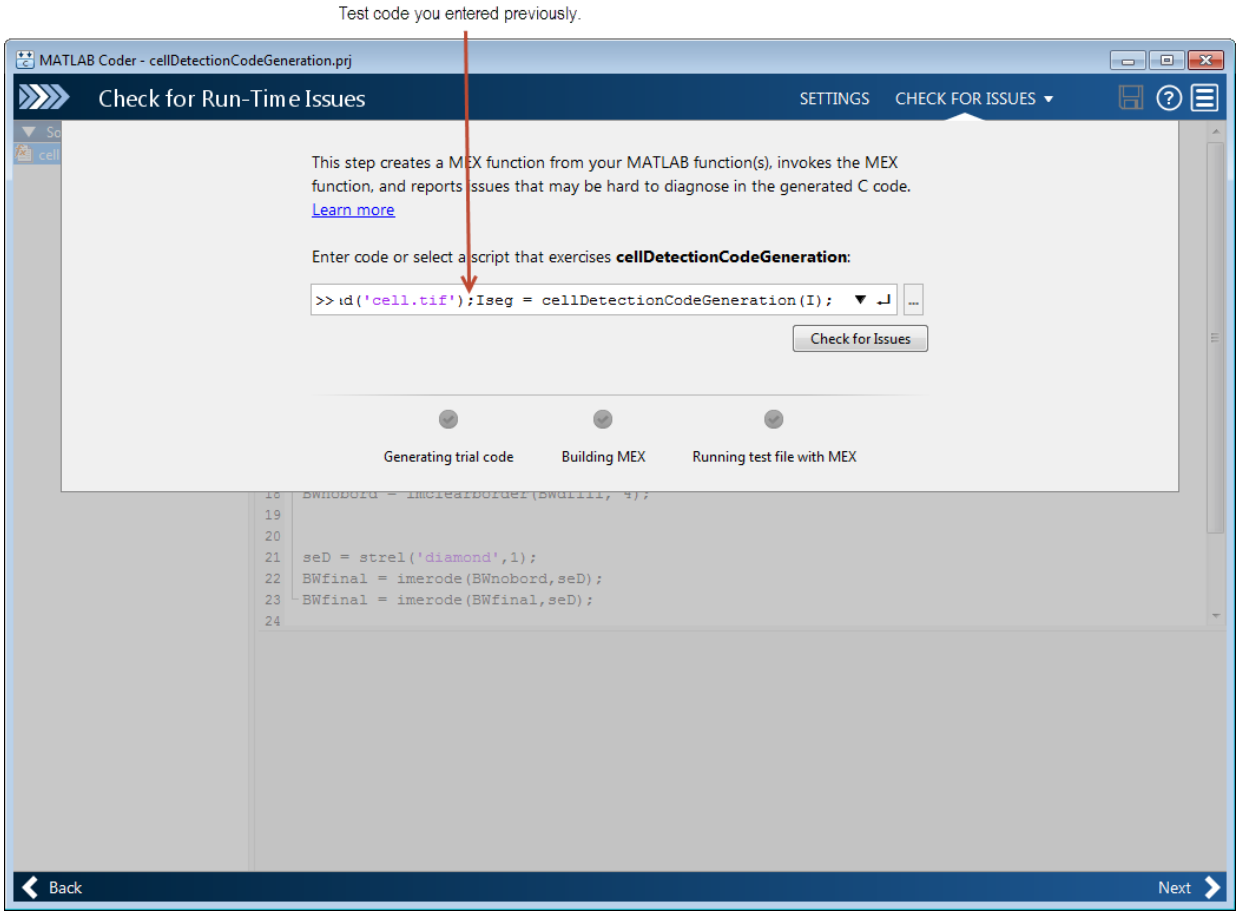
Define the size and data type of the inputs to your function. Every input to your code must be specified to be of fixed size, variable size or a constant. There are several ways to specify the size of your input argument but the easiest way is by giving MATLAB Coder an example of calling your function. Enter a script that calls your function in the text entry field. For this example, enter the following code and press **Enter**.

```
I = imread('cell.tif');
Iseg = cellDetectionCodeGeneration(I);
```

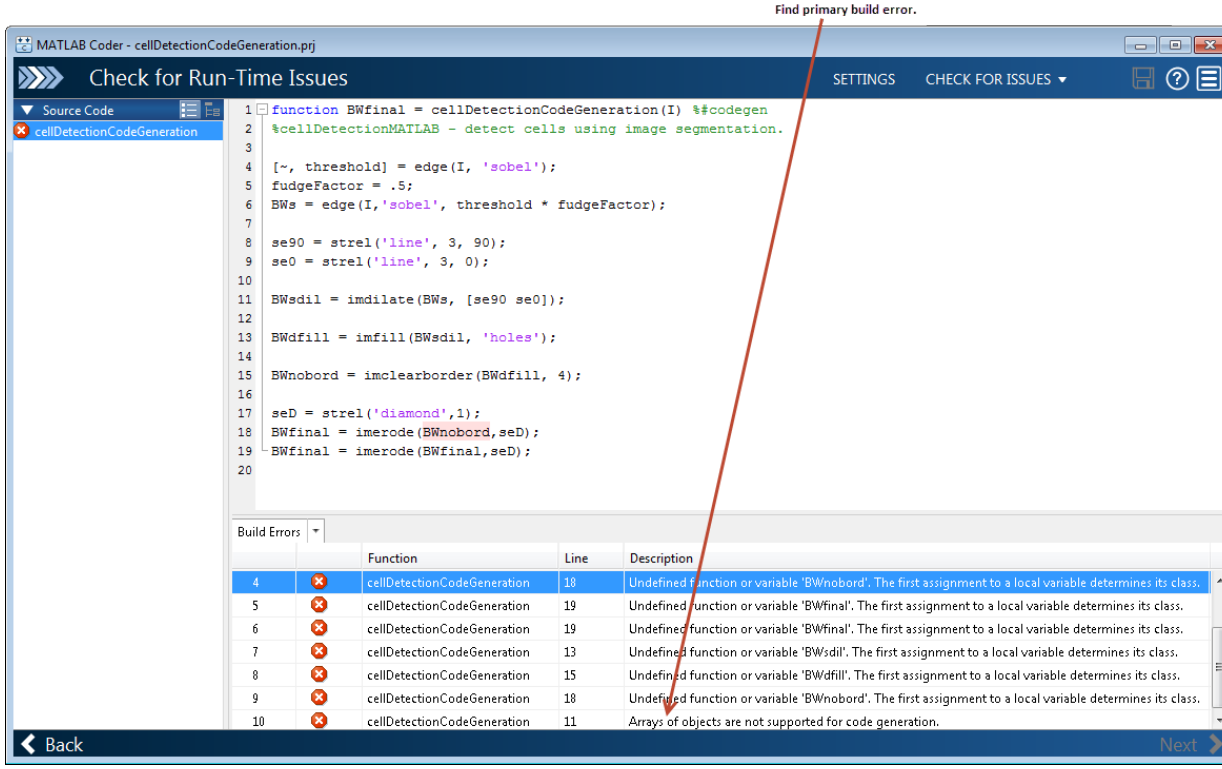
For more information about defining inputs, see the MATLAB Coder documentation. After MATLAB Coder returns with the input type definition, click **Next**.



Check for run-time issues. 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. You can use the same code you entered to define input types (which is preloaded into the dialog box). Click **Check for Issues**.



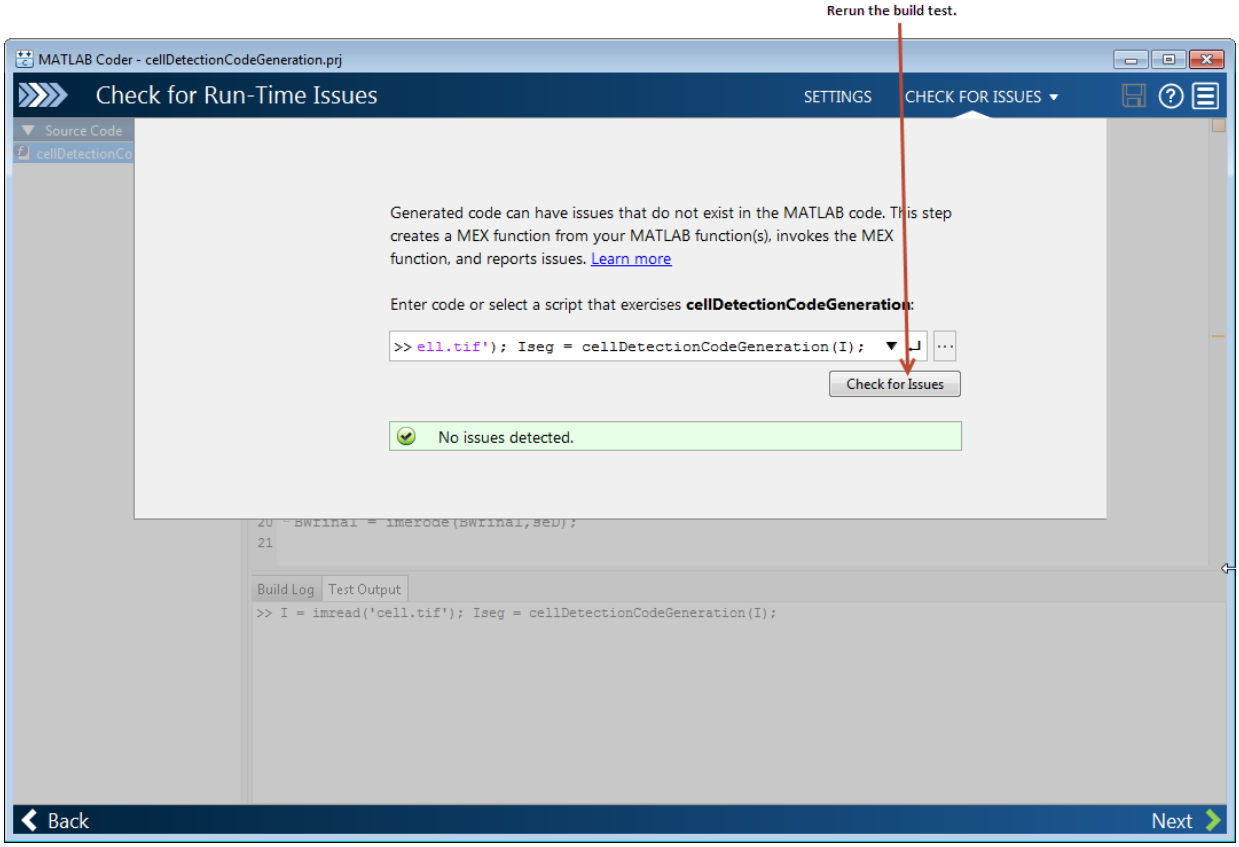
This example contains a build issue: it passes an array of `strel` objects to `imdilate` and arrays of objects are not supported for code generation.



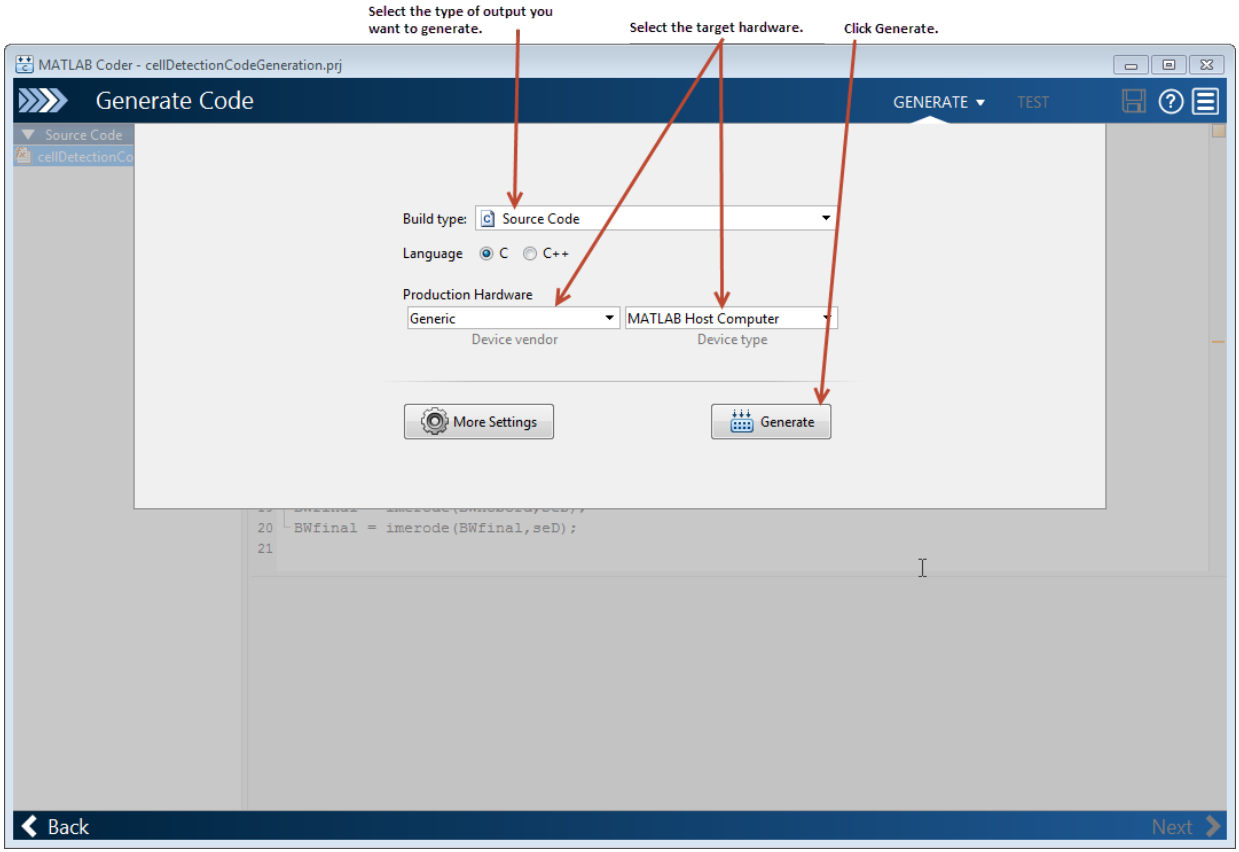
Address the build issues identified. For this example, you must modify the call to `imdilate` to avoid passing an array of strel objects. This is accomplished by replacing the single call to `imdilate` with two separate calls to `imdilate` where you pass one strel object with each call, highlighted in 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);
```

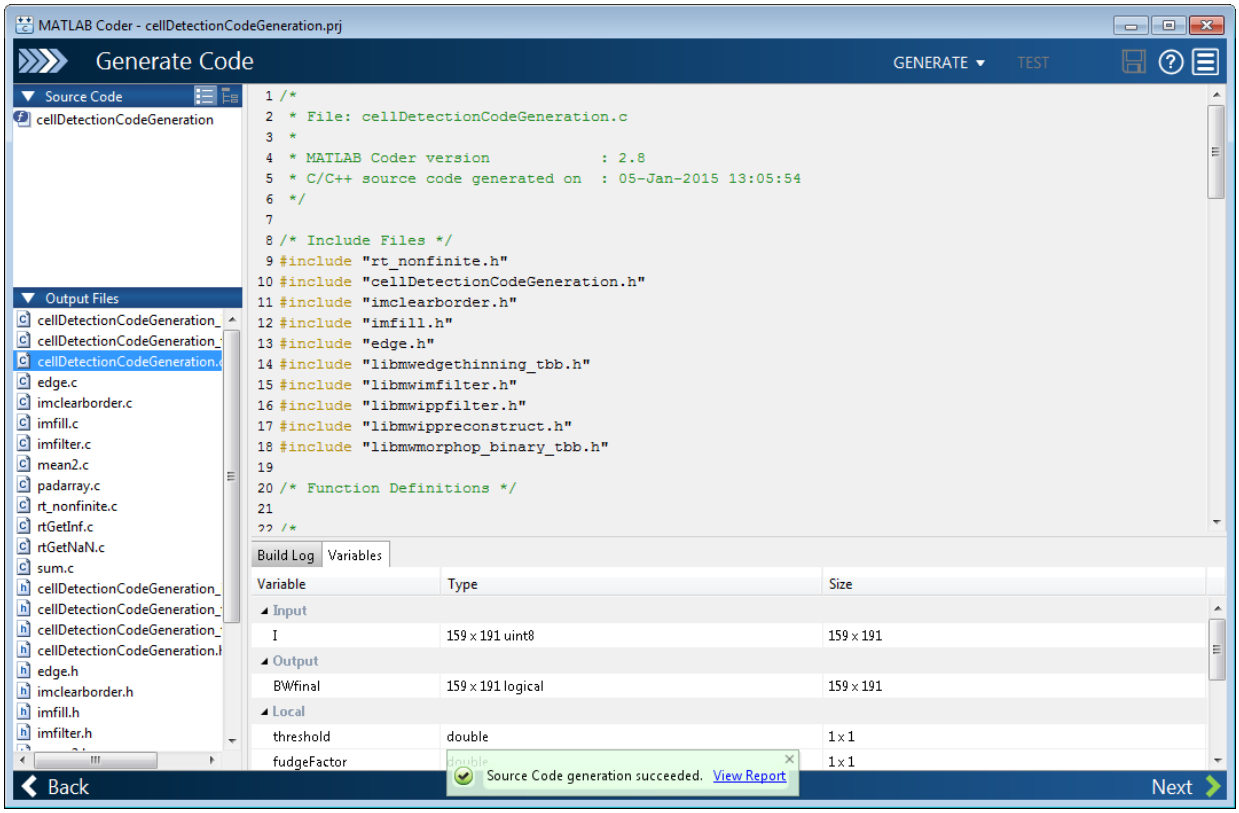
Rerun the test build to make sure your changes fixed the issue. Click **Check for Issues**. MATLAB Coder displays a message declaring that no issues were detected. You are now ready to generate code. Click **Next**.



Choose the type of code you want to generate and select the target platform. MATLAB Coder can generate C or C++ source code, a MEX file, a static library, a shared library, or a standalone executable. For Production Hardware, you can select from many choices including ARM and Intel processors. For this example, select Source Code and, for language, leave the **C** option selected, and select Generic for production hardware and MATLAB Host Computer for device type. When you choose MATLAB Host Computer, MATLAB Coder generates code that depends on a precompiled shared library. Image Processing Toolbox functions use a shared library to preserve performance optimizations. When you have made your selections, click **Generate**.



Generate code. When its done,MATLAB Coder displays the generated code.



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:

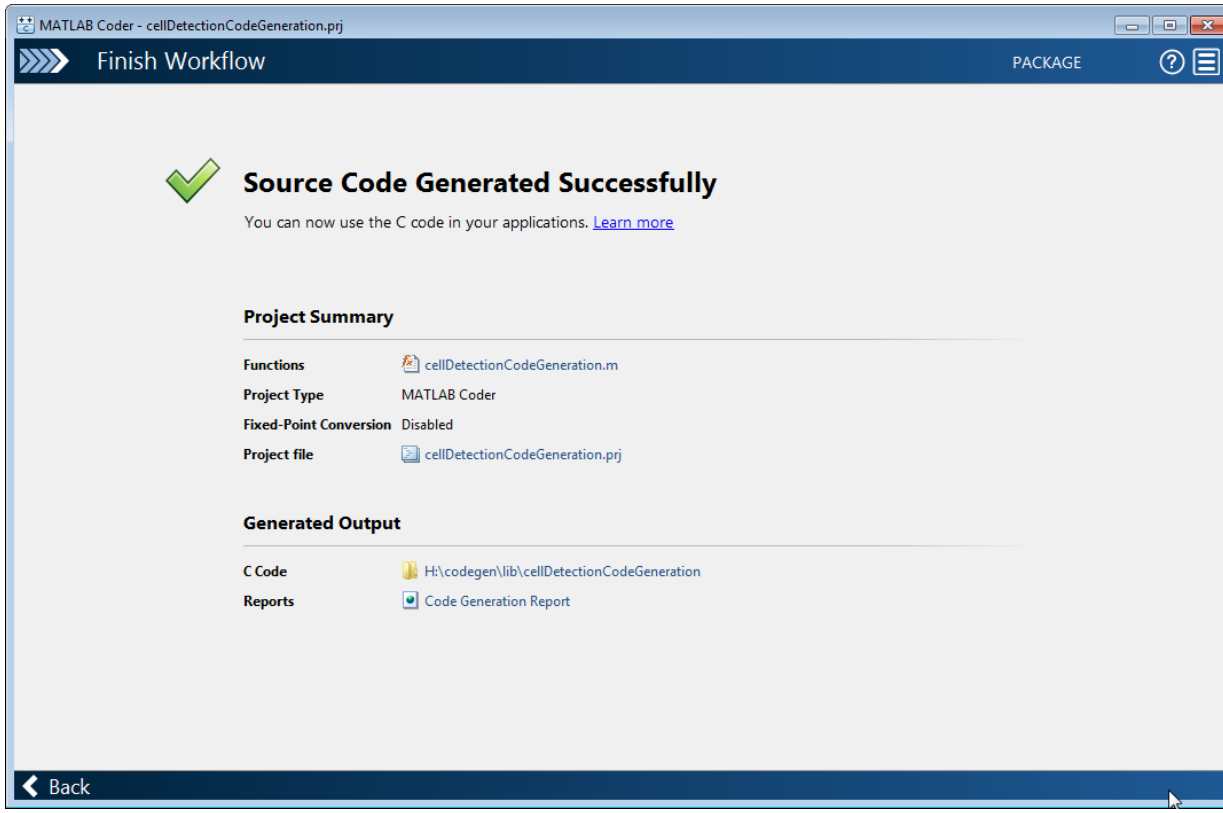
```
[ 1  2
  3  4 ]
```

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.

Click **Next** to complete the process. MATLAB Coder displays information about what it generated. By default, 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.



See Also

More About

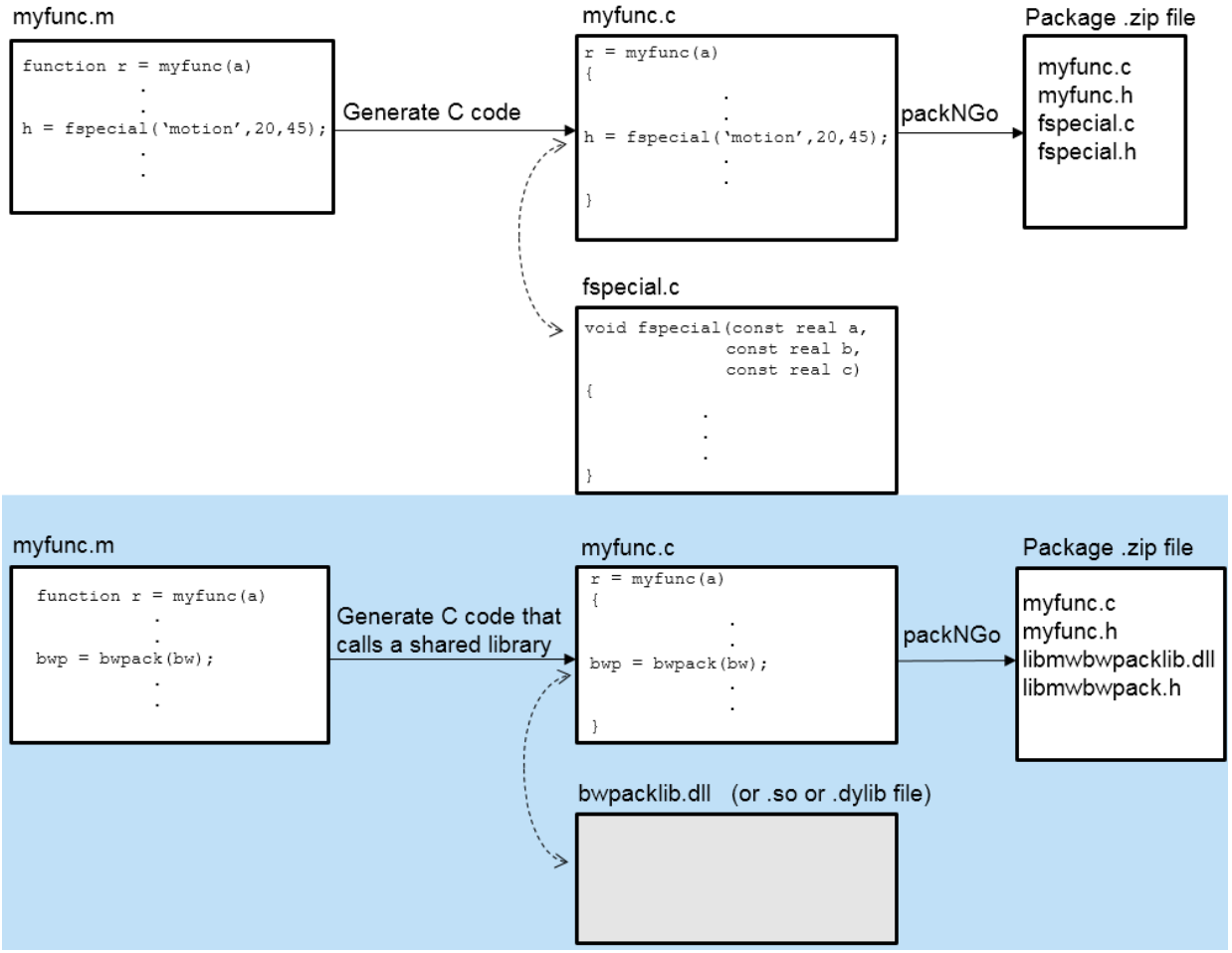
- “Code Generation Workflow” (MATLAB Coder)

Understand Code Generation with Image Processing Toolbox

The Image Processing Toolbox includes many functions that support the generation of efficient C code using MATLAB Coder. These functions support code generation in several ways:

- Some functions generate standalone C code that can be incorporated into applications that run on many platforms, such as ARM processors.
- Some functions generate 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 code or generate 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 C code.

The following diagram illustrates the difference between generating 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 Usage Notes” on page 18-3.

19

GPU Computing with Image Processing Toolbox Functions

- “Image Processing on a GPU” on page 19-2
- “List of Supported Functions with Limitations and Other Notes” on page 19-4
- “Perform Thresholding and Morphological Operations on a GPU” on page 19-7
- “Perform Element-Wise Operations on a GPU” on page 19-11

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 run image processing code on a graphics processing unit (GPU), you must have the Parallel Computing Toolbox software. To perform an image processing operation on a GPU, follow these steps:

- Move the data from the CPU to the GPU. Use the `gpuArray` function to transfer an array from MATLAB to the GPU. For more information, see “Create GPU Arrays from Existing Data” (Parallel Computing Toolbox).
- 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 19-4.
- Move the data back onto the CPU from the GPU. Use the `gather` function to retrieve an array from the GPU and transfer the array to the MATLAB workspace as a regular MATLAB array.

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.

To learn about integrating custom CUDA kernels directly into MATLAB to accelerate complex algorithms, see “Run CUDA or PTX Code on GPU” (Parallel Computing Toolbox).

See Also

`gather` | `gpuArray`

Related Examples

- “Perform Thresholding and Morphological Operations on a GPU” on page 19-7
- “Perform Element-Wise Operations on a GPU” on page 19-11

More About

- “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox)
- “List of Supported Functions with Limitations and Other Notes” on page 19-4

List of Supported Functions with Limitations and Other Notes

Hundreds of functions in MATLAB and dozens of functions in Image Processing Toolbox run automatically on a GPU if you supply a `gpuArray` argument. Whenever any of these functions is called with at least one `gpuArray` as an input argument, the function executes on the GPU and generates a `gpuArray` as the result. You can mix inputs using both `gpuArray` and MATLAB arrays in the same function call; the MATLAB arrays are transferred to the GPU for the function execution.

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
`bwdist`	Input images must be 2-D and have less than 2^{32} elements. Euclidean is the only distance metric supported.
`bwlabel`	—
`bwlookup`	—
`bwmorph`	—
`corr2`	—
`edge`	Canny method is not supported on the GPU.
`histeq`	—
`im2double`	—
`im2int16`	—
`im2single`	—
`im2uint8`	—
`im2uint16`	—
`imabsdiff`	—
`imadjust`	—
`imbothat`	`gpuArray` input must be of type `uint8` or `logical` and the structuring element must be flat and two-dimensional.

Function	Remarks/Limitations
`imclose`	`gpuArray` input must be of type `uint8` or `logical` and the structuring element must be flat and two-dimensional.
`imcomplement`	—
`imdilate`	`gpuArray` input must be of type `uint8` or `logical` and the structuring element must be flat and two-dimensional The `PACKOPT` syntaxes are not supported on the GPU.
`imerode`	`gpuArray` input must be of type `uint8` or `logical` and the structuring element must be flat and two-dimensional The `PACKOPT` syntaxes are not supported on the GPU.
`imfill`	Inputs must be 2-D, supporting only the 2-D connectivities (4 and 8). Does not support the interactive hole filling syntax.
`imfilter`	Input kernel must be 2-D
`imgaussfilt`	—
`imgaussfilt3`	—
`imgradient`	—
`imgradientxy`	—
`imhist`	When running on a GPU, `imhist` does not display the histogram. To display the histogram, use `stem(binLocations,counts)`.
`imlincomb`	—
`imnoise`	—
`imopen`	`gpuArray` input must be of type `uint8` or `logical` and the structuring element must be flat and two-dimensional.
`imreconstruct`	Inputs must be 2-D, supporting only the 2-D connectivities (4 and 8).
`imregdemons`	The parameter `'DisplayWaitbar'` is not supported on the GPU.
`imregionalmax`	Inputs must be 2-D, supporting only the 2-D connectivities (4 and 8).
`imregionalmin`	Inputs must be 2-D, supporting only the 2-D connectivities (4 and 8).
`imresize`	`gpuArray` input must be of type `single` or `double`. Only bicubic interpolation is supported on GPU and the function always performs antialiasing.

Function	Remarks/Limitations
`imrotate`	The `'bicubic'` interpolation mode used in the GPU implementation of this function differs from the default (CPU) bicubic mode. The GPU and CPU versions of this function are expected to give slightly different results.
`imshow`	—
`imtophat`	`gpuArray` input must be of type `uint8` or `logical` and the structuring element must be flat and two-dimensional.
`iradon`	The GPU implementation of this function supports only nearest-neighbor and linear interpolation methods.
`mat2gray`	—
`mean2`	—
`medfilt2`	Padding options are not supported on the GPU.
`normxcorr2`	—
`padarray`	—
`radon`	—
`rgb2gray`	—
`rgb2ycbcr`	—
`regionprops`	`gpuArray` input must be a 2-D logical matrix or a 2-D label matrix. The connected component structure (`CC`) input is not supported. The following properties are not supported: `'ConvexArea'`, `'ConvexHull'`, `'ConvexImage'`, `'EulerNumber'`, `'FilledArea'`, `'FilledImage'`, and `'Solidity'`. `'struct'` is the only return type supported.
`std2`	—
`stdfilt`	The filtering neighborhood must be two-dimensional.
`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.

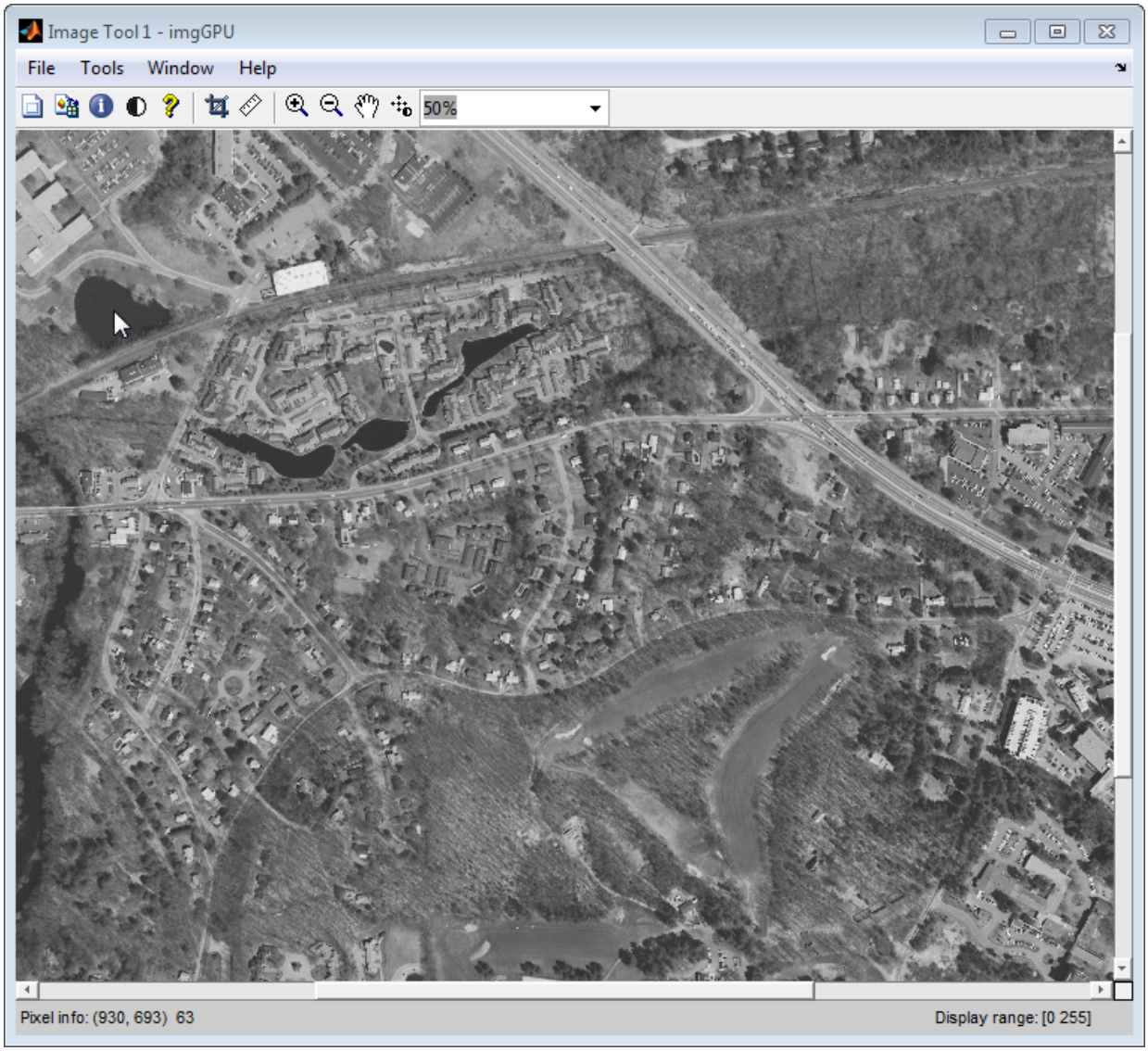
```
imGPU = gpuArray(imCPU);
```

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, then a function that has been GPU-enabled performs the operation on the GPU.

```
imGPUgray = rgb2gray(imGPU);
```

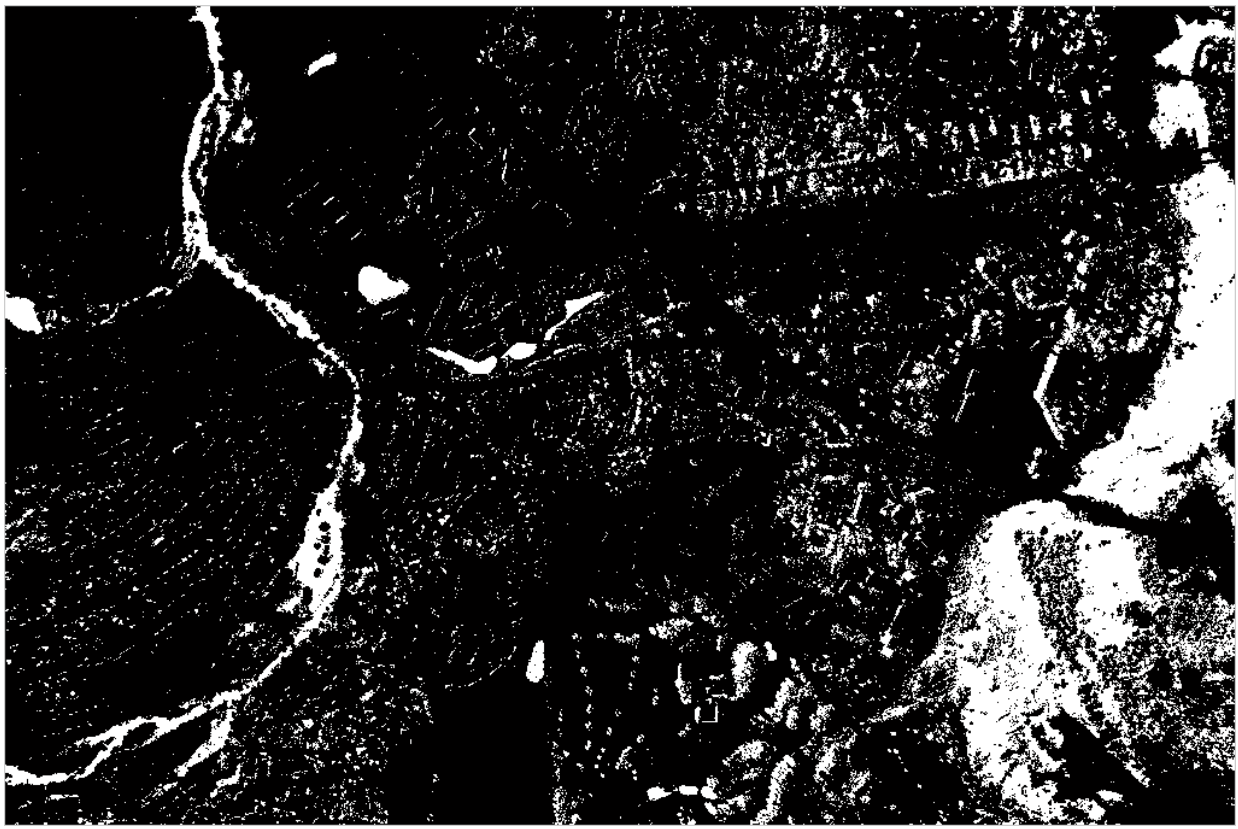
View the image in the **Image Viewer** and inspect the pixel values to find the value of watery areas. Note that you must bring the image data back onto the CPU, using the `gather` function, to use the Image Viewer. As you move the mouse over the image, you can view the value of the pixel under the cursor 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 by 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` object.

```
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);
```