Image Processing Toolbox™
User's Guide

MATLAB®

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.
1 Apple Hill Drive
Natick, MA 01760-2098

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)
March 2019	Online only	Revised for Version 10.4 (Release 2019a)
September 2019	Online only	Revised for Version 11.0 (Release 2019b)
March 2020	Online only	Revised for Version 11.1 (Release 2020a)
September 2020	Online only	Revised for Version 11.2 (Release 2020b)

Contents

1

Getting Started

2

Introduction

3

Reading and Writing Image Data

Displaying and Exploring Images

4

Building GUIs with Modular Tools

5

Geometric Transformations

6

Image Registration

7

Designing and Implementing Linear Filters for Image Data

8

Transforms

9

11

ROI-Based Processing

12

Image Segmentation

13

14

15

Big Image Processing

16

Neighborhood and Block Operations

17

Deep Learning

18

Hyperspectral Image Processing

19

Code Generation for Image Processing Toolbox Functions

20

GPU Computing with Image Processing Toolbox Functions

21

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

Perform image processing, visualization, and analysis

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, and image registration using deep learning and traditional image processing techniques. The toolbox supports processing of 2D, 3D, and arbitrarily large images.

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

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)
```
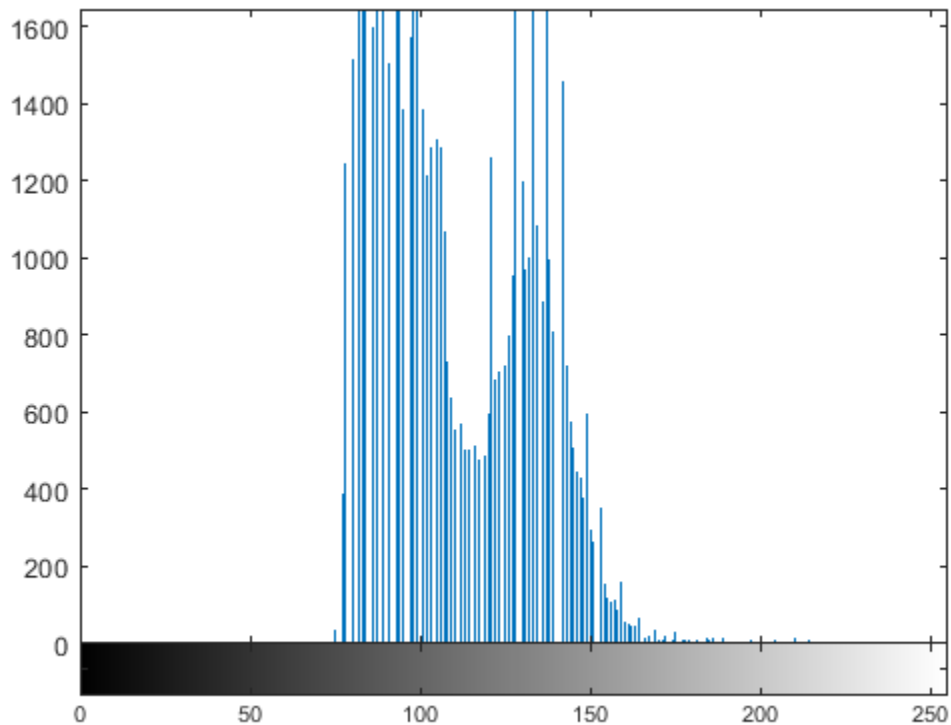

Improve the contrast in an image, using the `histeq` function. Histogram equalization spreads the intensity values over the full range of the image. Display the image. (The toolbox includes several other functions that perform contrast adjustment, including `imadjust` and `adapthisteq`, and interactive 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)
```
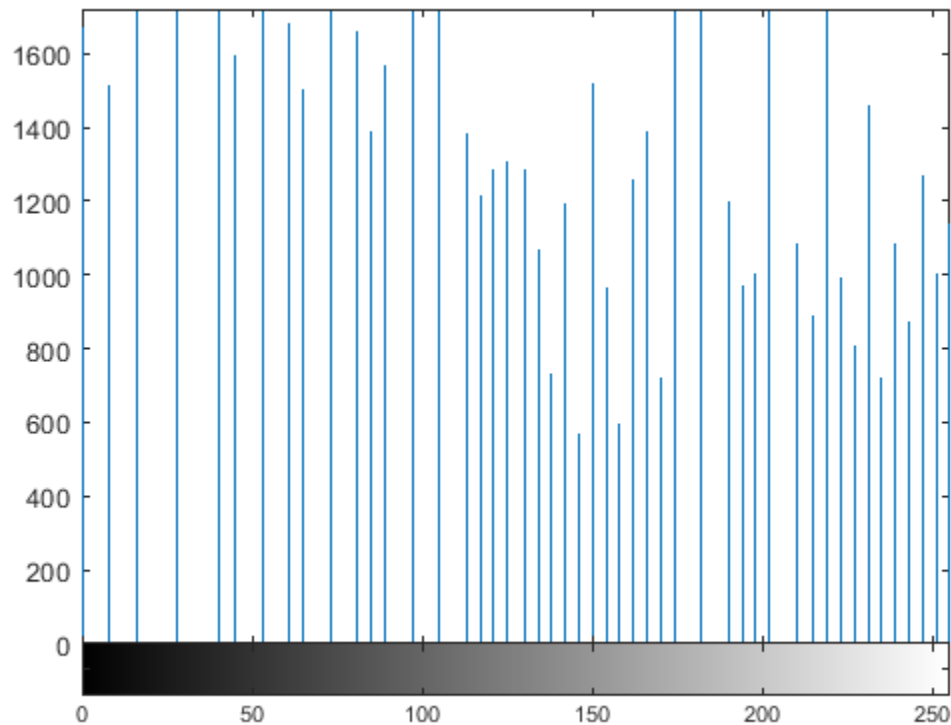
Step 4: Write the Adjusted Image to a Disk File

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\Bdoc20b_1465442_5924\ib8F4264\0\tpd8fc74e3\images-ex8950508(
             FileModDate: '25-Aug-2020 12:20:40'
                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: '25 Aug 2020 16:20:40 +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)
```
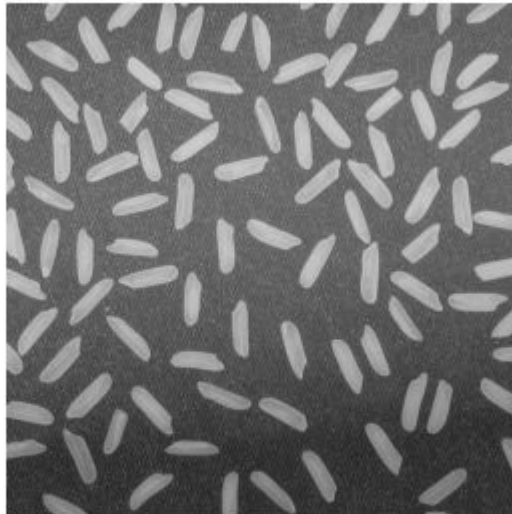

The background illumination is brighter in the center of the image than at the bottom. Preprocess the image to make the background illumination more uniform.

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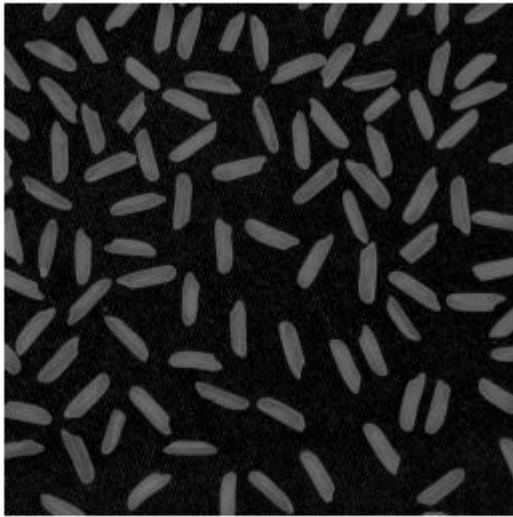
Use `imadjust` to increase the contrast of the processed image `I2` by saturating 1% of the data at both low and high intensities and by stretching the intensity values to fill the `uint8` dynamic range.

```
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)
```
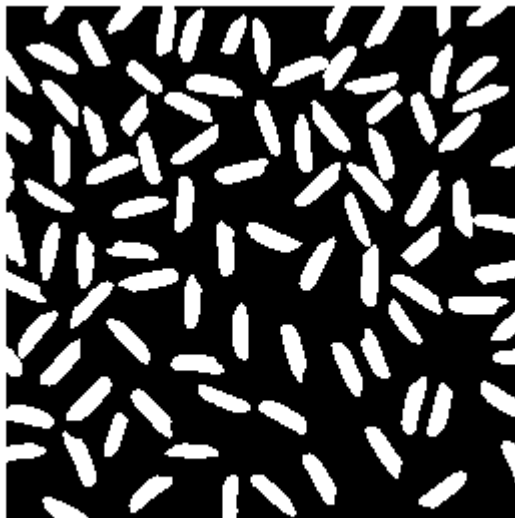

Identify Objects in the Image

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)
```
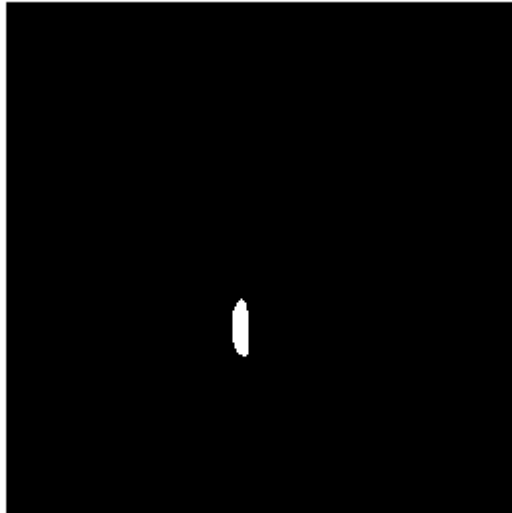

Visualize all the connected components in the image by creating a label matrix and then displaying it as a pseudocolor indexed image.

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)
```
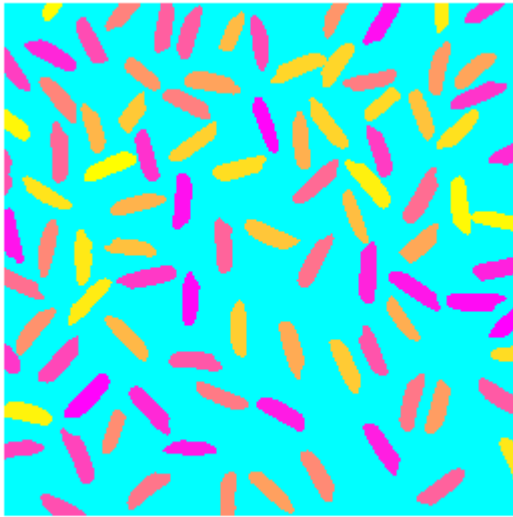
Compute Area-Based Statistics

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=95×1 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')
```

Histogram of Rice Grain Area
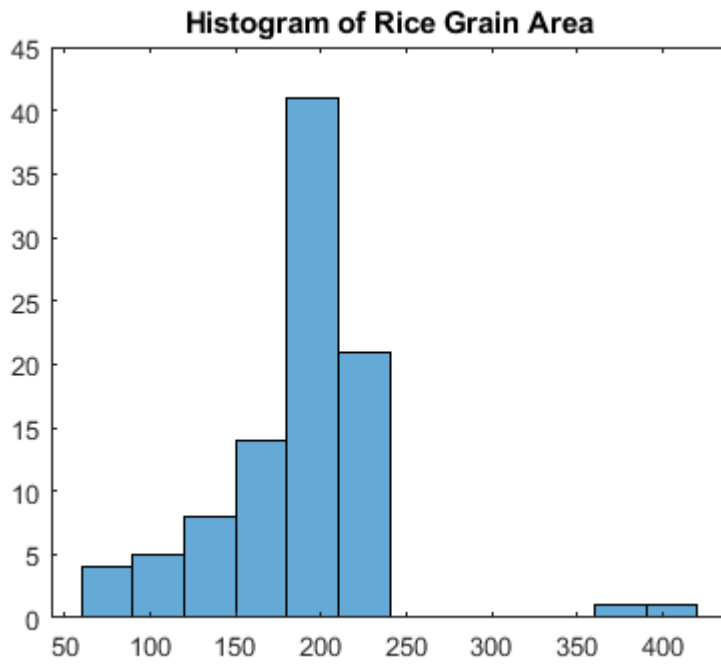
See Also
bwareaopen | bwconncomp | imadjust | imbinarize | imopen | imread | imshow | label2rgb | labelmatrix | regionprops

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

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 1-4

More About

- "Image Types in the Toolbox" on page 2-12

Image Coordinate Systems

You can access locations in images using several different image coordinate systems. You can specify locations using discrete pixel indices because images are stored as arrays. You can also specify locations using continuous spatial coordinates because images represent real-world scenes in continuous space.

Pixel Indices

As described in "Images in MATLAB" on page 2-2, MATLAB stores most images as arrays. Each (row, column) index of the array corresponds to a single pixel in the displayed image.

There is a one-to-one correspondence between pixel indices and subscripts for the first two matrix dimensions. Just like for array indexing in MATLAB, pixel indices are integer values and range from 1 to the length of the row or column. The indices are ordered from top to bottom, and from left to right.
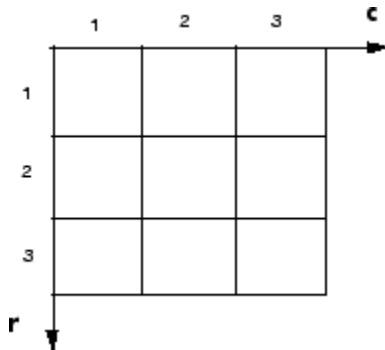
For example, the data for the pixel in the fifth row, second column is stored in the matrix element (5,2). You use normal MATLAB matrix subscripting to access values of individual pixels. For example, the MATLAB code

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

returns the value of the pixel at row 2, column 15 of the single-channel image I. Similarly, the MATLAB code

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

returns the color values of the pixel at row 2, column 15 of the multi-channel image RGB.

Spatial Coordinates

In a *spatial coordinate system*, locations in an image are positions on a continuous plane. Locations are described in terms of Cartesian *x* and *y* coordinates (not row and column indices as in the pixel indexing system). From this Cartesian perspective, an (*x*,*y*) location such as (3.2,5.3) is meaningful and distinct from the coordinate (5,3).

The Image Processing Toolbox defines two types of spatial coordinate systems depending on the frame of reference. Intrinsic coordinates specify locations with respect to the image's frame of reference. World coordinates specify locations with respect to an external world observer.

Intrinsic Coordinates

By default, the toolbox defines spatial image coordinates using the *intrinsic coordinate system*. This spatial coordinate system corresponds to the image's pixel indices. 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 $x = 3.0$, $y = 5.0$. Be aware, however, that the order of order of intrinsic coordinate (3.0,5.0) is reversed relative to pixel indices (5,3).
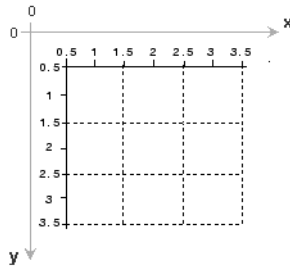
The intrinsic coordinates of the center of every pixel are integer valued. The center of the upper left pixel has intrinsic coordinates (1.0, 1.0). The center of the lower right pixel has intrinsic coordinates (*numCols*, *numRows*), where *numCols* and *numRows* are the number of rows and columns in the image. In general, the center of the pixel with pixel indices (m, n) falls at the point $x = n$, $y = m$ in the intrinsic coordinate system.

Because the size of each pixel in the intrinsic coordinate system is one unit, the boundaries of the image have fractional coordinates. The upper left corner of the image is located at (0.5,0.5), not at (0,0). Similarly, the lower right corner of the image is located at (`numCols` + 0.5, `numRows` + 0.5).

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).

World Coordinates

In some situations, you might want to use a *world coordinate system* (also called a nondefault spatial coordinate system). Some situations when you might want to use a world coordinate system include:

- When you perform a geometric operation, such as translation, on an image and want to preserve information about how the new position relates to the original position.
- When pixels are not square. For example, in magnetic resonance imaging (MRI), data you can collect data such that pixels have a higher sampling rate in one direction than an orthogonal direction.
- When you know how the extent of pixels aligns with positions in the real world. For example, in an aerial photograph, every pixel might cover a specific 5-by-5 meter patch on the ground.
- When you want to reverse the direction of the x-axis or y-axis. This is a common technique to use with geospatial data.

There are several ways to define a world coordinate system. You can use spatial referencing objects, which encode the location of the image in a world coordinate system, the image resolution, and how the image extent relates to intrinsic and world coordinates. You can also specify the maximum and minimum coordinate in each dimension. For more information, see "Define World Coordinate System of Image" on page 2-6.

See Also

Related Examples

- "Shift X- and Y-Coordinate Range of Displayed Image" on page 2-9

More About

- "Images in MATLAB" on page 2-2
- "Define World Coordinate System of Image" on page 2-6

Define World Coordinate System of Image

The world coordinate system is a continuous spatial coordinate system that specifies the location in an image independently of the pixel indices of the image. For more information about coordinate systems in Image Processing Toolbox, see "Image Coordinate Systems" on page 2-3.

Define Spatial Referencing Objects

To specify a world coordinate system for an image, you can use spatial referencing objects. Spatial referencing objects define the location of the image in a world coordinate system and specify how the extents of the image relate to intrinsic and world limits. You can use these objects to specify nonsquare pixel dimensions by specifying a different image resolution in each dimension. Spatial referencing objects also enable you to convert between coordinate systems.

Image Processing Toolbox uses includes two spatial referencing objects, `imref2d` and `imref3d`. The table describes the properties of the 2-D spatial referencing object, `imref2d`. The 3-D spatial referencing object, `imref3d`, includes these properties as well as corresponding properties for the *Z* dimension.

Property	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 spatial referencing, this sample code creates a spatial referencing object associated with a 2-by-2 image. The code specifies the pixel extent in the horizontal and vertical directions as 4 units/pixel and 2 units/pixel, respectively. The object calculates the world limits, image extent in world coordinates, and image extent in intrinsic coordinates.

```
R = imref2d([2 2],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 figure illustrates how these properties map to elements of the image.
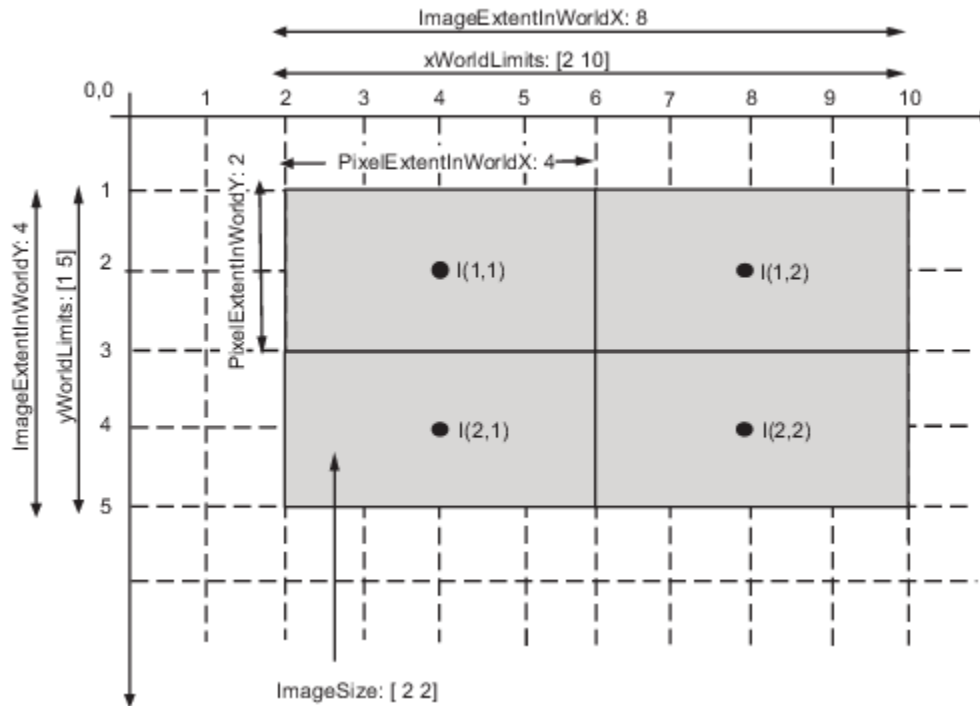
Specify Minimum and Maximum Image Extent

Image objects (such as obtained when displaying an image using imshow) define the world extent using the XData and YData properties. Each property is a two-element vector that specifies the center coordinate of the outermost pixels in that dimension. For more information, see Image.

By default, the intrinsic coordinates, world coordinates, and MATLAB axes coordinates of an image coincide. For an image A, the default value of XData is [1 size(A,2)] and the default value of YData is [1 size(A,1)]. For example, if A is a 100 row by 200 column image, the default XData is [1 200] and the default YData is [1 100].

To define a nondefault world coordinate system for an image, specify the image XData and YData properties with the range of coordinates spanned by the image in each dimension. When you do this, the MATLAB axes coordinates become identical to the world coordinates and no longer coincide with the intrinsic coordinates. For an example, see "Shift X- and Y-Coordinate Range of Displayed Image" on page 2-9.

Note that the values in XData and YData are actually the coordinates for the center point of the boundary pixels, not the outermost edge of the boundary pixels. Therefore, the actual coordinate range spanned by the image is slightly larger. For instance, if XData is [1 200] and the image is 200 pixels wide, as for the intrinsic coordinate system, then each pixel is one unit wide and the interval in X spanned by the image is [0.5 200.5]. Similarly, if XData is [1 200] and the image is 50 pixels wide,

as for a nondefault world coordinate system, then each pixel is four units wide and the interval in X spanned by the image is [–1 202].

You can 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 XData to [200 1].

See Also

imref2d | imref3d | imregister | imregtform | imshow | imwarp

Related Examples

- "Shift X- and Y-Coordinate Range of Displayed Image" on page 2-9

More About

- "Image Coordinate Systems" on page 2-3

Shift X- and Y-Coordinate Range of Displayed Image

This example shows how to specify a nondefault world coordinate system by changing the `XData` and `YData` properties of a displayed image.

Read an image.

```
I = imread("peppers.png");
```

Display the image using the intrinsic coordinate system, returning properties of the image in `ax`. Turn on the axis to display the coordinate system.

```
figure
ax = imshow(I);
title('Image Displayed with Intrinsic Coordinates')
axis on
```
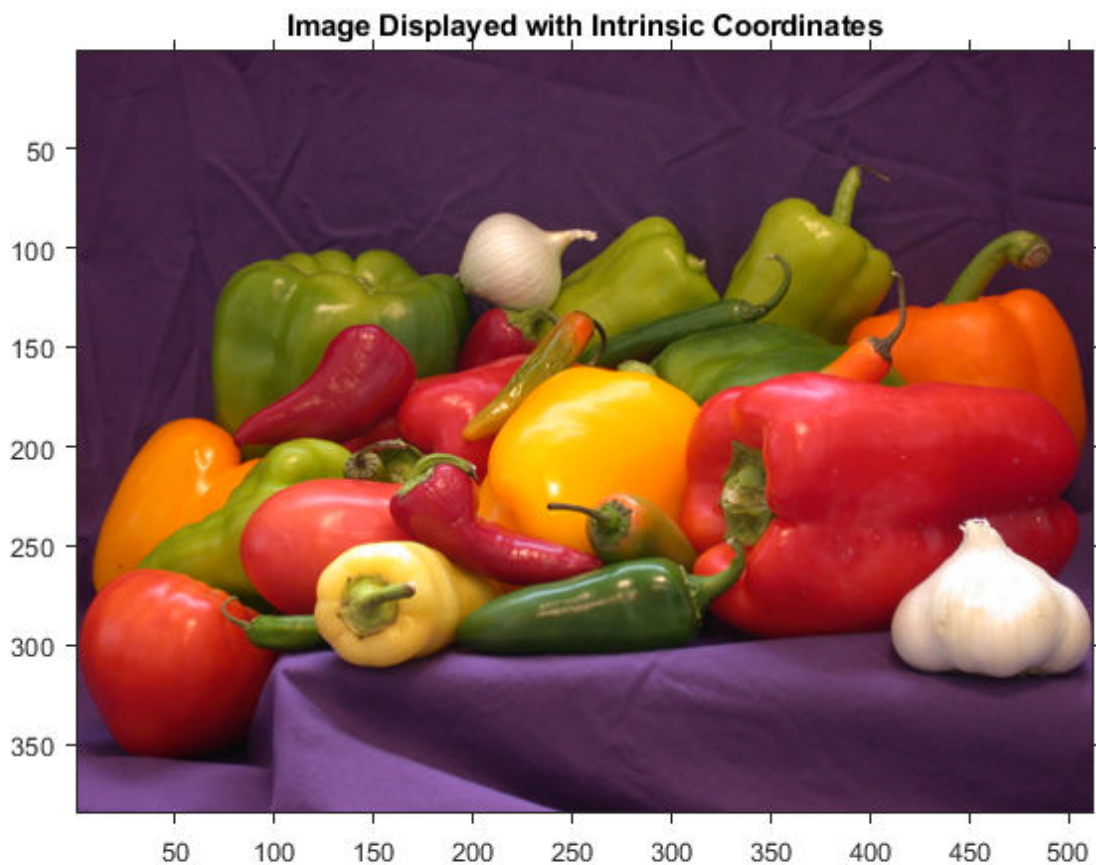

Check the range of the x- and y-coordinates, which are stored in the XData and YData properties of `ax`. The ranges match the dimensions of the image.

```
xrange = ax.XData
```

xrange = 1×2

```
       1    512
```

```
yrange = ax.YData
```

```
yrange = 1×2
```

```
       1    384
```

Change the range of the x- and y-coordinates. This example shifts the image to the right by adding 100 to the x-coordinates and shifts the image up by subtracting 25 from the y-coordinates.

```
xrangeNew = xrange + 100;
yrangeNew = yrange - 25;
```

Display the image, specifying the shifted spatial coordinates.

```
figure
axNew = imshow(I,'XData',xrangeNew,'YData',yrangeNew);
title('Image Displayed with Nondefault Coordinates');
axis on
```
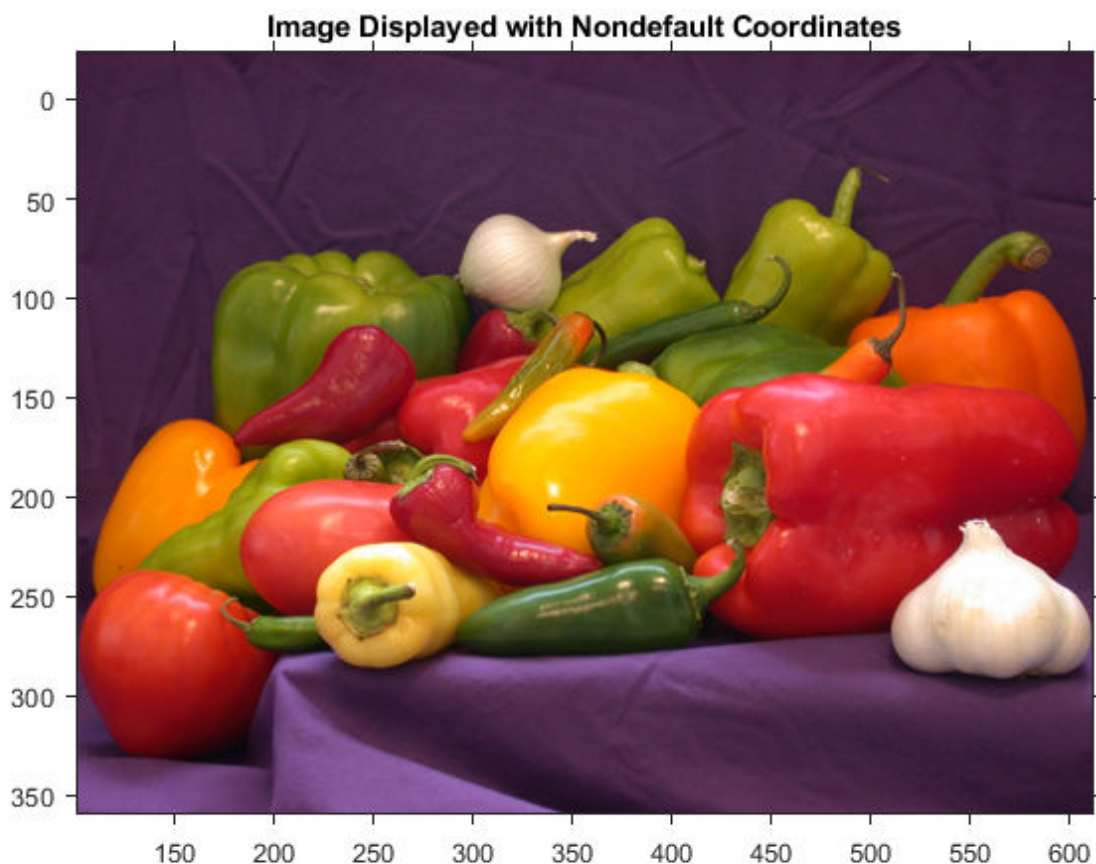

Confirm that the ranges of the x- and y-coordinates of the new image match the shifted ranges specified by xrangeNew and yrangeNew.

```
axNew.XData
```

ans = *1×2*

```
   101   612
```

```
axNew.YData
```

ans = *1×2*

```
   -24   359
```

See Also

More About

- "Image Coordinate Systems" on page 2-3
- "Define World Coordinate System of Image" on page 2-6

Image Types in the Toolbox

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

All images in Image Processing Toolbox are assumed to have nonsparse values. Numeric and logical images are expected assumed to be real-valued unless otherwise specified.

Image Type	Interpretation
"Binary Images" on page 2-13	Image data are stored as an *m*-by-*n* logical matrix in which values of 0 and 1 are interpreted as black and white, respectively. Some toolbox functions can also interpret an *m*-by-*n* numeric matrix as a binary image, where values of 0 are black and all nonzero values are white.
"Indexed Images" on page 2-13	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 color map is a *c*-by-3 array of class `double`.
"Grayscale Images" on page 2-14 (Also known as *intensity* images)	Image data are stored as an *m*-by-*n* numeric matrix whose elements specify intensity values. The smallest value indicates black and the largest value indicates white. • 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].
"Truecolor Images" on page 2-15 (Commonly referred to as *RGB* images)	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 15-15.

Image Type	Interpretation
High Dynamic Range (HDR) Images on page 2-15	HDR images are stored as an *m*-by-*n* numeric matrix or *m*-by-*n*-by-3 numeric array, similar to grayscale or RGB images, respectively. HDR images have data type `single` or `double` but data values are not limited to the range [0, 1] and can contain `Inf` values. For more information, see "Work with High Dynamic Range Images" on page 3-29.
Multispectral and Hyperspectral Images on page 2-16	Image data are stored as an *m*-by-*n*-by-*c* numeric array, where *c* is the number of color channels.
Label Images on page 2-17	Image data are stored as an *m*-by-*n* categorical matrix or numeric matrix of nonnegative integers.

Binary Images

In a binary image, each pixel has one of only two discrete values: 1 or 0. Most functions in the toolbox interpret pixels with value 1 as belonging to a region of interest, and pixels with value 0 as the background. Binary images are frequently used in conjunction with other image types to indicate which portions of the image to process.

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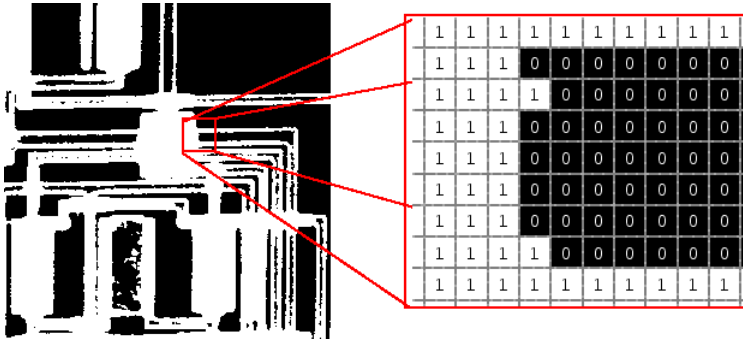
Indexed Images

An indexed image consists of an image matrix and a color map.

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

The pixel values in the image matrix are direct indices into the color map. Therefore, the color of each pixel in the indexed image is determined by mapping the pixel value in the image matrix to the corresponding color in the color map. The mapping depends on the class of the image matrix:

- If the image matrix is of class `single` or `double`, the color map normally contains integer values in the range [1, *p*], where *p* is the length of the color map. The value 1 points to the first row in the color map, the value 2 points to the second row, and so on.
- If the image matrix is of class `logical`, `uint8` or `uint16`, the color map normally contains integer values in the range [0, *p*–1]. The value 0 points to the first row in the color map, the value 1 points to the second row, and so on.

A color map 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 color map into the workspace as separate variables, you must keep track of the association between the image and color map. However, you are not limited to using the default color map—you can use any color map that you choose.

The figure illustrates an indexed image, the image matrix, and the color map, respectively. The image matrix is of class `double`, so the value 7 points to the seventh row of the color map.
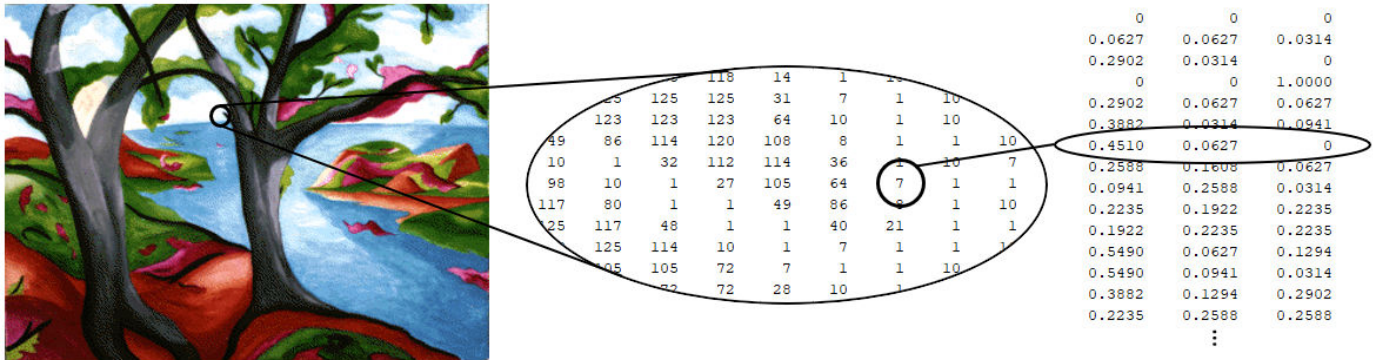
Grayscale Images

A grayscale image is a data matrix whose values represent intensities of one image pixel. While grayscale images are rarely saved with a color map, MATLAB uses a color map to display them.

You can obtain a grayscale image directly from a camera that acquires a single signal for each pixel. You can also convert truecolor or multispectral images to grayscale to emphasize one particular aspect of the images. For example, you can take a linear combination of the red, green, and blue channels of an RGB image such that the resulting grayscale image indicates the brightness, saturation, or hue of each pixel. You can process each channel of a truecolor or multispectral image independently by splitting the channels into separate grayscale images.

The figure depicts a grayscale image of class `double` whose pixel values are in the range [0, 1].
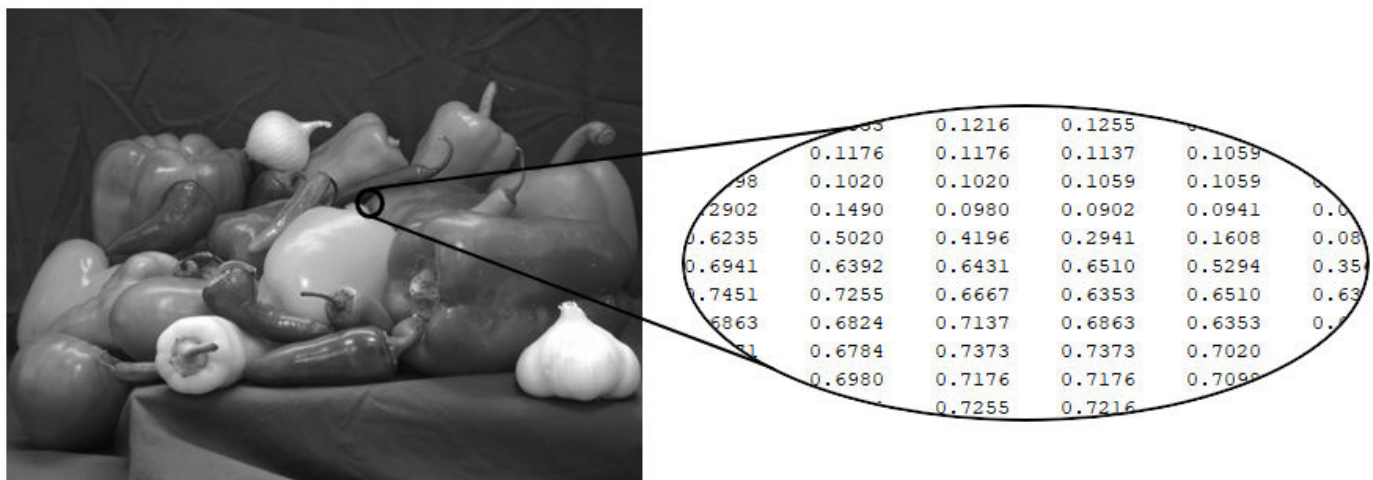
Truecolor Images

A truecolor image is an image in which each pixel has a color specified by three values. Graphics file formats store truecolor images as 24-bit images, where three color channels 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.

RGB images are the most common type of truecolor images. In RGB images, the three color channels are red, green, and blue. For more information about the RGB color channels, see "Display Separated Color Channels of RGB Image" on page 2-18.

There are other models, called color spaces, that describe colors using three different 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 15-15.

Truecolor images do not use a color map. The color of each pixel is determined by the combination of the intensities stored in each color channel at the pixel's location.

The figure depicts the red, green, and blue channels of a floating-point RGB image. Observe that pixel values are in the range [0, 1].

To determine the color of the pixel at (row, column) coordinate (2,3), you would look at the RGB triplet stored in the vector (2,3,:). 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`

HDR Images

Dynamic range refers to the range of brightness levels. 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.

The figure depicts the red, green, and blue channels of a tone-mapped HDR image with original pixel values in the range [0, 3.2813]. Tone mapping is a process that reduces the dynamic range of an HDR image to the range expected by a computer monitor or screen.

Multispectral and Hyperspectral Images

A multispectral image is a type of color image that stores more than three channels. For example, a multispectral image can store three RGB color channels and three infrared channels, for a total of six channels. The number of channels in a multispectral image is usually small. In contrast, a hyperspectral image can store dozens or even hundreds of channels.

The figure depicts a multispectral image with six channels consisting of red, green, blue color channels (depicted as a single RGB image) and three infrared channels.
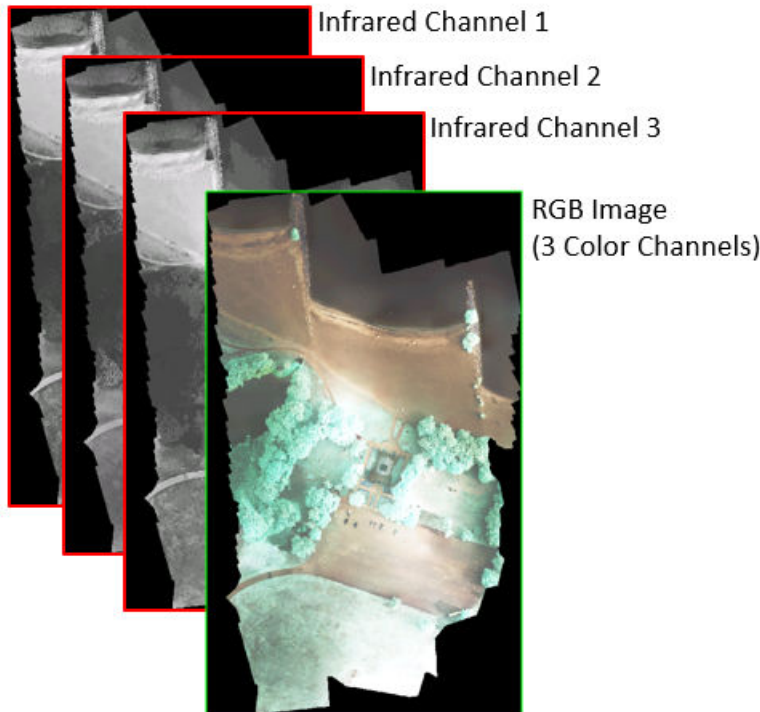
Label Images

A label image is an image in which each pixel specifies a class, object, or region of interest (ROI). You can derive a label image from an image of a scene using segmentation techniques.

- A numeric label image enumerates objects or ROIs in the scene. Labels are nonnegative integers. The background typically has the value 0. The pixels labeled 1 make up one object; the pixels labeled 2 make up a second object; and so on.
- A `categorical` label image specifies the class of each pixel in the image. The background is commonly assigned the value `<undefined>`.

The figure depicts a label image with three categories: petal, leaf, and dirt.
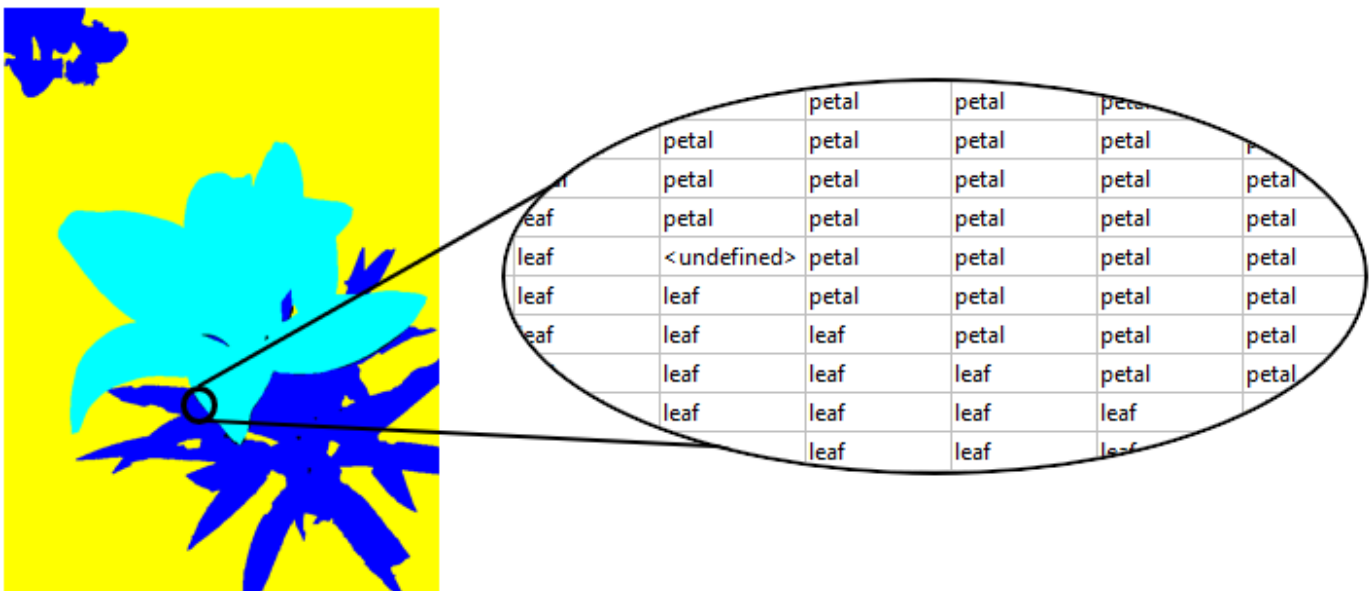
See Also

More About

Display Separated Color Channels of RGB Image

This example creates a simple RGB image and then separates the color channels. The example displays each color channel as a grayscale intensity image and as a color image.

Create an RGB image with uninterrupted areas of red, green, and blue. Display the image.

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


Separate the three color channels.

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

Display a grayscale representation of each color channel. 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, in other words, when R == 0.

```
figure
subplot(1,3,1)
imshow(R)
title('Red Channel')

subplot(1,3,2)
imshow(G)
title('Green Channel')

subplot(1,3,3)
imshow(B)
title('Blue Channel')
```
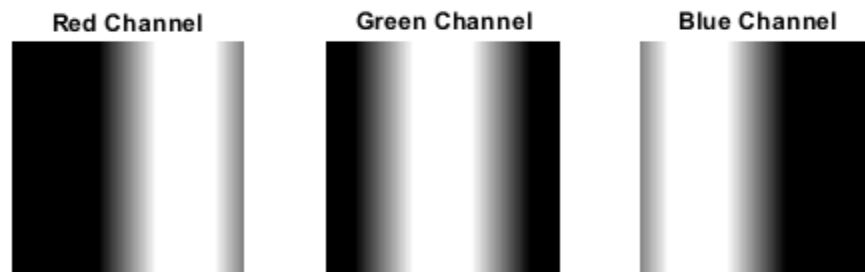
Display a color representation of each color channel. In these images, the desired color channel maintains its original intensity values and pixel values in the other two color channels are set to 0.

Create an all-black channel.

```
allBlack = zeros(size(RGB,1,2),class(RGB));
justR = cat(3,R,allBlack,allBlack);
justG = cat(3,allBlack,G,allBlack);
justB = cat(3,allBlack,allBlack,B);
```

Display all the channels in a montage.

```
figure
montage({justR,justG,justB},'Size',[1 3], ...
    "BackgroundColor",'w',"BorderSize",10);
title('Color Representation of the Red, Green, and Blue Color Channels');
```

Color Representation of the Red, Green, and Blue Color Channels
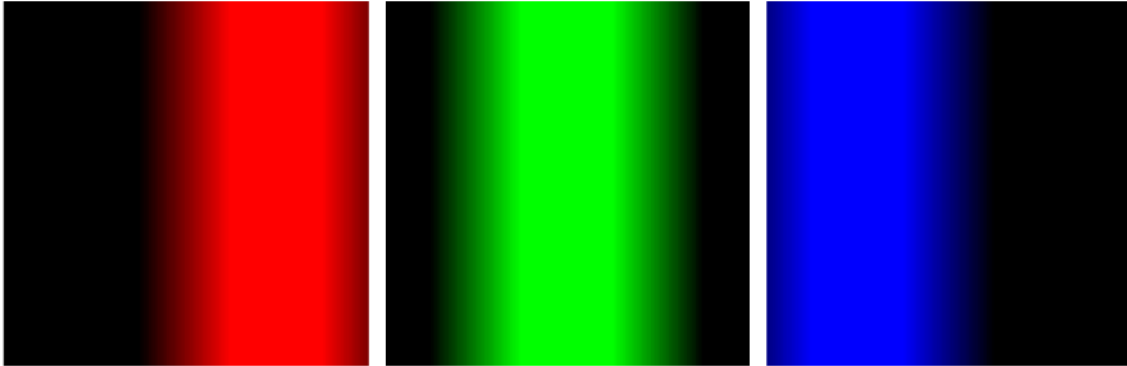
See Also
`imsplit`

More About

- "Image Types in the Toolbox" on page 2-12

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.
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 15-15.
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 color map, 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 color map, to reduce the number of distinct colors in the image. See "Reduce Colors of Indexed Image Using imapprox" on page 15-7 for more information.

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
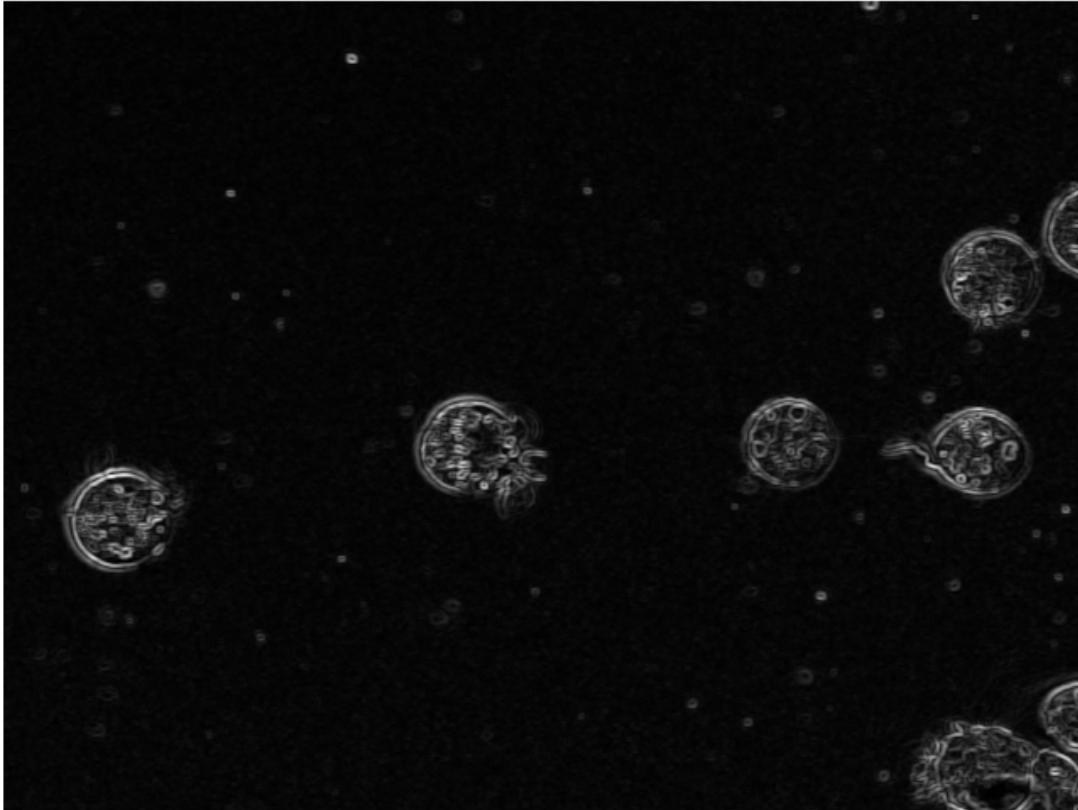
Detecting Cars in a Video of Traffic

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 `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
      NumFrames: 120

   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
      NumFrames: 120

   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');`
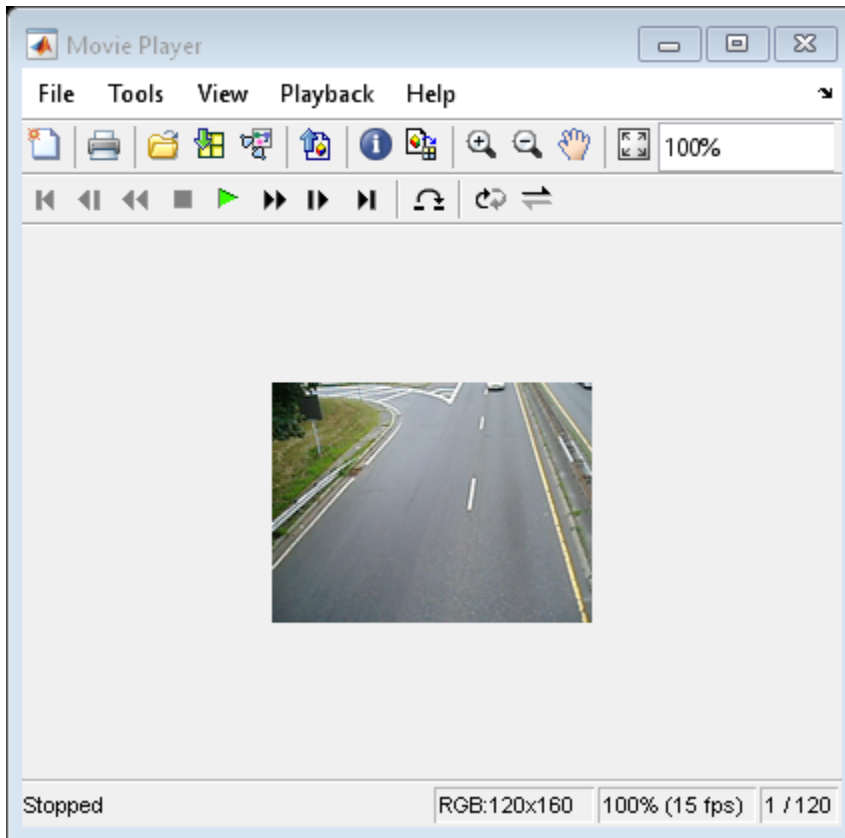
Step 3: Develop Your Algorithm

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 "Process Image Sequences" on page 2-49.

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);
```
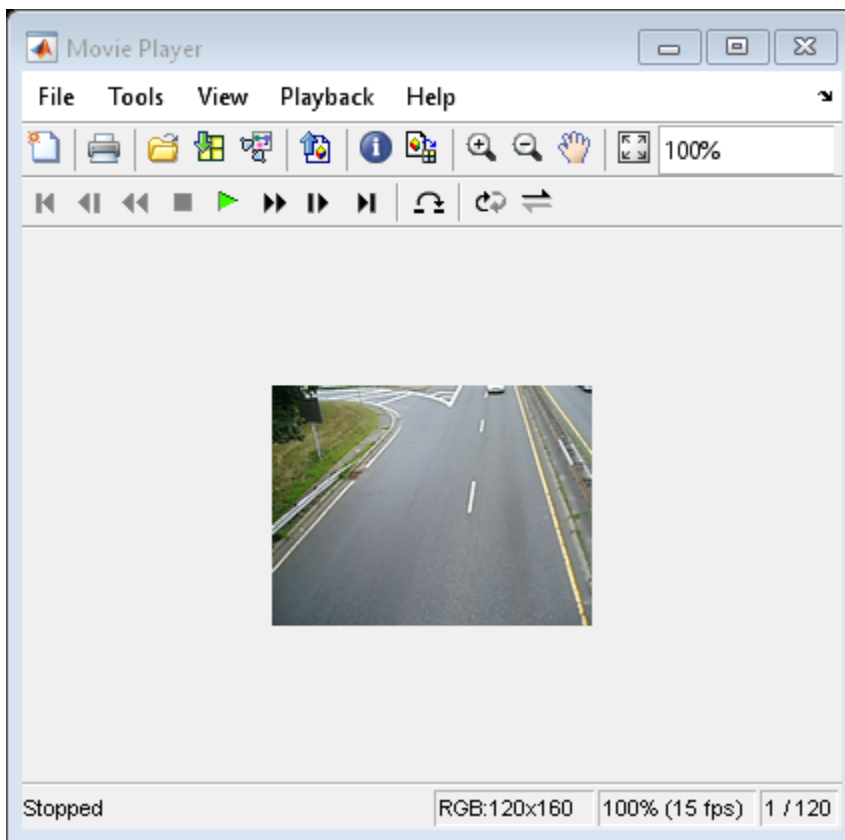

See Also

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

More About

- "Work with Image Sequences as Multidimensional Arrays" on page 2-48
- "Perform an Operation on a Sequence of Images" on page 2-23

Process Folder of Images Using 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.

Create a new folder in an area where you have write permission and copy a set of 10 images from the **Image Processing Toolbox** imdata folder into the new folder.

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

Load Images into Image Batch Processor App

Open the **Image Batch Processor** app from the MATLAB® toolstrip. On the **Apps** tab, in the **Image Processing and Computer Vision** section, click **Image Batch Processor** .
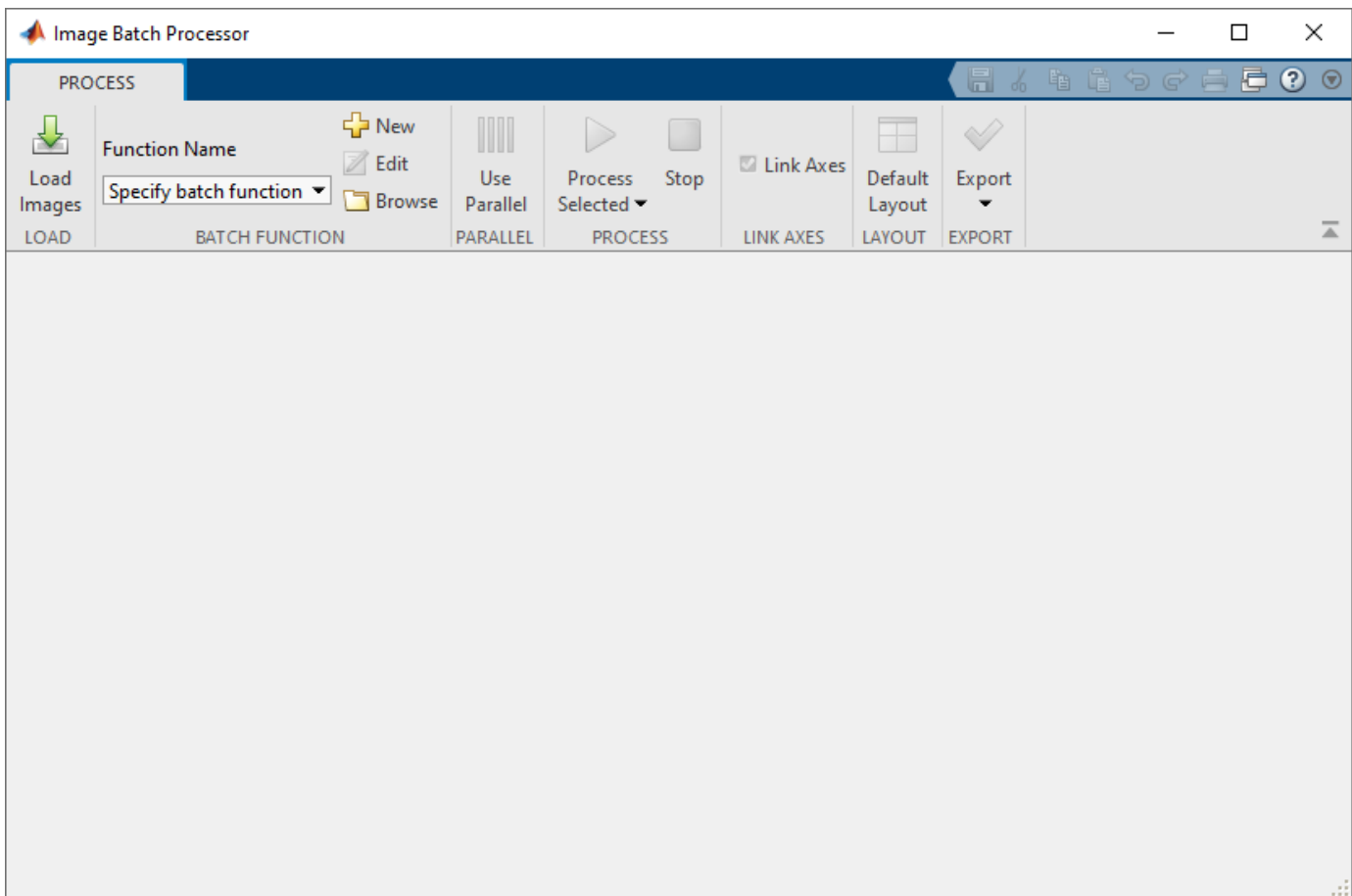
Load images into the app. In the app toolstrip, click **Load Images**. In the Load Images from Folder dialog box, specify the folder containing the images you want to load. For this example, specify the folder that you created in the first step, cellprocessing. By default, the app includes images in subfolders. To change this behavior, clear **Include images in subfolders**. Then, click **Load**.

The **Image Batch Processor** app creates thumbnails of the images in the folder and displays them in a scrollable tab in the left pane. The app displays the first selected image (highlighted in blue) in larger resolution in the Input Image tab in the right pane.
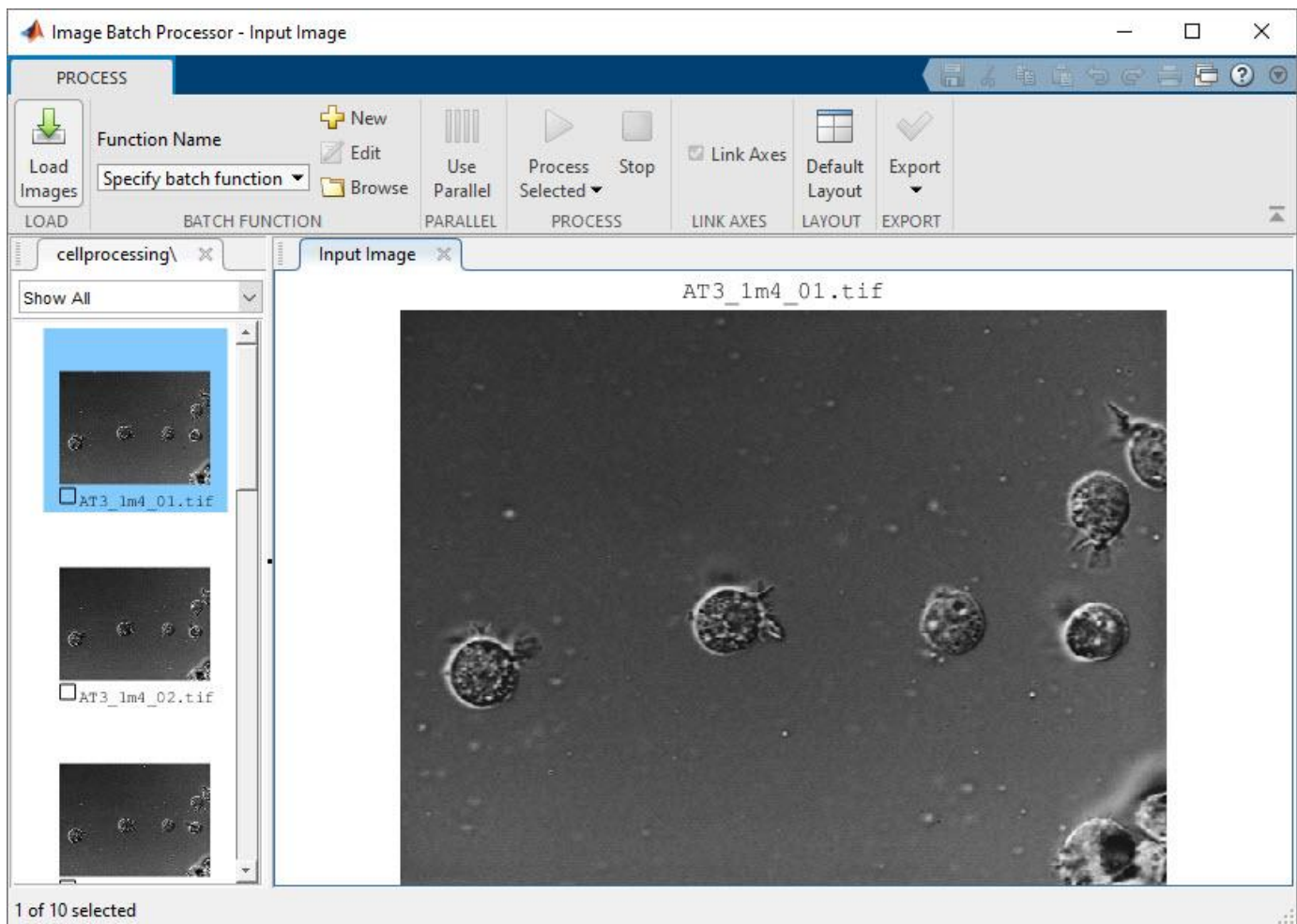
Specify Batch Processing Function

Specify the name of the function you want to use to process the images in the folder. To specify an existing function, type the name in the **Function Name** box in the Batch Function section of the app toolstrip. You can also click the folder icon next to the box to browse and select the function. To create a new batch processing function, click **New** in the Batch Function section of the app toolstrip.

When you do this, the app opens the batch processing function template in the MATLAB® Editor. For this example, click **New** to create a new function.
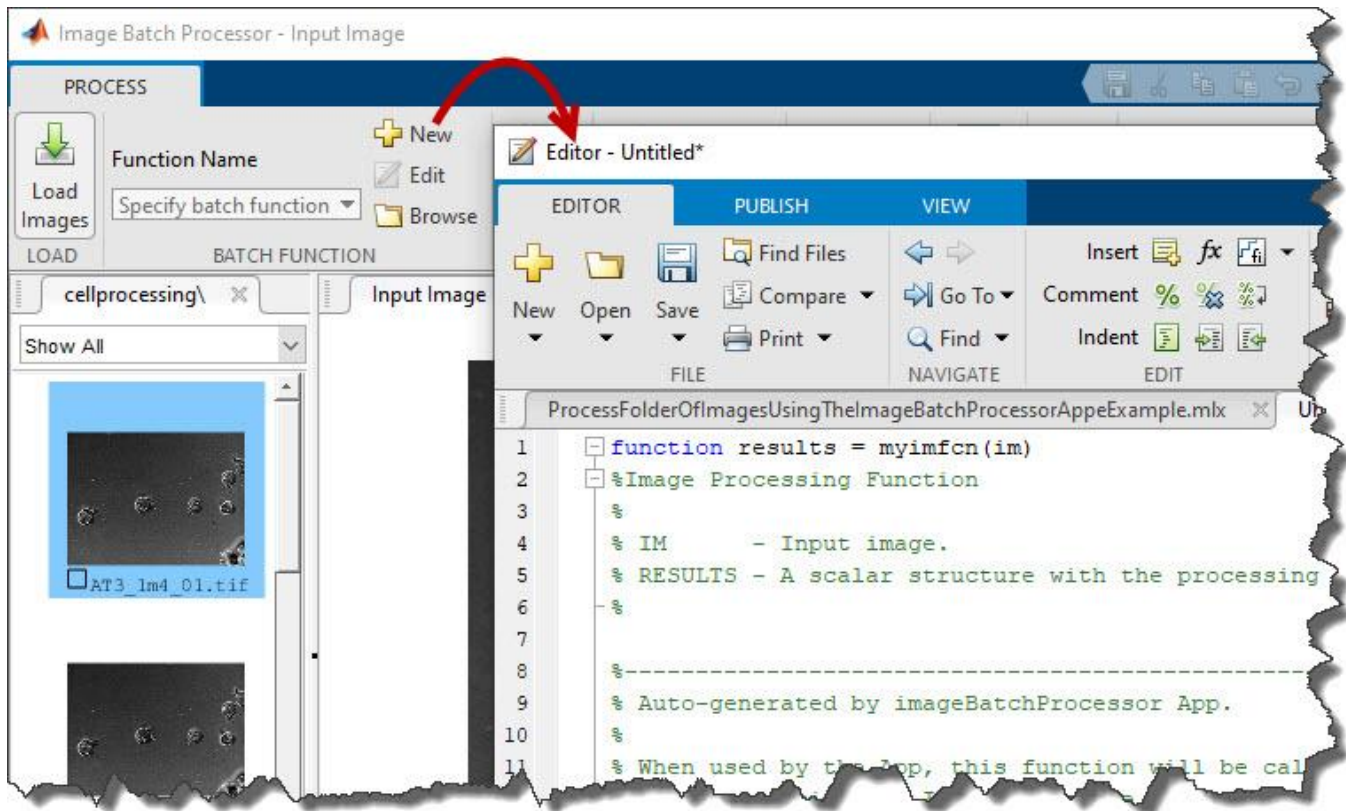
In the batch processing function template, enter code for the new function into the space reserved in the template file and click **Save**. This example uses the default name for the batch processing function, `myimfcn`, but you can specify any name. For this example, the code specifies a function that 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;
end
```

Save the file. After saving, the app displays the name of this new function in the **Function Name** box on the app toolstrip.
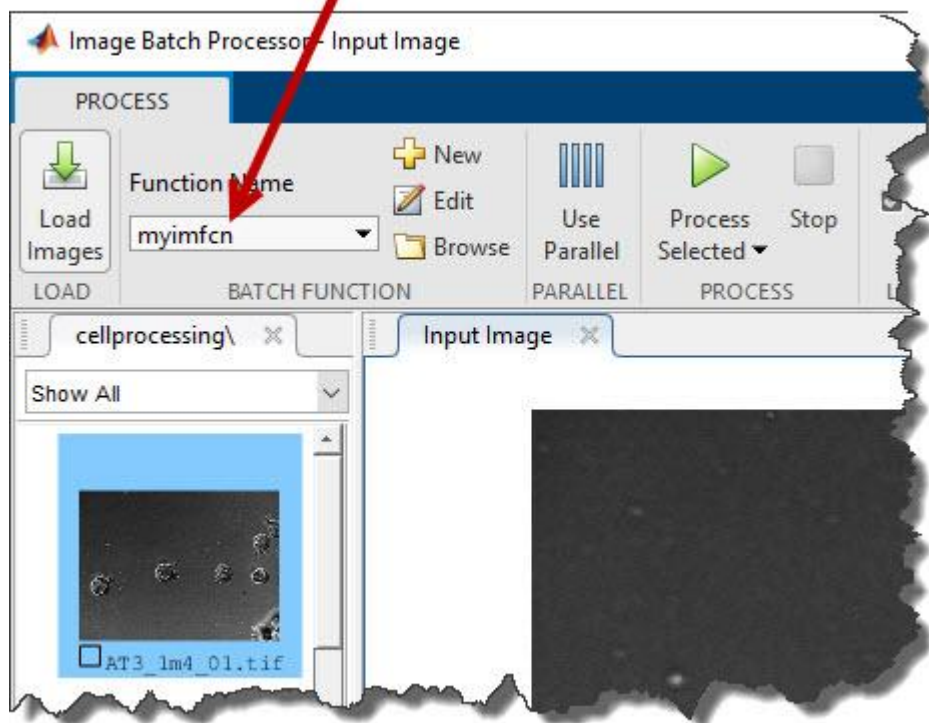
Process Images Using Batch Processing Function

Test the new function by running the batch processor on one of your images. With one image selected (highlighted in blue), click **Process Selected** to process the selected image. 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.

Processed image   Results tab
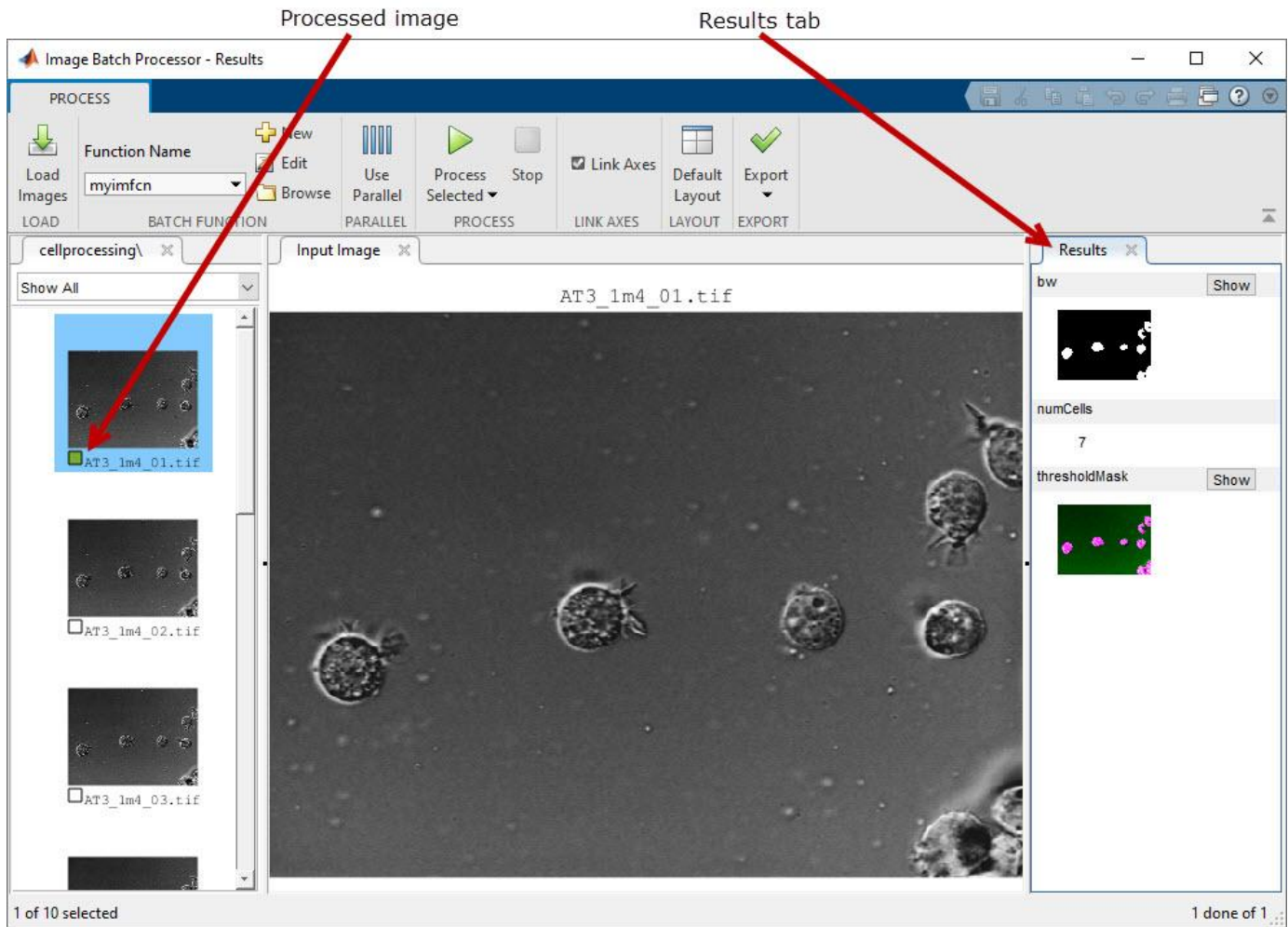
To get a closer view of the image results, click **Show** for that particular result in the **Results** tab. The app open a larger resolution version of the image in a new tab in a bottom-center pane. For this example, view the binary mask results be clicking **Show** for bw in the **Results** tab. To explore the results, move the cursor over the image to access the pan and zoom controls. When zooming and panning, the app links the result image to the original image—panning or zooming on one image causes the other image to move as well. If you do not want this behavior, clear **Link Axes** in the app toolstrip.
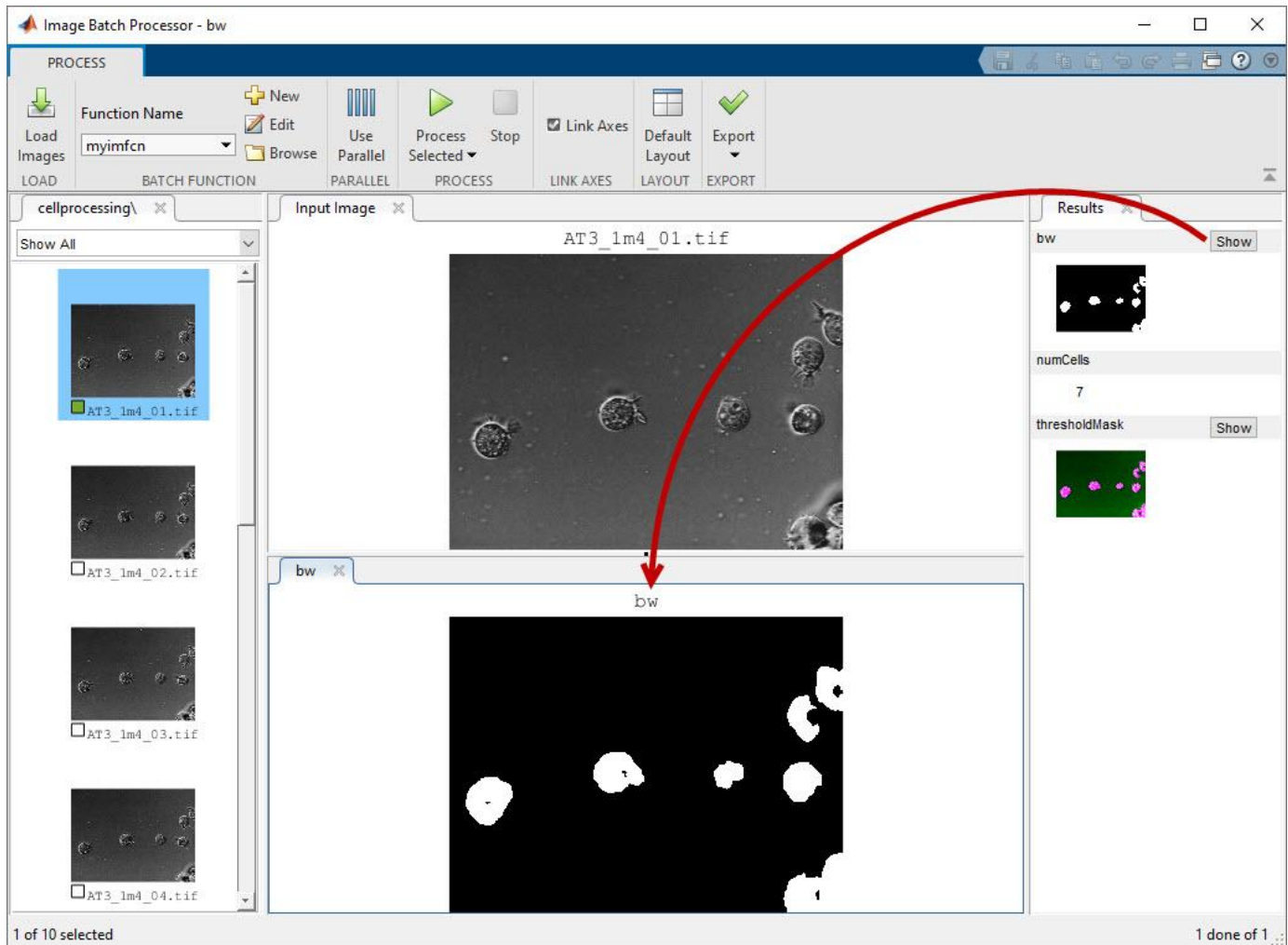
If the results of the test run on one image are successful, then execute the function on all of the images in the folder. To process all the images at once, on the app toolstrip, click **Process Selected** and select **Process All**. To process only a subset of the images, click **Process Selected**. You can select images to process either by pressing **Ctrl** and clicking the desired images or by clicking one image to start, pressing **Shift**, and clicking another image to select all images in between the starting and ending images. If you have Parallel Computing Toolbox™, you can click **Use Parallel** on the app toolstrip to process the images on a local parallel pool. For this example, process all of the images.

The app processes all the images in the specified folder. A filled-in green square next to a thumbnail indicates the app successfully processed that image. The **Results** tab contains the results of the selected image (highlighted in blue). A status bar at the bottom-right of the app reports on the number of images processed.

Export Processed Images and Processing Pipeline

To save the results, 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.

Save the results in a workspace variable. On the app toolstrip, click Export and select **Export result of all processed images to workspace** option. In the dialog box that opens, select the results you

want to export. A common approach is to export the nonimage results to the workspace and save the images that result from the processing in files. This example saves the cell count along with the name of the input file to the workspace variable `numCells`.
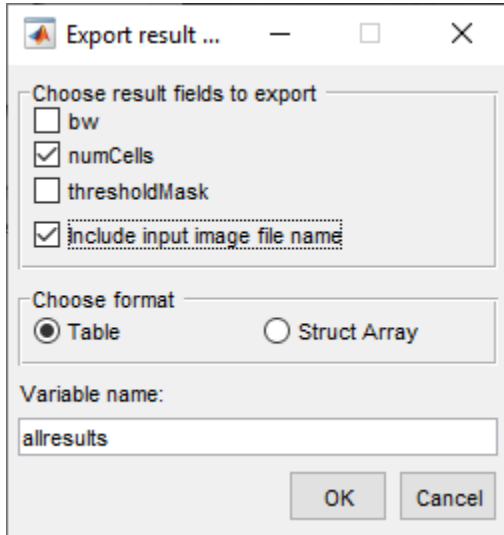
By default, the app returns the results you select in a table named `allresults`. To store the results in a structure instead of a table, select Struct Array in the **Choose format** section of the dialog box. To specify another name for the result variable, change **Variable name** in the dialog box. If you select **Include input image file name**, the app includes the name of the image associated with the results in the structure or table. After specifying exporting details, click **OK**.

To get the MATLAB® code that the app used to process your files, on the app toolstrip, click **Export** and select the **Generate function**. 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.

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 Parallel Server™ to run parallel MapReduce programs on Hadoop® clusters. The example shows how to test your algorithm on a local system on a subset of the images before moving it to the Hadoop cluster.

Download Sample Data

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. The names 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. For example, find the number of images in the data set that use the `w1` stain.

```
d = dir('C:\Temp\BBBCdata\BBBC005_v1_images\*w1*');
numel(d)
```

```
ans = 9600
```

Load Image Files into the Image Batch Processor

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 section, click **Image Batch Processor**. You can also open the app from the command line using the `imageBatchProcessor` command.

In the **Image Batch Processor** app, click **Load Images** and navigate to the folder in which you stored the downloaded data set.
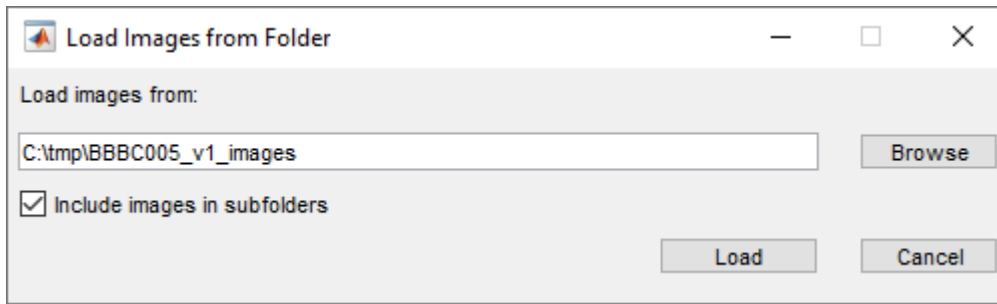
The **Image Batch Processor** app displays thumbnails of the images in the folder in the left pane and a higher-resolution version of the currently selected image in the Input Image tab. View some of the images to get familiar with the data set.
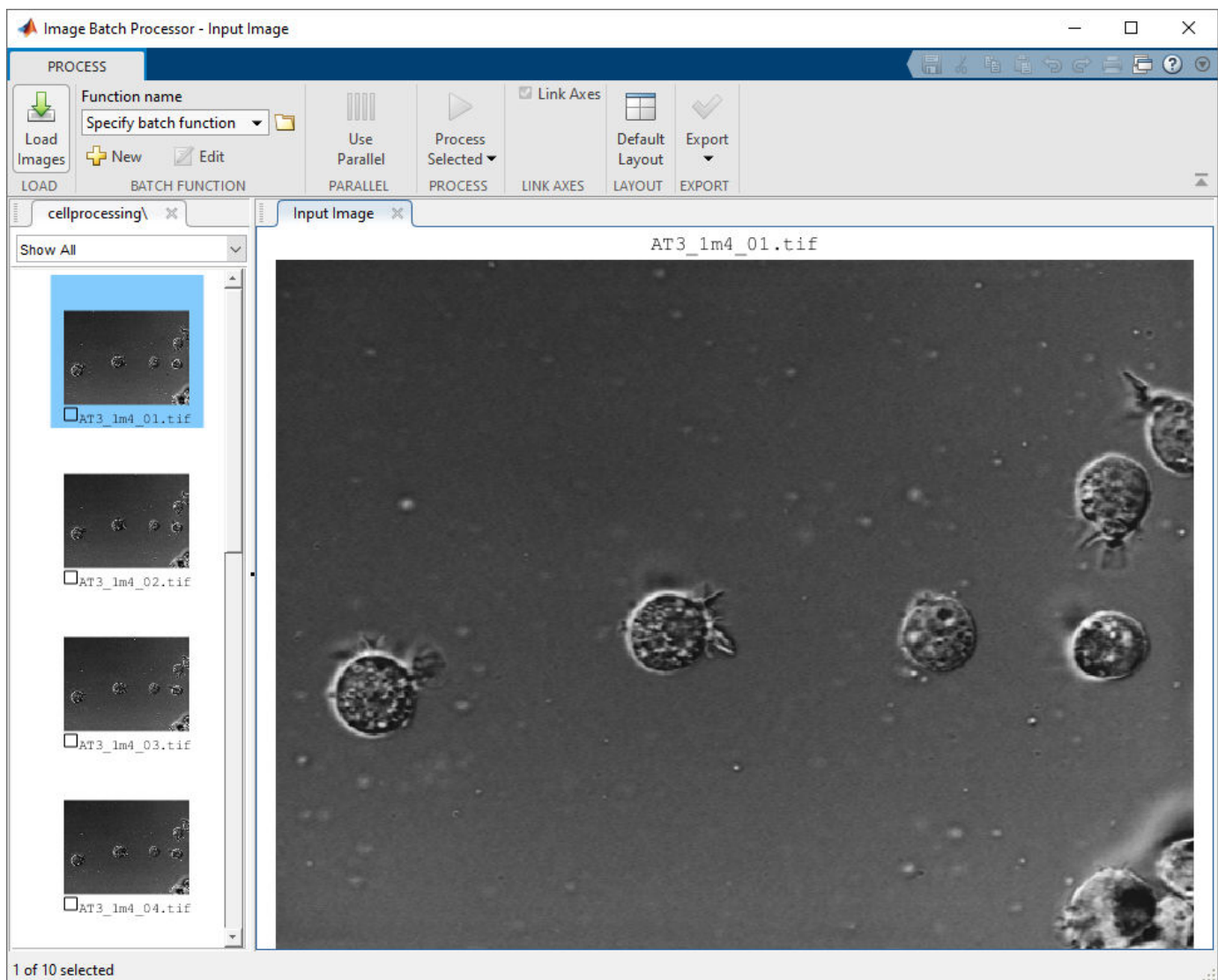
Specify Segmentation Function

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**. The app opens the batch function template in the MATLAB® editor. For this example, create a new function containing the following image segmentation code. Click **Save** to create the batch function. The app updates to display the name of the function you created in the Batch Function section of the app toolstrip.

```matlab
function imout = cellSegmenter(im)  % A simple cell segmenter
    % Otsu thresholding
    bw = imbinarize(im);

    % 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);
end
```

Test Your Segmentation Function on Sample Image

Select the thumbnail of an image displayed in the app and click **Process Selected** to execute a test run of your algorithm. For this example, choose only an image with the "w1" stain (identifiable in the file name). The segmentation algorithm works best with these images.

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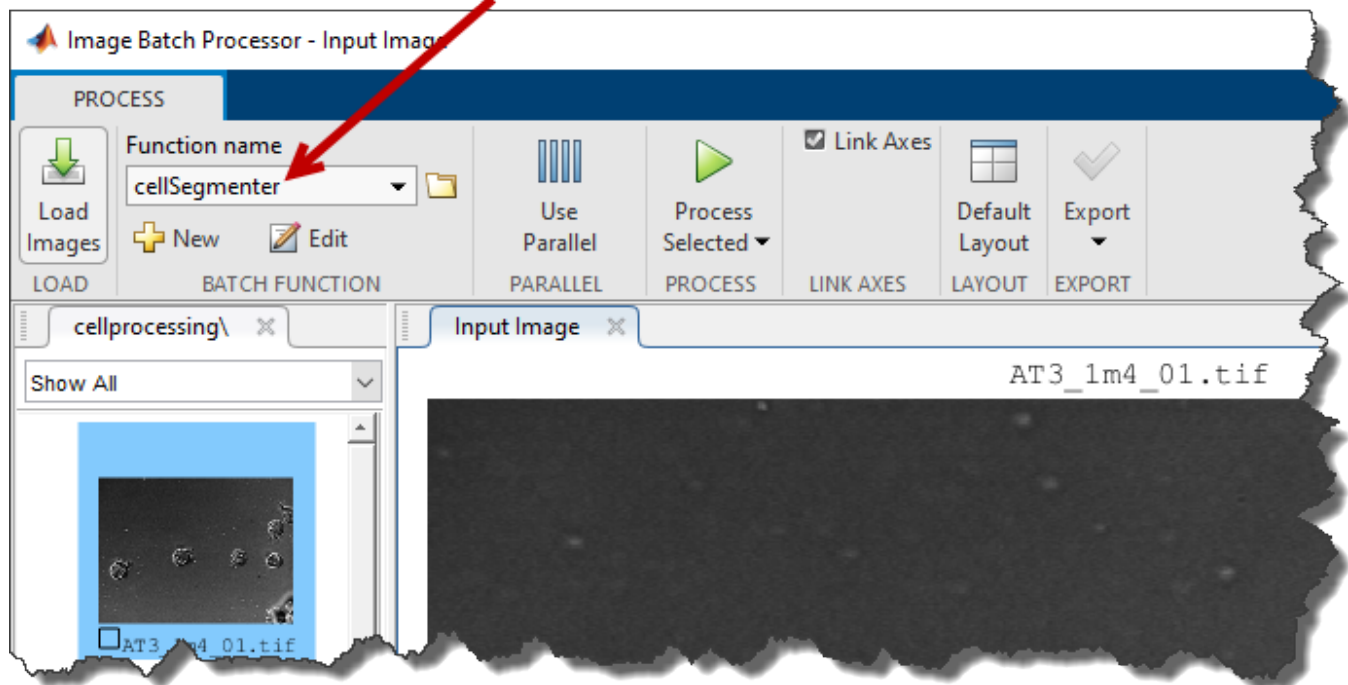
Examine the results of running your algorithm to verify that your segmentation algorithm found the correct number of cells in the image. The names of the images contain the cell count in the C number. For example, the image named `SIMCEPImages_A05_C18_F1_s05_w1.TIF` contains 18 cells. Compare this number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

Test Your MapReduce Framework Locally

After assuring that your segmentation code works as expected on one image, 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.

First, create an image datastore, using the `ImageDatastore` function, containing a small subset of your images. MapReduce uses a datastore to process data in small chunks that individually fit into memory. Move to the folder containing the images and create an image datastore. 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.

```
localimds = imageDatastore(fullfile('/your_data/broad_data/BBBC005_v1-images','*w1*'));
```

Even limiting the selection to file with "w1" in their names, the image data store still contains over 9000 files. Subset the list of images further, selecting every 100th file from the thousands of files in the data set.

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

Repackage the Sample Set into an Hadoop Sequence File

Once you have created the image datastore, convert the sample subset of images into Hadoop sequence files, a format used by the Hadoop cluster. 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).

To convert the image datastore to an Hadoop sequence file, 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`.

```
mapreducer(0);
```

When run locally, `mapreduce` creates a key-value datastore backed by MAT-files.

```
localmatds = mapreduce(localimds, @identityMap,
@identityReduce,'OutputFolder', pwd);
```

Test Your MapReduce Framework Locally

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 and remove the display of the image.

```
function cellCount = cellCounter(im)
    % Otsu thresholding
    bw = imbinarize(im);

    % 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;
```

```matlab
    cellCount = sum(round(cellsPerBlob));
end
```

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.

```matlab
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.

```matlab
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, focusBlue, meanDiff);
end
```

Run the `mapreduce` job on your local system.

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

Gather the results.

```matlab
focusErrorTbl = readall(focusErrords);
```

Get the average error values.

```matlab
averageErrors = cell2mat(focusErrorTbl.Value);
```

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, as seen in this plot of the results.

```matlab
function plot_errors()
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');
end
```
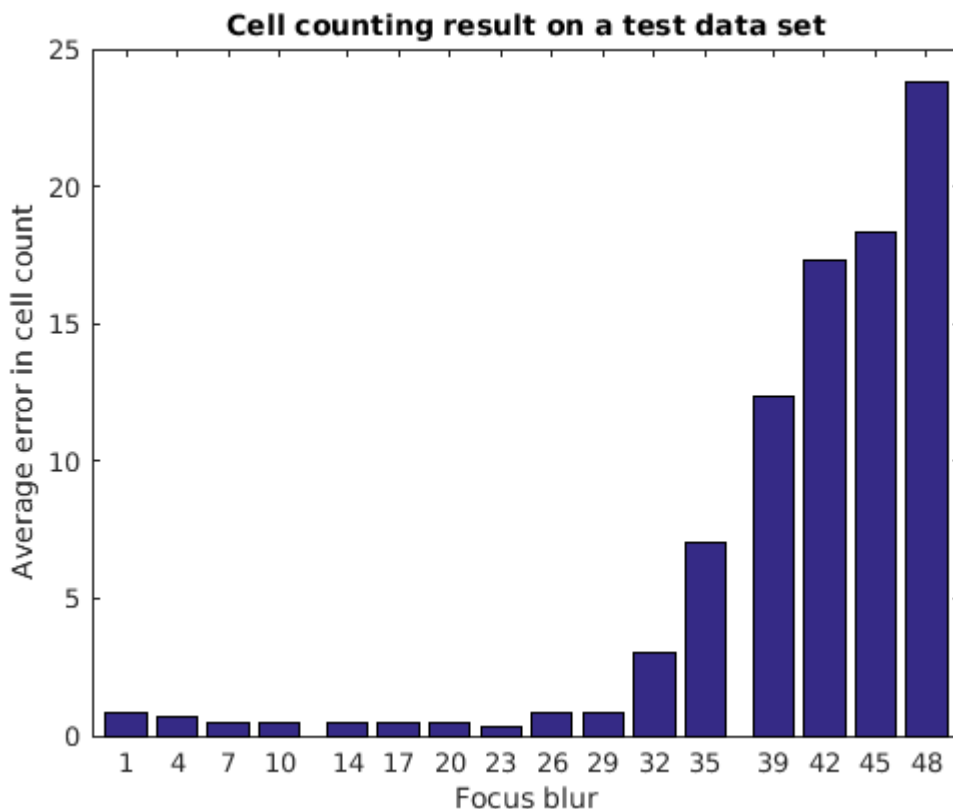

Run MapReduce Framework on Hadoop Cluster

Now that you've verified the processing of your algorithm on a subset of your data, run your algorithm on the full dataset on a Hadoop cluster.

Load Data into the Hadoop File System

Load all the image data into the Hadoop file system and run your MapReduce framework on a Hadoop cluster, using the following shell commands. To run this command, replace `your_data` with the location on your computer.

```
hadoop fs -mkdir /user/broad_data/
```

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

Set Up Access to MATLAB Parallel Server Cluster

Set up access to the MATLAB Parallel Server cluster. To run this command, replace `'your/hadoop/install'` with the location on your computer.

```
setenv('HADOOP_HOME','/your/hadoop/install');

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 Remote Cluster

Change the `mapreduce` execution environment to point to the remote cluster.

```
mapreducer(cluster);
```

Convert All Image Data into Hadoop Sequence Files

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.

```
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_sequence';
```

Convert the images to a key-value datastore.

```
seqds = mapreduce(imageDS, @identityMap, @identityReduce,'OutputFolder',
seqFolder);
```

Run Cell Counting Algorithm on Entire Data Set

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

First, specify the output location for error count.

```
output = 'hdfs://hadoop01glnxa64:54310/user/broad_data/
BBBC005_focus_vs_errorCount';
```

Run your algorithm on the Mapreduce framework. Use the `tic` and `toc` functions to record how long it takes to process the set of images.

```
tic;
```

```
focusErrords = mapreduce(seqds, @mapImageToMisCountError,
@reduceErrorCount,'OutputFolder',output);
```

```
toc
```

Gather results.

```
focusErrorTbl = readall(focusErrords);
```

```
averageErrors = cell2mat(focusErrorTbl.Value);
```

Plot the results, as before.

```
function reduceErrorCountAll(key, intermValueIter, outKVStore)
    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');
end
```
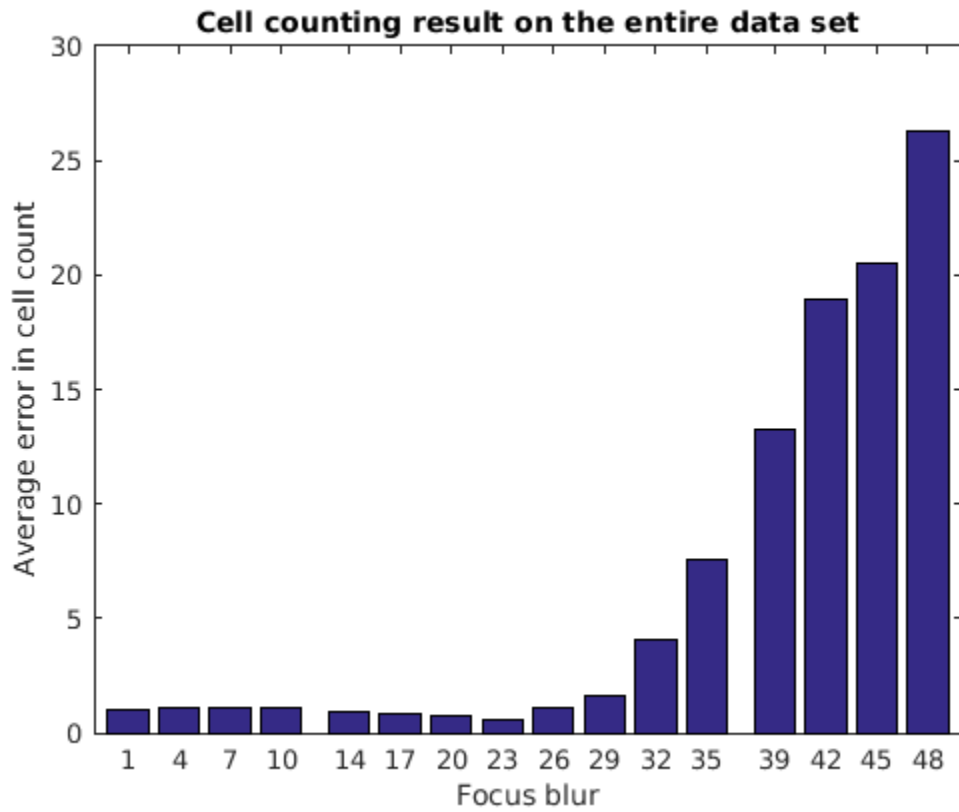
See Also

`ImageDatastore` | `mapreduce`

More About

* "Getting Started with MapReduce"
* "Work with Remote Data"

Work with Image Sequences as Multidimensional Arrays

Create Multidimensional Array Representing Image Sequence

Multidimensional arrays are a convenient way to display and process image sequences. Create a multidimensional array by concatenating the individual images of an image sequences. Each image must be the same size and have the same number of color channels. If you are storing a sequence of indexed images, each image must use the same color map.

- If you have a sequence of 2-D grayscale, binary, or indexed images, then concatenate the images in the third dimension to create a 3-D array of size m-by-n-by-p. Each of the p images has size m-by-n.

- If you have a sequence of 2-D RGB images, then concatenate the images along the fourth dimension to create a 4-D array of size m-by-n-by-3-by-p. Each of the p images has size m-by-n-by-3.

The figure depicts 2-D images concatenated as planes of a 3-D array.
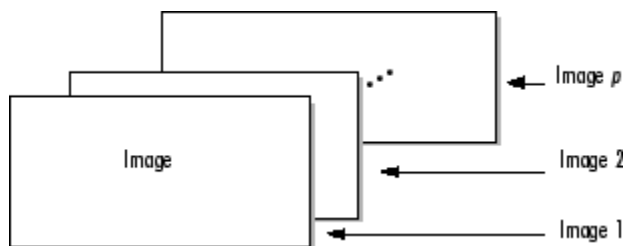
Use the `cat` function to concatenate individual images. For example, this code concatenates a group of RGB images along the fourth dimension.

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

Note Some functions work with a particular type of multidimensional array, call a multiframe array. In a multiframe array, images are concatenated along the fourth dimension regardless of the number of color channels that the images have. A multiframe array of grayscale, binary, or indexed images has size m-by-n-by-1-by-p. If you need to convert a multiframe array of grayscale images to a 3-D array for use with other toolbox functions, then you can use the `squeeze` function to remove the singleton dimension.

Display Image Sequences

There are several ways to display image sequences. To display one frame at a time, use the Image Viewer app or `imshow`. To display all the frames in an image sequence simultaneously, use the `montage` function.

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.

Process Image Sequences

Many toolbox functions can operate on multidimensional arrays and, consequently, can operate on image sequences. For example, if you pass a multidimensional 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 multidimensional 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 table lists these toolbox functions and provides guidelines about how to use them to process image sequences.

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.
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.

Function	Image Sequence Dimensions	Guideline When Used with an Image Sequence
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.
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, that is,[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.

See Also

More About

- "Perform an Operation on a Sequence of Images" on page 2-23
- "View Image Sequences in Video Viewer App" on page 4-65
- "Process Folder of Images Using Image Batch Processor App" on page 2-31

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 clip return values to fit the data type.

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 data types. See the reference pages for the individual arithmetic functions for more information.

See Also

More About
- "Image Arithmetic Clipping Rules" on page 2-52
- "Nest Calls to Image Arithmetic Functions" on page 2-53

Image Arithmetic Clipping 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 clipped, or truncated, 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 table lists some additional examples.

Result	Class	Clipped Value
300	uint8	255
-45	uint8	0
10.5	uint8	11

See Also

More About

- "Nest Calls to Image Arithmetic Functions" on page 2-53

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 clips 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 clips the final result.

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

See Also
`imlincomb`

More About
• "Image Arithmetic Clipping Rules" on page 2-52

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
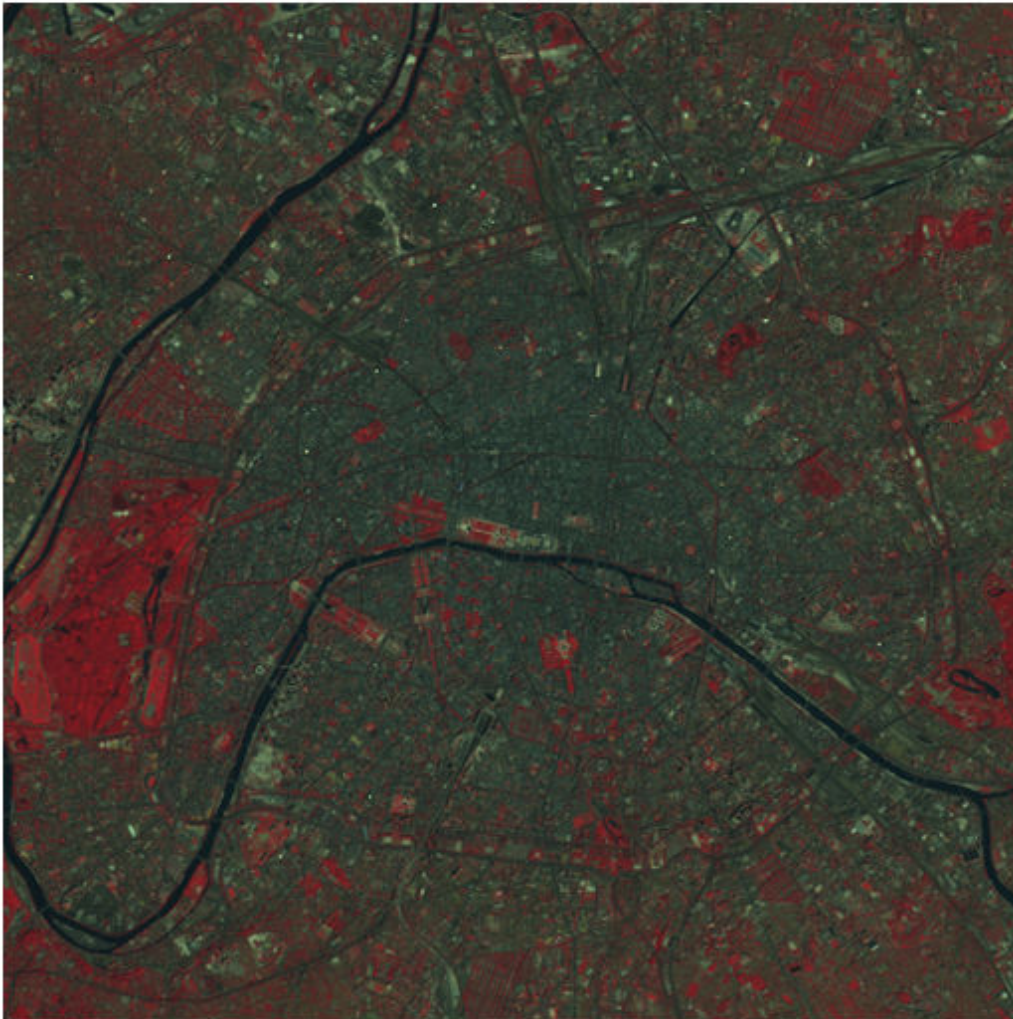
Image courtesy of Space Imaging, LLC

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')
```


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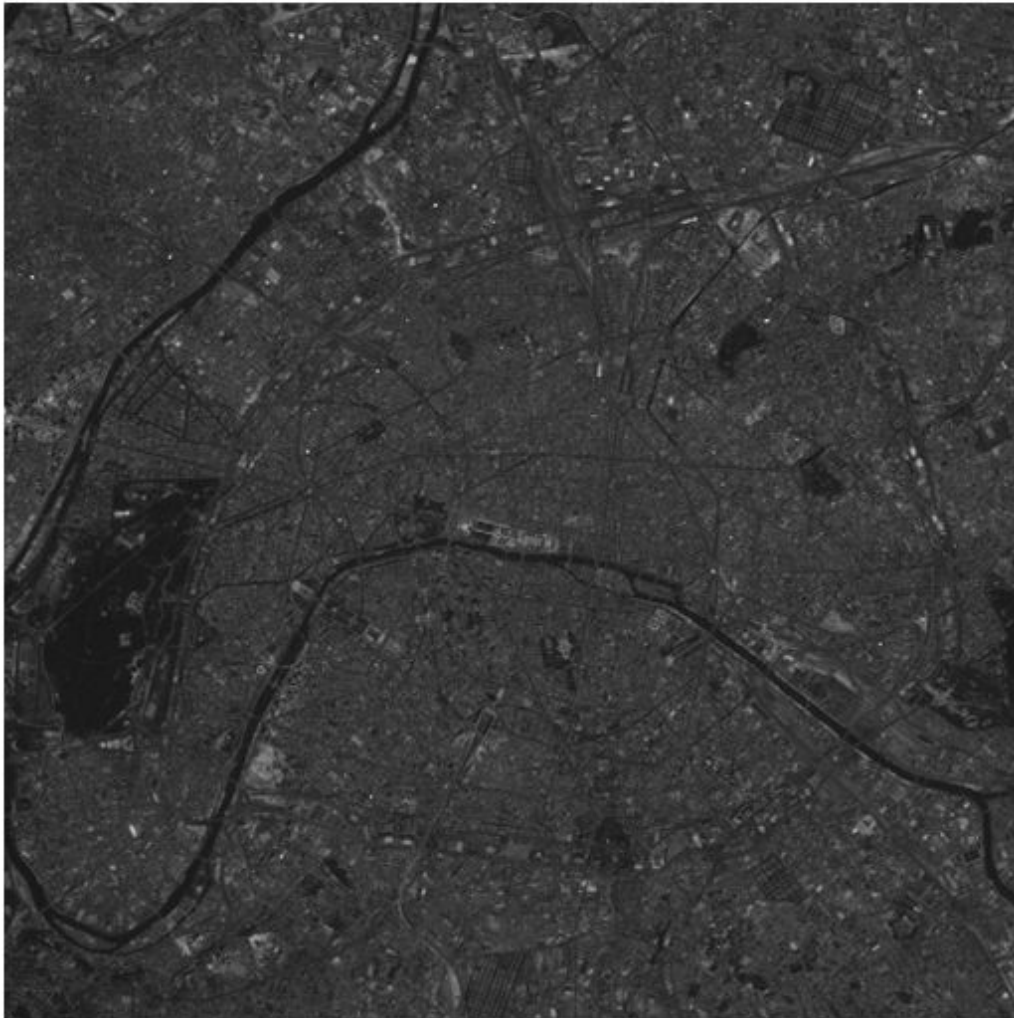
```
imshow(NIR)
title('Near Infrared Band')
```

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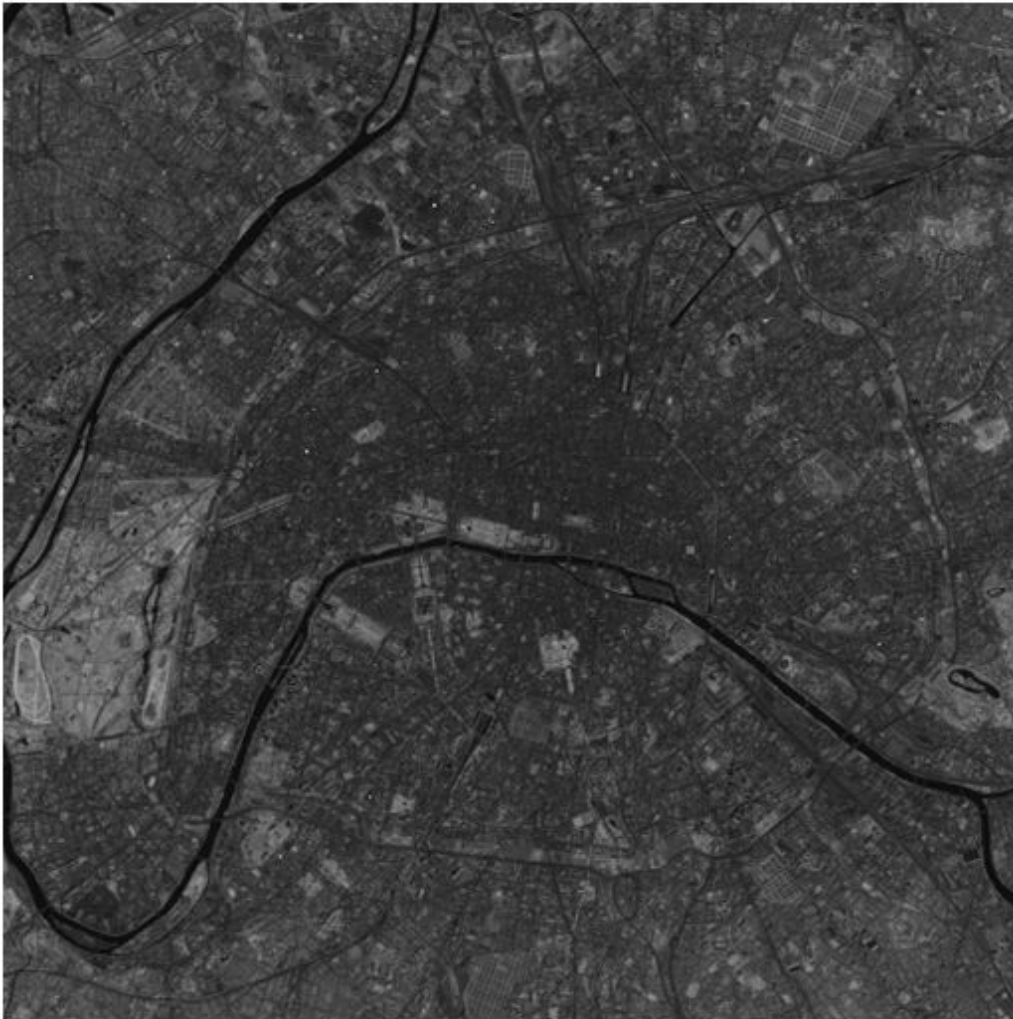
With one simple call to the `plot` command in MATLAB, you can create a scatter plot displaying one point per pixel (as a blue cross, in this case), with its x-coordinate determined by its value in the red channel and its y-coordinate by the value its value in the NIR channel.

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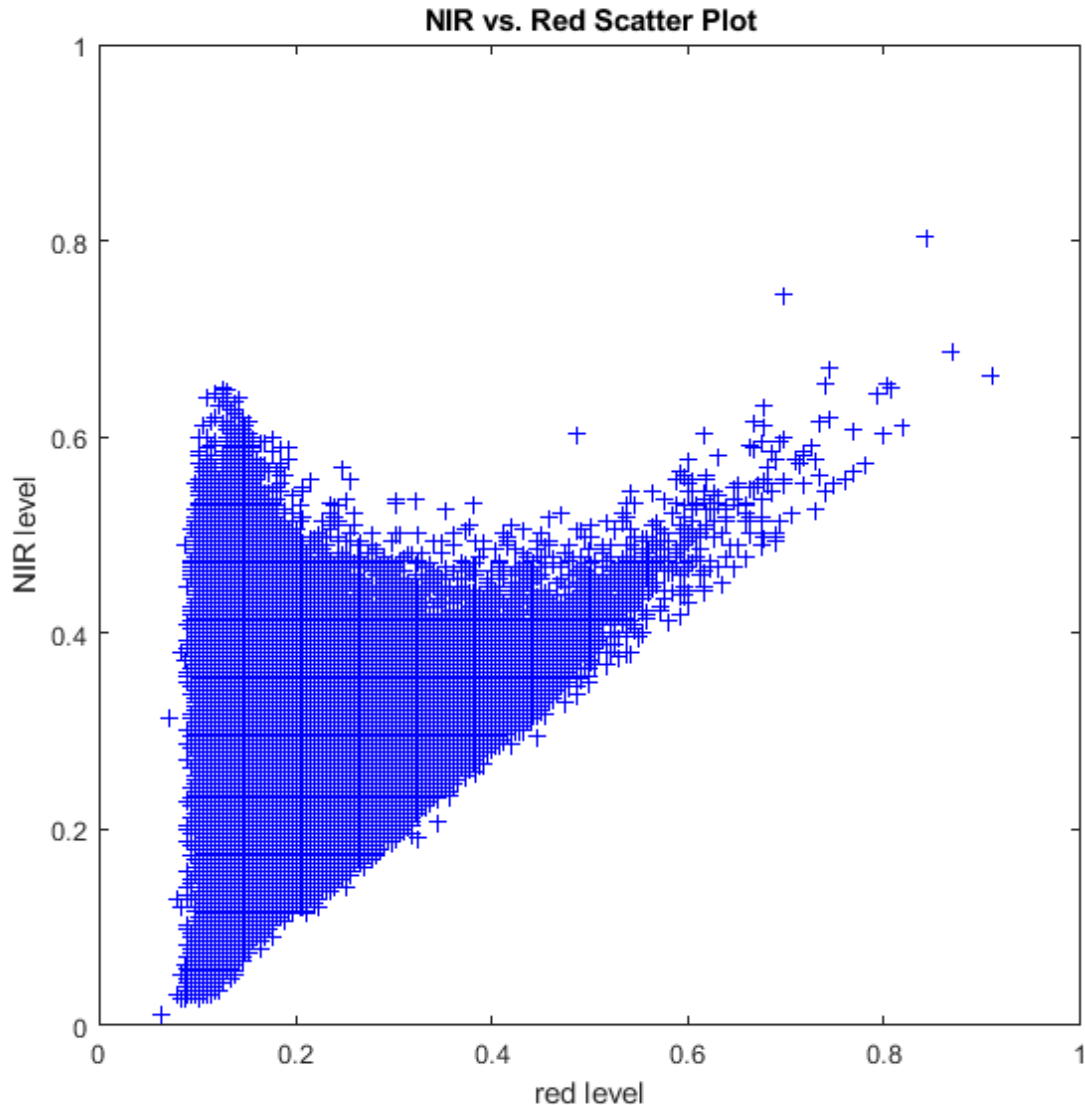
The appearance of the scatter plot of the Paris scene is characteristic of a temperate urban area with trees in summer foliage. There's a set of pixels near the diagonal for which the NIR and red values are nearly equal. This "gray edge" includes features such as road surfaces and many rooftops. Above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

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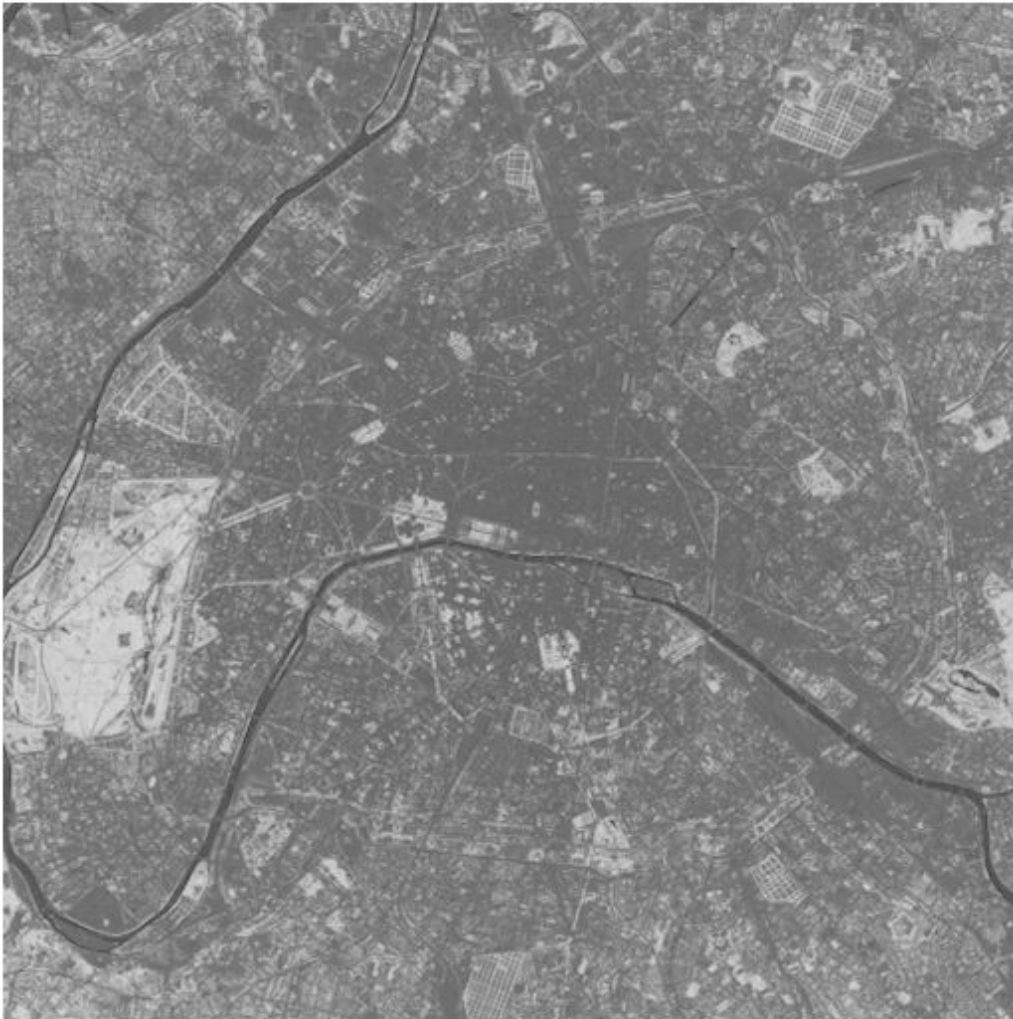
The Seine River appears very dark in the NDVI image. The large light area near the left edge of the image is the park (Bois de Boulogne) noted earlier.

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')
```


NDVI with Threshold Applied
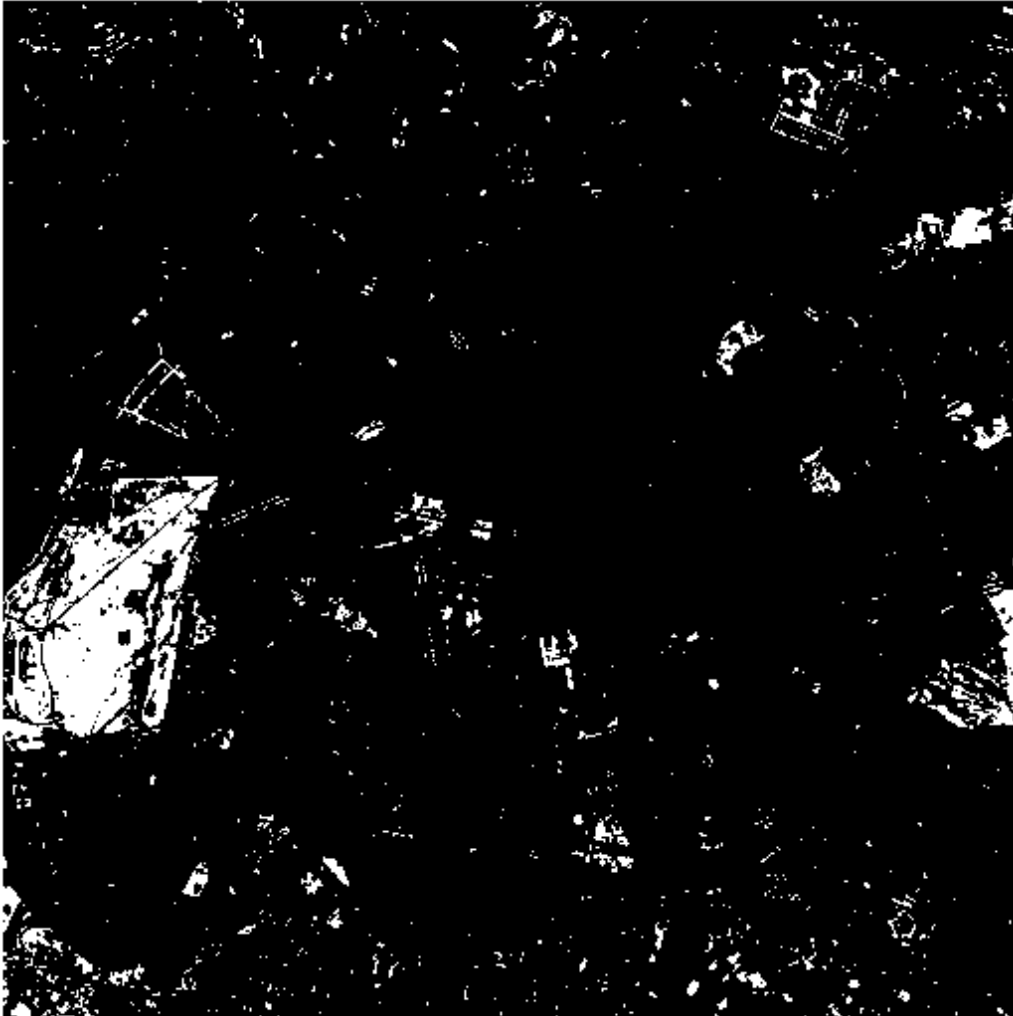
Step 5: Link Spectral and Spatial Content

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')
```
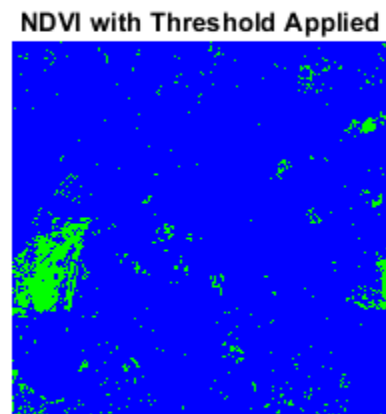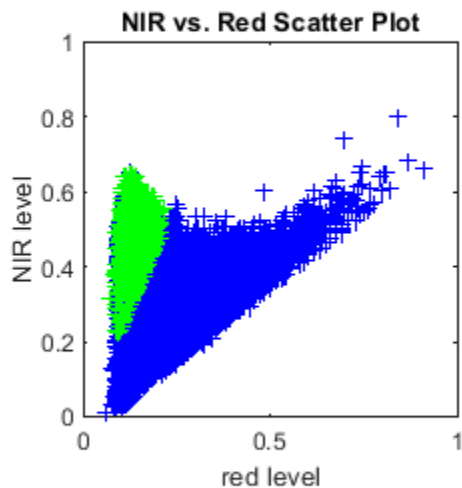

See Also

`im2single` | `imshow` | `multibandread`

Related Examples

- "Enhance Multispectral Color Composite Images" on page 11-93

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.

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 2-12
- "Read Multiple Images from a Single Graphics File" on page 3-4
- "Write Image Data to File in Graphics Format" on page 3-6

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 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              4566  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 2-12
- "Read Image Data into the Workspace" on page 3-2

Write Image Data to File in Graphics Format

This example shows how to write image data from the 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, `trees.mat`, along with the associated colormap `map`.

```
load trees
whos
```

```
  Name          Size            Bytes  Class      Attributes

  X             258x350         722400  double
  caption         1x66             132  char
  map           128x3             3072  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,'trees.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 2-12
- "Read Image Data into the Workspace" on page 3-2

DICOM Support in 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.

See Also

Apps
DICOM Browser

Functions
`dicomCollection` | `dicomread` | `dicomreadVolume` | `dicomwrite`

More About

- "Read Metadata from DICOM Files" on page 3-8
- "Read Image Data from DICOM Files" on page 3-10
- "Create New DICOM Series" on page 3-18
- "Remove Confidential Information from DICOM File" on page 3-14
- "Write Image Data to DICOM Files" on page 3-12

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 3-10. 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 3-9.

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 (such as 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.

See Also

Apps
DICOM Browser

Functions
dicomdict | dicomdisp | dicominfo | dicomread | dicomreadVolume

More About

- "Read Image Data from DICOM Files" on page 3-10
- "Create New DICOM Series" on page 3-18
- "Remove Confidential Information from DICOM File" on page 3-14
- "Write Image Data to DICOM Files" on page 3-12

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',[])
```


See Also

Apps
DICOM Browser

Functions
dicominfo | dicomread | dicomreadVolume

More About

- "Read Metadata from DICOM Files" on page 3-8
- "Create New DICOM Series" on page 3-18

- "Remove Confidential Information from DICOM File" on page 3-14
- "Write Image Data to DICOM Files" on page 3-12

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 (such as X-ray, radiotherapy, or nuclear medicine) to a file. However, `dicomwrite` does not perform any validation of this data.

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.

Specify Value Representation

Each field of DICOM metadata (known as an *attribute* or *data element*), includes a tag that identifies the attribute, information about the length of the attribute, and the attribute data. The attribute

optionally includes a two-letter value representation (VR) that identifies the format of the attribute data. For example, the format can be a single-precision binary floating point number, a character vector that represents a decimal integer, or a character vector in the format of a date-time.

To include the VR in the attribute when using `dicomwrite`, specify the `'VR'` name-value pair argument as `'explicit'`. If you do not specify the VR, then `dicomwrite` infers the value representation from the data dictionary.

The figure shows an attribute with and without the VR.
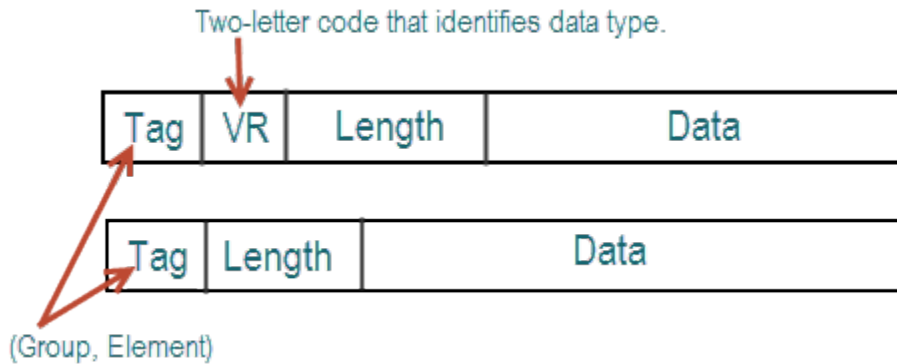
See Also

Apps
DICOM Browser

Functions
dicomanon | dicominfo | dicomread | dicomuid | dicomwrite

More About

- "Read Metadata from DICOM Files" on page 3-8
- "Read Image Data from DICOM Files" on page 3-10
- "Create New DICOM Series" on page 3-18
- "Remove Confidential Information from DICOM File" on page 3-14

Remove Confidential Information from DICOM File

This example shows how to anonymize 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.

Read an image from a DICOM file into the workspace.

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

Display the image. Because the DICOM image data is signed 16-bit data, automatically scale the display range so that the minimum pixel value is black and the maximum pixel value is white.

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

```
Fri Apr 30 06:27:24 1993 3D          JFK IMAGING CENTER
Ex: 2411                    AIL                    ****
Se: 3
Surface                                   M 26  151832

DFOV 16.0cm
BONE




R                                                       L
I                                                       S




kv 120
mA 200

3.0 mm/1.0sp
Tilt: -0.0

W = 1492 L = 906           PSR
```

Read the metadata from the DICOM file.

```
info = dicominfo(dicomFile);
```

The DICOM file in this example has already been anonymized for patient privacy. To create an informative test DICOM file, set the PatientName with an artificial value using the Person Name (PN) value representation.

```
info.PatientName = 'Doe^John';
```

Write the image with modified metadata to a new DICOM file.

```
dicomFileNotAnon = 'ankle_notAnon.dcm';
dicomwrite(I,dicomFileNotAnon,info);
```

Read the metadata from the non-anonymous DICOM file, then confirm that the patient name in the new file is not anonymous.

```
infoNotAnon = dicominfo(dicomFileNotAnon);
infoNotAnon.PatientName
```

```
ans = struct with fields:
    FamilyName: 'Doe'
     GivenName: 'John'
```

To identify the series to which the non-anonymous image belongs, display the value of the SeriesInstanceUID property.

```
infoNotAnon.SeriesInstanceUID
```

```
ans =
'1.2.840.113619.2.1.2411.1031152382.365.736169244'
```

Anonymize the file using the `dicomanon` function. The function creates a new series with new study values, changes some of the metadata, and then writes the image to a new file.

```
dicomFileAnon = 'ankle_anon.dcm'
```

```
dicomFileAnon =
'ankle_anon.dcm'
```

```
dicomanon(dicomFileNotAnon,dicomFileAnon);
```

Read the metadata from the anonymized DICOM file.

```
infoAnon = dicominfo(dicomFileAnon);
```

Confirm that the patient name information has been removed.

```
infoAnon.PatientName
```

```
ans = struct with fields:
    FamilyName: ''
     GivenName: ''
    MiddleName: ''
    NamePrefix: ''
    NameSuffix: ''
```

Confirm that the anonymous image belongs to a new study by displaying the value of the SeriesInstanceUID property.

```
infoAnon.SeriesInstanceUID
```

```
ans =
'1.3.6.1.4.1.9590.100.1.2.332764594910612584641813594964057281315'
```

See Also

Apps
DICOM Browser

Functions
dicomanon | dicominfo | dicomread | dicomuid | dicomwrite

More About

- "Read Image Data from DICOM Files" on page 3-10
- "Read Metadata from DICOM Files" on page 3-8
- "Create New DICOM Series" on page 3-18
- "Write Image Data to DICOM Files" on page 3-12
- "DICOM Support in Image Processing Toolbox" on page 3-7

Create New DICOM Series

This example shows how to create a new DICOM series for a modified DICOM image.

In the DICOM standard, images can be organized into series. By default, when you write an image with metadata to a DICOM file, `dicomwrite` puts the image in the same series. You typically only start a new DICOM series when you modify the image in some way. To make the modified image the start of a new series, assign a new DICOM unique identifier to the SeriesInstanceUID metadata field.

Read an image from a DICOM file into the workspace.

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

Display the image. Because the DICOM image data is signed 16-bit data, automatically scale the display range so that the minimum pixel value is black and the maximum pixel value is white.

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

Read the metadata from the DICOM file.

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

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

```
info.SeriesInstanceUID
```

```
ans =
'1.2.840.113619.2.1.2411.1031152382.365.736169244'
```

This example modifies the image by removing all of the text from the image. Text in the image appears white. Find the maximum value of all pixels in the image, which corresponds to the white text.

```
textValue = max(I(:));
```

The background of the image appears black. Find the minimum value of all pixels in the image, which corresponds to the background.

```
backgroundValue = min(I(:));
```

To remove the text, set all pixels with the maximum value to the minimum value.

```
Imodified = I;
Imodified(Imodified == textValue) = backgroundValue;
```

View the processed image.

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

To write the modified image as a new series, you need a new DICOM unique identifier (UID). Generate a new UID using the `dicomuid` function. `dicomuid` is guaranteed to generate a unique UID.

```
uid = dicomuid
```

```
uid =
'1.3.6.1.4.1.9590.100.1.2.265328955116359303213367544491626002787'
```

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

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

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 written image 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.

See Also

Apps
DICOM Browser

Functions
dicomanon | dicominfo | dicomread | dicomuid | dicomwrite

More About

*   "Remove Confidential Information from DICOM File" on page 3-14
*   "DICOM Support in Image Processing Toolbox" on page 3-7

Create Image Datastore Containing DICOM Images

This example shows how to create an image datastore from a collection of DICOM files containing 2-D images.

Specify the location of a directory containing 2-D DICOM image files.

```
dicomDir = fullfile(matlabroot,'toolbox/images/imdata/dog');
```

Create an `imageDatastore`, specifying the read function as a handle to the `dicomread` function.

```
dicomds = imageDatastore(dicomDir, ...
    'FileExtensions','.dcm','ReadFcn',@(x) dicomread(x));
```

Read and display the first image in the datastore.

```
I = read(dicomds);
```

Display the image. The image has signed 16-bit data, so scale the display range to the pixel values in the image.

```
imshow(I,[])
```
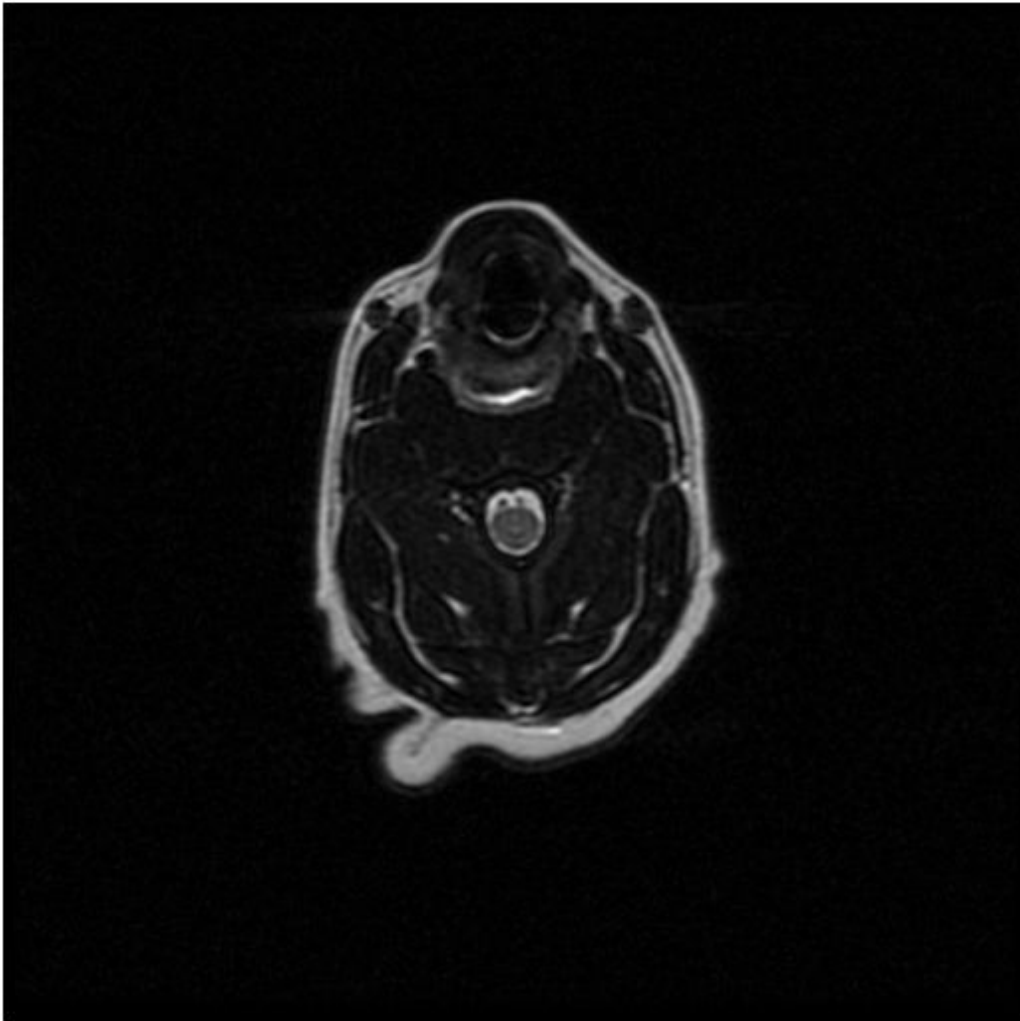
See Also
dicomread | imageDatastore

More About
• "Create Image Datastore Containing Single and Multi-File DICOM Volumes" on page 3-24

Create Image Datastore Containing Single and Multi-File DICOM Volumes

This example shows how to create an image datastore containing volumetric DICOM data stored as a single file and as multiple files.

Specify the location of a directory containing DICOM data. The data includes 2-D images, a 3-D volume in a single file, and a multi-file 3-D volume.

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

Gather details about the DICOM files by using the `dicomCollection` function. This function returns the details as a table, where each row represents a single study. For multi-file DICOM volumes, the function aggregates the files into a single study.

```
collection = dicomCollection(dicomDir,"IncludeSubfolders",true)
```

```
collection=6×14 table
           StudyDateTime              SeriesDateTime              PatientName        PatientSex
        _____    _____    _____    _____

    s1    {0×0 double          }    {0×0 double          }     ""                  ""
    s2    {[30-Apr-1993 11:27:24]}  {[30-Apr-1993 11:27:24]}   "Anonymized"        ""
    s3    {[14-Dec-2013 15:47:31]}  {[14-Dec-2013 15:54:33]}   "GORBERG MITZI"     "F"
    s4    {[03-Oct-2011 19:18:11]}  {[03-Oct-2011 18:59:02]}   ""                  "M"
    s5    {[03-Oct-2011 19:18:11]}  {[03-Oct-2011 19:05:04]}   ""                  "M"
    s6    {[30-Jan-1994 11:25:01]}  {0×0 double          }     "Anonymized"        ""
```

Create a temporary directory to store the processed DICOM volumes.

```
matFileDir = fullfile(pwd,'MATFiles');
if ~exist(matFileDir,'dir')
    mkdir(matFileDir)
end
```

Loop through each study in the collection.

```
for idx = 1:size(collection,1)
```

Get the file names that comprise the study. For multi-file DICOM volumes, the file names are listed as a string array.

```
    dicomFileName = collection.Filenames{idx};
    if length(dicomFileName) > 1
        matFileName = fileparts(dicomFileName(1));
        matFileName = split(matFileName,filesep);
        matFileName = replace(strtrim(matFileName(end))," ","_");
    else
        [~,matFileName] = fileparts(dicomFileName);
    end
    matFileName = fullfile(matFileDir,matFileName);
```

Read the data. Try different read functions because the images have a different number of dimensions and are stored in different formats.

1) Try reading the data of the study by using the `dicomreadVolume` function.

- If the data is a multi-file volume, then `dicomreadVolume` runs successfully and returns the complete volume in a single 4-D array. You can add this data to the datastore.
- If the data is contained in a single file, then `dicomreadVolume` does not run successfully.

2) Try reading the data of the study by using the `dicomread` function.

- If `dicomread` returns a 4-D array, then the study contains a complete 3-D volume. You can add this data to the datastore.
- If `dicomread` returns a 2-D matrix or 3-D array, then the study contains a single 2-D image. Skip this data and continue to the next study in the collection.

```
try
    data = dicomreadVolume(collection,collection.Row{idx});
catch ME
    data = dicomread(dicomFileName);
    if ndims(data)<4
        % Skip files that are not volumes
        continue;
    end
end
```

For complete volumes returned in a 4-D array, write the data and the absolute file name to a MAT file.

```
save(matFileName,'data','dicomFileName');
```

End the loop over the studies in the collection.

```
end
```

Create an `imageDatastore` from the MAT files containing the volumetric DICOM data. Specify the `ReadFcn` property as the helper function `matRead`, defined at the end of this example.

```
imdsdicom = imageDatastore(matFileDir,'FileExtensions','.mat', ...
    'ReadFcn',@matRead);
```

Read the first DICOM volume from the image datastore.

```
[V,Vinfo] = read(imdsdicom);
[~,VFileName] = fileparts(Vinfo.Filename);
```

The DICOM volume is grayscale. Remove the singleton channel dimension by using the `squeeze` function, then display the volume by using the `volshow` function.

```
V = squeeze(V);
ViewPnl = uipanel(figure,'Title',VFileName);
volshow(V,'Parent',ViewPnl);
```
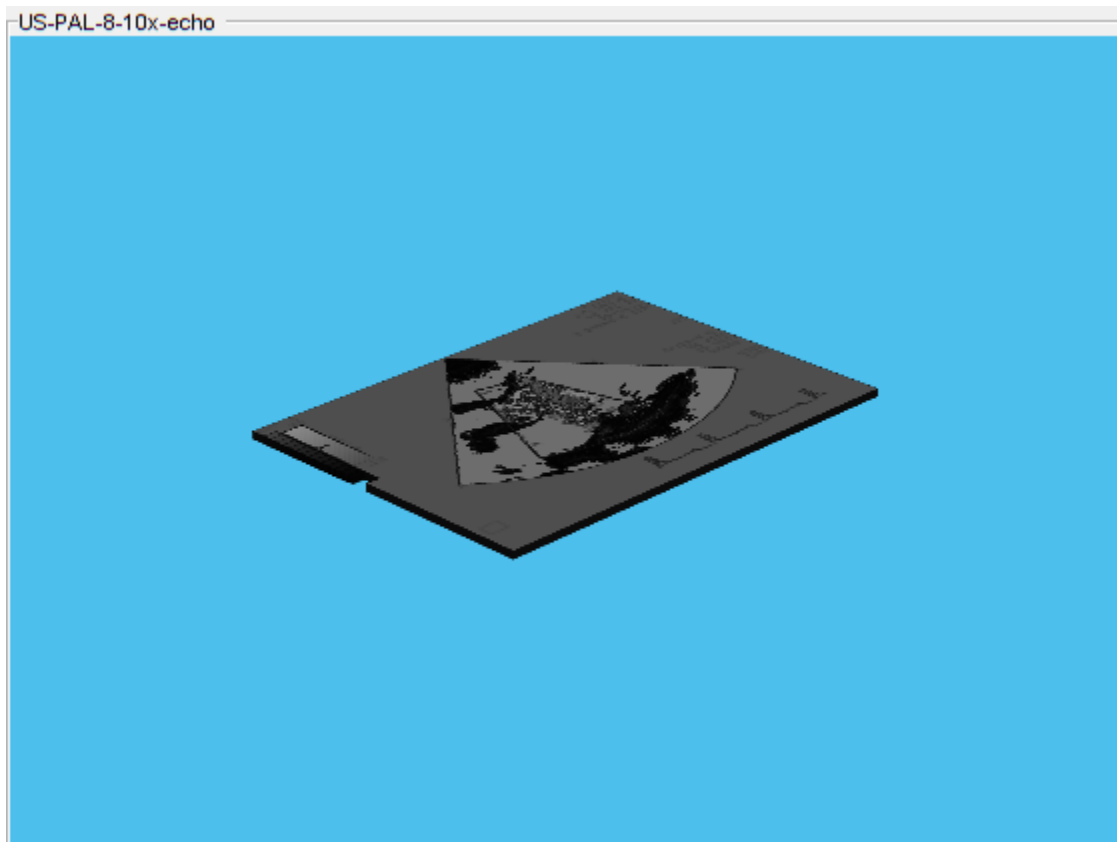
Supporting Functions

The `matRead` function loads data from the first variable of a MAT file with file name `filename`.

```
function data = matRead(filename)
    inp = load(filename);
    f = fields(inp);
    data = inp.(f{1});
end
```

See Also

dicomCollection | dicominfo | dicomread | dicomreadVolume | imageDatastore | volshow

More About

- "Create Image Datastore Containing DICOM Images" on page 3-22
- "Preprocess Volumes for Deep Learning" (Deep Learning Toolbox)

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 3-28.

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. For more information, see `interfileinfo` or `interfileread`.

An Interfile data set consists of two files:

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

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 3-27.

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. The toolbox also includes tone-map operators for creating low dynamic range (LDR) images from HDR images for processing and display.

Read HDR Image

To read an HDR 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 data type `single`.

```
whos
```

```
  Name         Size            Bytes    Class     Attributes

  hdr_image    665x1000x3      7980000  single
```

The range of data exceeds the range [0, 1] expected of LDR data.

```
hdr_range = [min(hdr_image(:)) max(hdr_image(:))]

hdr_range =

  1×2 single row vector

       0    3.2813
```

Display and Process HDR Image

Many toolbox functions assume that images of data type `single` and `double` are LDR images with data in the range [0, 1]. Since HDR data is not bound to the range [0, 1] and can contain `Inf` values, you must examine the behavior of each function carefully when working with HDR data.

- Some functions clip values outside the expected range before continuing to process the data. These functions can return unexpected results because clipping causes a loss of information.
- Some functions expect data in the range [0, 1] but do not adjust the data before processing it. These functions can return incorrect results.
- Some functions expect real data. If your HDR image contains values of `Inf`, then these functions can return unexpected results.
- Some functions have no limitations for the range of input data. These functions accept and process HDR data correctly.

To work with functions that require LDR data, you can reduce the dynamic range of an image using a process called tone mapping. Tone mapping scales HDR data to the range [0, 1] while attempting to preserve the appearance of the original image. Tone mapping functions such as `tonemap`, `tonemapfarbman`, and `localtonemap` give more accurate results than simple linear rescaling such

as performed by the `rescale` function. However, note that tone mapping incurs a loss of subtle information and detail.

To display HDR images, you must perform tone mapping. For an example, see "Display High Dynamic Range Image" on page 3-31.

Create High Dynamic Range Image

To create an HDR 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 an HDR image from the workspace into a file, use the `hdrwrite` function.

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

See Also
`hdrread` | `hdrwrite` | `localtonemap` | `makehdr` | `tonemap` | `tonemapfarbman`

Related Examples

More About

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)
```

See Also

`localtonemap` | `tonemap` | `tonemapfarbman`

More About

- "Work with High Dynamic Range Images" on page 3-29

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 Figure Window" on page 4-3
- "Display Multiple Images" on page 4-7
- "View Thumbnails of Images in Folder or Datastore" on page 4-11
- "Get Started with Image Viewer App" on page 4-17
- "Get Pixel Information in Image Viewer App" on page 4-26
- "Measure Distance Between Pixels in Image Viewer App" on page 4-33
- "Adjust Image Contrast in Image Viewer App" on page 4-36
- "Crop Image Using Image Viewer App" on page 4-40
- "Explore 3-D Volumetric Data with Volume Viewer App" on page 4-44
- "Explore 3-D Labeled Volumetric Data with Volume Viewer App" on page 4-57
- "View Image Sequences in Video Viewer App" on page 4-65
- "Convert Multiframe Image to Movie" on page 4-72
- "Display Different Image Types" on page 4-73
- "Add Color Bar to Displayed Grayscale Image" on page 4-78
- "Print Images" on page 4-80
- "Manage Display Preferences" on page 4-81

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 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 Figure Window" on page 4-3. 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. **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 "Get Started with Image Viewer App" on page 4-17.

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 color map color, and the value `max(I(:))` is displayed using the last color map 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`
`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 color map	Set to a grayscale color map whose values range from black to white	Ignored when `CData` is 3-D

Display an Image in Figure Window

Overview

To display image data, use the `imshow` function. The following example reads an image into the workspace and then displays the image in a figure window using the `imshow` function.

```
moon = imread('moon.tif');
imshow(moon);
```
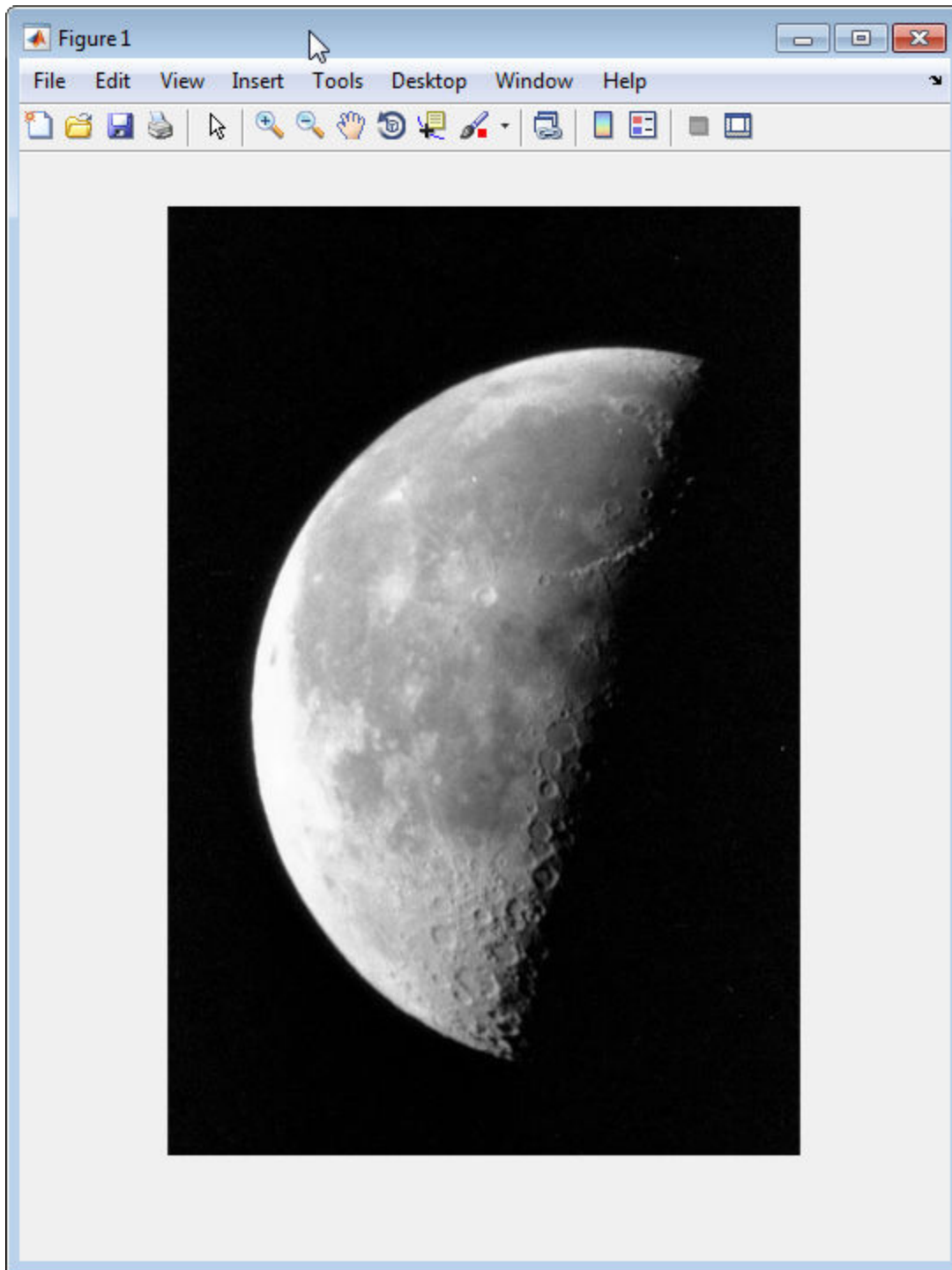
You can also pass `imshow` the name of a file containing an image.

```
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 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 4-73.

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. 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.
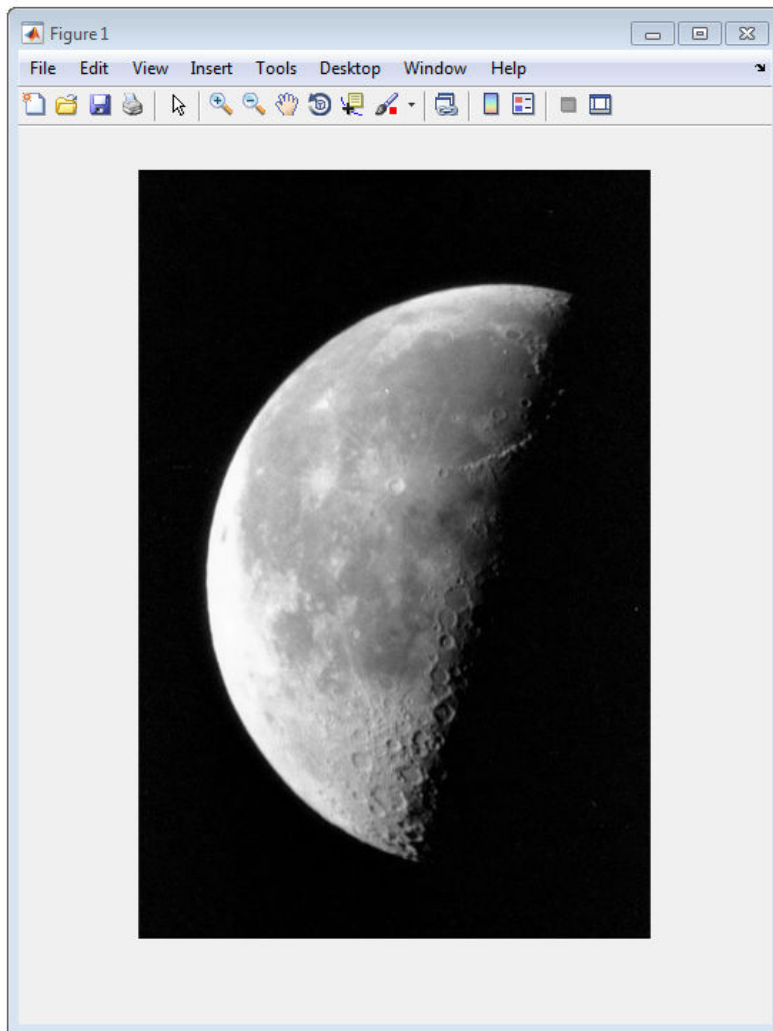
'Loose'
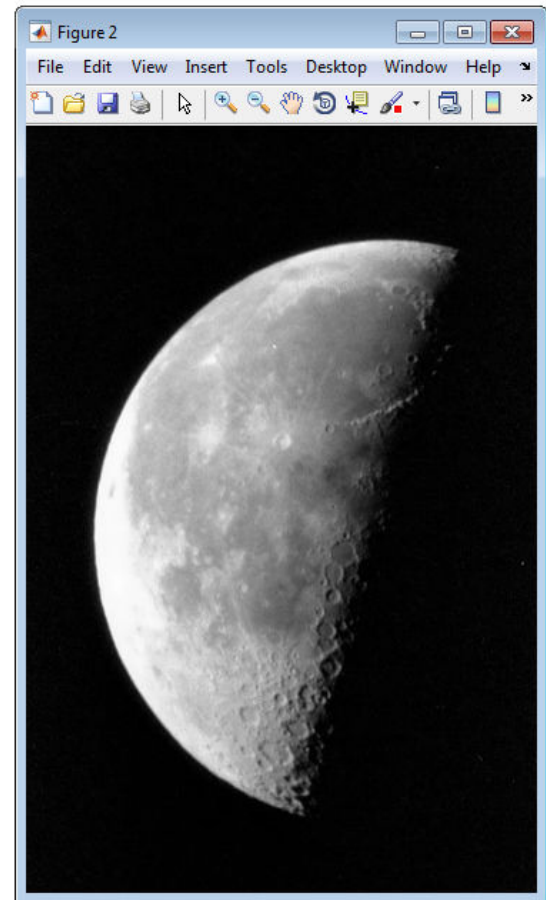
'Tight'

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 4-7

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 `montage` function. By default, `montage` scales the images, depending on the number of images and the size of your screen, and arranges them to form a square. `montage` preserves the aspect ratio of the original images. You can specify the size of the thumbnails using the `ThumbnailSize` parameter.

The images in the montage can be of different types and sizes. `montage` converts indexed images to RGB using the color map 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 "Work with Image Sequences as Multidimensional Arrays" on page 2-48.

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)
```
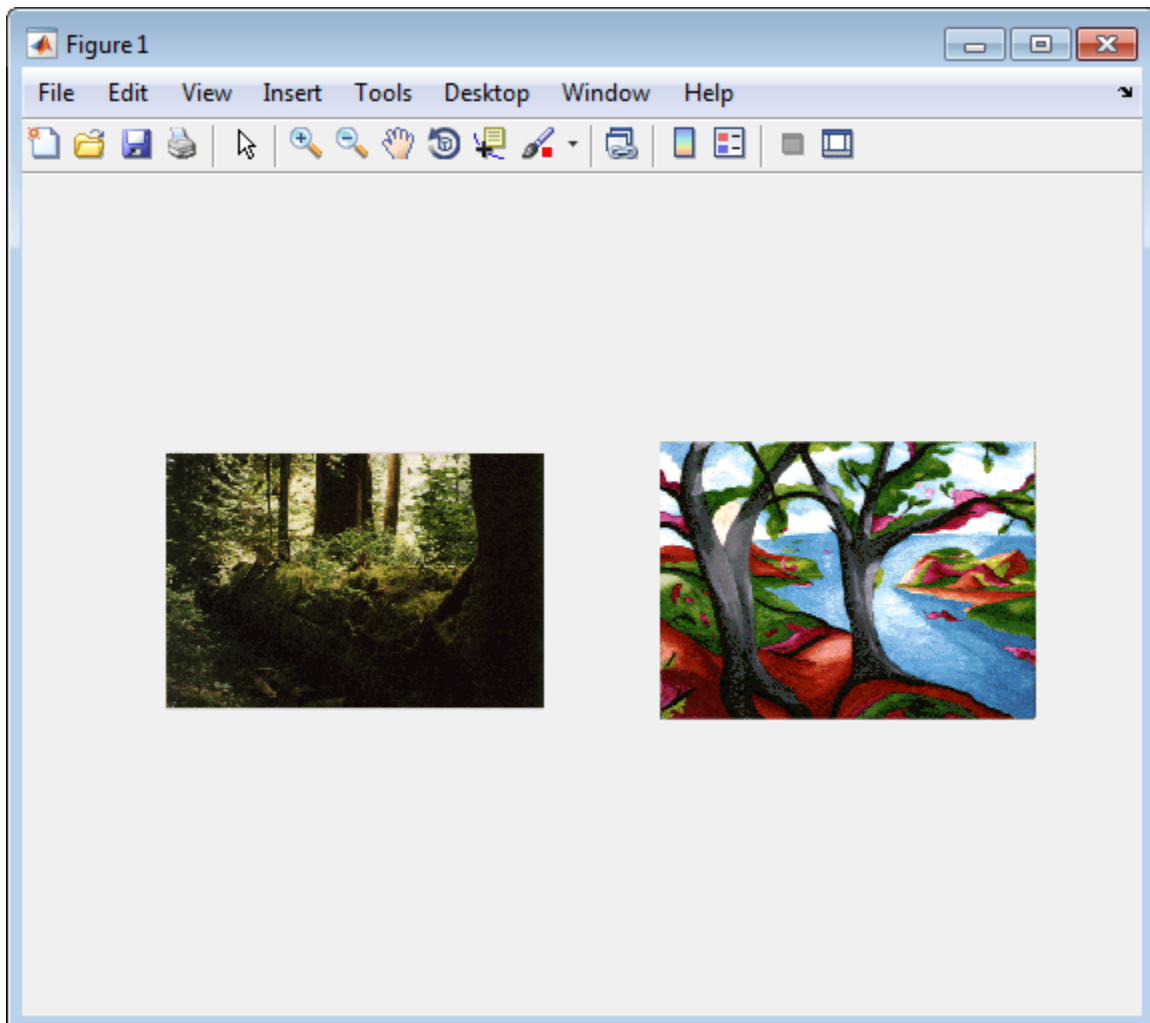
Compare a Pair of Images

The `imshowpair` function displays a pair of images in the same figure window. This display can be helpful when comparing 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
imshow | imshowpair | montage

More About
- "Display an Image in Figure Window" on page 4-3
- "Display Different Image Types" on page 4-73

View Thumbnails of Images in Folder or Datastore

This example shows how to use the **Image Browser** app to view reduced-size versions, called *thumbnails*, of all the images in a folder or in an image datastore. You can also view larger resolution versions of individual images. In addition, you can use the **Image Browser** app to open a selected image in another app, such as the **Image Region Analyzer** app**.**

View Thumbnails of Images in Folder

To view thumbnails of all the images in a folder, open the **Image Browser** app from the MATLAB® toolstrip. On the **Apps** tab, in the **Image Processing and Computer Vision** section, click **Image Browser** .
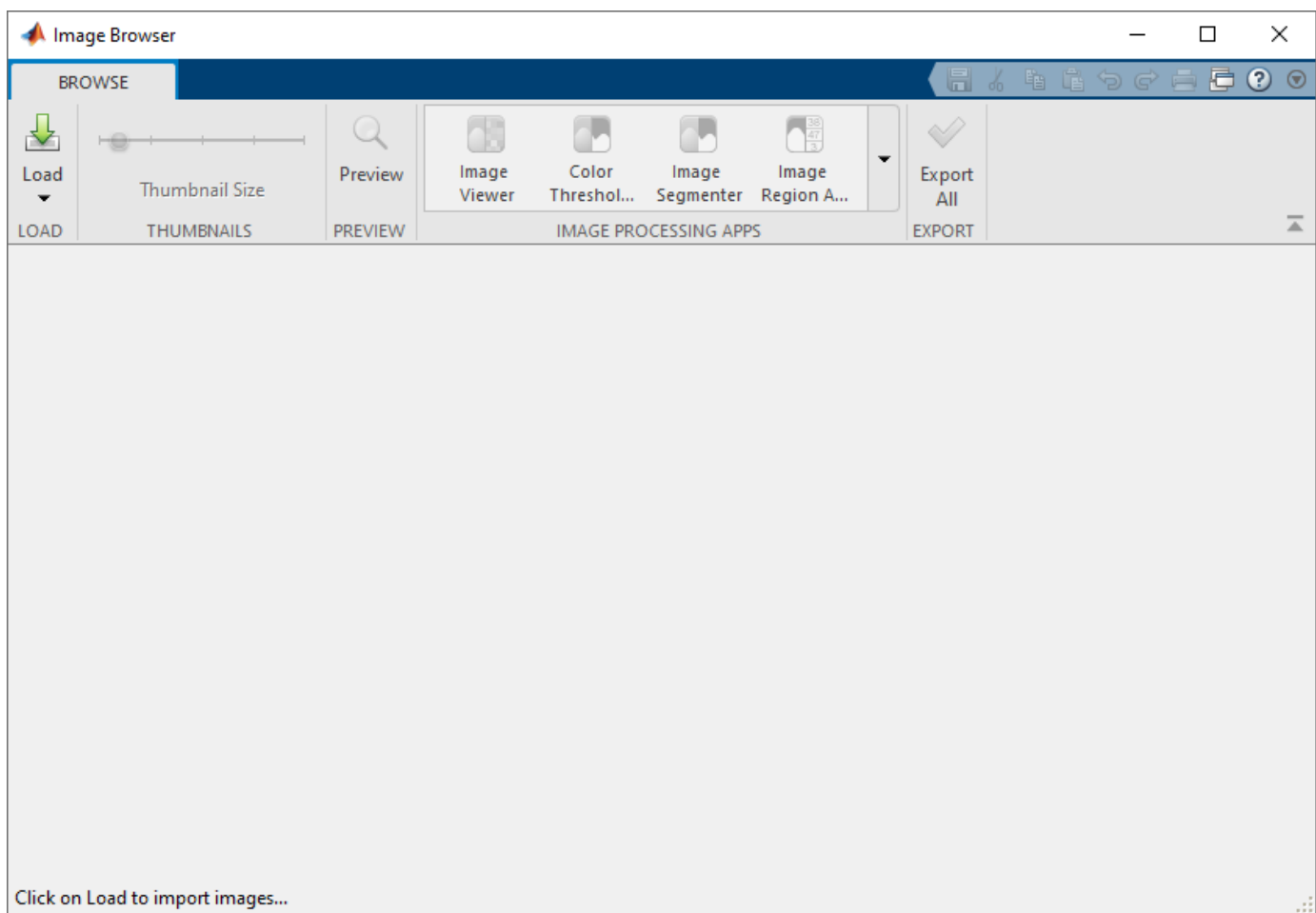
Load images into the app by first clicking **Load** and then select **Load folder of images**. The app displays a file explorer window. Navigate to the folder you want to view. For this example, select the sample image folder, `imdata`.

You can also open the app at the command line using the `imageBrowser` function, specifying the name of the folder you want to view. For example, to view all the images in the sample image folder, use this command: `imageBrowser(fullfile(matlabroot,'toolbox/images/imdata/'));`.

The **Image Browser** app displays thumbnails of all the images in the folder. To adjust the size of the image thumbnails, use the **Thumbnail Size** slider in the app toolstrip.
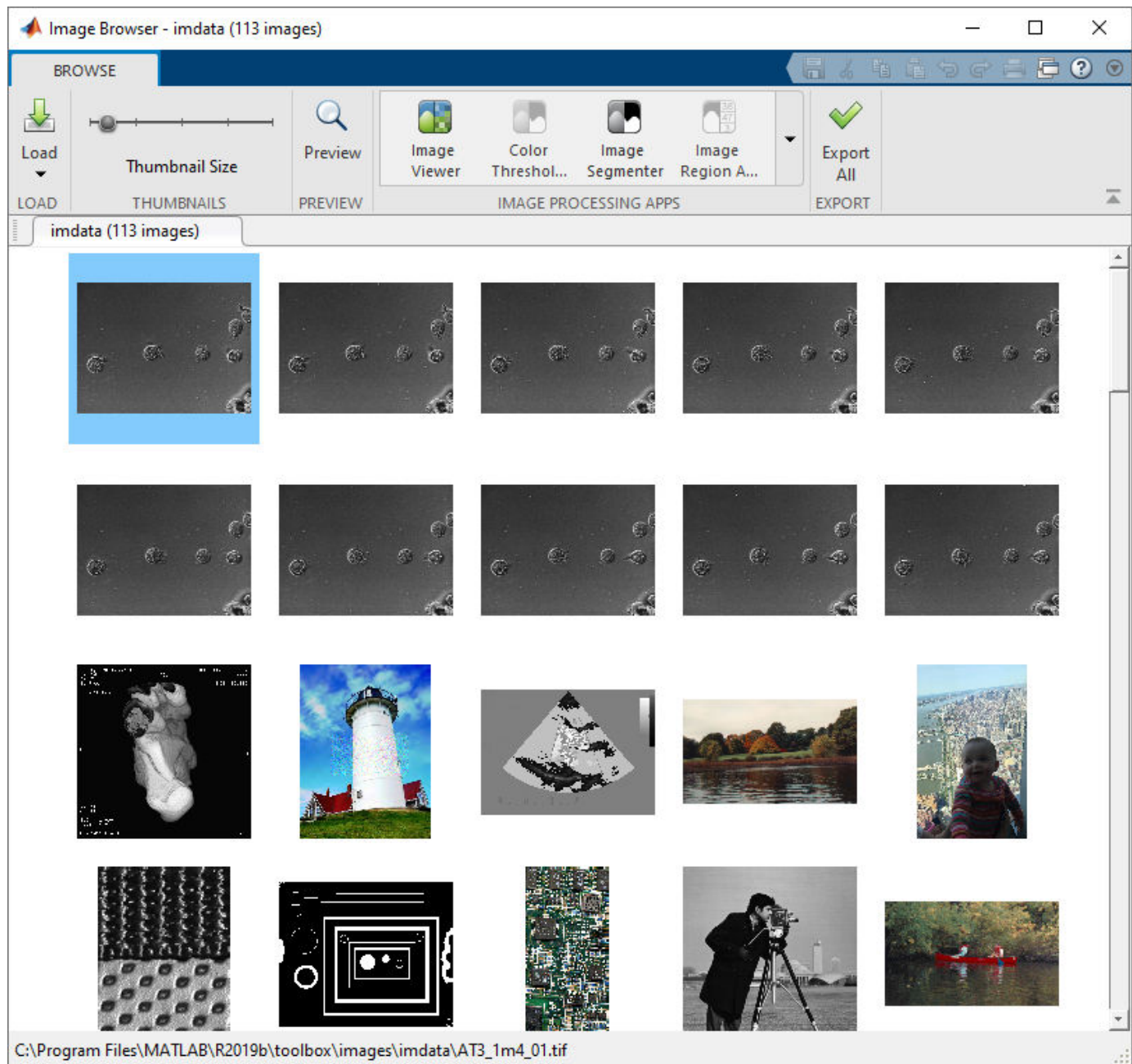
View Thumbnails of Images in Image Datastore

To view thumbnails of all the images in an image datastore, open the **Image Browser** app from the MATLAB® toolstrip. On the **Apps** tab, in the **Image Processing and Computer Vision** section, click **Image Browser** .

Load images into the app by first clicking **Load** and then select **Load image datastore from workspace**.

For this example, create an image datastore by using the `imageDatastore` function containing the images in the `imdata` folder.

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

In the dialog box, select the image datastore variable and click **OK**. The **Image Browser** app displays thumbnails of all the images in the folder.
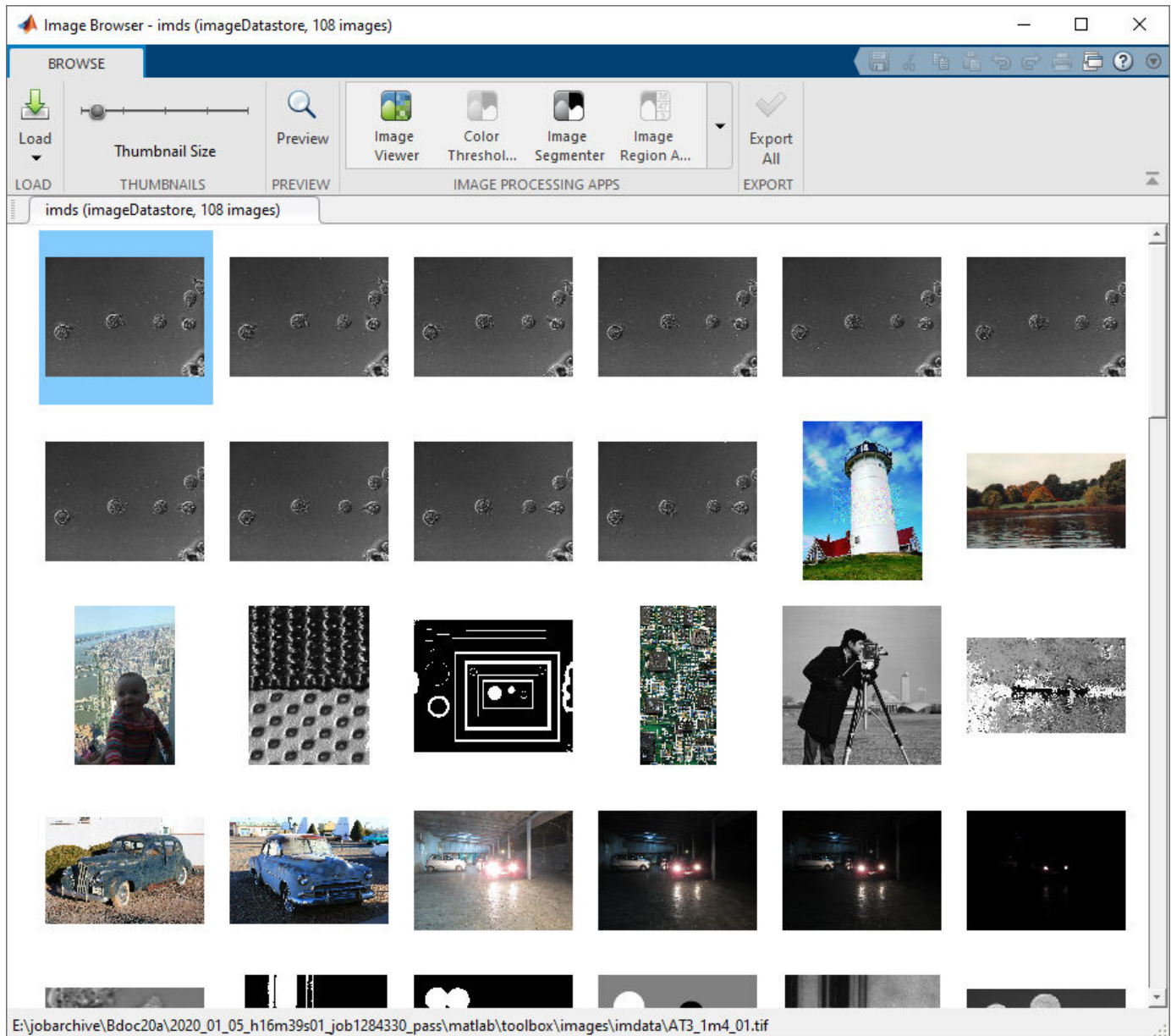
You can also open the app at the command line using the `imageBrowser` function, specifying the name of the image datastore that you want to view. For example, to view all the images in the image datastore created from the sample image folder, use this command: `imageBrowser(imds);`

Explore an Image in More Detail

To get a closer look at an image displayed in the Image Browser app, select the image and click **Preview**. You can also get a preview by double-clicking an image. The app displays the image at a higher resolution in a Preview tab. For example, view the blobs.png binary image in a Preview tab.
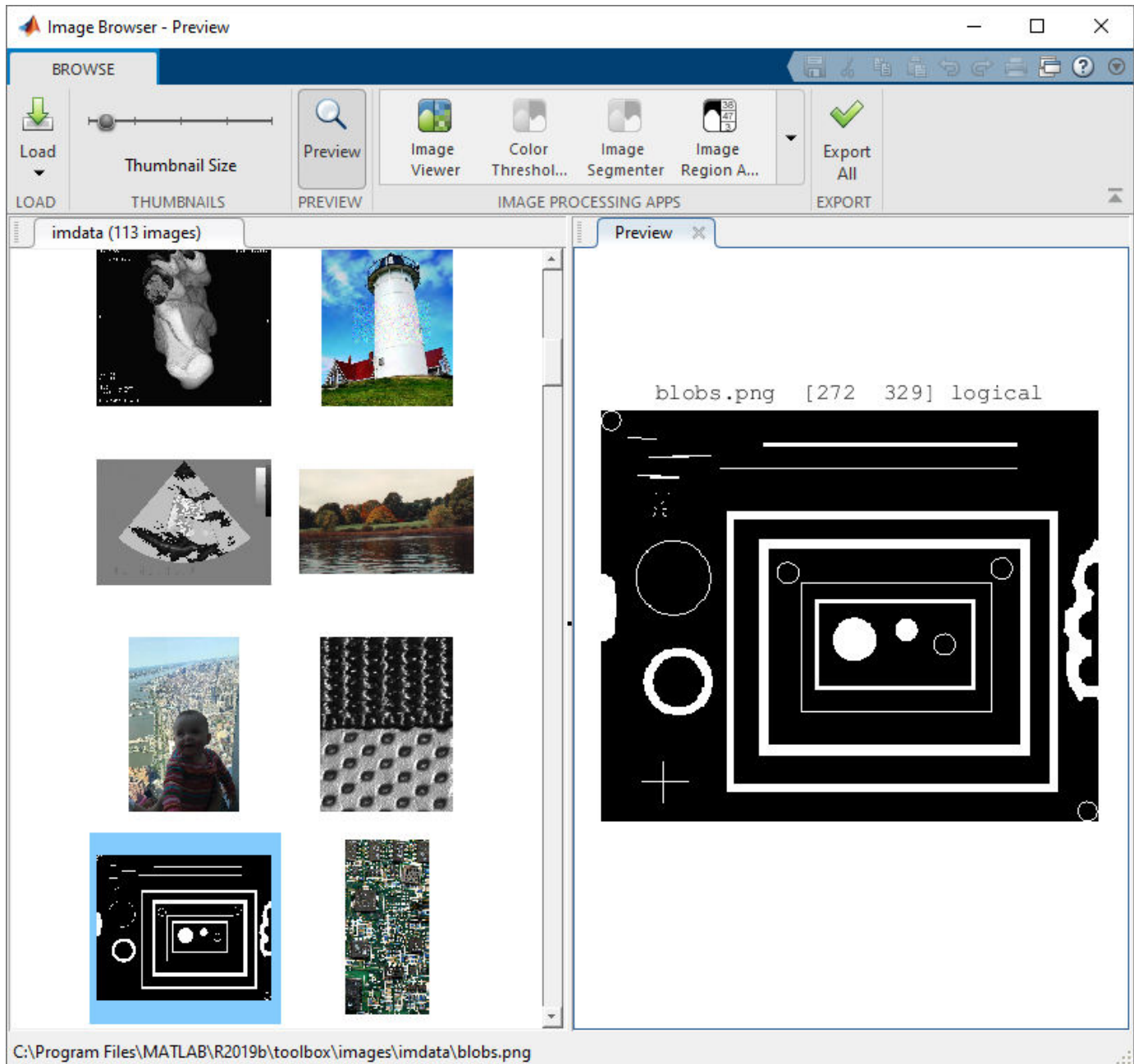
To explore the image displayed in the Preview tab, use the zoom and pan options visible over the top-right corner of the image when you pause over the image.
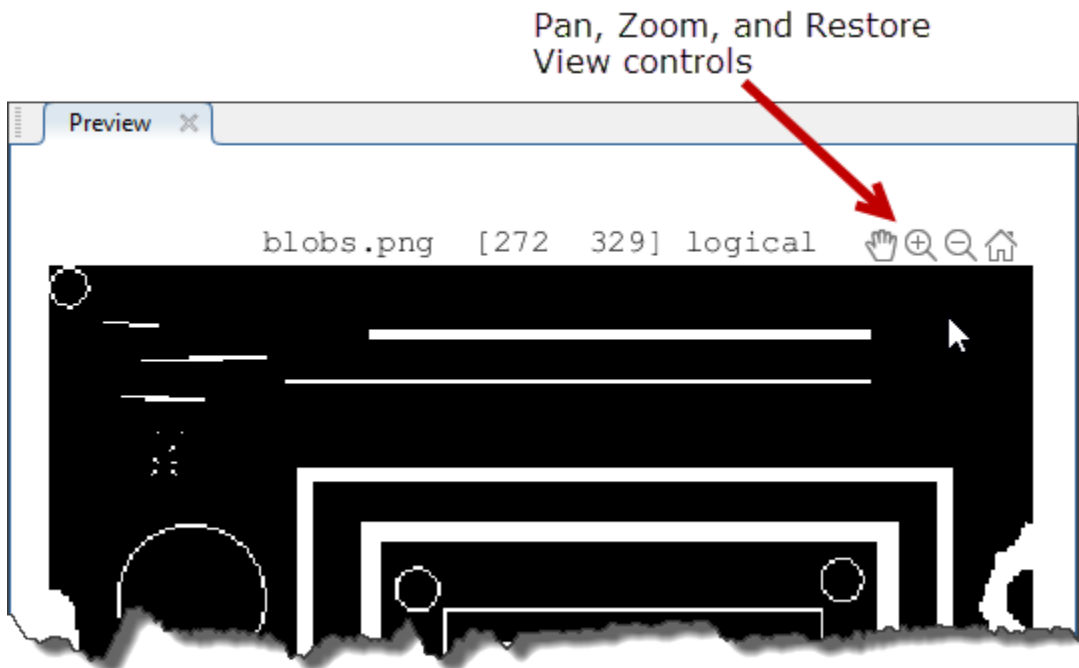
Export Images to the Workspace or an Image Datastore

If you are viewing images in a folder, you can export all the images in an image datastore. On the app toolstrip, click **Export All** and specify the name of the image datastore. To export an individual image from the folder, right-click the image and choose **Export image to workspace.** Specify the name of the workspace variable in which you want to store the image.

If you are viewing images in an image datastore, you can export an individual image into the workspace. Right-click the image, one of the images to the workspace, select the image, right-click, and then 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.
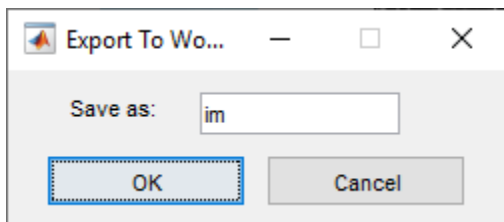
Customize View of Folder

If you are viewing images in a folder in the Image Browser app, you can modify the display of thumbnails. For example, you can delete some of the image thumbnails and then save the modified display in an image datastore. Click **Export All** in the toolstrip and specify the variable name you want use for the datastore. When you open this image datastore in the future, the **Image Browser** app displays only the images you saved. The **Image Browser** app does not delete the images from the file system—it only removes the thumbnails from the display.

4-15

Launch Another App from the Image Browser App

You can select an image in the **Image Browser** app and then open the image in another app. The Image Browser app lets you open the **Image Viewer** app, the **Color Thresholder** app, the **Image Segmenter** app, and the **Image Region Analyzer** app.

For example, in the **Image Browser** app select the blobs.png image. In the app toolstrip, click the **Image Region Analyzer** app. The **Image Region Analyzer** app opens containing the blobs.png image.
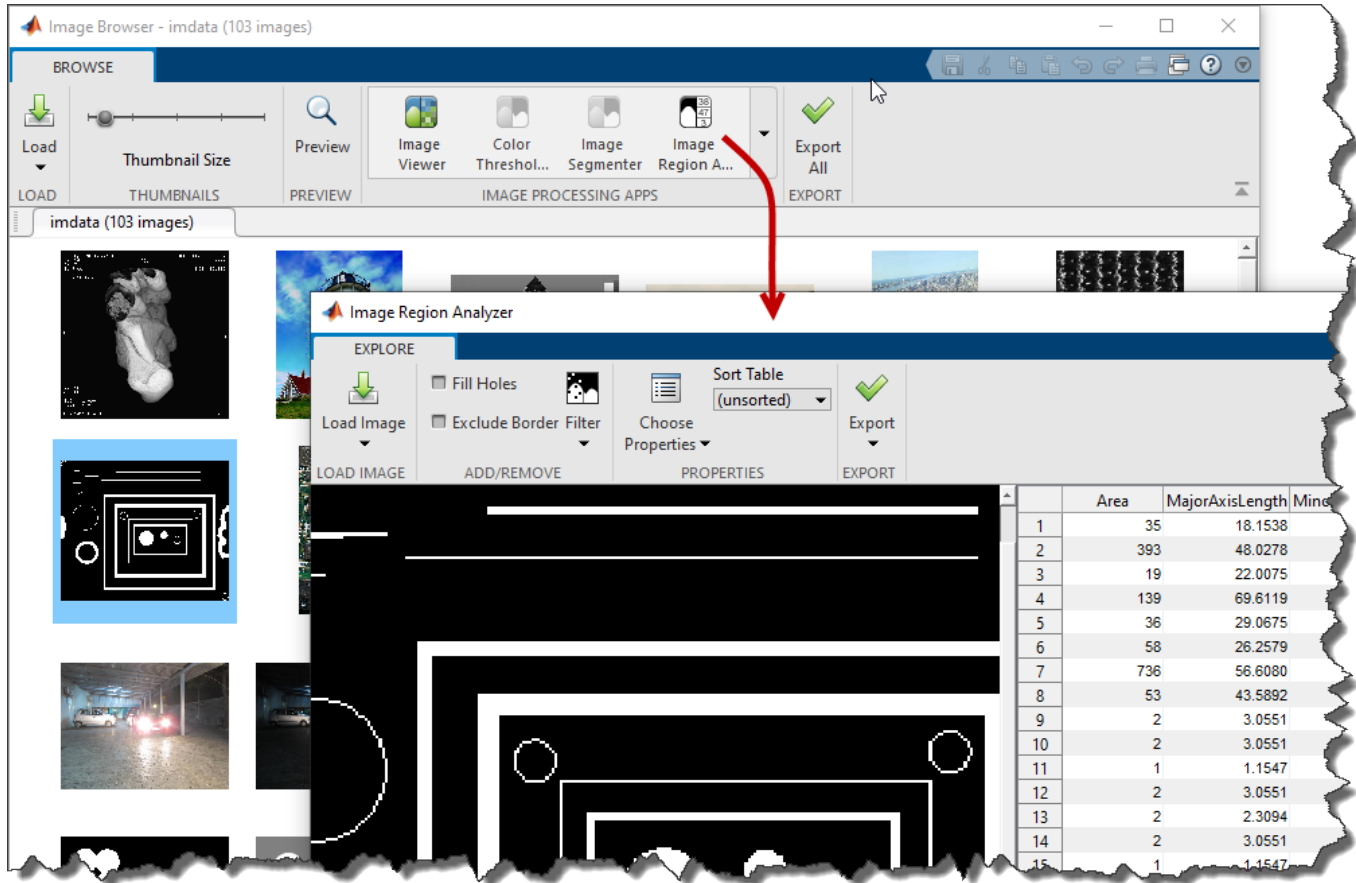
See Also

Image Browser | imageDatastore

More About

- "Getting Started with Datastore"

Get Started with Image Viewer App

In this section...
"Open Image Viewer App" on page 4-18
"Navigate Image in Image Viewer App" on page 4-19
"Get Information about Image Data" on page 4-21
"Modify Image Data" on page 4-23
"Save and Export Results" on page 4-24

The **Image Viewer** app presents an integrated environment for displaying images and performing common image processing tasks. The workflow for using **Image Viewer** typically involves a combination of these steps:

- Open the app and read an image on page 4-18
- Navigate the image on page 4-19
- Get information about the image data on page 4-21
- Modify the image data on page 4-23
- Save and export results on page 4-24

The figure shows an image displayed in **Image Viewer** with many of the related tools open and active.
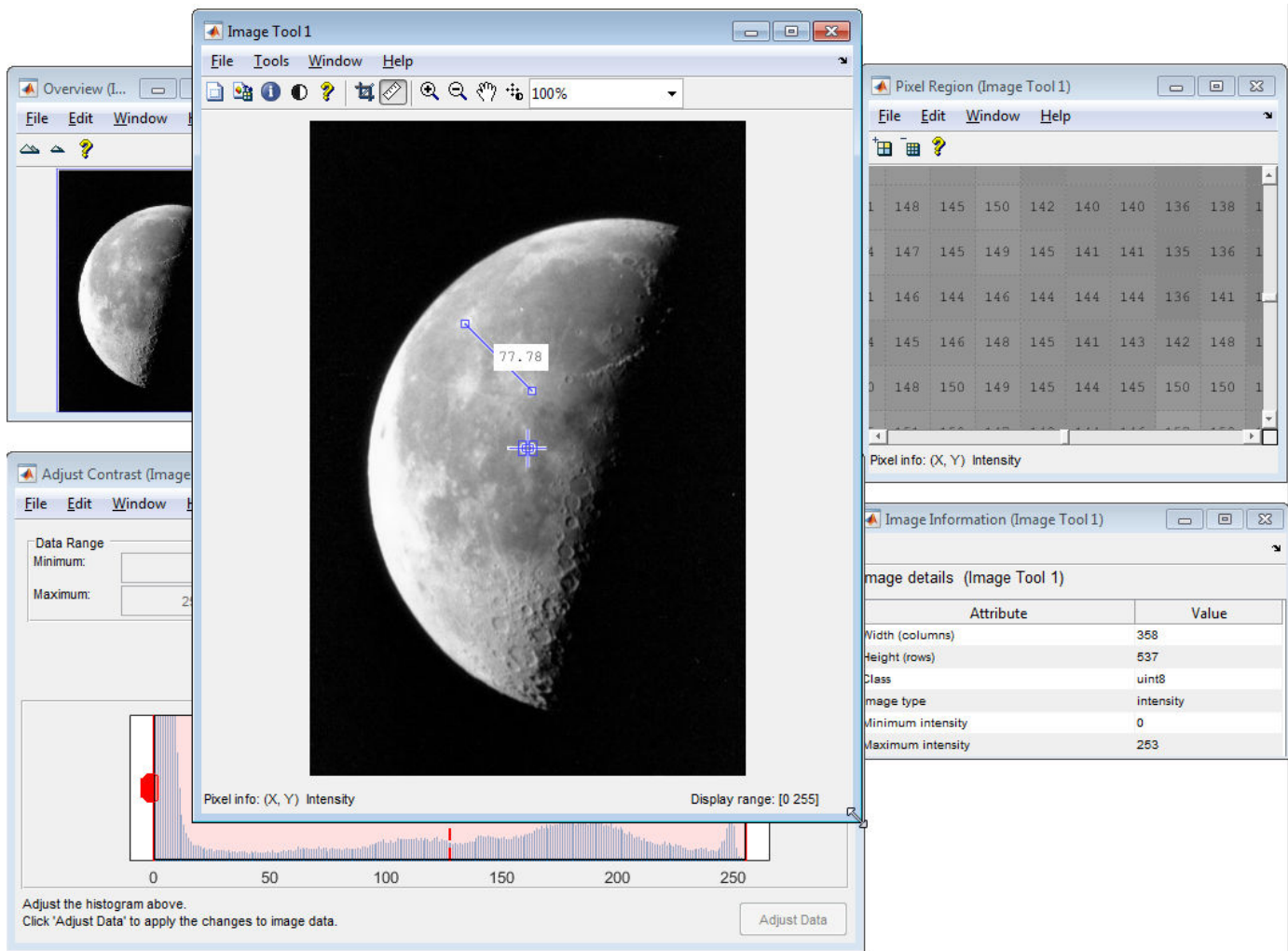
Note You can also access individual tools outside the **Image Viewer** app. To do so, display an image in a figure window using a function such as `imshow`, then create one or more tools using toolbox functions. For example, you can build an image processing app with custom layout and behavior using a combination of individual tools. For more information, see "Interactive Tool Workflow" on page 5-6

Open Image Viewer App

There are three ways to open the **Image Viewer** app. In each case, you can select an image from a variable in the workspace or specify the name of the image file. **Image Viewer** can open any file that can be read by `imread`.

- You can open the **Image Viewer** app from the command line by using the `imtool` function. Use this function when you want to control various aspects of the initial image display, such as the initial magnification, color map, or display range. For example, this code opens **Image Viewer** and loads the image with file name `cameraman.tif`.

```
imtool('cameraman.tif')
```

- You can open the **Image Viewer** app from the Apps tab, under Image Processing and Computer Vision. To bring image data into **Image Viewer** from a file name, select `Open` from the **File** menu. To bring image data into **Image Viewer** from the workspace, select `Import from Workspace` from the **File** menu. Optionally filter the variables in the workspace to show only images of a desired type, such as binary, indexed, intensity (grayscale), or truecolor (RGB) images.
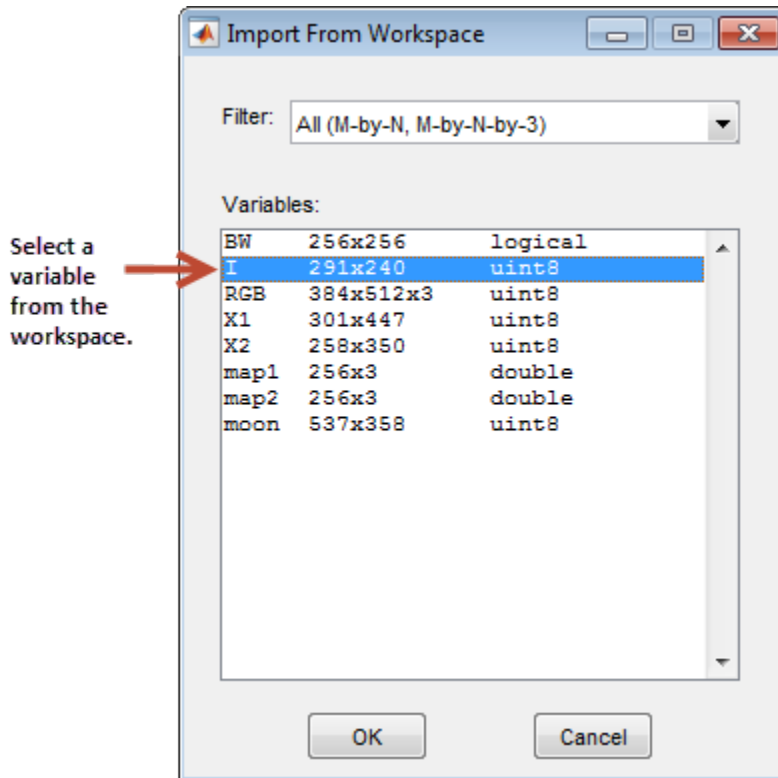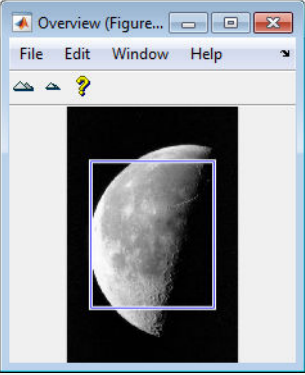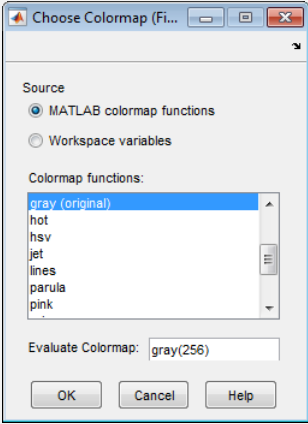
- You can start a new **Image Viewer** from within an existing **Image Viewer** by using the **New** option from the **File** menu.

Note When you specify a file name, **Image Viewer** does not save the image data in a workspace variable. However, you can export the image from **Image Viewer** to the workspace. For more information, see "Save and Export Results" on page 4-24.

Navigate Image in Image Viewer App
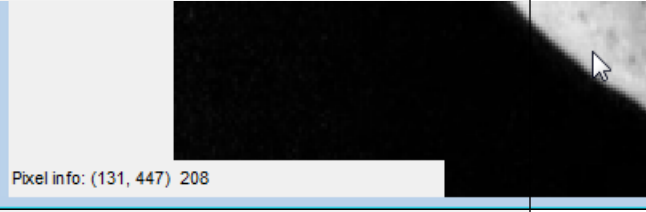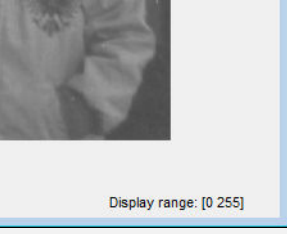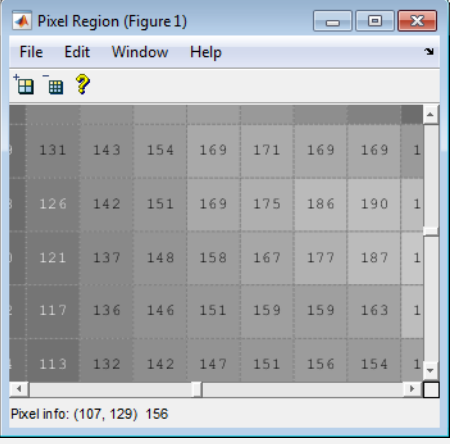
After you open **Image Viewer**, the image appears in the app window. **Image Viewer** provides navigation aids and tools that can help you explore images in detail.
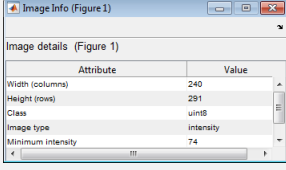
Action	Tools and Navigation Aids	Depiction of Tool
See image overview	To see what portion of the image is currently visible when the image is magnified, use the Overview tool. The Overview tool displays the entire image, scaled to fit. Superimposed over this view of the image is a *detail rectangle* that indicates which portion of the image is currently visible in **Image Viewer**. You can pan and zoom the image visible in the **Image Viewer** window by moving and resizing the detail rectangle in the Overview tool. To achieve better contrast of the detail rectangle against the underlying image, you can change the color of rectangle. Right-click anywhere inside the rectangle and select a color from the **Set Color** option on the context menu. To get the current position and size of the detail rectangle, right-click anywhere inside the rectangle and select **Copy Position** from the context menu. The tool copies the position as a four-element vector of the form `[xmin ymin width height]` to the clipboard. You can paste this position vector into the workspace or another application. To 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 4-80 for more information.	

Action	Tools and Navigation Aids	Depiction of Tool
Magnify image	To enlarge or shrink an image by specifying a scale factor, use the Magnification box. You can also set the magnification such that the image completely fills the figure window. This tool appears automatically on the right on the toolstrip. To enlarge or shrink an image by clicking the image, use the Zoom tool. The tool centers the new view of the image on the spot where you clicked. **Note** You can also zoom by using the **Ctrl+Plus** or **Ctrl+Minus** keys. These are the **Plus(+)** and **Minus(-)** keys on the numeric keypad of your keyboard. When **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 interpolation methods used for resizing images, see `imresize`.	
Pan image	To pan an image that is too large to fit in the **Image Viewer** window, use scroll bars or the Pan tool.	
Choose image color map	To enhance the visibility of features in grayscale and indexed images, you can change the image color map using the Colormap tool. You can select a MATLAB color map or a color map variable from the workspace. You can also create a color map by entering a MATLAB command. **Image Viewer** does not provide a color bar. To add a color bar, open the image in another figure window. For more information, see "Save and Export Results" on page 4-24.	

Get Information about Image Data

Image Viewer provides tools that can help you get information about pixel values and other aspects of the image data.
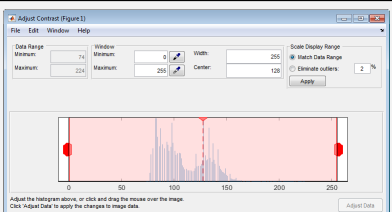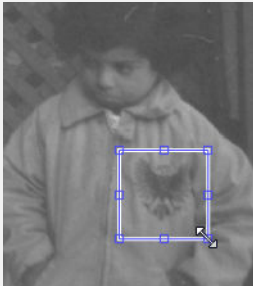
Tool	Description	Depiction of Tool
Pixel Information tool	Get the (*x*, *y*) coordinates and the value of a single pixel under the pointer. For more information, see "Determine Individual Pixel Values in Image Viewer" on page 4-26.	Pixel info: (131, 447) 208
Display Range tool	Determine the display range of grayscale image data. The tool is not enabled for RGB, indexed, or binary images. For more information, see "Determine Image Display Range in Image Viewer" on page 4-31.	Display range: [0 255]
Pixel Region tool	Get information about a group of pixels. For more information, see "Determine Pixel Values in an Image Region" on page 4-27.	Pixel Region (Figure 1) — File Edit Window Help. 131 143 154 169 171 169 169; 126 142 151 169 175 186 190; 121 137 148 158 167 177 187; 117 136 146 151 159 159 163; 113 132 142 147 151 156 154. Pixel info: (107, 129) 156
Distance tool	Measure the Euclidean distance between two pixels. For more information, see "Measure Distance Between Pixels in Image Viewer App" on page 4-33	128.97

Tool	Description	Depiction of Tool
Image Information tool	Get information about image and image file metadata. The Image Information tool always provides basic information about the width, height, class, and image type. For grayscale and indexed images, this information includes the minimum and maximum intensity values. If you select an image to open in **Image Viewer** by specifying a file name, then the Image Information tool also displays image metadata. This metadata is the same information returned by the `imfinfo` function or the `dicominfo` function.	

Modify Image Data

Image Viewer provides tools that can help you adjust the image contrast and crop an image.

By default, when you close **Image Viewer**, the app does not save the modified image data. However, you can export the modified image to a file or save the modified data in a workspace variable. For more information, see "Save and Export Results" on page 4-24.

Tool	Description	Depiction of Tool
Adjust Contrast tool	Adjust the contrast of an image by setting a window over a histogram of pixel values. For more information, see "Adjust Image Contrast in Image Viewer App" on page 4-36.	
Window/ Level tool	Adjust the contrast of an image by interacting with the image. For more information, see "Adjust Image Contrast in Image Viewer App" on page 4-36.	
Crop Image tool	Crop an image to a rectangular region of interest. For more information, see "Crop Image Using Image Viewer App" on page 4-40.	

Save and Export Results

Image Viewer enables you to export image data to the workspace, save the image data to file, and open images in a new figure window. When saving and exporting image data, changes to the display range are not preserved. If you would like to preserve your changes, then use the `imcontrast` function.

Destination	Procedure

Create workspace variable	There are three ways to create a workspace variable from the image data in **Image Viewer**.
	
	• You can use the **Export to Workspace** option on the **Image Viewer File** menu.
	
	• If you start the app by using the `imtool` function and specify a handle to the tool, then you can use the `getimage` function and specify the handle to the tool. For example, this code opens the image with file name `moon.tif` in an **Image Viewer** then exports the image to the variable `moon`.
	
	`t = imtool('moon.tif');`
	`moon = getimage(t);`
	
	• If you start the app without specifying a handle to the tool, then you can use the `getimage` function and specify a handle to the image object within the figure. For example, this code opens the image with file name `moon.tif` in an **Image Viewer** then exports the image to the variable `moon`.
	
	`imtool('moon.tif')`
	`moon = getimage(imgca);`

Save to file	Use the Save Image tool by selecting the **Save as** option on the **Image Viewer File** menu. This tool enables you to navigate your file system to determine where to save the file, specify the file name, and choose the file format.

Destination	Procedure
Open new figure window	Select the **Print to Figure** option from the **File** menu. You can use this figure window to see a color bar and print the image. For more information, see "Add Color Bar to Displayed Grayscale Image" on page 4-78 and "Print Images" on page 4-80.

See Also
Image Viewer | imtool

Get Pixel Information in Image Viewer App

In this section...
"Determine Individual Pixel Values in Image Viewer" on page 4-26
"Determine Pixel Values in an Image Region" on page 4-27
"Determine Image Display Range in Image Viewer" on page 4-31

The **Image Viewer** app provides tools that enable you to see the pixel values and coordinates of individual pixels and groups of pixels. You can save the pixel location and value information.

Determine Individual Pixel Values in Image Viewer

Image Viewer displays information about the location and value of individual pixels in an image in the bottom left corner of the tool. The pixel value and location information represent the pixel under the current location of the pointer. **Image Viewer** updates this information as you move the pointer over the image.

The format of the pixel information depends on the image type.

Image Type	Pixel Information	Example
Intensity	(X,Y) Intensity	(13,30) 82
Indexed	(X,Y) <index> [R G B]	(2,6) <4> [0.29 0.05 0.32]
Binary	(X,Y) BW	(12,1) 0
Truecolor	(X,Y) [R G B]	(19,10) [15 255 10]
Floating point image with `CDataMapping` property set to `direct`	(X,Y) value <index> [R G B]	(19,10) 82 <4> [15 255 10]

The figure shows **Image Viewer** with pixel location and grayscale pixel value displayed in the Pixel Information tool.

Pixel under the pointer

Pixel location and intensity value
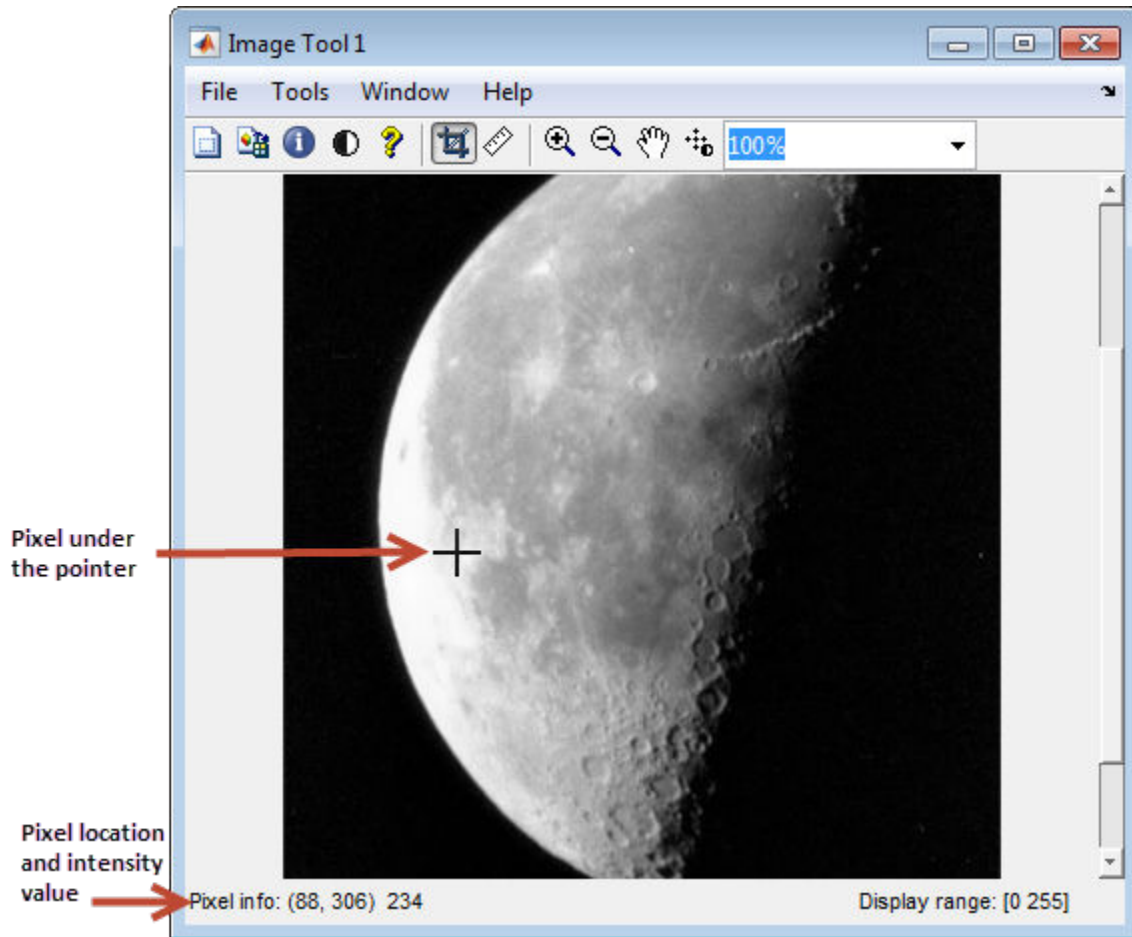
Note You can also obtain pixel value information from a figure with `imshow` by using the `impixelinfo` function.

To save the pixel location and value information displayed, right-click a pixel in the image and choose the **Copy pixel info** option. **Image Viewer** copies the *x*- and *y*-coordinates and the pixel value to the clipboard. You can paste this pixel information into the MATLAB workspace or another application by right-clicking and selecting **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 **Image Viewer**, use the Pixel Region tool. The Pixel Region tool superimposes a rectangle, called the *pixel region rectangle*, over the image displayed in **Image Viewer**. This rectangle defines the group of pixels that are displayed, in extreme close-up view, in the Pixel Region tool window.

The figure shows **Image Viewer** with the Pixel Region tool. Note how the Pixel Region tool includes its own Pixel Information tool in the display.

Pixel Region tool button

Pixel Region rectangle
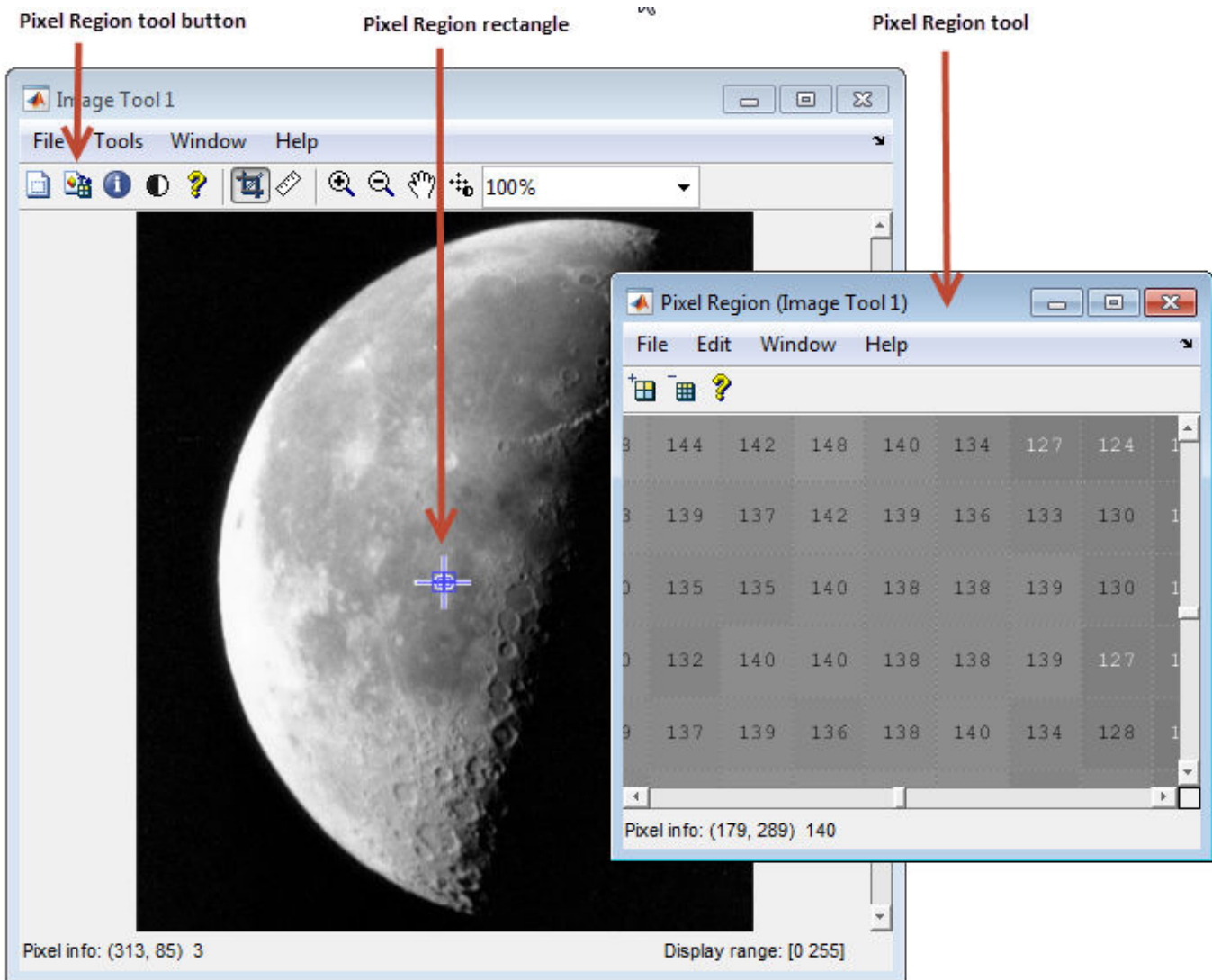
Pixel Region tool

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

- "Select a Region" on page 4-29
- "Customize the View" on page 4-29
- "Determine the Location of the Pixel Region Rectangle" on page 4-30
- "Print the View of the Image in the Pixel Region Tool" on page 4-31

Note You can also obtain pixel region information from a figure, such as displayed using `imshow`, by using the `impixelregion` function.

Select a Region

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. **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 "Determine the Location of the Pixel Region Rectangle" on page 4-30.

Using the mouse, position the pointer over the pixel region rectangle. The pointer changes to the fleur shape, ✛.

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.

Customize 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.
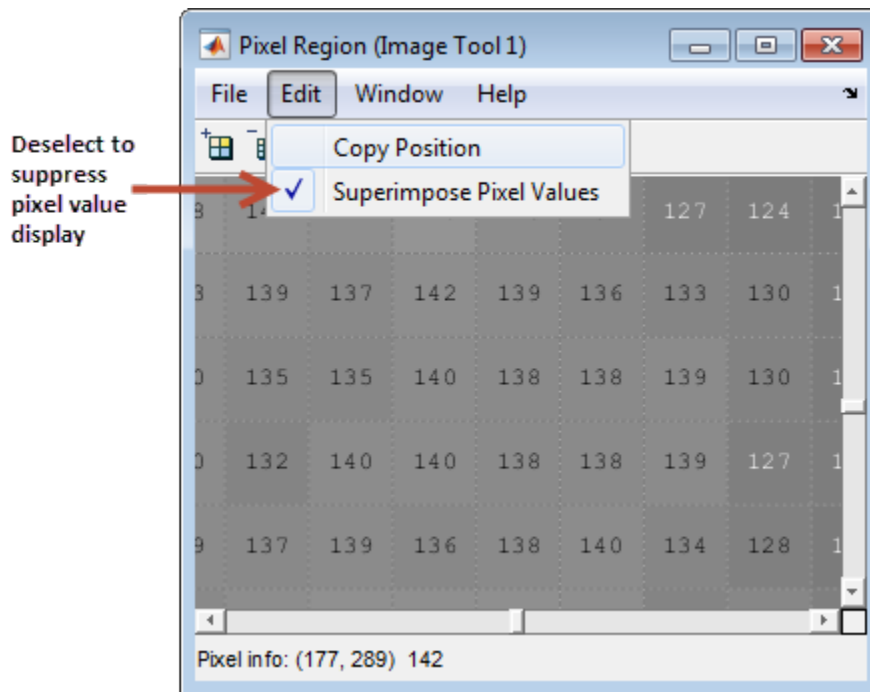
Determine the Location of the Pixel Region Rectangle

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 figure shows these components of the Pixel Region tool.
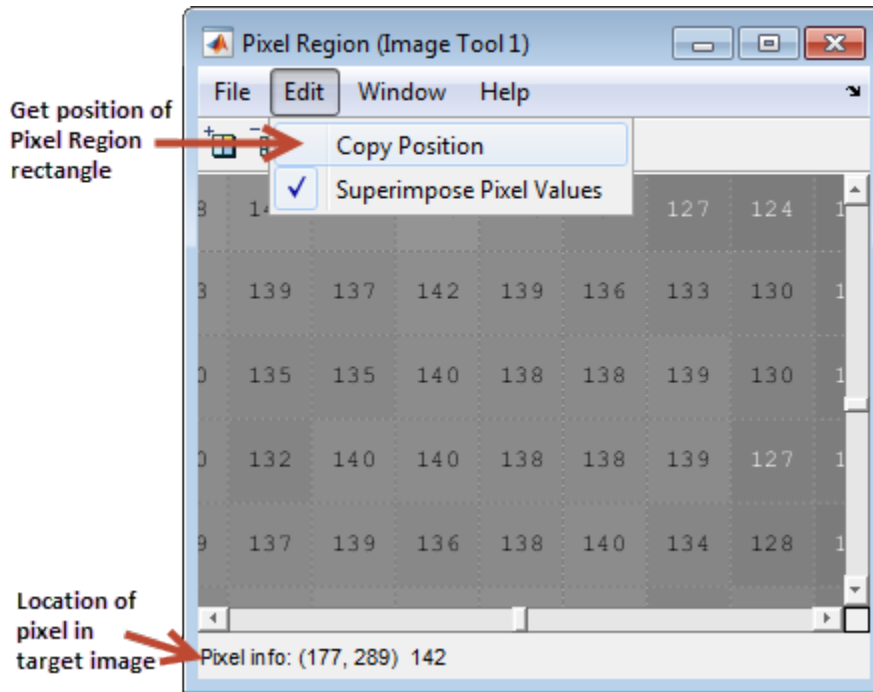
Print the View of the Image in the Pixel Region Tool

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 4-80 for more information.

Determine Image Display Range in Image Viewer

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 color map. `CLim` is a two-element vector `[cmin cmax]` specifying the `CData` value to map to the first color in the color map (`cmin`) and the `CData` value to map to the last color in the color map (`cmax`). Data values in between are linearly scaled.

For grayscale images, **Image Viewer** displays this information in the Display Range tool at the bottom right corner of the window. **Image Viewer** does not show the Display Range tool for indexed, truecolor, or binary images.

The figure shows an image with display range information in **Image Viewer**.
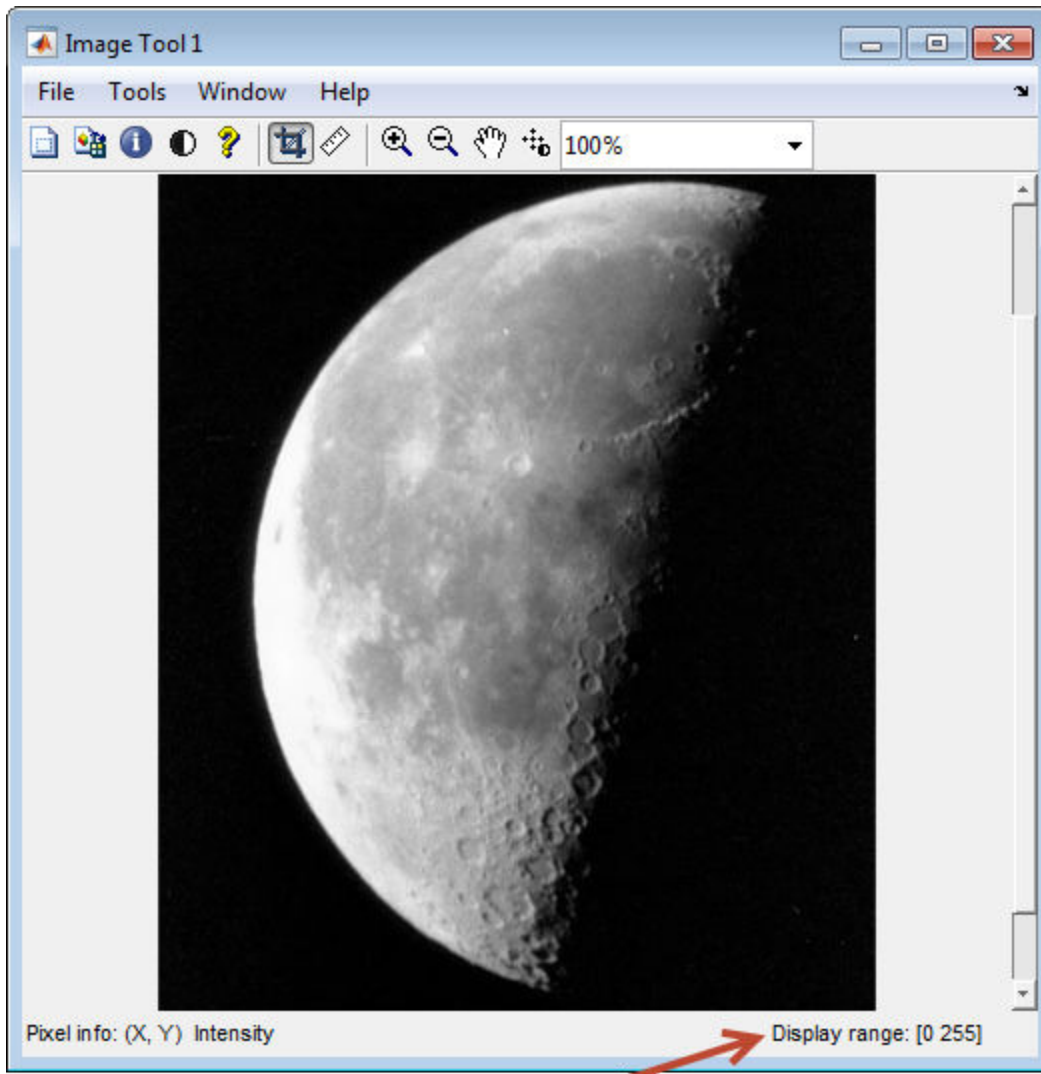
Display Range tool

Note You can also obtain the image display range from a figure, such as displayed using `imshow`, by using the `imdisplayrange` function.

You can change the image contrast and brightness by adjusting the display range. For more information, see "Adjust Image Contrast in Image Viewer App" on page 4-36.

See Also

Image Viewer | `imdisplayrange` | `impixelinfo` | `impixelregion` | `imtool`

More About

- "Get Started with Image Viewer App" on page 4-17

Measure Distance Between Pixels in Image Viewer App

In this section...
"Determine Distance Between Pixels Using Distance Tool" on page 4-33
"Export Endpoint and Distance Data" on page 4-34
"Customize the Appearance of the Distance Tool" on page 4-35

The **Image Viewer** app enables you to measure the Euclidean distance between two pixels using the Distance tool. The Distance tool displays the line, the endpoints, and label with the distance measurement. The tool specifies the distance in data units determined by the XData and YData properties, which is pixels, by default. You can save the endpoint locations and distance information.

Determine Distance Between Pixels Using Distance Tool

After you open an image in **Image Viewer**, open the Distance tool by clicking **Distance** tool button in the **Image Viewer** toolbar or selecting **Measure Distance** from the **Tools** menu.. For more information about opening an image in **Image Viewer**, see "Open Image Viewer App" on page 4-18.

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 first endpoint, hold down the mouse button, drag the cross hairs to the second endpoint, and release the button.

The figure shows a distance line with the line endpoints and distance measurement label.

Distance tool button
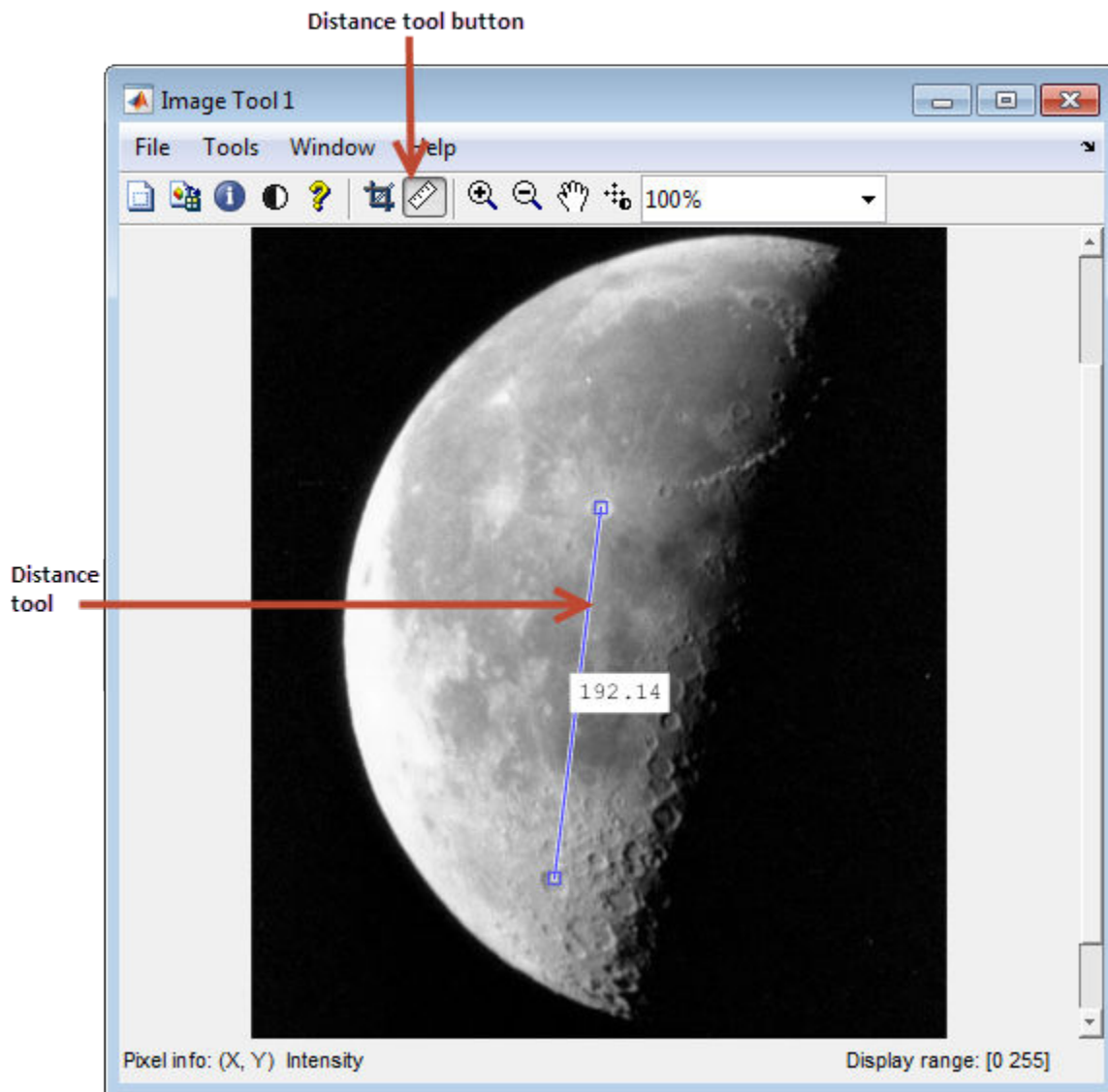
Note   You can also use a Distance tool with a figure, such as displayed using `imshow`, by using the `imdistline` function.

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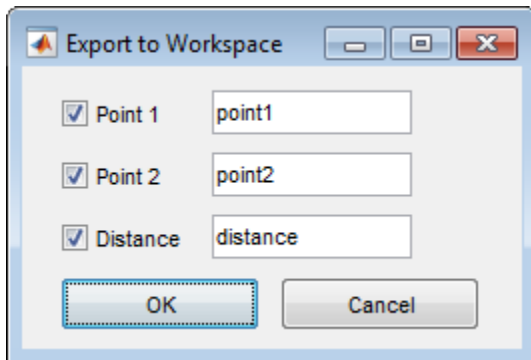
After you click **OK**, the Distance tool creates the variables in the workspace, as in the following example.

```
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.

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

See Also
Image Viewer | imdistline | imtool

More About
- "Get Started with Image Viewer App" on page 4-17

Adjust Image Contrast in Image Viewer App

In this section...
"Adjust Contrast and Brightness using Adjust Contrast Tool" on page 4-36
"Adjust Contrast and Brightness Using Window/Level Tool" on page 4-38
"Make Contrast Adjustments Permanent" on page 4-38

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. Contrast adjustment works by manipulating the *display range* of the image. Pixels with value equal to or less than the minimum value of the display range appear black. Pixels with value equal to or greater than the maximum value of the display range appear white. Pixels within the display range appear as a smooth gradient of shades of gray. Adjusting the contrast spreads the pixel values across the display range, revealing much more detail in the image.

The default display range of an image is determined by the image class. For instance, the display range for images with data type `uint8` is 0 to 255. If the *data range*, or actual minimum and maximum pixel values of the image data, is more narrow than the display range, then the displayed image does not display all shades of gray from black to white. In this case, you can improve the image contrast by shrinking the display range to match the data range. To highlight certain image features, you can further shrink the display range to be less than the data range.

The **Image Viewer** app offers two tools that enable you to change the contrast or brightness of an image interactively. The Window/Level tool enables you to change contrast and brightness by simply dragging the mouse over the image. The Adjust Contrast tool displays a histogram of image pixel values and graphical representation of the display range so that you can see how the display range relates to pixel values.

Adjust Contrast and Brightness using Adjust Contrast Tool

After you open an image in **Image Viewer**, open the Adjust Contrast tool by clicking **Adjust Contrast** ◖ in the **Image Viewer** toolbar. For more information about opening an image in **Image Viewer**, see "Open Image Viewer App" on page 4-18.

The Adjust Contrast tool displays a histogram of pixel values and information about the data range. The data range is a fixed property of the image data and does not change when you adjust the display range using the Adjust Contrast tool. The Adjust Contrast tool also displays a red-tinted rectangular box, called a *window*, overlaid on the histogram. The window directly corresponds to the display range of the image. The tool also shows the precise values of the precise values of the range and location of the window on the histogram.

For example, in the figure, the histogram for the image shows that the data range of the image is 7 to 253. The display range is the default display range for the `uint8` data type, 0 to 255. Therefore, no pixels appear as black or white. Further, the histogram shows that many pixel values are clustered in the middle of the display range, which explains why it is challenging to distinguish between the medium-gray pixel values.
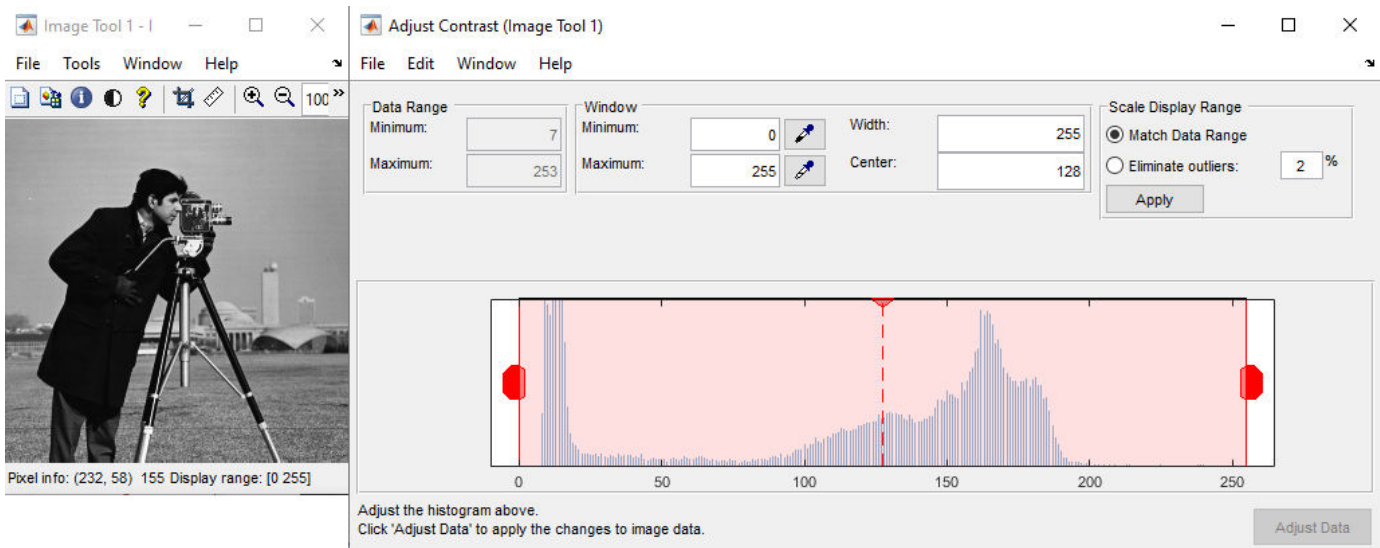
To increase the contrast of the image, narrow the range of pixel values. To increase the brightness of an image, shift the range towards large pixel values. There are three ways to adjust the display range of the image.

- You can adjust the display range interactively by manipulating the window. Adjust the minimum and maximum value of the display range by clicking and dragging the left and right edges of the window. Change the position of the window by clicking and dragging the interior of the window.

- You can enter specific values for the extent, width, and center of the window. 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.

- You can let the Adjust Contrast tool scale the display range automatically. When you select the **Match data range** option, the tool makes the display range equal to the data range of the image. When you select the **Eliminate outliers** option, the tool removes an equal percentage of pixels from the top and bottom of the display range. By default, the tool eliminates 2% of pixels, in other words, the top 1% and the bottom 1% of pixels. (You can perform this same operation using the `stretchlim` function.)

The display range of the image displayed in the **Image Viewer** window updates in real time when you change the display range. **Image Viewer** also updates the display range values displayed in the lower right corner of the app window.

For example, in the figure, the window indicates that the display range of the image is 12 to 193. Bright pixels appear much brighter than in the original image. There is better contrast between pixels with similar gray values, such as for the pillars and roof of the domed buildings.
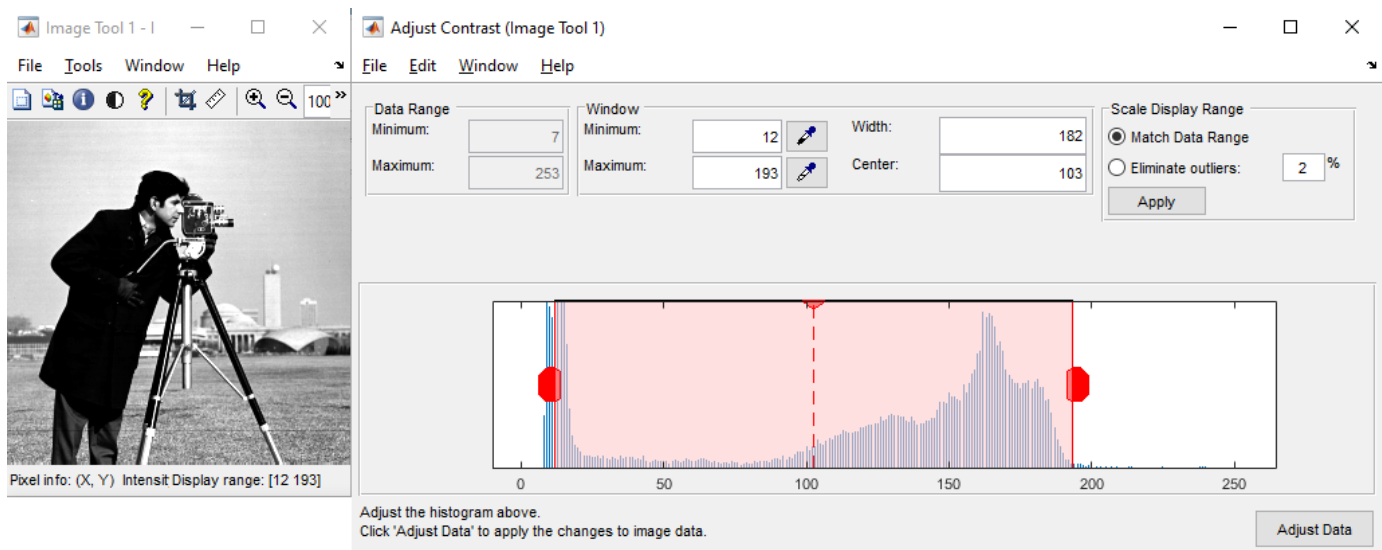
Adjust Contrast and Brightness Using Window/Level Tool

To start the Window/Level tool, click **Window/Level** ⊕ in the **Image Viewer** toolbar or select the **Window/Level** option from the **Image Viewer Tools** menu.

Move the pointer over the image. The pointer changes to the Window/Level cursor ⊕. To adjust the image contrast, click and drag the mouse horizontally. To adjust image brightness, click and drag the mouse vertically.

If you also have the Adjust Contrast tool open, then contrast adjustments you make using the Window/Level tool immediately adjust the window in the Adjust Contrast tool. For example, if you increase the brightness using the Window/Level tool, then the window in the Adjust Contrast tool shifts to the right.

When you close the Adjust Contrast tool, the Window/Level tool remains active. To stop the Window/ Level tool, click the **Window/Level** button ⊕ or any of the navigation buttons in the **Image Viewer** toolbar.

Make Contrast Adjustments Permanent

The Adjust Contrast tool and Window/Level tool adjust the values of the pixels used to display the image in **Image Viewer** but do not change the actual image data.

When using the Adjust Contrast tool, you can modify pixel values in the image to reflect the contrast adjustments by clicking the **Adjust Data** button. When you click the **Adjust Data** button, the histogram updates. 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.

When you close **Image Viewer**, the app 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. For more information, see "Save and Export Results" on page 4-24.

See Also
Image Viewer | imcontrast | imtool

More About
- "Get Started with Image Viewer App" on page 4-17

Crop Image Using Image Viewer App

Cropping an image means creating a new image from a part of an original image. The **Image Viewer** app enables you to crop an image interactively by using Crop Image tool.

After you open an image in **Image Viewer**, start the Crop Image tool by clicking **Crop Image** 🗗 in the **Image Viewer** toolbar or by selecting **Crop Image** from the **Image Viewer Tools** menu. For more information about opening an image in **Image Viewer**, see "Open Image Viewer App" on page 4-18.

When you move the pointer over the image, the pointer changes to cross hairs $+$. 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 figure shows a crop rectangle being defined using the Crop Image tool.
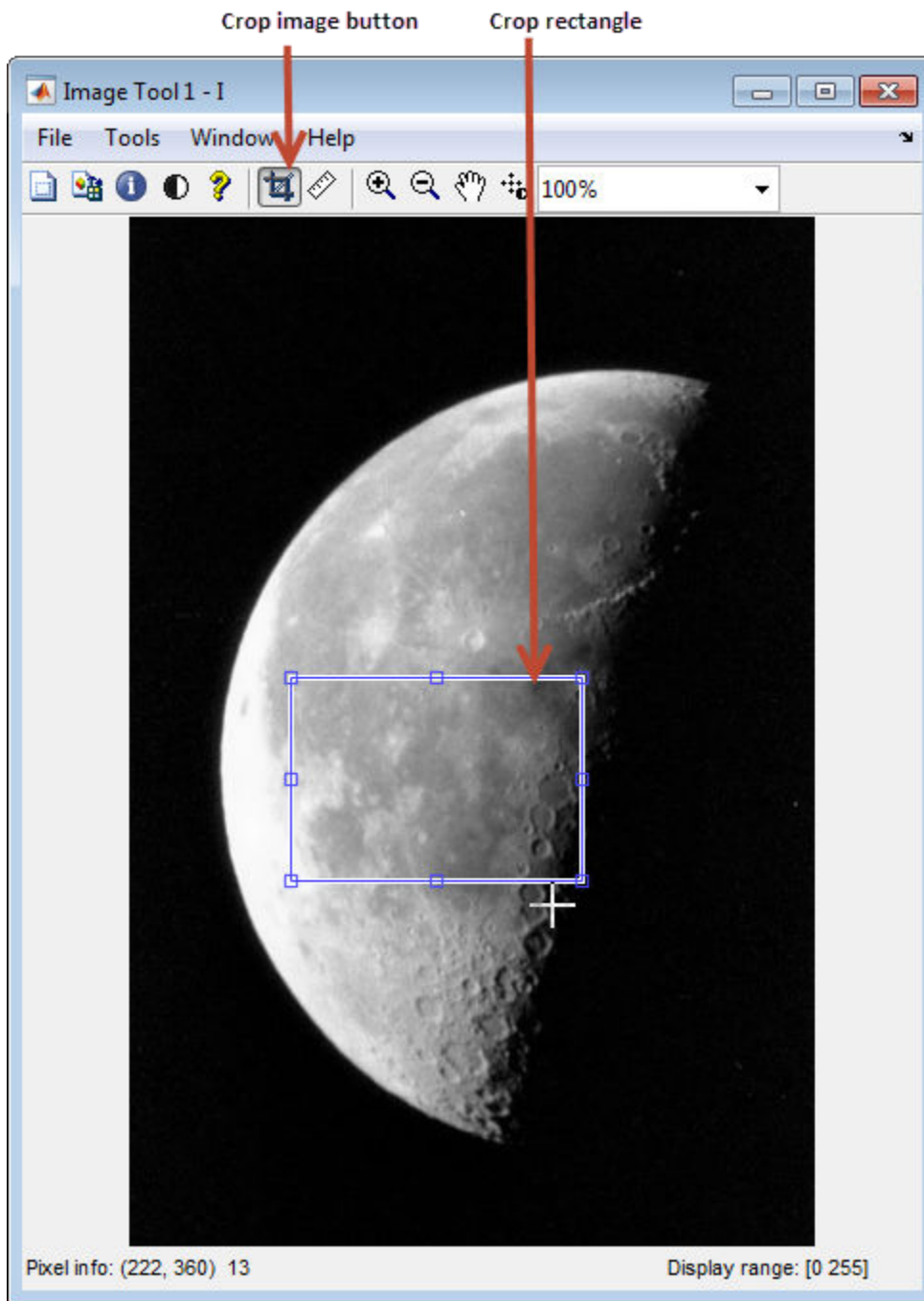
When you are finished defining the crop region, perform the crop operation. Double-click the left mouse button or right-click inside the region and select **Crop Image** from the context menu. **Image Viewer** displays the cropped image. If you have other modular interactive tools open, then they will update to show the newly cropped image.

By default, if you close **Image Viewer**, then the app 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.

See Also
Image Viewer | `imcrop` | `imtool`

More About

- "Get Started with Image Viewer App" on page 4-17

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 4-44
"View the Volume Data in the Volume Viewer" on page 4-46
"Adjust View of Volume Data in Volume Viewer" on page 4-49
"Refine the View with the Rendering Editor" on page 4-51
"Save Volume Viewer Rendering and Camera Configuration Settings" on page 4-56

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.
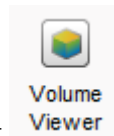
```
load mri

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 toolstrip , open the Apps tab and under Image
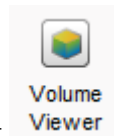
Processing and Computer Vision, click [Volume Viewer]. You can also open the app using the `volumeViewer` command.

Load volumetric data into Volume Viewer app. Click **Import 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.
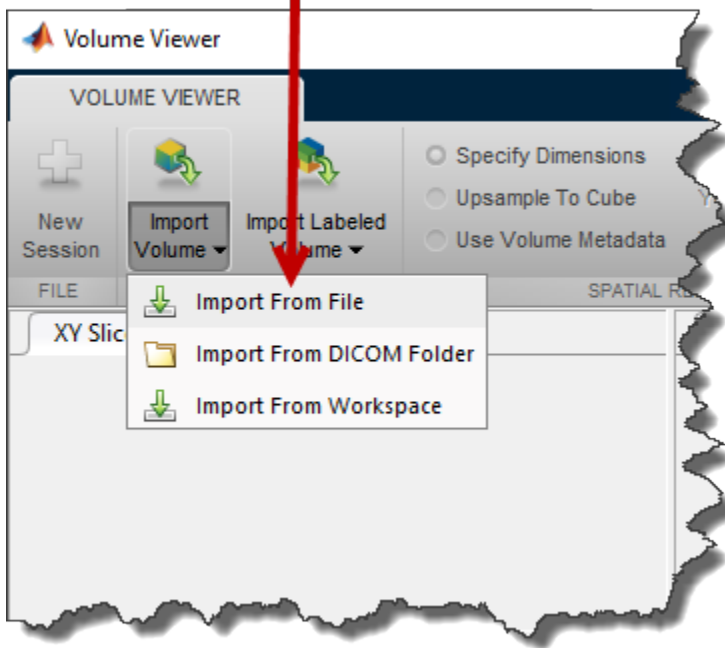
Select the workspace variable in the **Import from Workspace** dialog box and click **OK**.
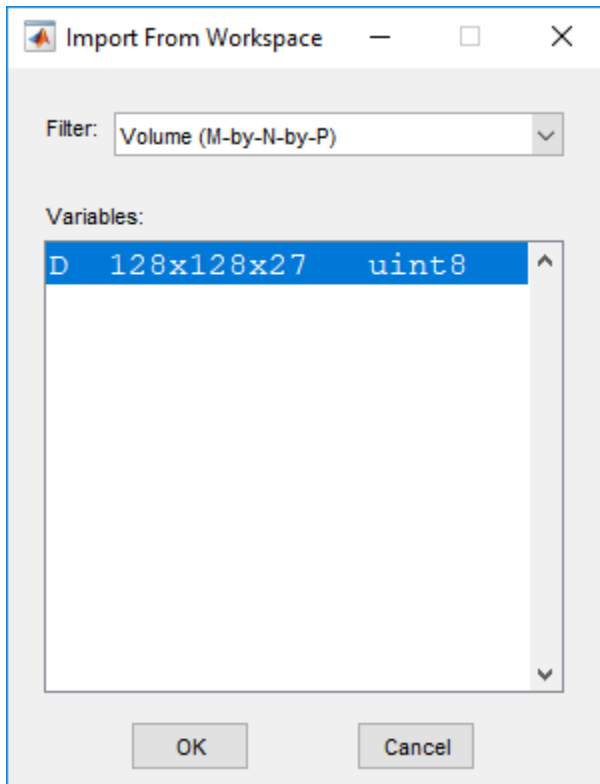
To start a new instance of the Volume Viewer, click **New Session**.
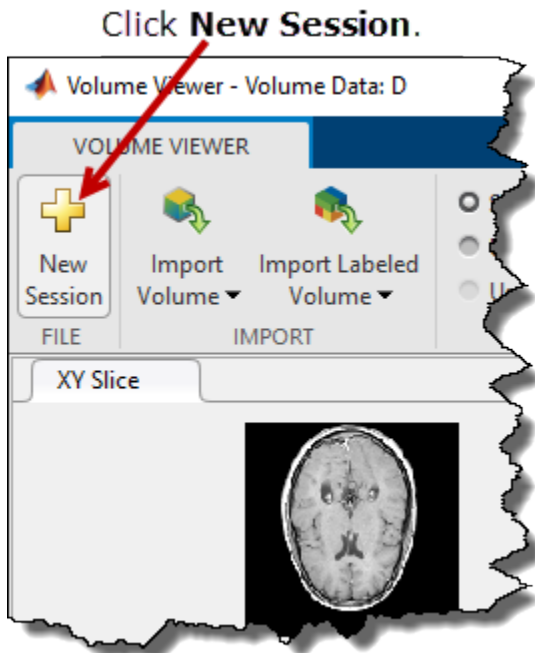
When you create a new session, this option deletes all the data currently in the viewer. Click **Yes** to create the new session.
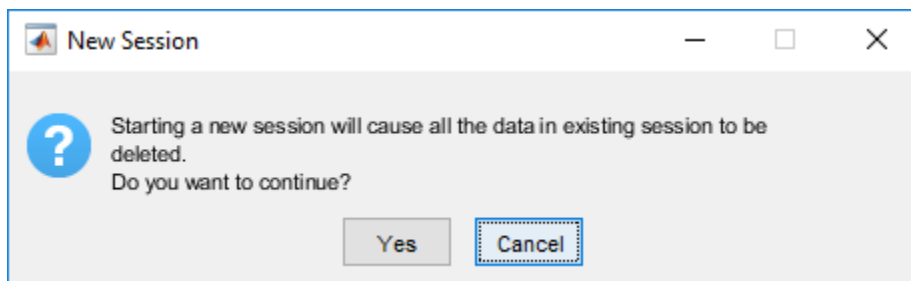
View the Volume Data in the Volume Viewer

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 position of the axes in the **Orientation Axes** window reflects the spatial orientation of the image as you rotate it.
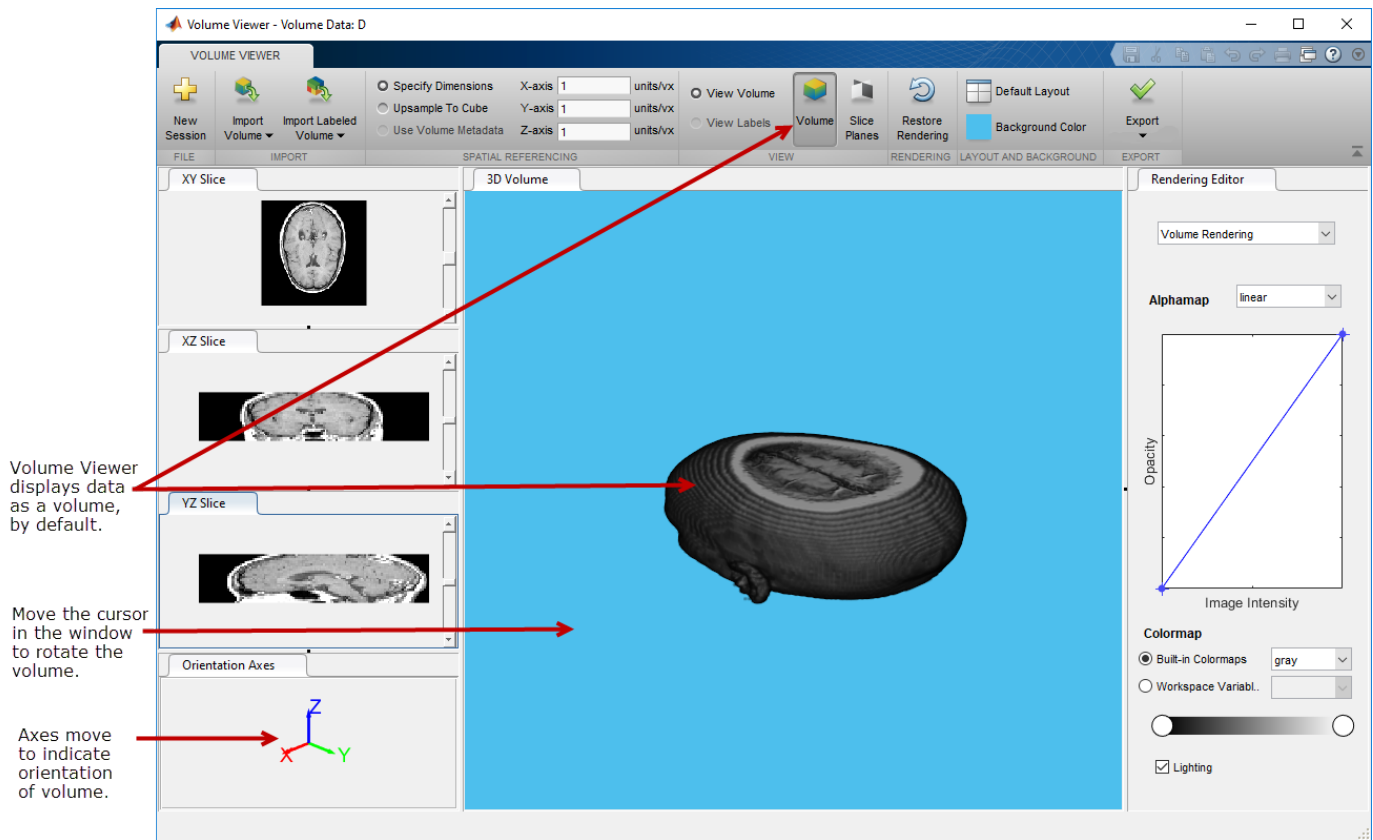
To change the background color used in the display window, click **Background Color** and select a color.

Select color for background.
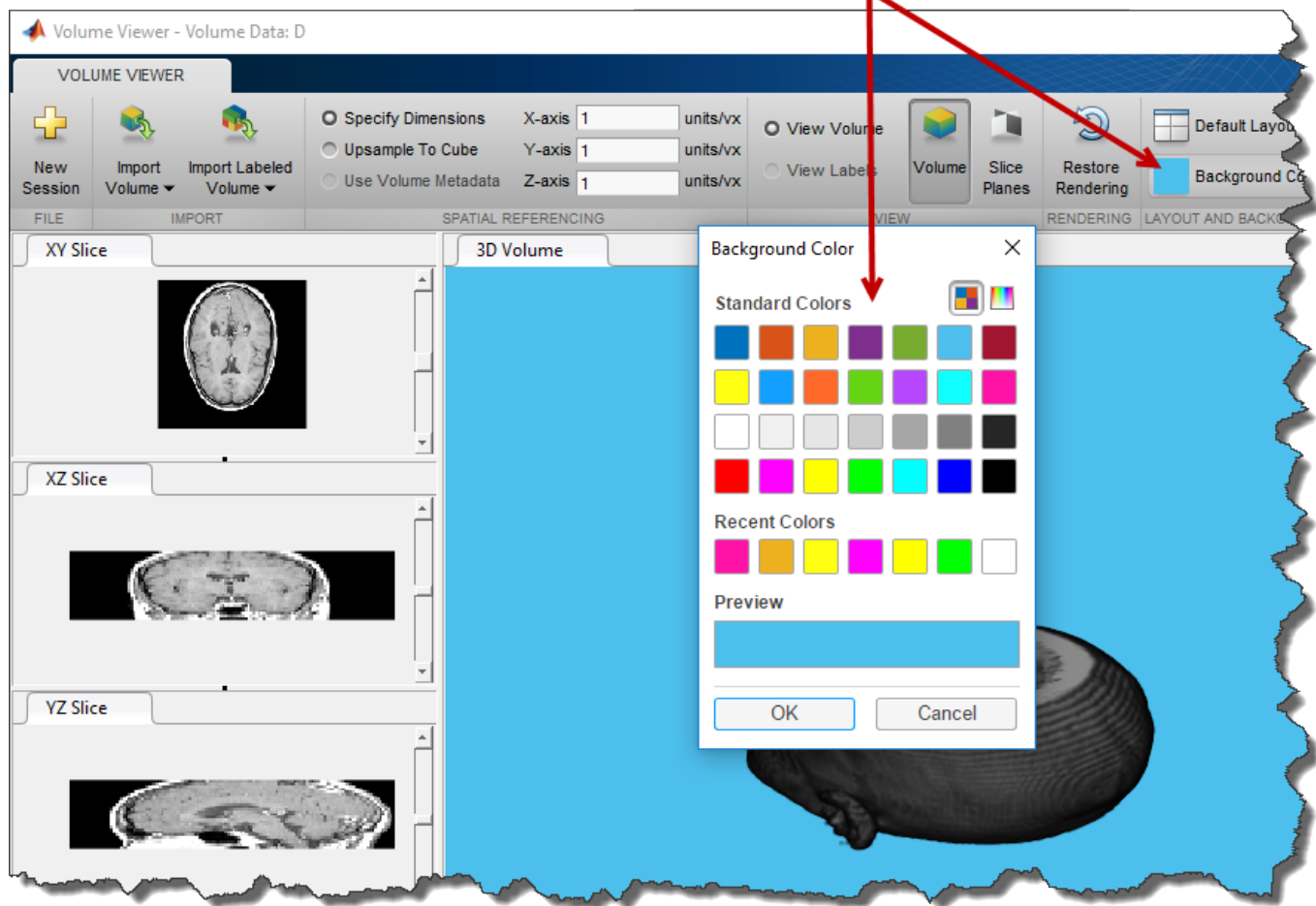
View the MRI data as a set of slice planes. Click **Slice Planes**. You can also zoom in and rotate this view of the data. Use the scroll bars in the three slice windows to view individual slices in any of the planes.
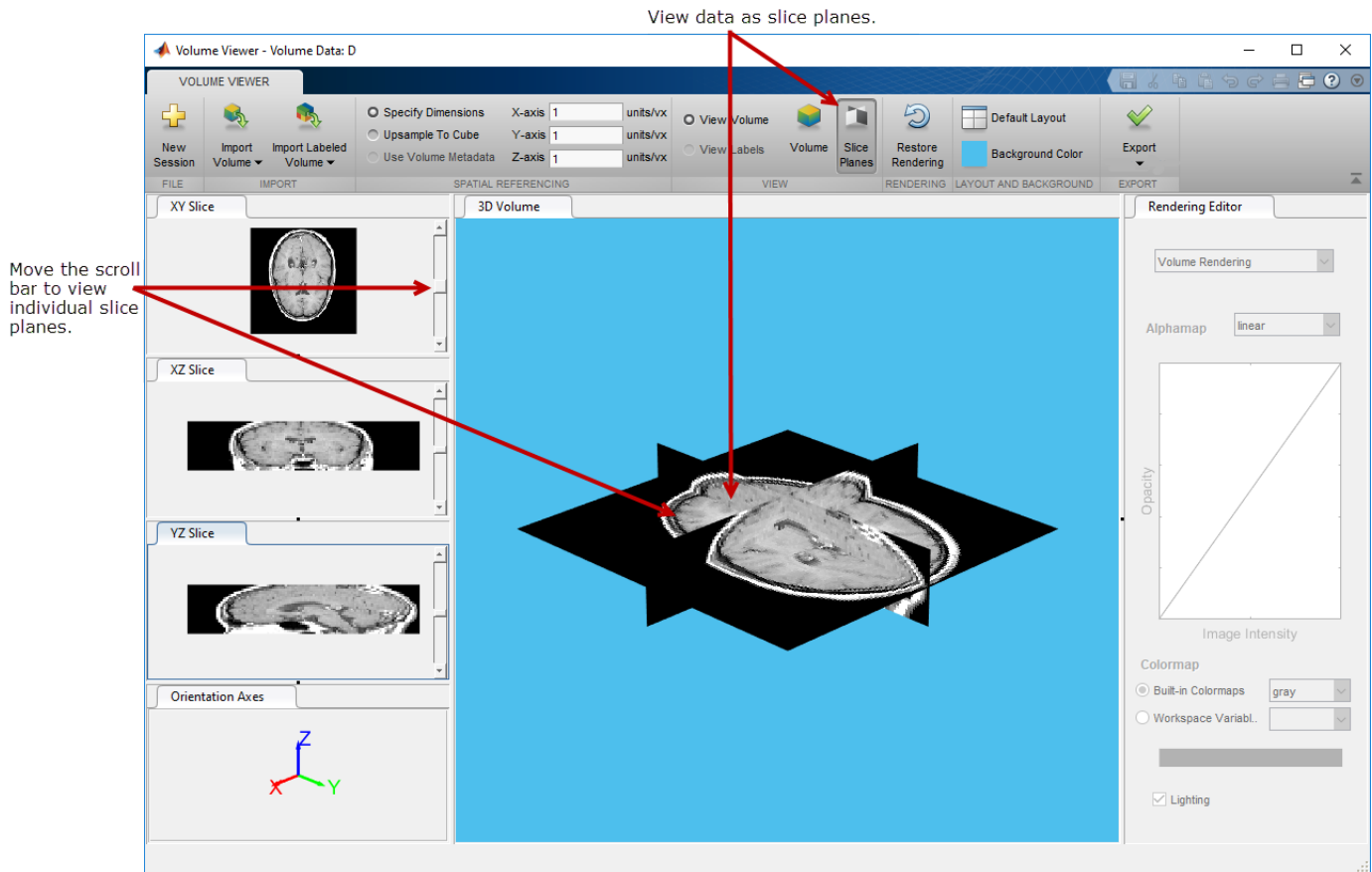
Continue using Volume Viewer capabilities until you achieve the best view of your data.

Adjust View of Volume Data in Volume Viewer

In this part of the example, you adjust the view of 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 get a more realistic view of the head volume. (The head appears flattened in the default view.)

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

Select this option and specify size of voxel dimensions in *x*, *y*, and *z* directions.

View impact of change on visualization.
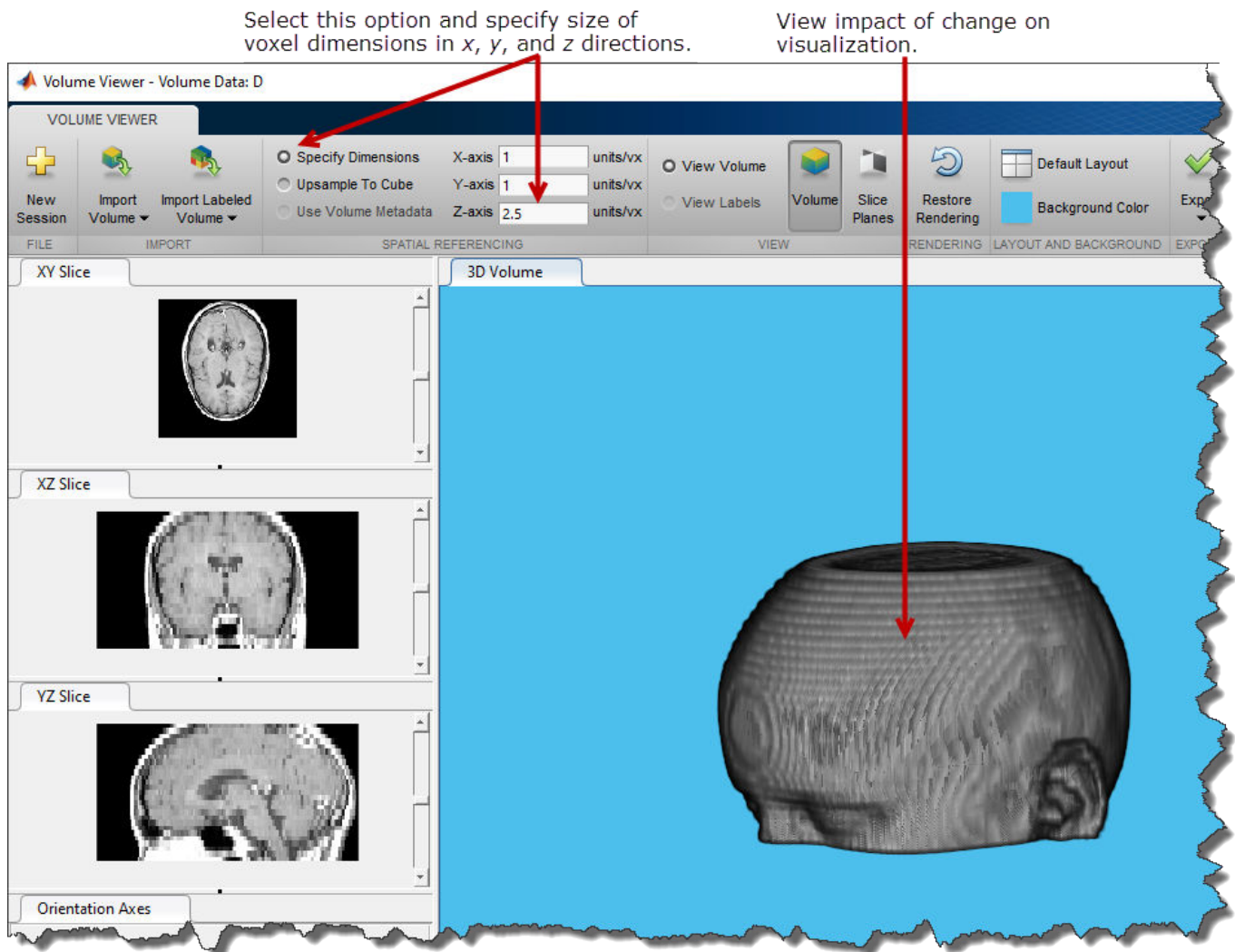
- **Upsample To Cube**--The Volume Viewer calculates a scale factor that makes the number of samples in each dimension the same as the largest dimension in the volume. This setting can make non-isotropically sampled data appear scaled more correctly.
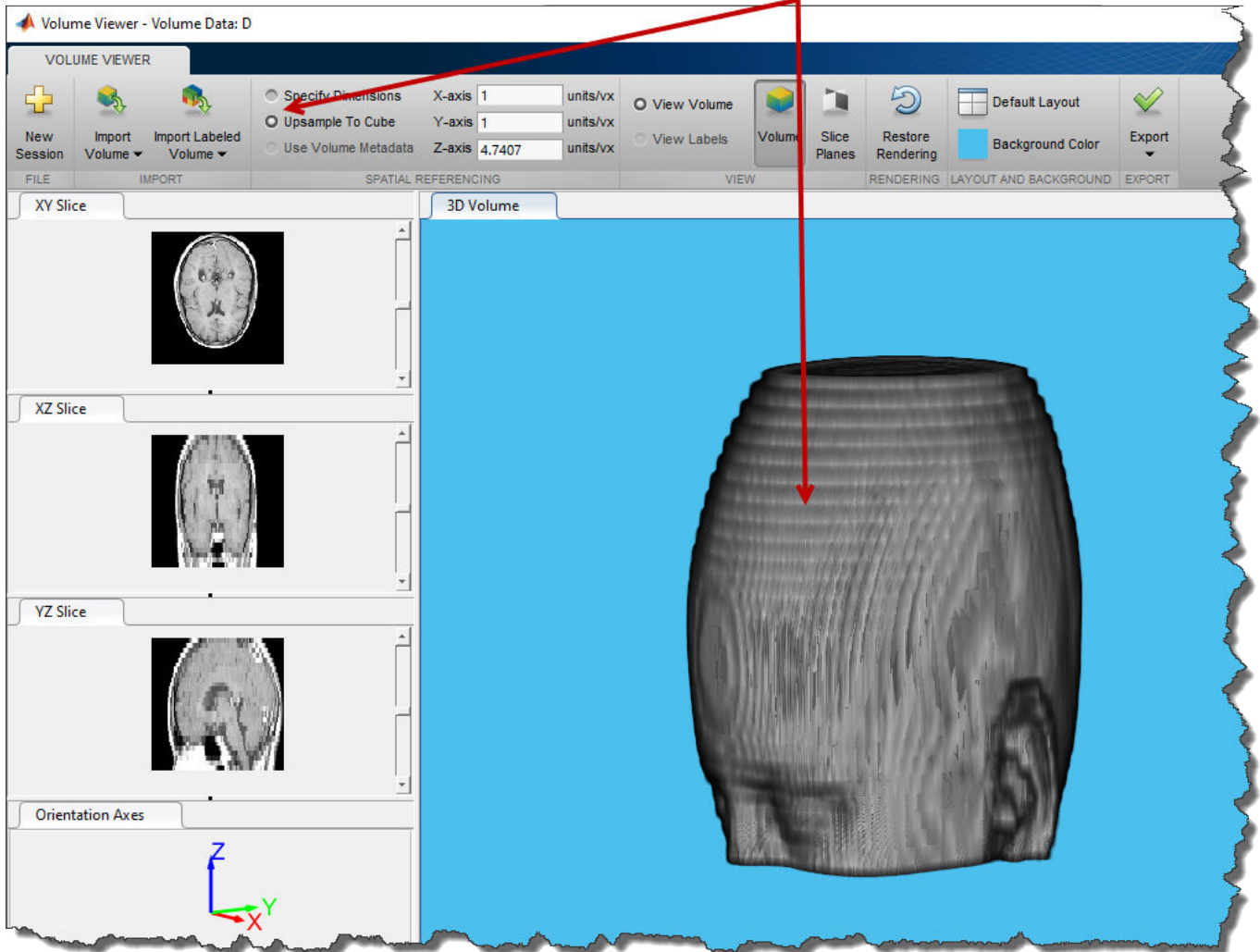
- **Use Volume Metadata**--If the data file includes resolution data in its metadata, the Volume Viewer uses the metadata and displays the volume true to scale. The Volume Viewer selects the **Use Volume Metadata** option, by default, if metadata is present.

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. Using the Rendering Editor, you can:

- Choose the overall viewing approach: Volume Rendering, Maximum Intensity Projection, or Isosurface.
- Modify the alphamap by specifying a preset alphamap, such as ct-bone, or by customizing the alphamap using the Opacity/Image Intensity curve.
- Specify the color map used in the visualization.
- Specify the lighting in the visualization.

Choose the viewing approach.

The Volume Viewer offers several viewing approaches for volumes. The **Maximum Intensity Projection** (MIP) option looks 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. You can also view the volume as an **Isosurface**.
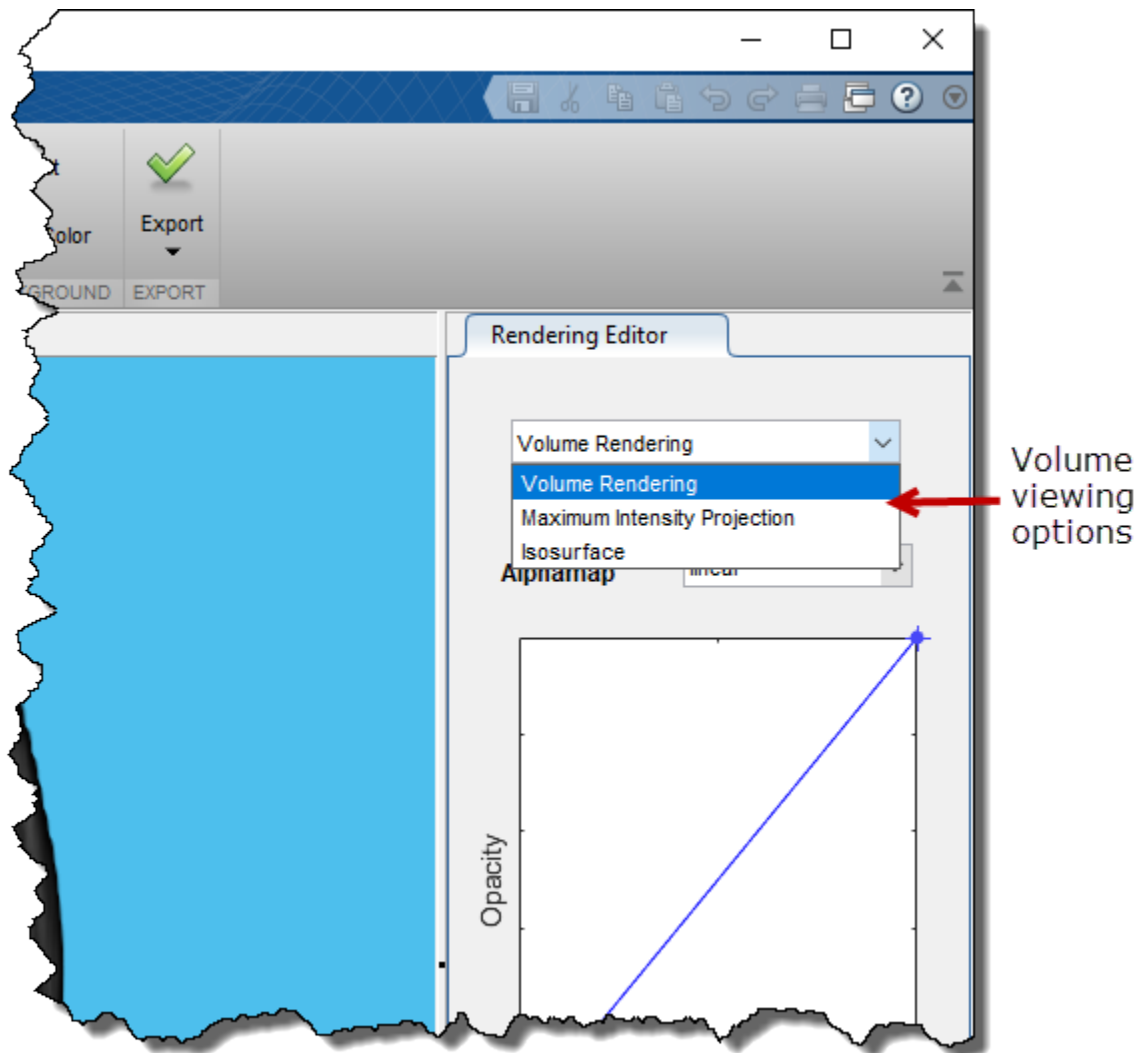
Specify the alphamap.

Volume rendering is highly dependent on defining an appropriate alphamap so that structures you want to see are opaque and structures you do not want to see are transparent. The Rendering Editor lets you define the opacity and transparency of voxel values throughout the volume. You can choose from a set of alphamap presets that automatically achieve certain well-defined effects. For example, to define a view that works well with CT bone data, select the **CT Bone** rendering 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. You customize the alphamap by manipulating the plot directly.

Specify the color map.

Color, when used with voxel intensity and opacity, is an important element of volume visualization. In the Rendering Editor, you can select from a list of predefined MATLAB color maps, such as `jet` and `parula`. You can also specify a custom color map that you have defined as a variable in the workspace. You can also change the color mapping for any color map by using the interactive color bar scale. For example, to lighten the color values in a visualization, click on the color bar to create a circular slider. To modify the color mapping so that more value map to lighter colors, move the slider to the left. You can create multiple sliders on the color bar to define other color mappings.
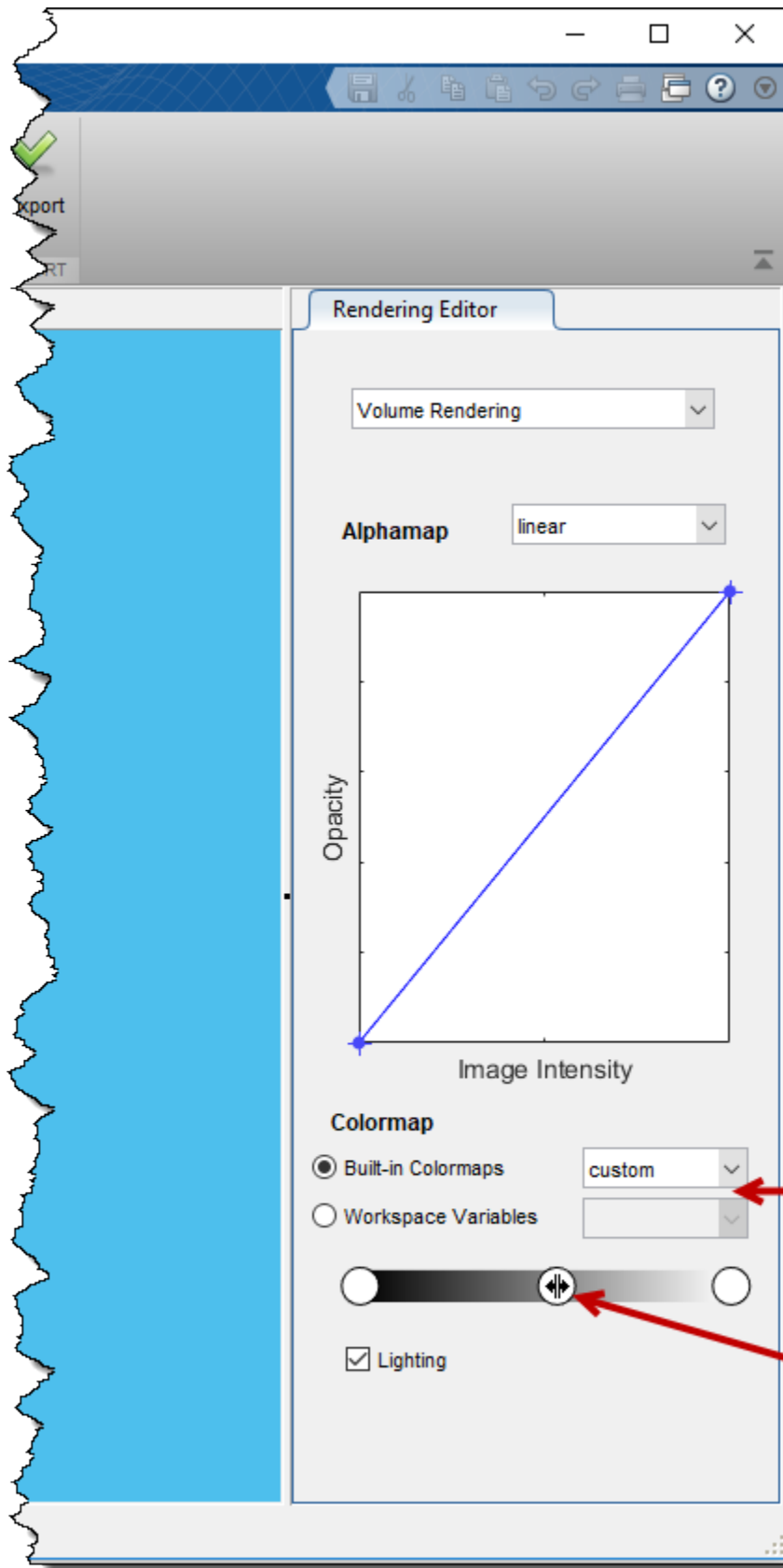
Select a built-in colormap or specify a colormap variable.

Click colorbar to create circular slider.

Move slider to change mapping of values.

Modify lighting effects. By default, the Volume Viewer uses certain lighting effects on the volume display. You can turn off these lighting effects by clearing the **Lighting** check box.

Save Volume Viewer Rendering and Camera Configuration Settings

After working in the Volume Viewer to achieve the best view of your data, you can save your rendering settings and camera configuration. The Volume Viewer stores this information in a structure, called `config` by default, that it writes to the workspace. You can use this structure with the `volshow` and `labelvolshow` functions to recreate the view you achieved in the Volume Viewer.

To save rendering and camera configuration settings, click **Export** and click the **Rendering and Camera Configurations** option.
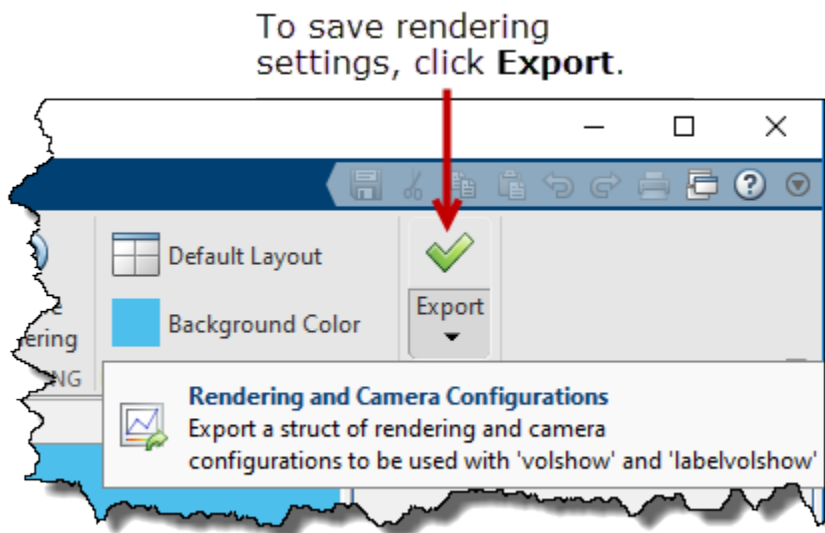
Specify the name for the structure that the Volume Viewer creates or accept the default name (`config`) and click **OK**.
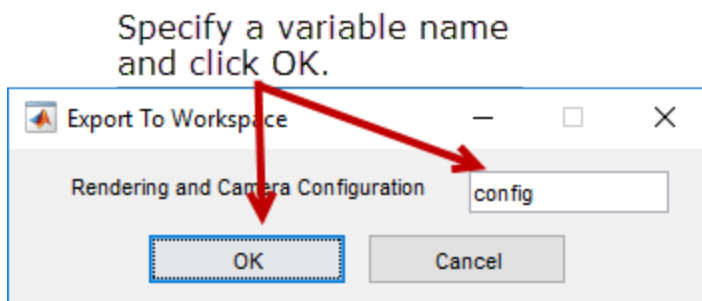
Explore 3-D Labeled Volumetric Data with Volume Viewer App

This example shows how to explore 3-D labeled volumetric data using the Volume Viewer app. Using the app, you can view the labeled volume by itself or view the labels as an overlay on the intensity volume. To illustrate, the example loads an intensity volume that shows the human brain with labeled areas that show the location and type of tumors found in the brain.

In this section...
"Load Labeled Volume and Intensity Volume into Volume Viewer" on page 4-57
"View Labeled Volume in Volume Viewer" on page 4-59
"Embed Labeled Volume with Intensity Volume" on page 4-60

Load Labeled Volume and Intensity Volume into Volume Viewer

This part of the example shows how to load a labeled volume and an intensity volume into the Volume Viewer.

Load the MRI intensity data of a human brain and the labeled volume from MAT-files into the workspace. This operation creates two variables in the workspace: `vol` and `label`.
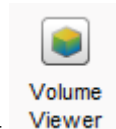
```
load(fullfile(toolboxdir('images'),'imdata','BrainMRILabeled','images','vol_001.mat'));
load(fullfile(toolboxdir('images'),'imdata','BrainMRILabeled','labels','label_001.mat'));
```

```
whos
```

```
Name        Size                      Bytes  Class      Attributes

  label      240x240x155              8928000  uint8
  vol        240x240x155             17856000  uint16
```

Open the **Volume Viewer** app. From the MATLAB toolstrip, open the Apps tab and under Image
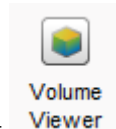
Processing and Computer Vision, click   . You can also open the app using the `volumeViewer` command.

Load the labeled volume into the Volume Viewer app. Click **Import Labeled Volume** to open the labeled volume. You can load an image in a 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.) For this example, choose the **Import From Workspace** option.

Click here to load labeled volume data.
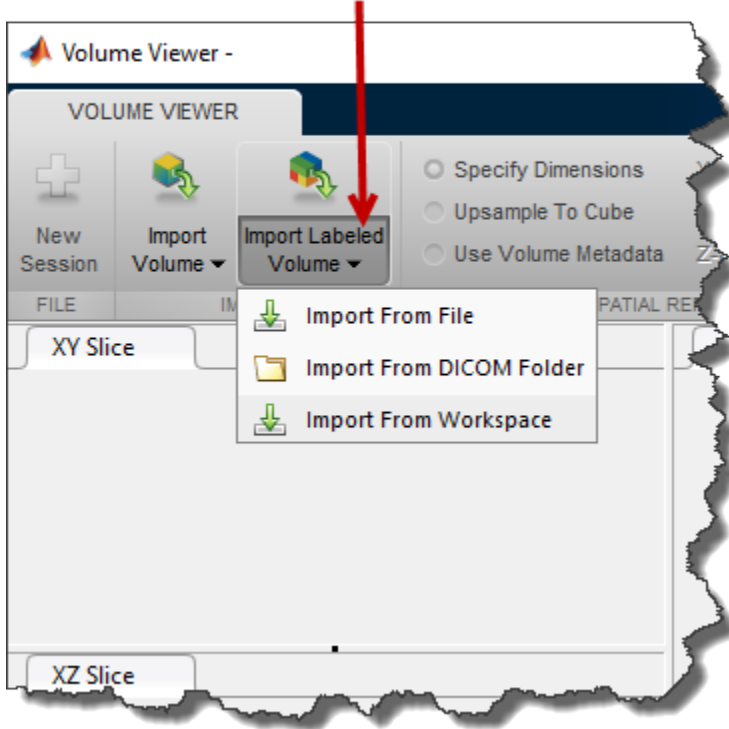
Select the workspace variable associated with the labeled volume data in the **Import from Workspace** dialog box and click **OK**.
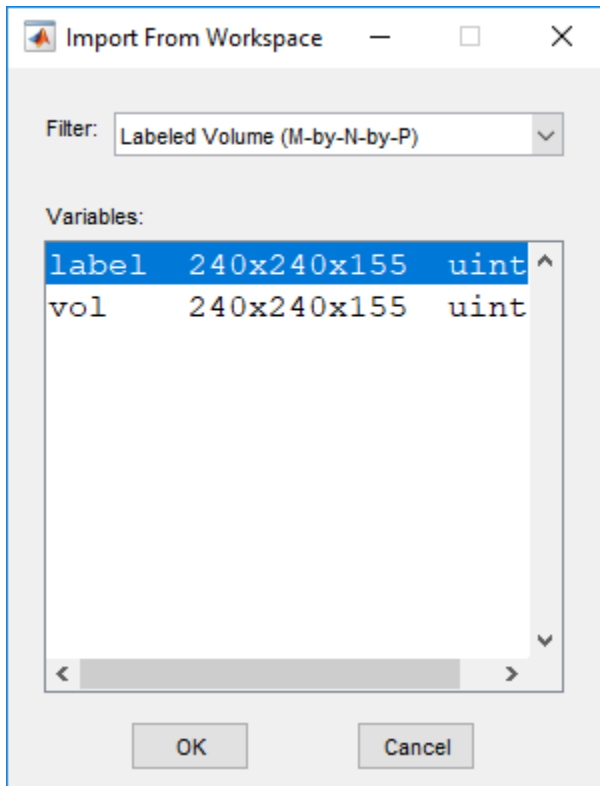
View Labeled Volume in Volume Viewer

In this part of the example, you view the labeled volume data in the Volume Viewer.

View the labeled volume in the Volume Viewer app. By default, the Volume Viewer displays the data as a labeled volume but you can also view it as slice planes. To explore the labeled 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 position of the axes in the **Orientation Axes** window reflects the spatial orientation of the labeled volume as you rotate it.

View the labeled volume.

Control color and opacity of the labels.
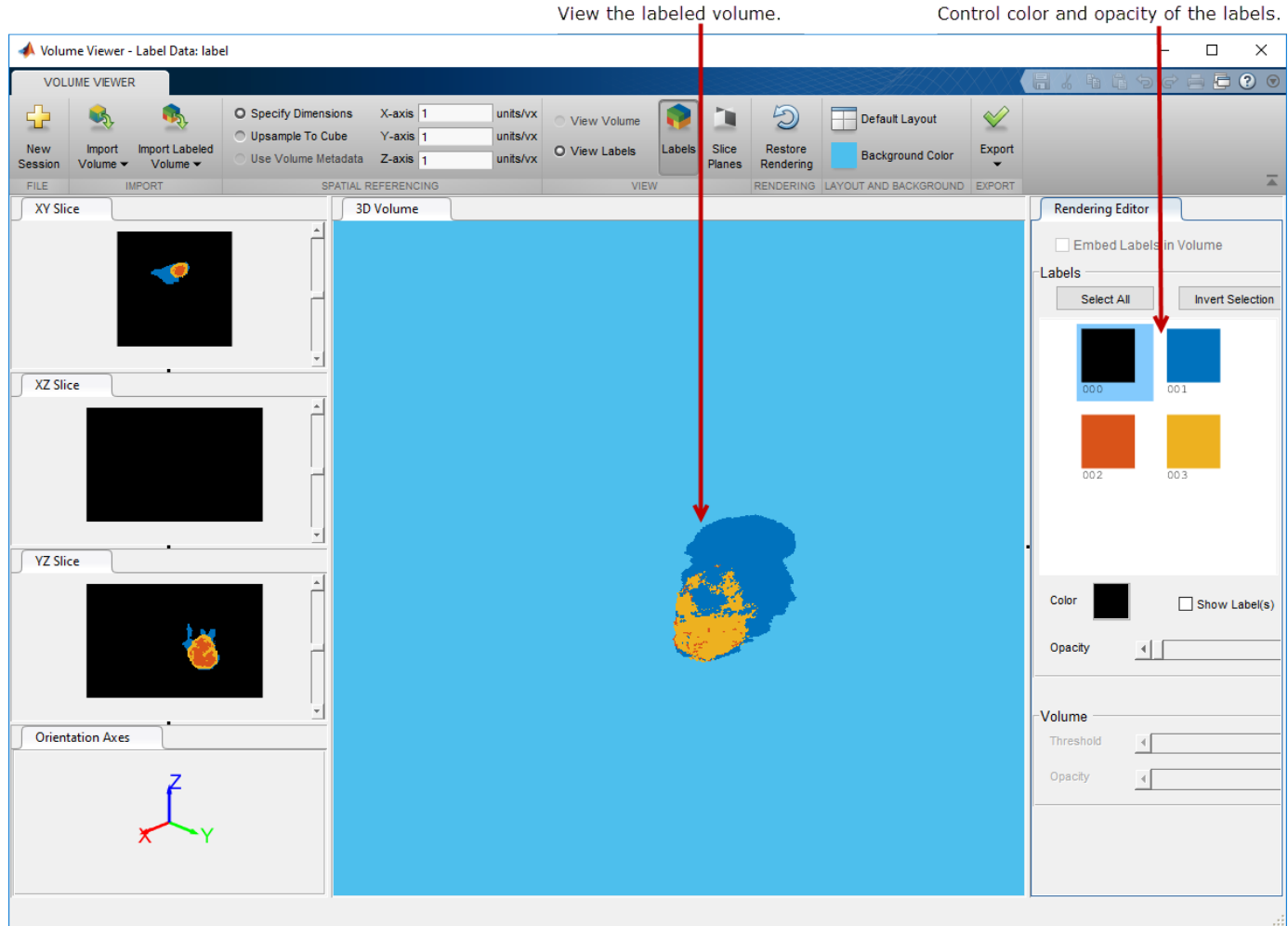
Refine your view of the labeled volume using the **Rendering Editor** part of the Volume Viewer. You can use the **Rendering Editor** to view certain labels and hide others, change the color, and modify the transparency of the labels. Label 000 is the background and it is typically not visible. When you select the background label, the **Show Labels** check box is clear, by default. To select all the visible labels at once, select the background label and click **Invert Selection**. To change the color of a label (or all the labels), select the label in the **Rendering Editor** and specify the color in the **Color** selector. You can also control the transparency of the label using the **Opacity** slider.

Embed Labeled Volume with Intensity Volume

In this part of the example, you view the labeled volume and the intensity volume in the Volume Viewer at the same time. Viewing the labeled volume overlaid on the intensity volume can provide context for the data.

With the labeled volume already in the Volume Viewer, load the intensity volume into the app. Click **Import Volume** and choose the **Import From Workspace** option.
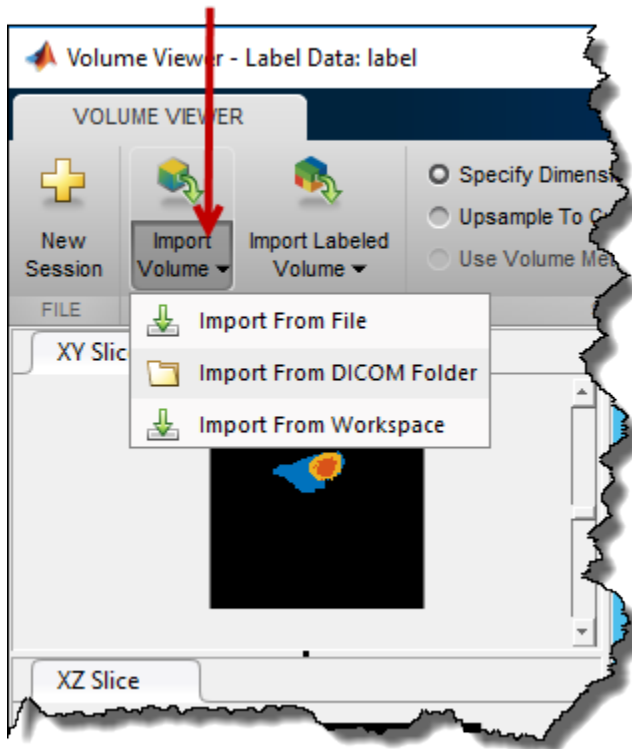
Select the workspace variable associated with the intensity volume in the **Import from Workspace** dialog box and click **OK**.
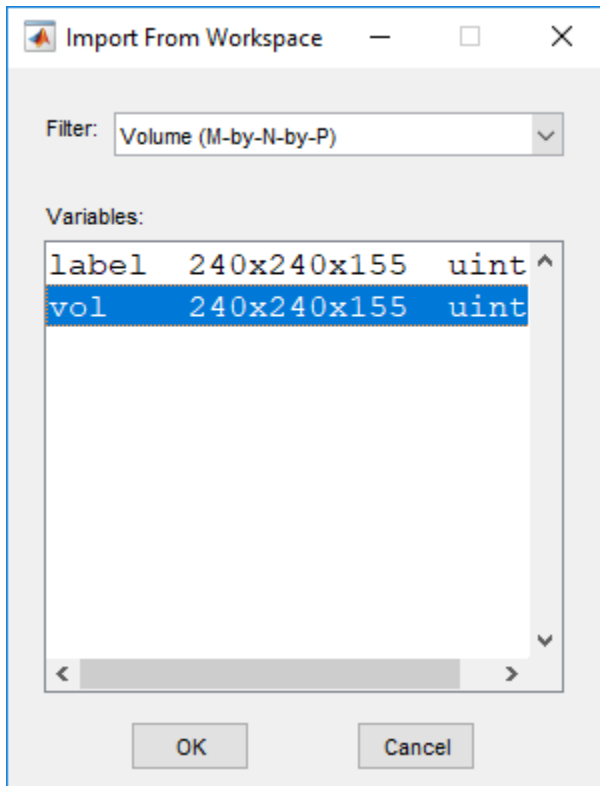
The Volume Viewer displays the labeled volume over the intensity volumetric data. By default, the Volume Viewer displays the label data and the intensity data as volumes but you can also view it as slice planes. To explore the labeled and intensity volumes, zoom in and out using the mouse wheel or a right-click. You can also rotate the volumes by positioning the cursor in the image window, pressing and holding the mouse, and moving the cursor. To view only the intensity volume, and hide the labeled volume, click **View Volume**.

View labeled volume and intensity volume together.
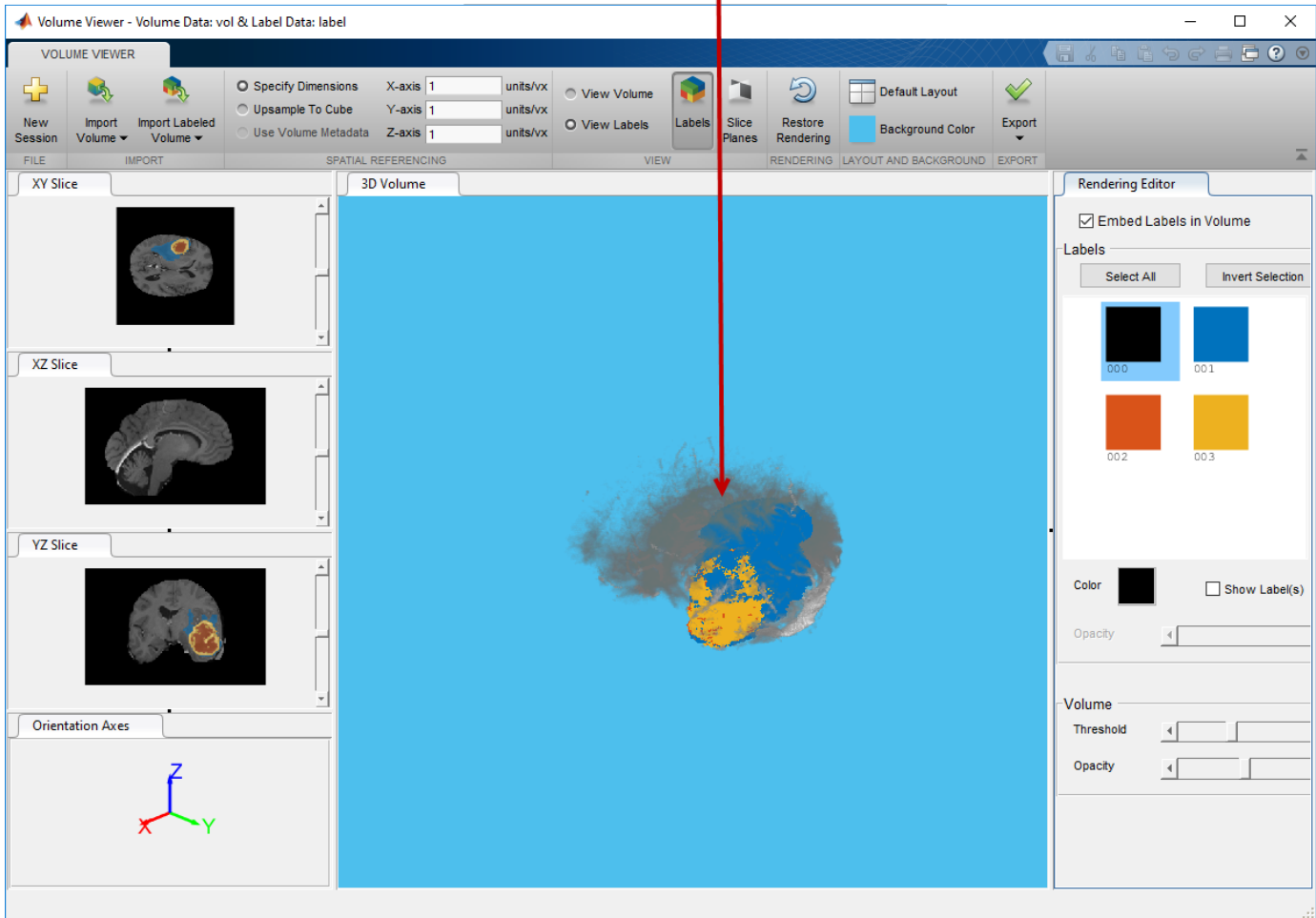
Refine your view of the labeled volume and the intensity volume using options in the **Rendering Editor**. To only view the labeled volume, and hide the intensity volume, clear the **Embed Labels in Volume** check box. In the **Labels** area of the **Rendering Editor**, you can select any of the labels and change its color or transparency. Label 000 is the background. By default, the background is set to black and is not visible. The **Show Labels** check box is clear. To select all the labels, click on the background and then click **Invert Selection**. If the intensity volume is visible, you can modify the threshold value and transparency using the sliders in the **Volume** area of the **Rendering Editor**.

To view the labeled volume and the intensity volume together, make sure **View labels** is selected.

To view only the labeled volume, clear this check box.

Select individual label or labels.

Determine visibility of label or labels.

Control view of intensity volume.

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.
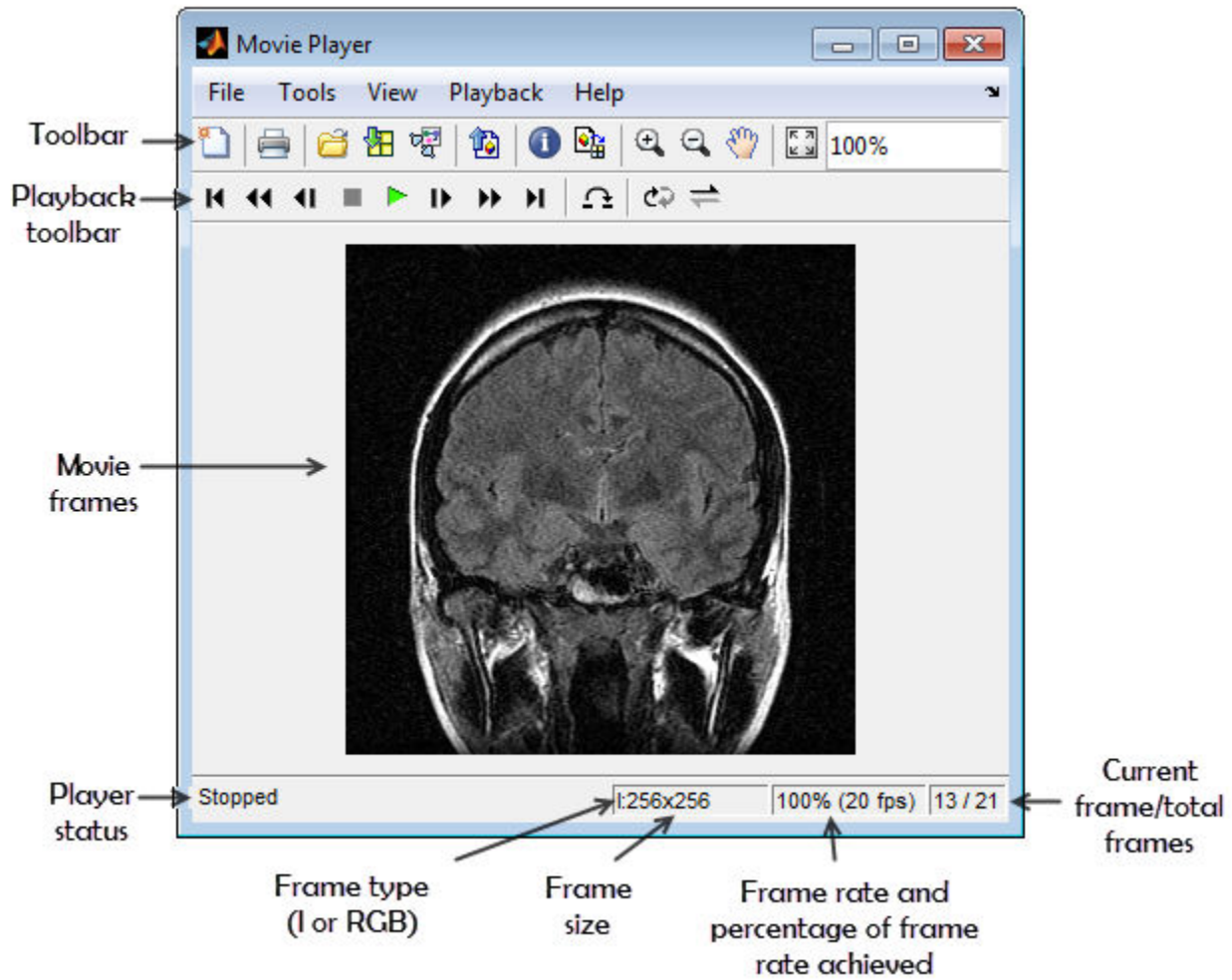
Explore the image sequence using Video Viewer Playback controls.

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.

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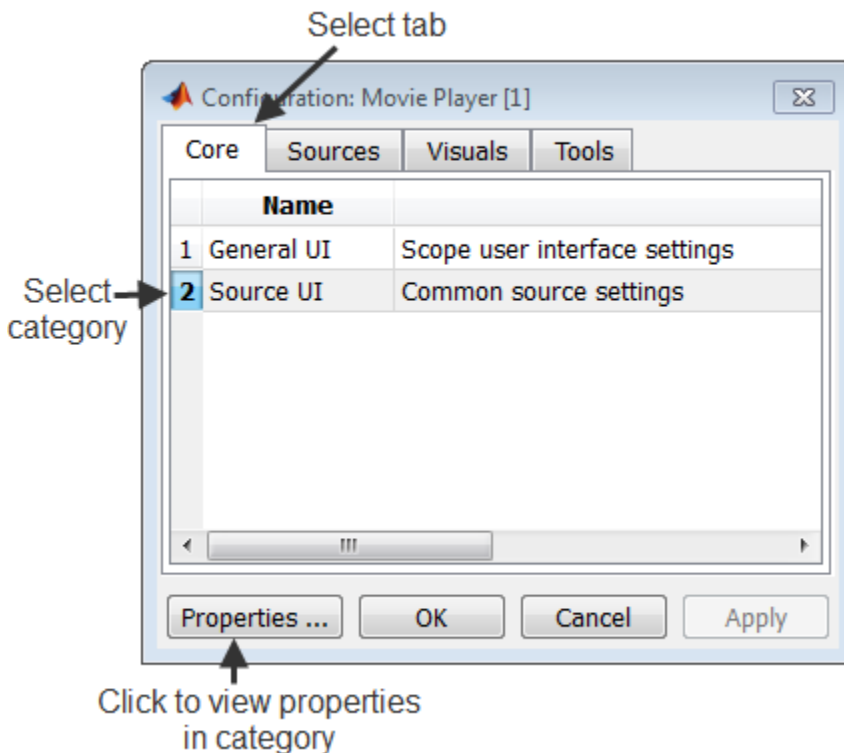
The following table lists the options that are available for each category on every pane.

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.
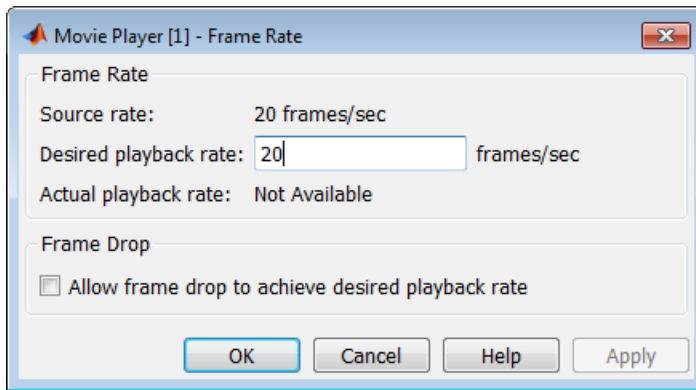
If you want to increase the actual playback rate, but your system's hardware cannot keep up with the desired rate, select the **Allow frame drop to achieve desired playback rate** check box. This parameter enables the Video Viewer app to achieve the playback rate by dropping frames. When you select this 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 color map 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 color map.
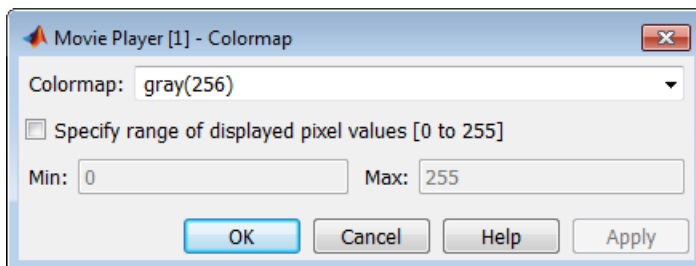
Use the **Colormap** parameter to specify a particular color map.

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.
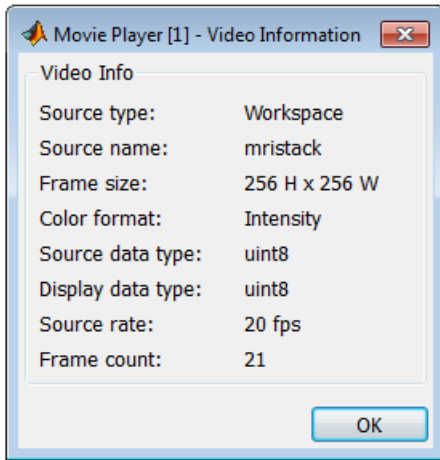
Convert Multiframe Image to Movie

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 4-73
"Display Grayscale Images" on page 4-73
"Display Binary Images" on page 4-75
"Display Truecolor Images" on page 4-76

If you need help determining what type of image you are working with, see "Image Types in the Toolbox" on page 2-12.

Display Indexed Images

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

```
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 color map, 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 color map, the value 1 points to the second row, and so on. This offset is handled automatically by the `imtool` and `imshow` functions.

If the color map contains a greater number of colors than the image, the functions ignore the extra colors in the color map. If the color map contains fewer colors than the image requires, the functions set all image pixels over the limits of the color map's capacity to the last color in the color map. For example, if an image of class `uint8` contains 256 colors, and you display it with a color map that contains only 16 colors, all pixels with a value of 15 or higher are displayed with the last color in the color map.

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 color map.

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 color map, but you normally do not specify a color map for a grayscale image. MATLAB displays grayscale images by using a grayscale system color map (where R=G=B). By default, the number of levels of gray in the

color map 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 (that is, [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 **Image Viewer**, using the automatic scaling option. If you execute this example, note the display range specified in the lower right corner of the **Image Viewer** window.

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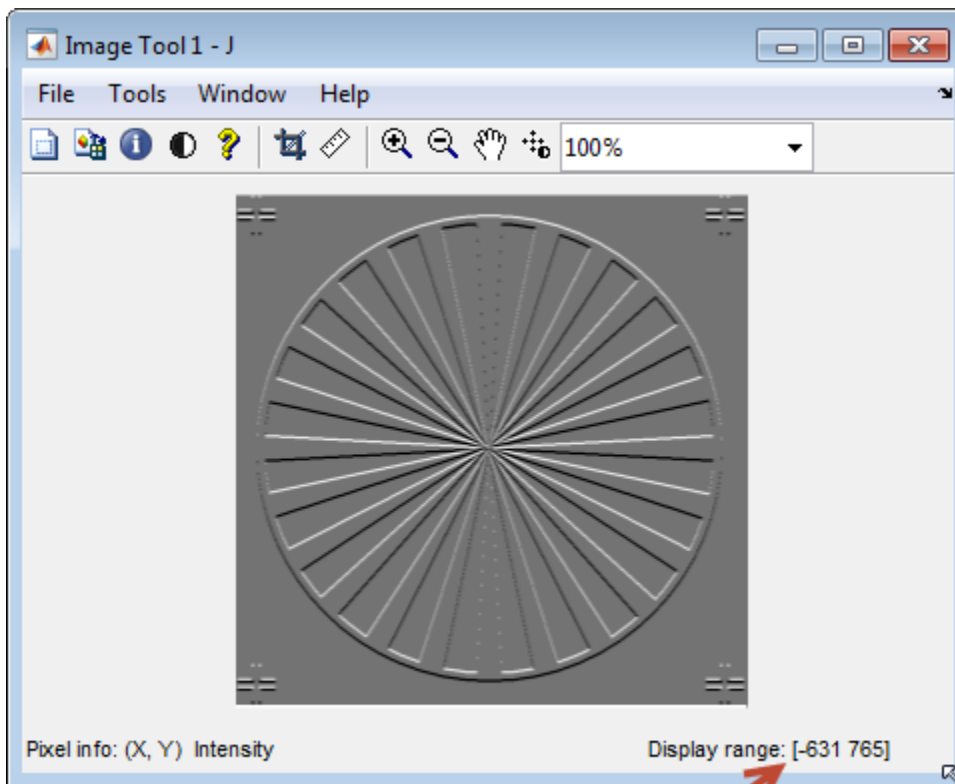
Unconventional display range

Display Binary Images

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

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

To display a binary image, call the `imshow` function or open the **Image Viewer** app. For example, this code reads a binary image into the MATLAB workspace and then displays the image. This documentation uses the variable name `BW` to represent a binary image in the workspace

```
BW = imread('circles.png');
imshow(BW)
```
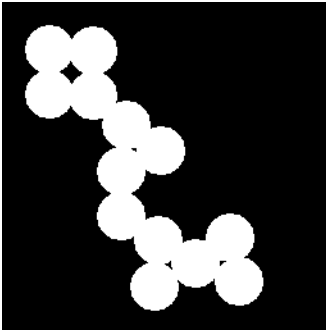
Change Display Colors of Binary Image

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

```
imshow(~BW)
```
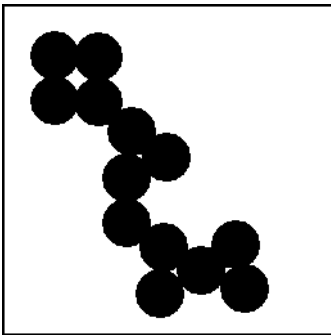

You can also display a binary image using the indexed image color map syntax. For example, the following command specifies a two-row color map that displays 0's as red and 1's as blue.

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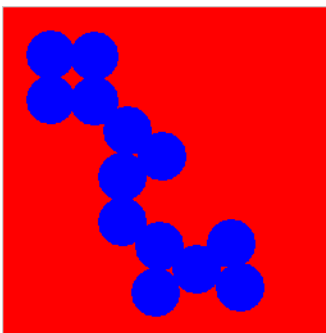

Display Truecolor Images

Truecolor images, also called RGB images, represent color values directly, rather than through a color map. 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 color map associated with the image. For more information, see "Display Indexed Images" on page 4-73.

Add Color Bar to Displayed Grayscale Image

This example shows how to display a grayscale image with a color bar that indicates the mapping of data values to colors. Seeing the correspondence between data values and the colors displayed by using a color bar is especially useful if you are displaying unconventional range data as an image.

Read and display a grayscale image.

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

Convert the image to data type `double`. Data is in the range [0, 1].

```
I = im2double(I);
dataRangeI = [min(I(:)) max(I(:))]
```

```
dataRangeI = 1×2

    0    1
```

Filter the image using an edge detection filter. The filtered data exceeds the default range [0, 1] because the filter is not normalized.

```
h = [1 2 1; 0 0 0; -1 -2 -1];
J = imfilter(I,h);
dataRangeJ = [min(J(:)) max(J(:))]
```

```
dataRangeJ = 1×2

   -2.5961    2.5451
```

Display the filtered image using the full display range of the filtered data. `imshow` displays the minimum data value as black and the maximum data value as white.

```
imshow(J,[])
```

Use the `colorbar` function to add the color bar to the image.

```
colorbar
```
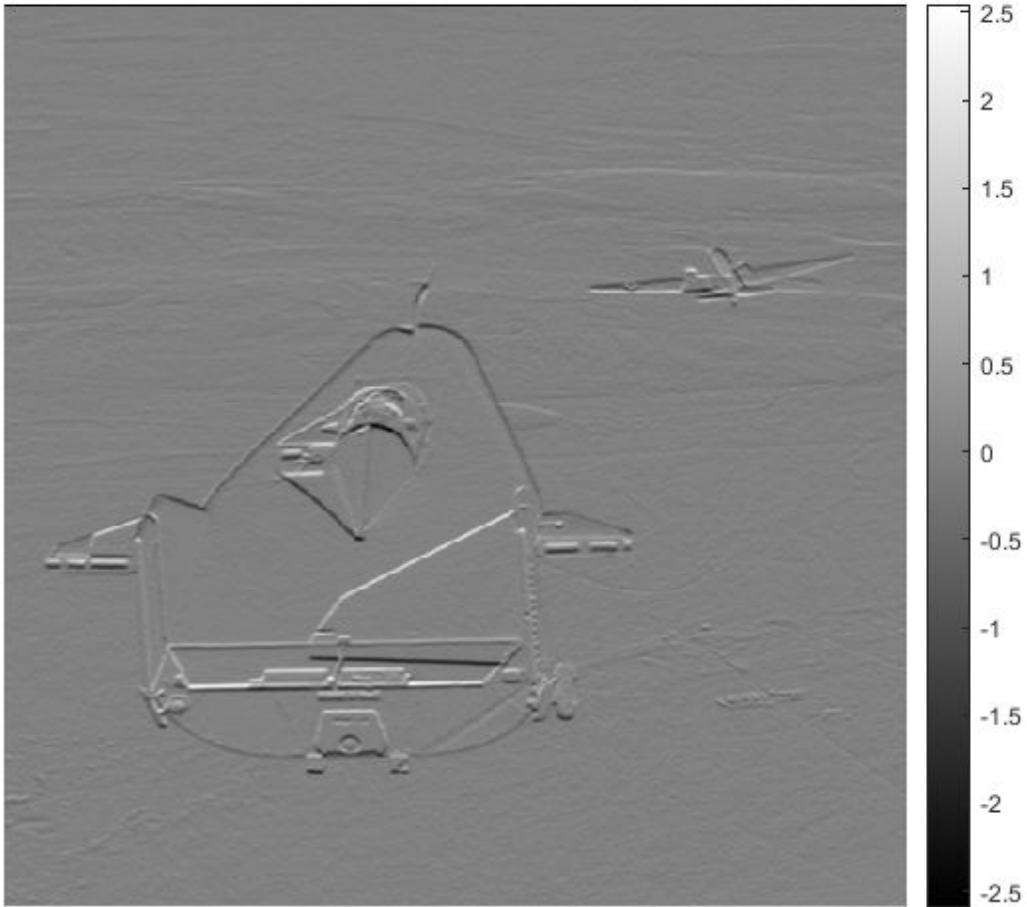
See Also

imshow

More About

*   "Display Grayscale Images" on page 4-73

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 3-6 for details.

If you want to print an image, use `imshow` to display the image in a MATLAB figure window. If you are using **Image Viewer**, then you must use the **Print to Figure** option on the **File** menu. When you choose this option, **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". 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 4-81
"Set Values of Toolbox Preferences" on page 4-81

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

Retrieve Values of Toolbox Preferences

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

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

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

```
iptgetpref('ImtoolInitialMagnification')

ans =

    100
```

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

Set Values of Toolbox Preferences

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

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

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

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

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

Building GUIs with Modular Tools

This chapter describes how to use interactive modular tools and create custom image processing applications.
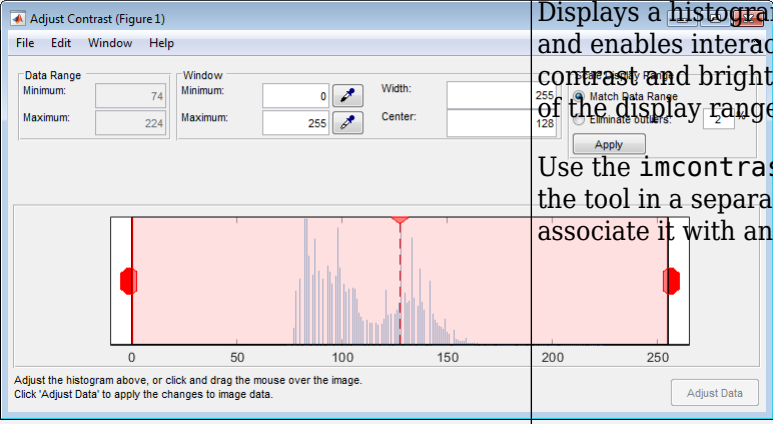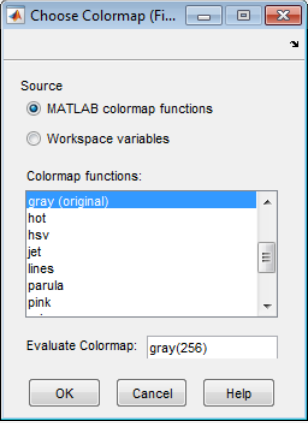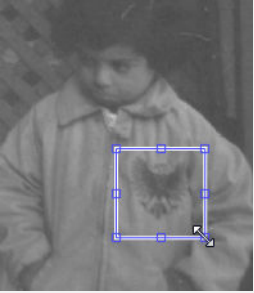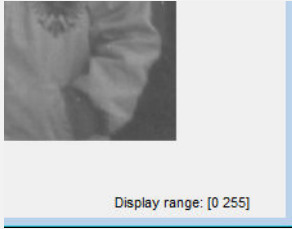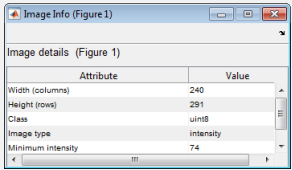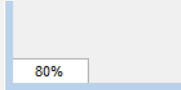
Interactive Image Viewing and Processing Tools
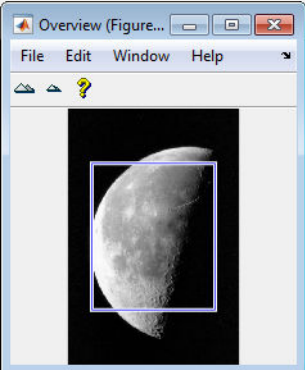
The toolbox includes several tools that you can use to interact with an image displayed in a MATLAB figure window. For example, you can use tools to adjust the display of the image, get information about the image data, and adjust the contrast or crop the image.
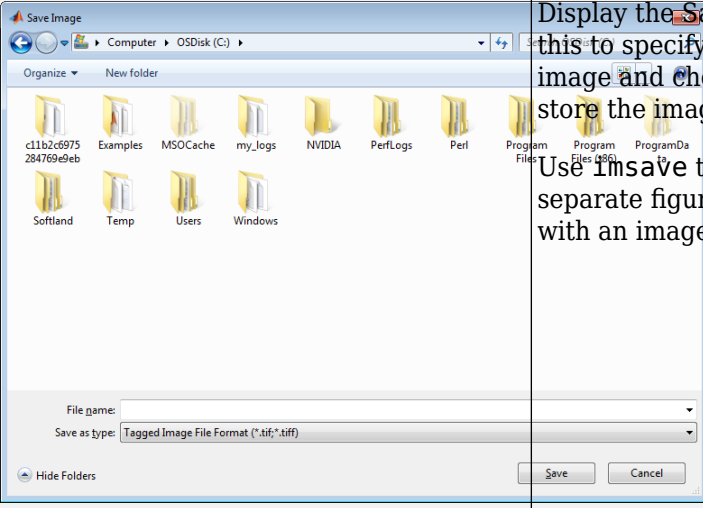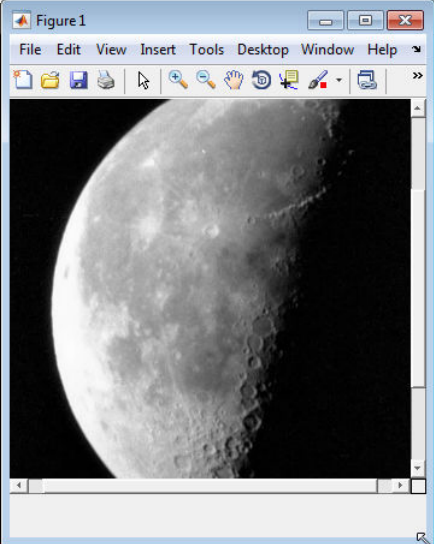
You can use the tools independently or in combination. You can create custom image processing apps that open a combination of tools and initialize their display and interactions. For more information, see "Interactive Tool Workflow" on page 5-6.

You can also access all tools using the **Image Viewer** app.

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 color map of the target figure. You can select one of the MATLAB color maps, select a color map variable from the MATLAB workspace, or enter a custom MATLAB function. Use the `imcolormaptool` function to launch the tool in a separate figure window.
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.

Tool	Example	Description
Display Range tool	Display range: [0 255]	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	128.97	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.
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	80%	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.

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		Displays information about the pixel the mouse is over in the target image. Use `impixelinfo` to create the tool, associate it with 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`.
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.

Tool	Example	Description
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.
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.

See Also

More About

- "Interactive Tool Workflow" on page 5-6
- "Get Started with Image Viewer App" on page 4-17

Interactive Tool Workflow

In this section...
"Display Target Image in Figure Window" on page 5-6
"Create the Tool" on page 5-6
"Position Tools" on page 5-7
"Add Navigation Aids" on page 5-8
"Customize Tool Interactivity" on page 5-8

Using the interactive tools typically involves the following steps.

Display Target Image in Figure Window

Display the image to be processed (called the target image) in a MATLAB figure window. The `imshow` function is recommended because it optimizes figure, axes, and image object properties for image display, but you can also use the `image` or `imagesc` functions.

Some of the tools add themselves to the figure window containing the image. Prevent the tools from displaying over the image by including a border. If you are using the `imshow` function, then make sure that the Image Processing Toolbox `ImshowBorder` preference is set to `'loose'` (this is the default setting).

Create the Tool

After you display an image in a figure window, create one or more tools using the corresponding tool creation functions. For a list of available tools, see "Interactive Image Viewing and Processing Tools" on page 5-2. The functions create the tools and automatically set up the interactivity connection between the tool and the target image.

Associate Tool with Target Image

When you create a tool, you can specify the target image or you can let the tool pick a suitable target image.

- To specify the target image, provide a handle to the target image as an input argument to the tool creation function. The handle can be a specific image object, or a figure, axes, or uipanel object that contains an image.
- To let the tool pick the target image, call the tool creation function with no input arguments. By default, the tool uses the image in the current figure as the target image. If the current figure contains multiple images, then the tool associates with the first image in the figure object's children (the last image created). Note that not all tools offer a no-argument syntax.

Some tools can work with multiple images in a figure. These are `impixelinfo`, `impixelinfoval`, and `imdisplayrange`.

Specify Parent of Tool

When you create a tool, 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 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. When you do not specify the parent, the tools use default behavior.

- Some of the smaller tools, such as the Display Range tool and 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 tools, such as the Adjust Contrast tool and Choose Colormap tool, open in separate figures of their own.
- Two tools, the Overview tool and Pixel Region tool, have different creation functions for specifying the parent figure. 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. For an example, see "Create Pixel Region Tool" on page 5-15.

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.

Position Tools

Each tool has 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 tools are constructed from graphics objects, such as uipanel objects, you can change their default positioning or other characteristics by setting properties of the objects. To specify the position of a tool or other graphics object, set the `Position` property as a four-element position vector `[left bottom width height]`. The values of `left` and `bottom` specify the distance from the lower left corner of the parent container object, such as a figure. The values of `width` and `height` specify the dimensions of the object.

When you specify 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 of the tool, not the exact location in pixels.

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])
```

For an example, see "Create Pixel Region Tool" on page 5-15.

Add Navigation Aids

The toolbox includes tools that you can use to add navigation aids to a GUI application.

The Scroll Panel is the primary navigation tool and 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 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.

Adding a scroll panel to an image display changes the relationship of the graphics objects used in the display. For more information, see "Add Scroll Panel to Figure" on page 5-10.

Note The toolbox 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.

Customize Tool Interactivity

When you create a tool and associate it with a target image, the tool automatically makes the necessary connections between the target image and the tool.

Some tools have a one-way connection to the target image. These tools get updated when you interact with the target image, but you cannot use the tool to modify the target image. For example, the Pixel Information tool receives information about the location and value of the pixel currently under the pointer.

Other tools have a two-way connection to the target image. These tools get updated when you interact with the target image, and you can update the target image by interacting with the tools. For example, the Overview tool sets up a two-way connection to the target image. For this tool, if you change the visible portion of the target image by scrolling, panning, or by changing the magnification, then the Overview tool changes the size and location of the detail rectangle to match the portion of the image that is now visible. Conversely, if you move the detail window in the Overview tool, then the tool updates the visible portion of the target image in the scroll panel.

The 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 tool by using the application programmer interface (API) associated with the tool to set up callbacks to get notification of events. For more information, see "Callbacks — Programmed Response to User Action" and "Overview Events and Listeners". For an example, see "Build Image Comparison Tool" on page 5-24.

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 each tool.

See Also

Related Examples
- "Create Pixel Region Tool" on page 5-15
- "Build App for Navigating Large Images" on page 5-21
- "Build App to Display Pixel Information" on page 5-19
- "Build Image Comparison Tool" on page 5-24

More About
- "Interactive Image Viewing and Processing Tools" on page 5-2
- "Add Scroll Panel to Figure" on page 5-10

Add Scroll Panel to Figure

The primary navigation tool of a figure is a scroll panel. 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.
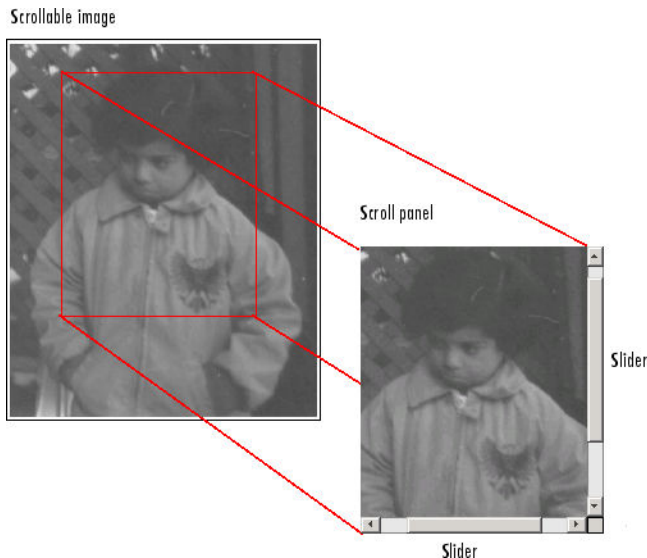
When you display an image in a scroll panel, it changes the object hierarchy of your displayed image. This diagram illustrates the typical object hierarchy for an image displayed in an axes object in a figure object.
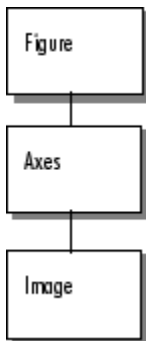
Object Hierarchy of Image Displayed in a Figure

When you call the `imscrollpanel` function to put the target image in a scrollable window, this object hierarchy changes. `imscrollpanel` inserts a new object into the hierarchy between the figure object and the axes object containing the image. The figure shows the object hierarchy after the call to `imscrollpanel`.
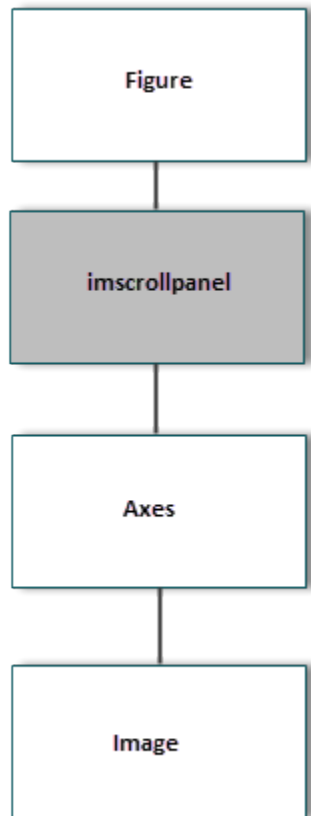
Object Hierarchy of Image Displayed in Scroll Panel

After you add a scroll panel to a figure, you can change the image data displayed in the scroll bar by using the `replaceImage` function in the `imscrollpanel` API.

The scroll panel navigation tool is not compatible with the figure window toolbar and menu bar. When you add a scroll panel to an image displayed in a figure window, suppress the toolbar and menu bar from the figure. This sample code demonstrates one way to do this.

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

See Also

immagbox | imoverview | imscrollpanel

Related Examples

- "Build App for Navigating Large Images" on page 5-21

More About

- "Interactive Image Viewing and Processing Tools" on page 5-2

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
```

See Also
imhandles

More About
- "Interactive Tool Workflow" on page 5-6

Create Pixel Region Tool

This example shows how to create a Pixel Region tool in a separate figure window and embedded in an existing figure window.

Create Pixel Region Tool in Separate Figure Window

Read an image into the workspace.

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

Display the image in a figure window. Return a handle to the target image, `himage`.

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

To create the Pixel Region tool in a separate window, use the `impixelregion` function.

```
hpixreg = impixelregion(himage);
```
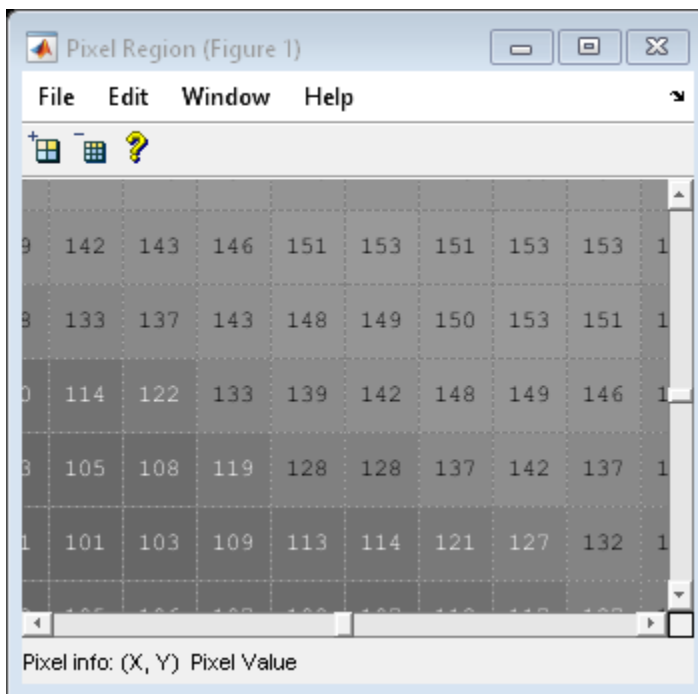
Embed Pixel Region Tool in Existing Figure

Create a new figure window and return a handle to the figure.

```
fig = figure;
```

Create an axes and display the target image in the axes.

```
ax = axes;
img = imshow(I);
```

To create the Pixel Region tool in the same figure as the target image, use the `impixelregionpanel` function. Specify the target image's parent figure, `fig`, as the parent of the Pixel Region tool.

```
pixregionobj = impixelregionpanel(fig,img);
```
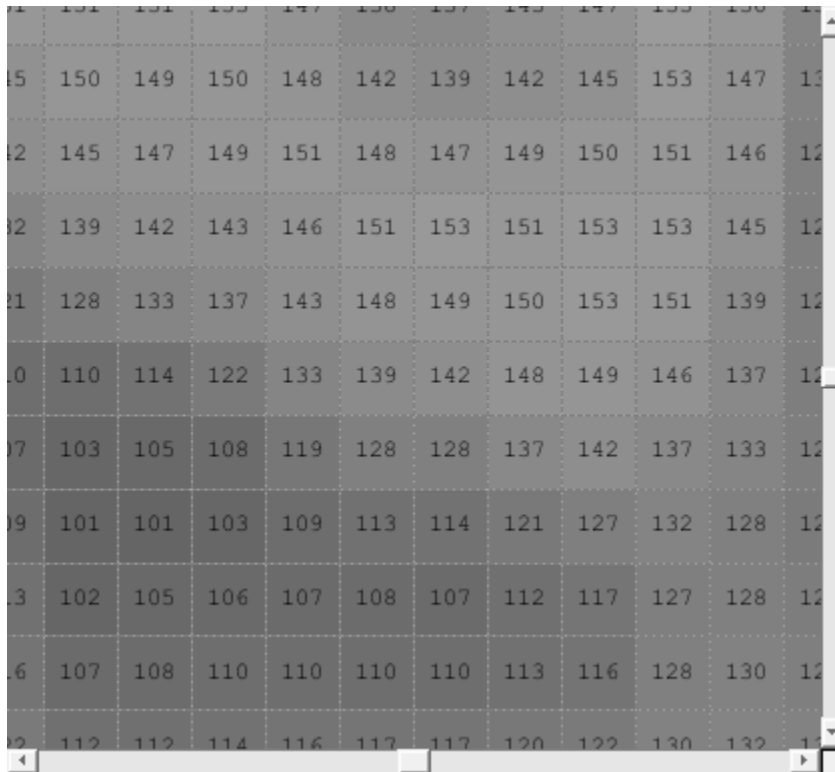
The Pixel Region tool overlaps and hides the original image. To see both the image and the tool, shift their positions so that they do not overlap.

```
set(ax,'Units','normalized','Position',[0 .5 1 .5]);
set(pixregionobj,'Units','normalized','Position',[0 .04 1 .4]);
```
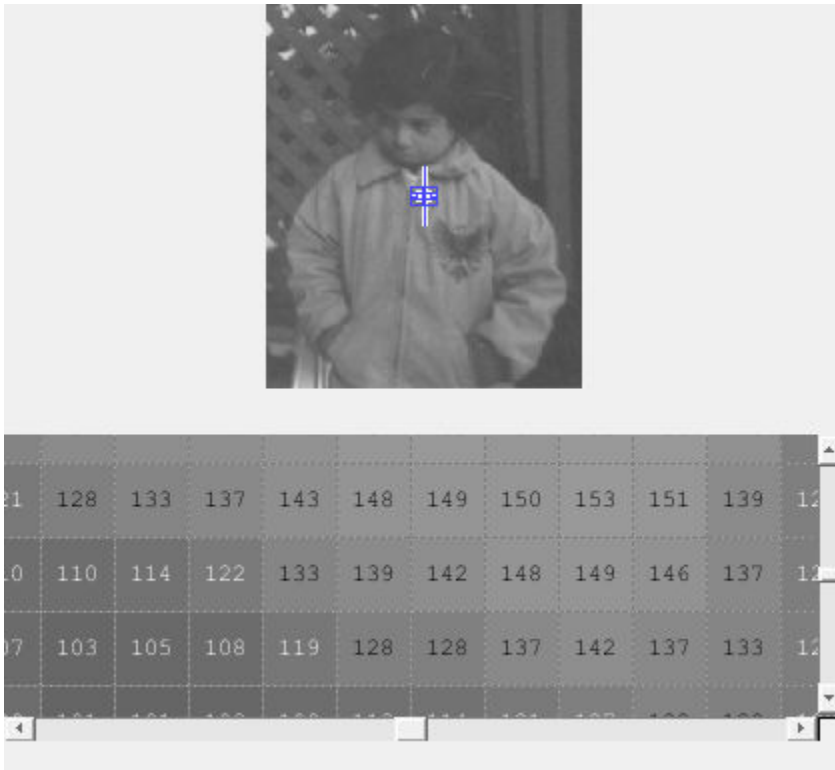
See Also

impixelregion | impixelregionpanel

More About
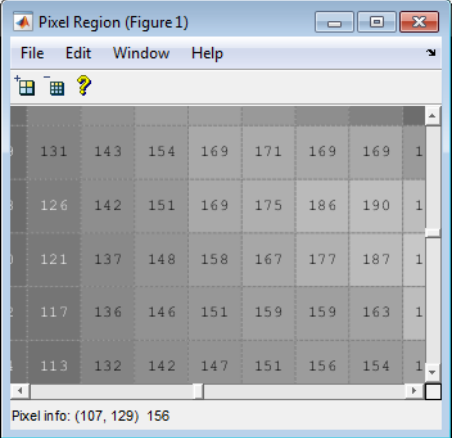
- "Position Tools" on page 5-7
- "Interactive Image Viewing and Processing Tools" on page 5-2

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.

First, define a function that builds the app. This example uses a function called `my_pixinfo_tool`, which is attached at the end of the example.

After you define the function that builds the app, test the app. Read an image into the workspace.

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

Display the image with pixel information tools in the app.

```
 my_pixinfo_tool(I)
```


App Creation Function

The `my_pixinfo_tool` function 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. Note that the function suppresses the toolbar and menu bar in the figure window because scrollable navigation is incompatible with standard MATLAB™ figure window navigation tools.

```
function my_pixinfo_tool(im)
% Create figure, setting up properties
fig = figure('Toolbar','none', ...
             'Menubar','none', ...
```

```matlab
                '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])

end
```

See Also

imdisplayrange | imdistline | impixelinfo | impixelregionpanel

Related Examples

- "Create Pixel Region Tool" on page 5-15
- "Build Image Comparison Tool" on page 5-24
- "Build App for Navigating Large Images" on page 5-21

More About

- "Interactive Tool Workflow" on page 5-6
- "Interactive Image Viewing and Processing Tools" on page 5-2

Build App for Navigating Large Images

This example shows how to build an app using modular tools that displays an image with navigation aids including scroll bars, and overview window, and a magnification box.

First, define a function that builds the app. This example defines a function called `my_large_image_display` at the end of the example.

After you define the function that builds the app, test the app. Read an image into the workspace.

```
I = imread('car1.jpg');
```

Display the image with navigation aids in the app.

```
my_large_image_display(I)
```

100%

App Creation Function

The `my_large_image_display` function accepts an image as an argument and displays the image in a figure window with scroll bars, an Overview tool, and a magnification box. Note that the function suppresses the toolbar and menu bar in the figure window because scrollable navigation is incompatible with standard MATLAB™ figure window navigation tools.

```matlab
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);

end
```

See Also

immagbox | imoverview | imscrollpanel

Related Examples

- "Build App to Display Pixel Information" on page 5-19
- "Create Image Comparison Tool Using ROIs" on page 12-37

More About

- "Interactive Tool Workflow" on page 5-6
- "Interactive Image Viewing and Processing Tools" on page 5-2
- "Add Scroll Panel to Figure" on page 5-10

Build Image Comparison Tool

This example shows how to make a GUI that displays two images side by side in scroll panels that are synchronized in location and magnification.

First, define a function that builds the app. This example uses a function called `my_image_compare_tool`, which is attached at the end of the example.

After you define the function that builds the app, test the app. Get two images.

```
I = imread('flamingos.jpg');
L = rgb2lightness(I);
Iedge = edge(L,'Canny');
```

Display the images in the app. When you move the detail rectangle in the Overview tool or change the magnification in one image, both images respond.

```
my_image_compare_tool(I,Iedge);
```
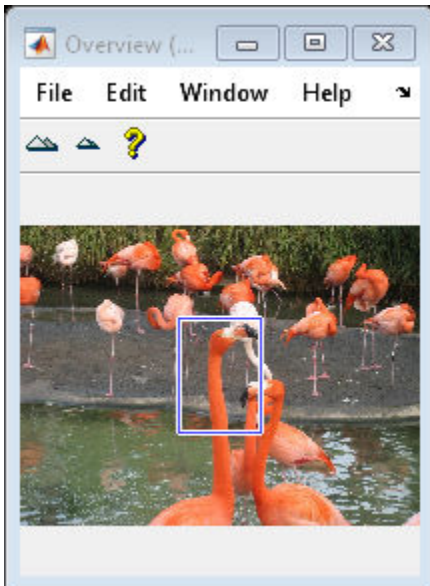
App Creation Function

The `my_image_compare_tool` function accepts two images as input arguments and displays the images in scroll panels. The custom tool also includes an Overview tool and a Magnification box. Note that the function suppresses the toolbar and menu bar in the figure window because scrollable navigation is incompatible with standard MATLAB™ figure window navigation tools.

To synchronize the scroll panels, the function makes the connections between tools using callbacks and the Scroll Panel API functions. The function specifies a callback function that executes 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.

```matlab
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);
```

```matlab
% 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 the left and right scroll panels
apiL.setMagnification(apiR.getMagnification())
apiL.setVisibleLocation(apiR.getVisibleLocation())

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

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

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

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

end
```

See Also

immagbox | imoverview | imscrollpanel

Related Examples

- "Build App to Display Pixel Information" on page 5-19
- "Build App for Navigating Large Images" on page 5-21

More About

- "Interactive Tool Workflow" on page 5-6
- "Interactive Image Viewing and Processing Tools" on page 5-2
- "Add Scroll Panel to Figure" on page 5-10

Create Angle Measurement Tool Using ROI Objects

Note The `impoly` function used in this example is not recommended. Use the new `drawpolyline` function and `Polyline` ROI object instead. See "Use Polyline to Create An Angle Measurement Tool" on page 12-72.

This example shows how to create an angle measurement tool using interactive 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.

```matlab
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);
```
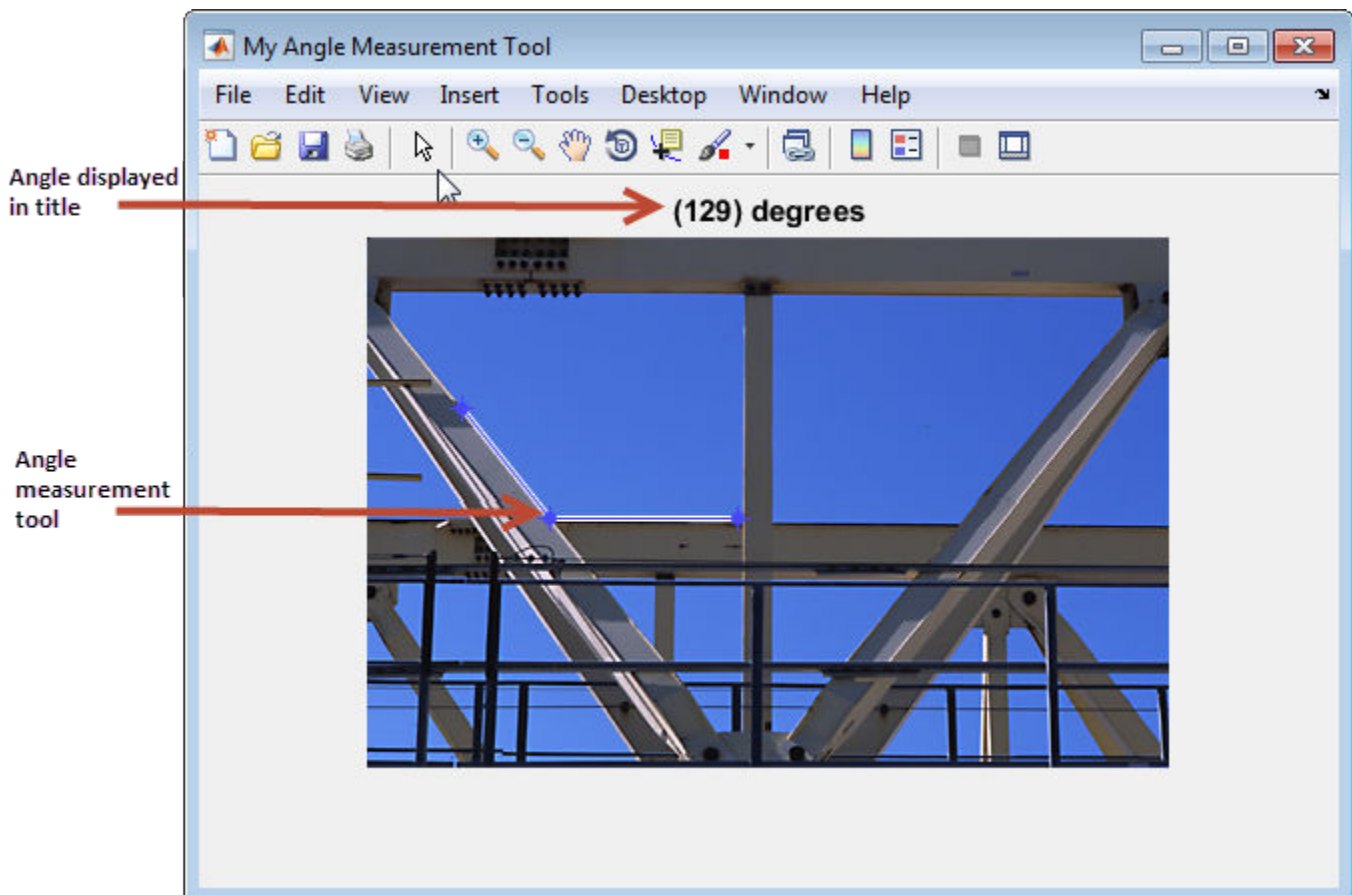
See Also

More About

- "ROI Creation Overview" on page 12-5
- "ROI Migration" on page 12-16
- "Use Polyline to Create An Angle Measurement Tool" on page 12-72

6

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

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
```
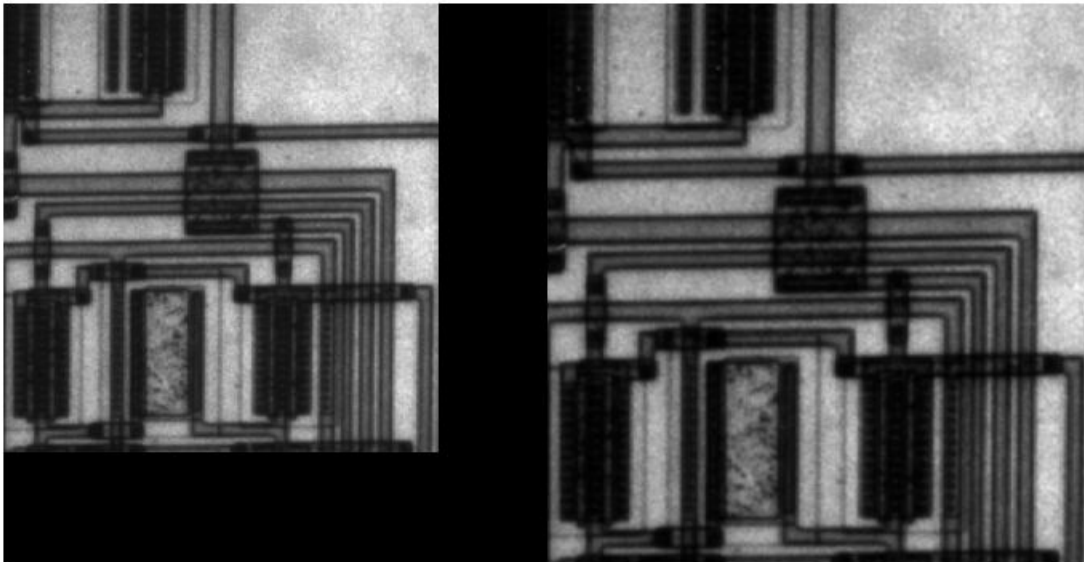

Specify the Size of the Output Image

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)
```
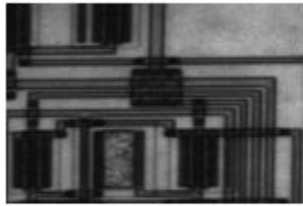
Specify the Interpolation Method

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)
```
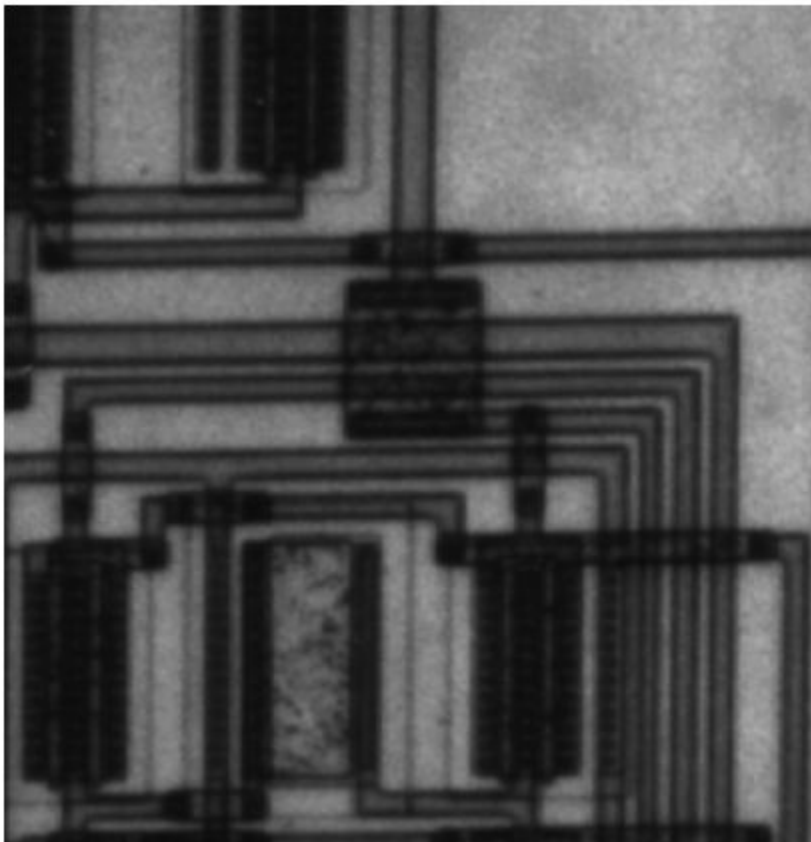
Prevent Aliasing When Shrinking an Image

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)
```
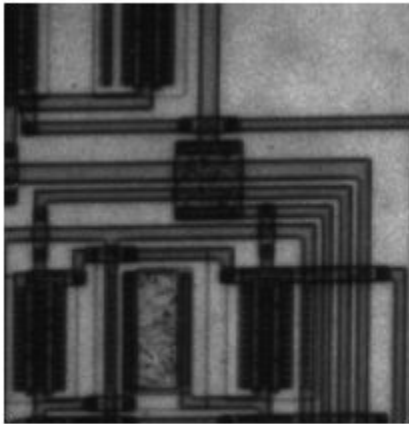
Rotate an Image

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')
```
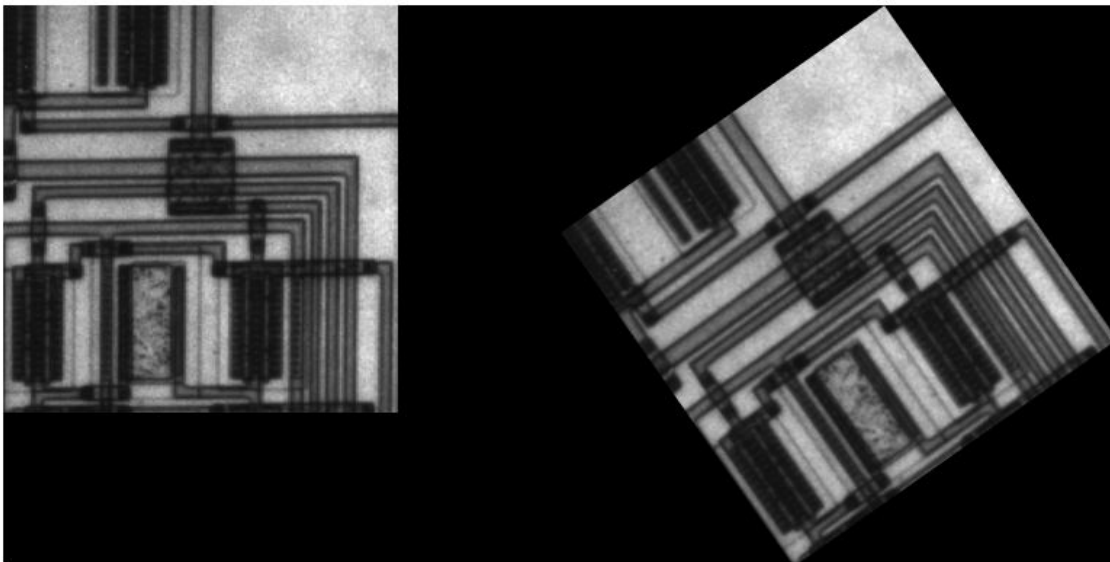
Crop a Rotated Image

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')
```
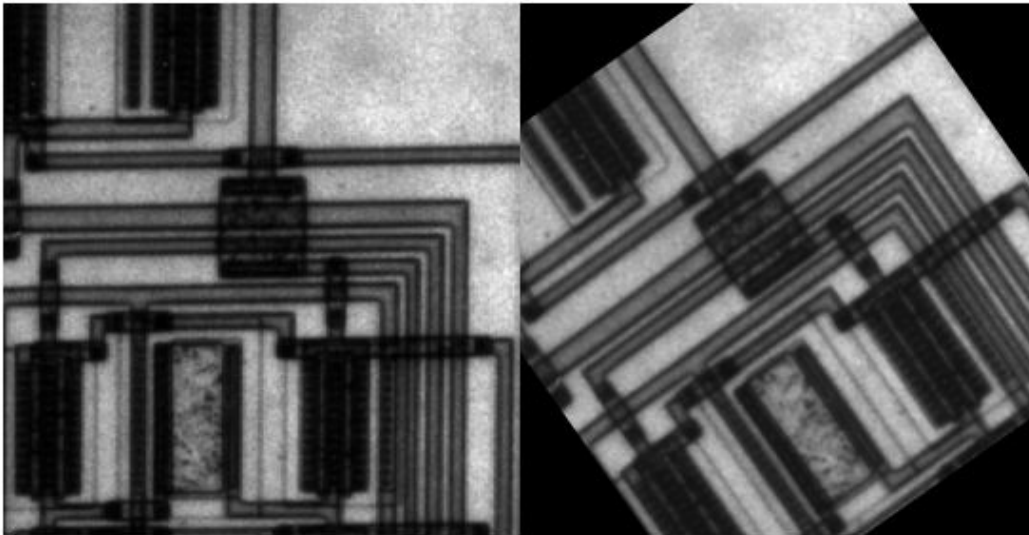
Crop an Image

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);
```
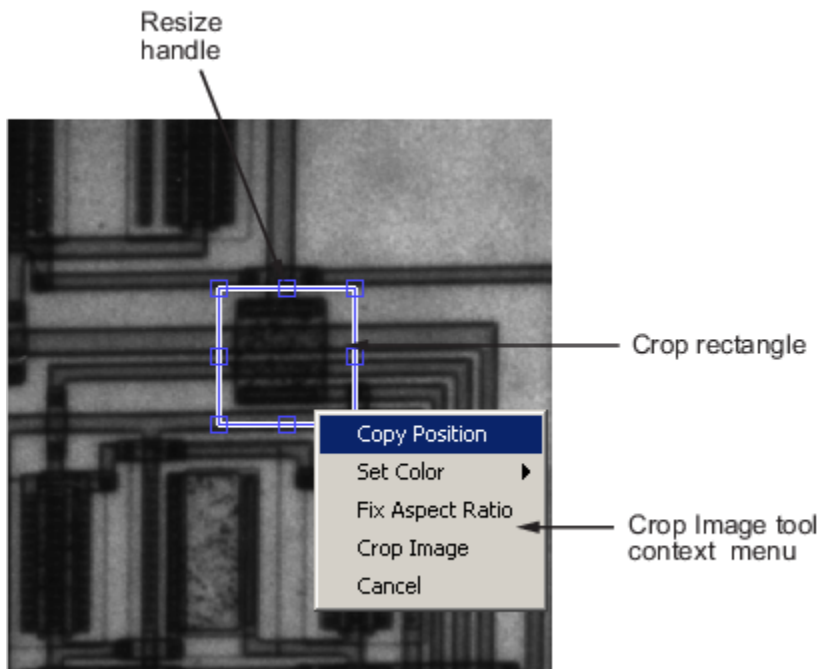

You can also specify the size and position of the crop rectangle as parameters when you call `imcrop`. Specify the crop rectangle as a four-element position vector, `[xmin ymin width height]`.

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')
```

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')
```

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')
```

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`.

```
Input Image          Geometric
                     Transformation Object
        |                    |
        v                    v
        +--------------------+
              imwarp
        +--------------------+
                 |
                 v
            Transformed
              Image
```
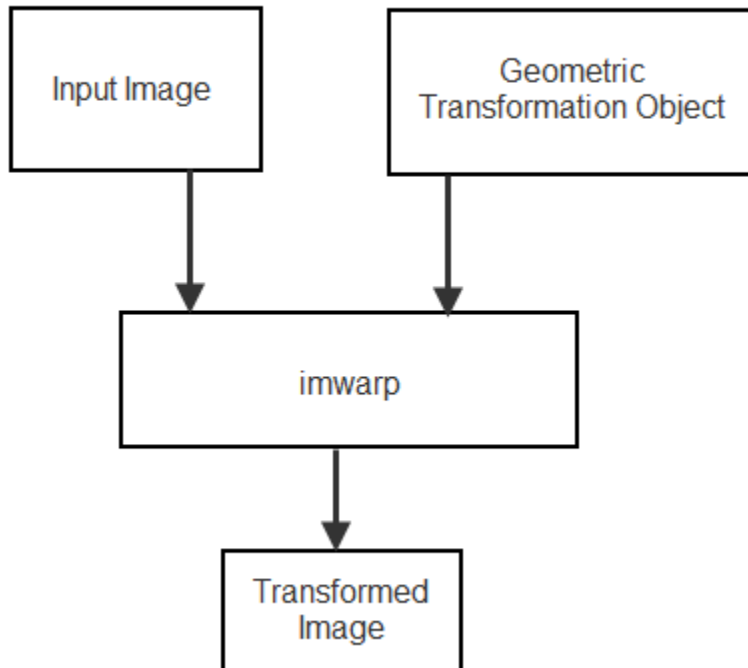
Create Geometric Transformation Object

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

- The `rigid2d`, `affine2d`, `projective2d`, `rigid3d`, 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 Transformation Object	affine2d	affine3d	rigid2d rigid3d	projective2d	geometricTransform2d geometricTransform3d	Other Transformation Objects
"Define Transformation Matrix" on page 6-14	X	X	X	X		
"Define Custom Point-Wise Mapping Function" on page 6-14					X	
"Estimate Transformation from Control Point Pairs" on page 6-15	X			X		X
"Estimate Transformation Using Similarity Optimization" on page 6-16	X	X				
"Estimate Transformation Using Phase Correlation" on page 6-16	X					

Define Transformation Matrix

If you know the transformation matrix for the geometric transformation you want to perform, then you can create a `rigid2d`, `affine2d`, `projective2d`, `rigid3d`, or `affine3d` geometric transformation object directly. For more information about creating a transformation matrix, see "Matrix Representation of Geometric Transformations" on page 6-17.

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: [function_handle]
        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: [function_handle]
        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 7-26.

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` | `rigid2d` | `rigid3d`

Related Examples
- "Perform Simple 2-D Translation Transformation" on page 6-25

More About
- "Matrix Representation of Geometric Transformations" on page 6-17
- "N-Dimensional Spatial Transformations" on page 6-29

Matrix Representation of Geometric Transformations
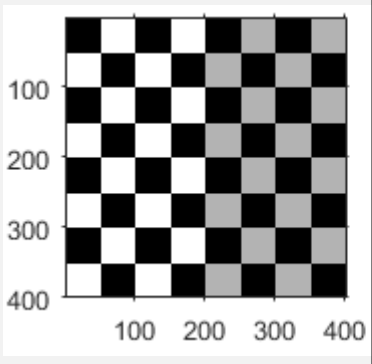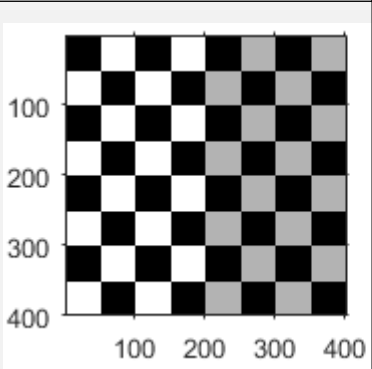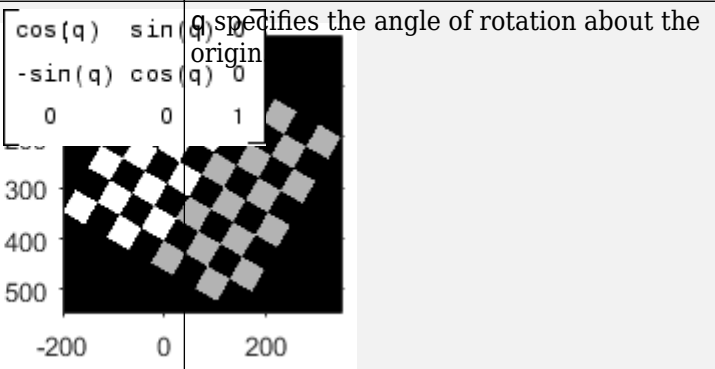
You can use a geometric transformation matrix to perform a global transformation of an image. First, define a transformation matrix and use it to create a geometric transformation object. Then, apply a global transformation to an image by calling `imwarp` with the geometric transformation object. For an example, see "Perform Simple 2-D Translation Transformation" on page 6-25.

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 any combination of 2-D transformation matrices to create an `affine2d` geometric transformation object. Use combinations of 2-D translation and rotation matrices to create a `rigid2d` geometric transformation object.
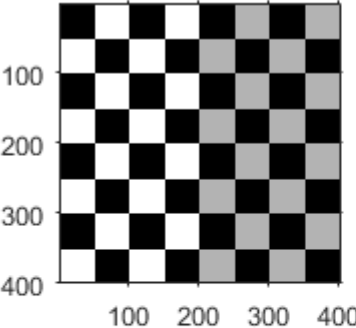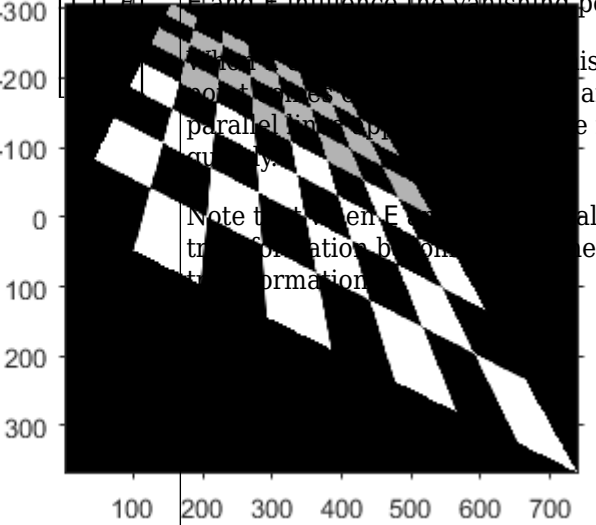
2-D Affine Transformation	Example (Original and Transformed Image)	Transformation Matrix	
Translation		$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$	t_x specifies the displacement along the *x* axis. t_y specifies the displacement along the *y* axis. For more information about pixel coordinates, see "Image Coordinate Systems" on page 2-3.
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.

2-D Affine Transformation	Example (Original and Transformed Image)	Transformation Matrix	
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 origin

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		$\begin{bmatrix} 1 & 0 & E \\ 0 & 1 & F \\ 0 & 0 & 1 \end{bmatrix}$ E and F influence the vanishing point. When E and F are large, the vanishing point comes closer to the origin and thus parallel lines appear to converge more quickly. Note that when E and F are equal to 0, the transformation becomes an affine transformation.

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)))
```
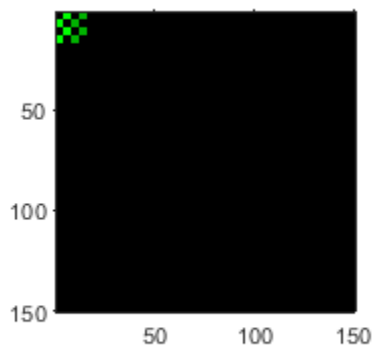
Create a translation matrix, and store it as an `affine2d` geometric transformation object. This translation will shift the image horizontally by 100 pixels.

```
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)))
```
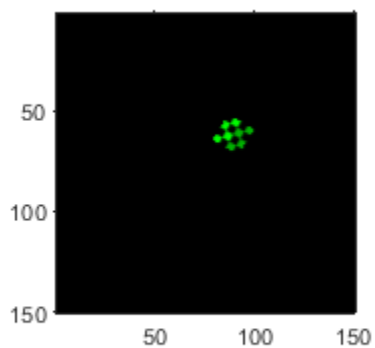
Rotation Followed by Translation

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)))
```
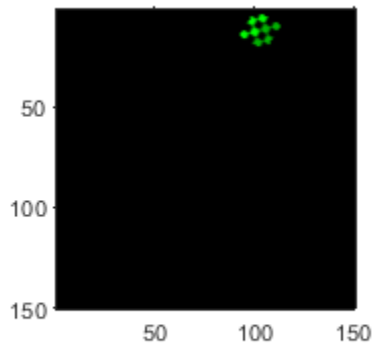

Notice how the spatial position of the transformed image is different than when translation was followed by rotation.

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 any combination of 3-D transformation matrices to create an `affine3d` geometric transformation object. Use combinations of 3-D translation and rotation matrices to create a `rigid3d` 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
`affine2d` | `affine3d` | `fitgeotrans` | `imwarp` | `projective2d` | `rigid2d` | `rigid3d`

Related Examples
- "Perform Simple 2-D Translation Transformation" on page 6-25

More About
- "2-D and 3-D Geometric Transformation Process Overview" on page 6-13

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)
```
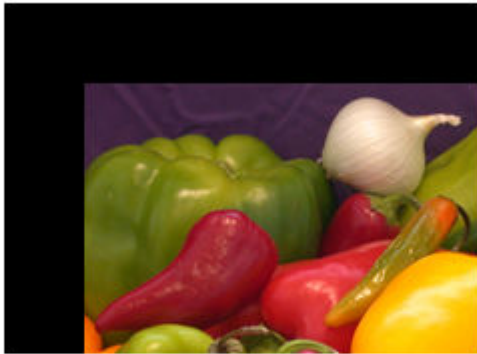
Now perform the transformation, this time specifying a fill value.

```
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 6-13
- "Matrix Representation of Geometric Transformations" on page 6-17

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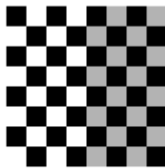
Get spatial referencing information about the image. This information is useful when you want to display the result of the transformation.

```
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)
```
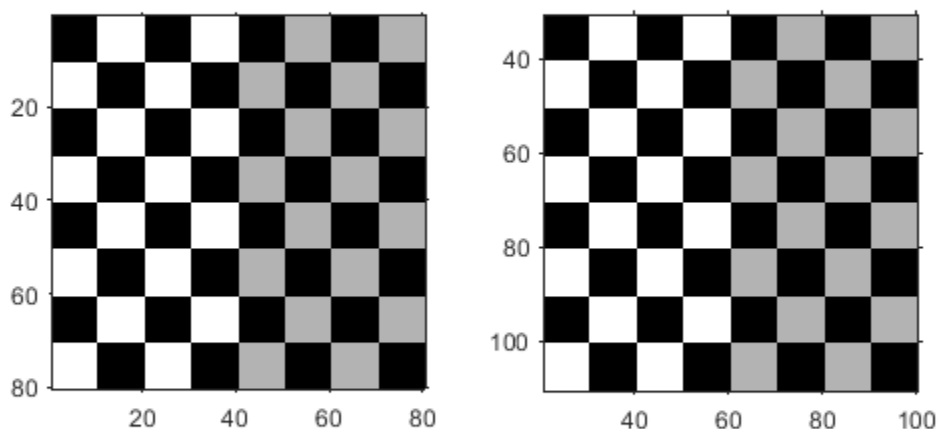
To see the entirety of the transformed image in the same relation to the origin of the coordinate space as the original image, use `imwarp` with the `'OutputView'` parameter, specifying a spatial referencing object. The spatial referencing object specifies the size of the output image and how much of 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)
```
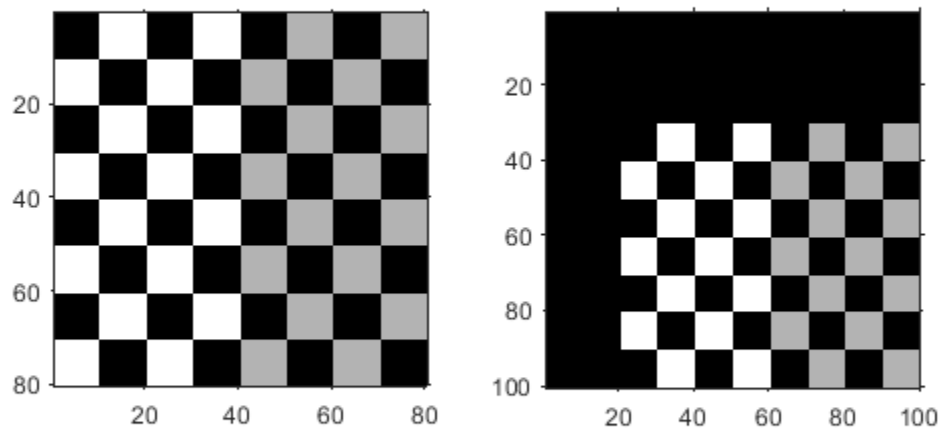

See Also

`affine2d` | `imref2d` | `imwarp`

More About

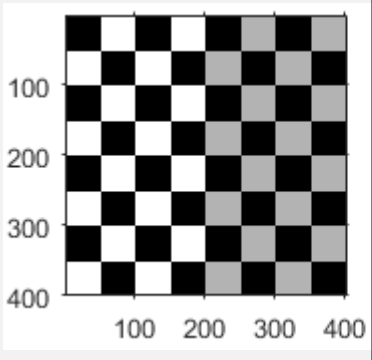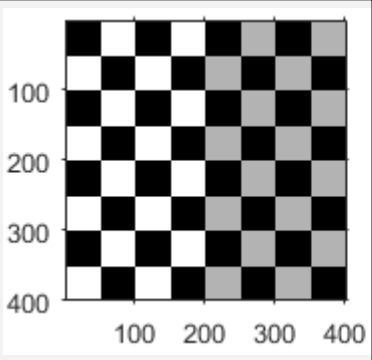- "2-D and 3-D Geometric Transformation Process Overview" on page 6-13
- "Matrix Representation of Geometric Transformations" on page 6-17

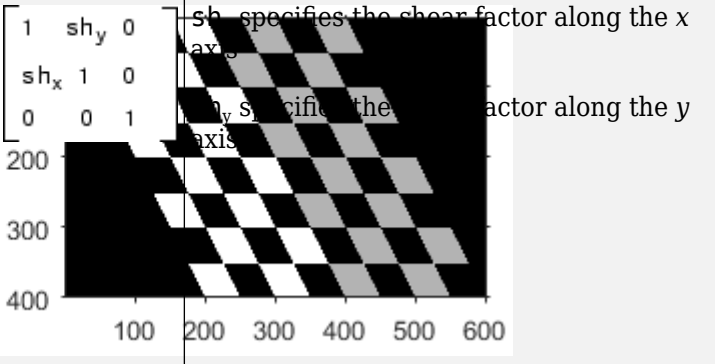# N-Dimensional Spatial Transformations

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)
```


Original image   Transformed image
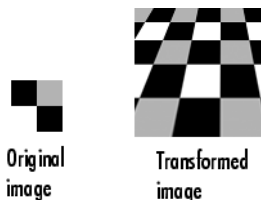
The `imtransform` function options let you control many aspects of the transformation. For example, note how the transformed image appears to contain multiple copies of the original image. This is

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 6-13

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');
```


Image courtesy of mPower3/Emerge
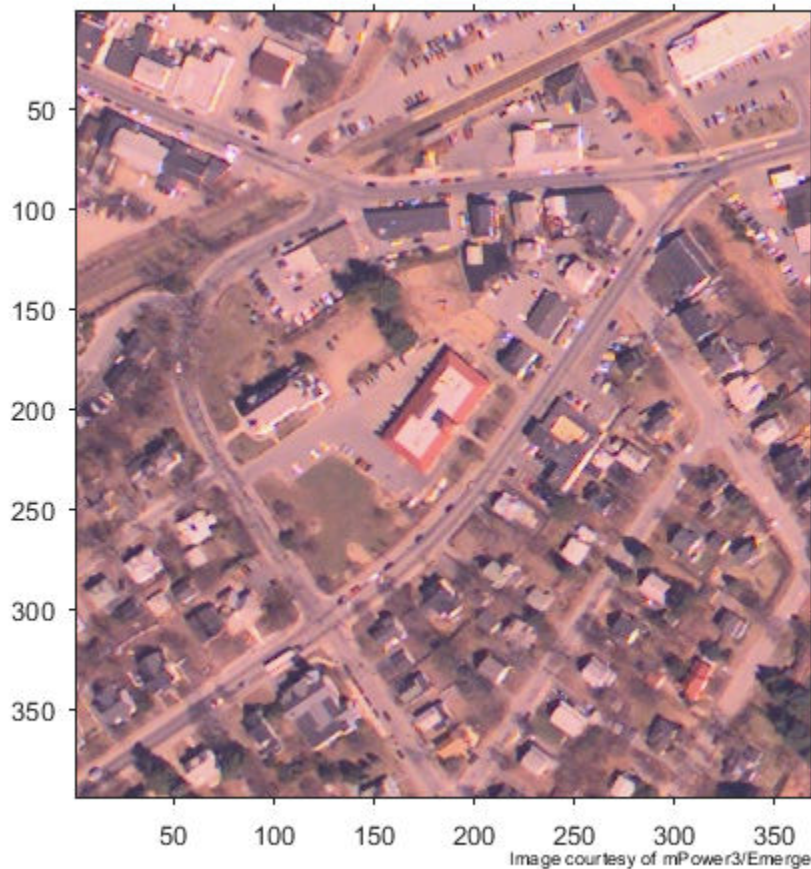
Load a MAT-file that contains preselected control points for the `fixed` and `moving` images and create a geometric transformation fit to the control points, using `fitgeotrans` .

```
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);
```
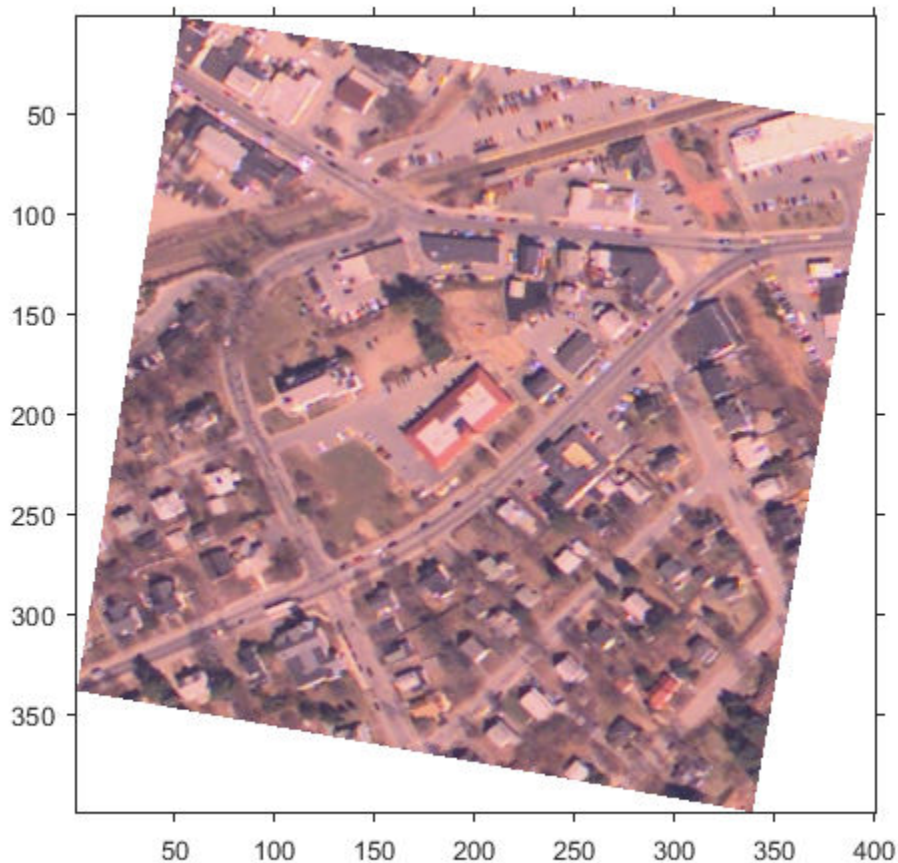

Overlay the transformed image, `registered`, over the `fixed` image, using `imshowpair`. Notice how the two images appear misregistered. This happens because `imshowpair` assumes that the images are both in the default intrinsic coordinate system. The next steps provide two ways to remedy this display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

```
figure, imshowpair(fixed,registered,'blend');
```
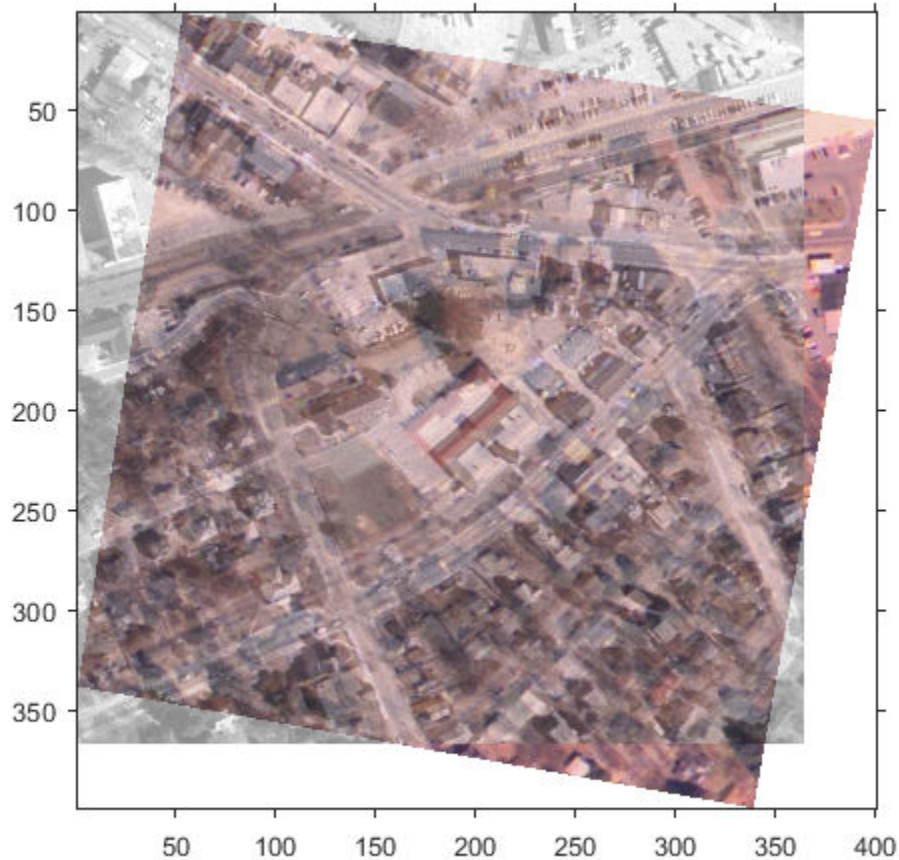
Constrain the transformed image, `registered`, to the same number of rows and columns, and the same spatial limits as the `fixed` image. This ensures that the registered image appears registered with the fixed image but areas of the registered image that would extrapolate beyond the extent of the fixed 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');
```
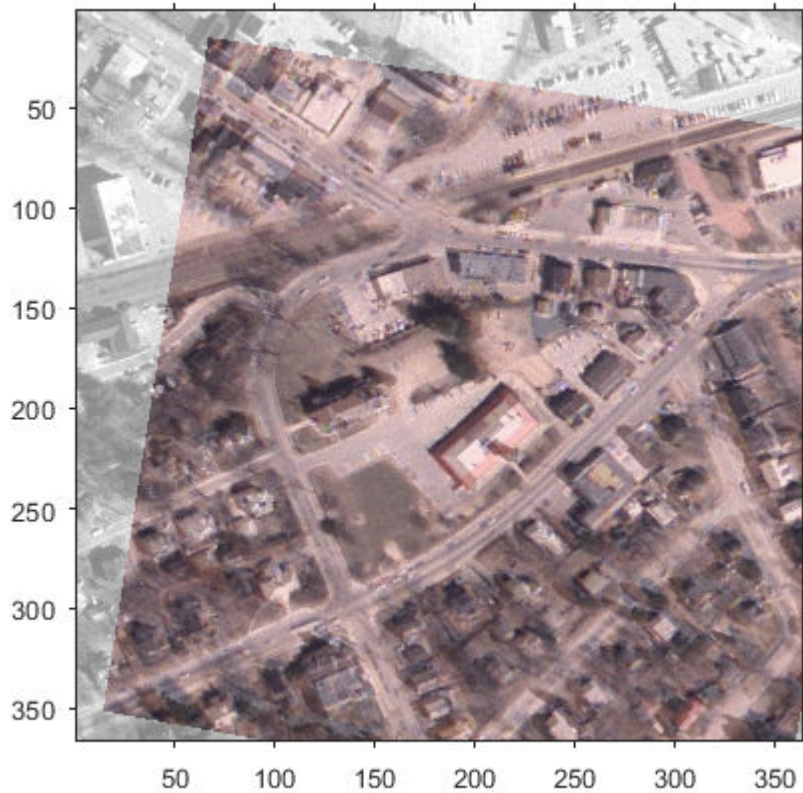
As an alternative, use the optional `imwarp` syntax that returns the output spatial referencing object that indicates the position of the full transformed image in the same default intrinsic coordinate system as the `fixed` image. Display the registered image over the fixed image and note that now the full `registered` image is visible.

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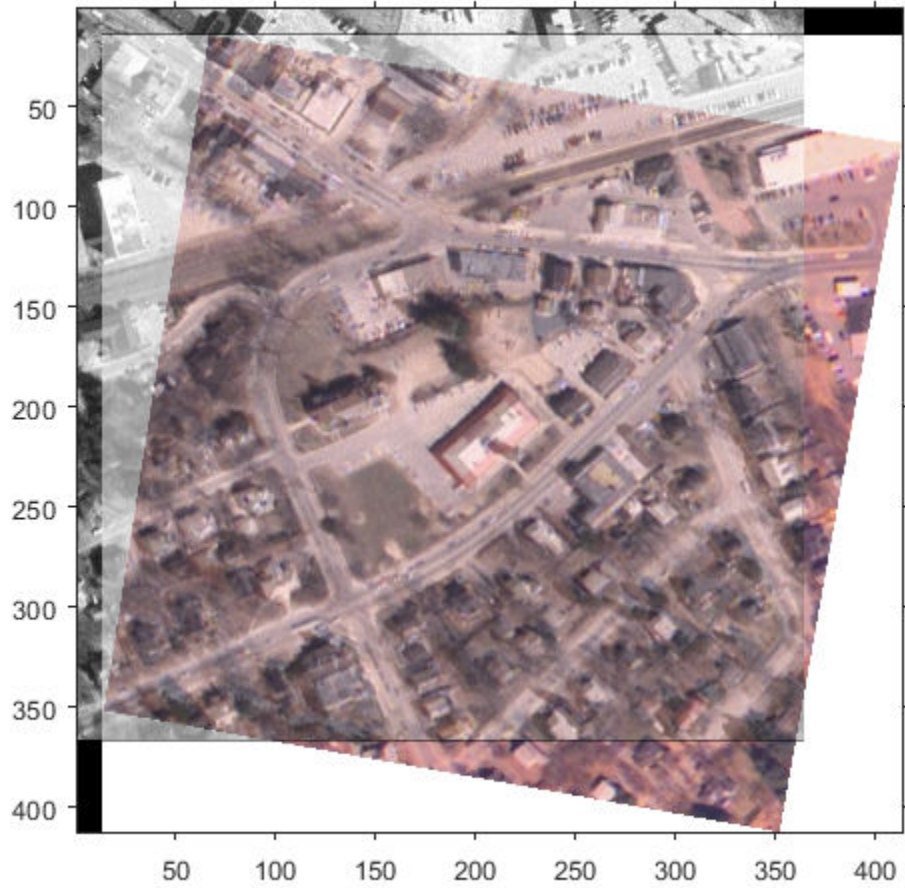
Clean up.

```
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')
```

Original
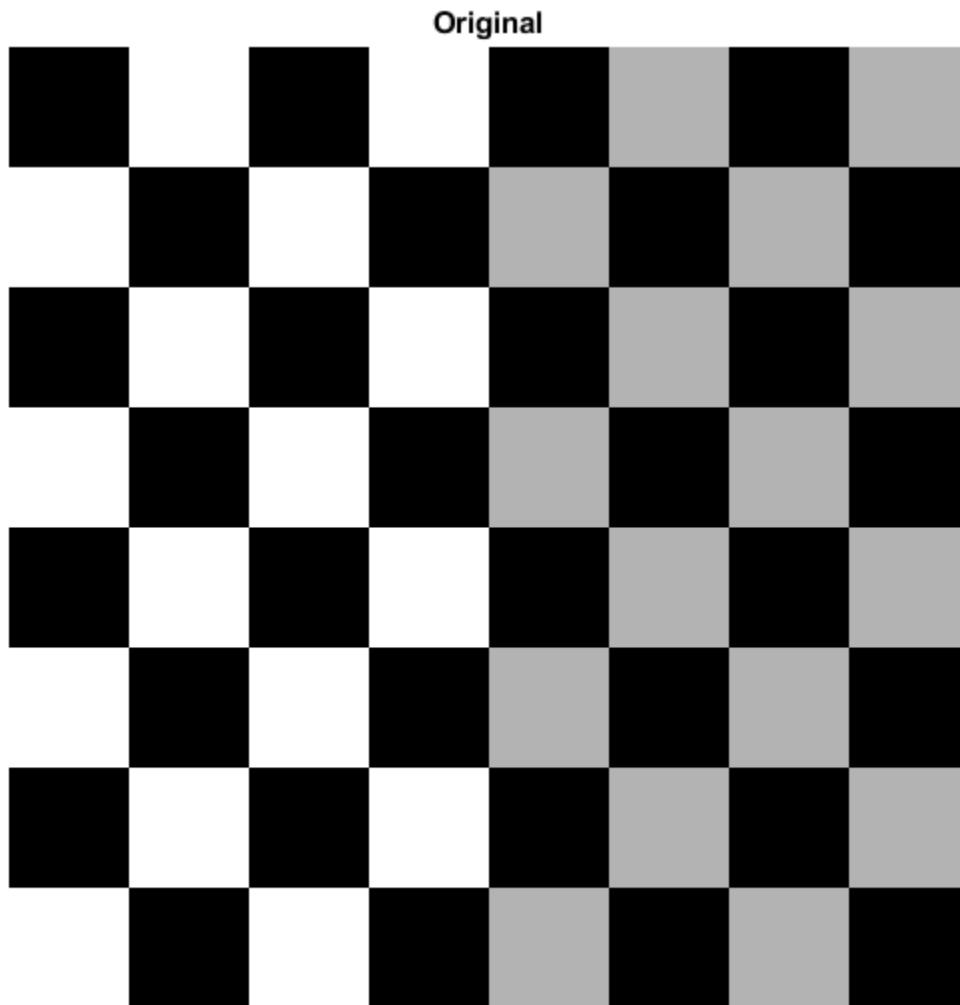
Image 2: Apply Nonreflective Similarity to Checkerboard

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')
```

Nonreflective Similarity
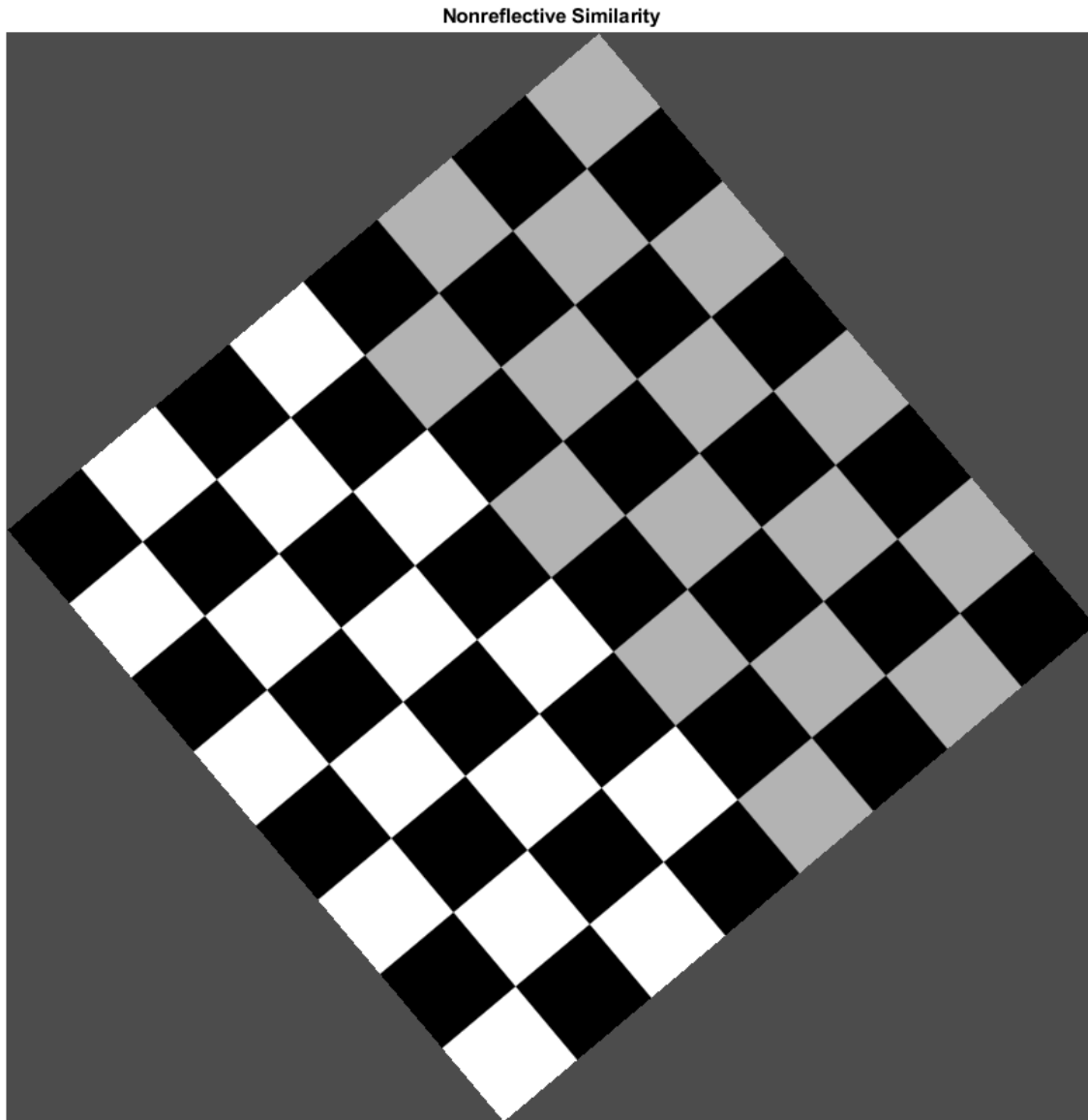
If you change either `tx` or `ty` to a non-zero value, you will notice that it has no effect on the output image. If you want to see the coordinates that correspond to your transformation, including the translation, include spatial referencing information:

```
[I_nonreflective_similarity,RI] = imwarp(I,t_nonsim,'FillValues',fill);

imshow(I_nonreflective_similarity,RI)
axis on
title('Nonreflective Similarity (Spatially Referenced)')
```

Nonreflective Similarity (Spatially Referenced)
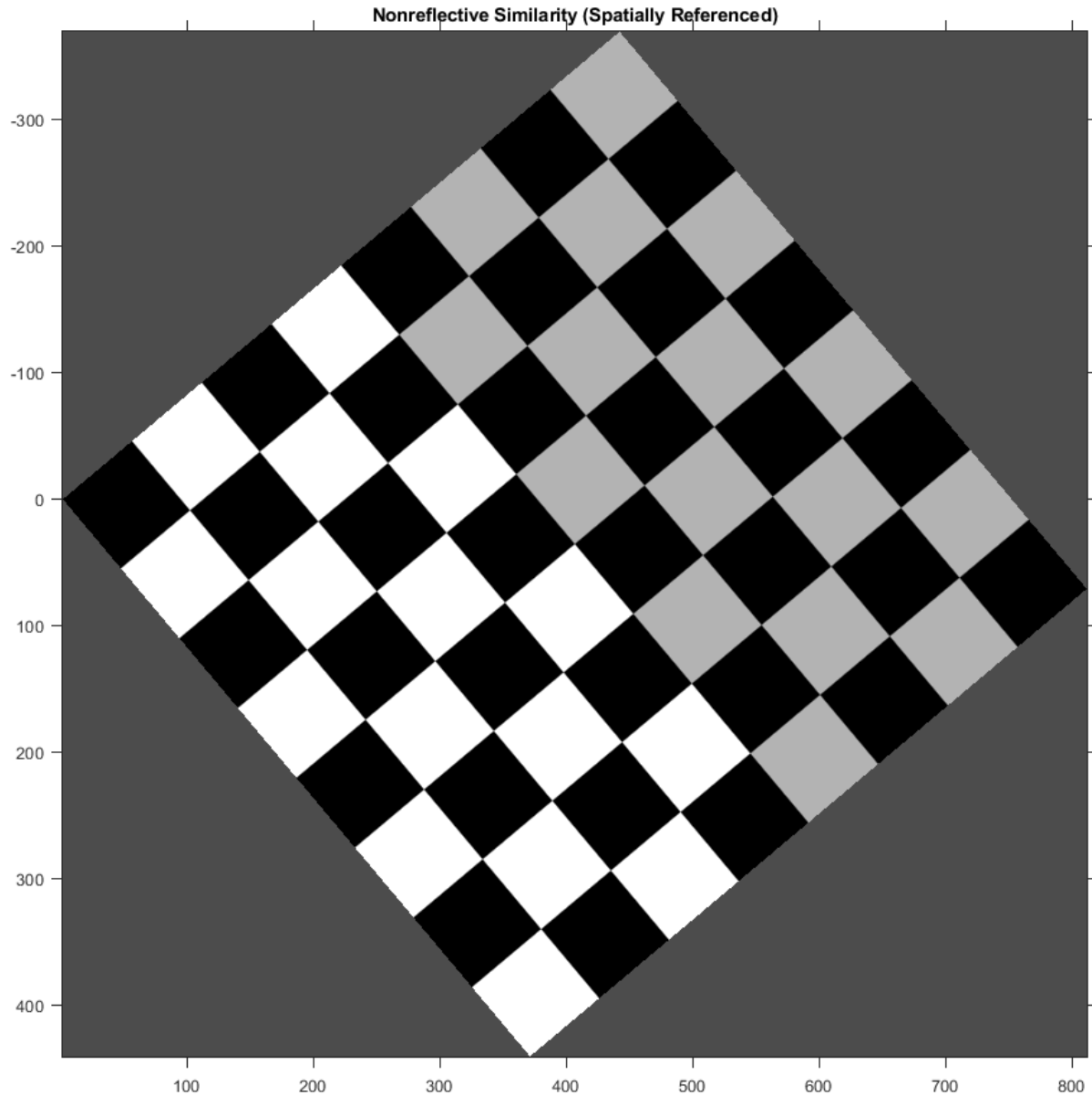
Notice that passing the output spatial referencing object `RI` from `imwarp` reveals the translation. To specify what part of the output image you want to see, use the 'OutputView' name-value pair in the `imwarp` function.

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')
```
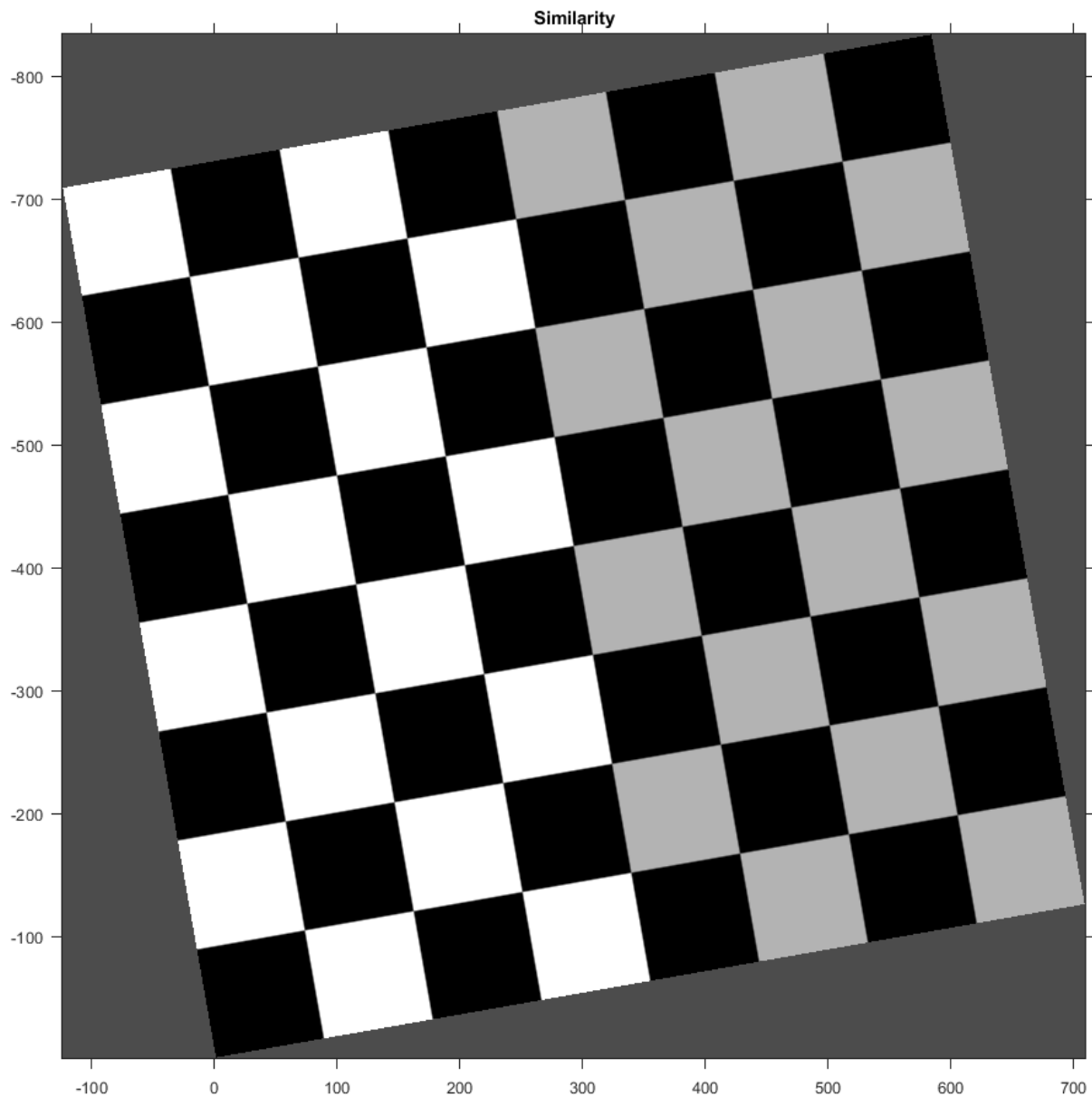
Image 4: Apply Affine Transformation to Checkerboard

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')
```
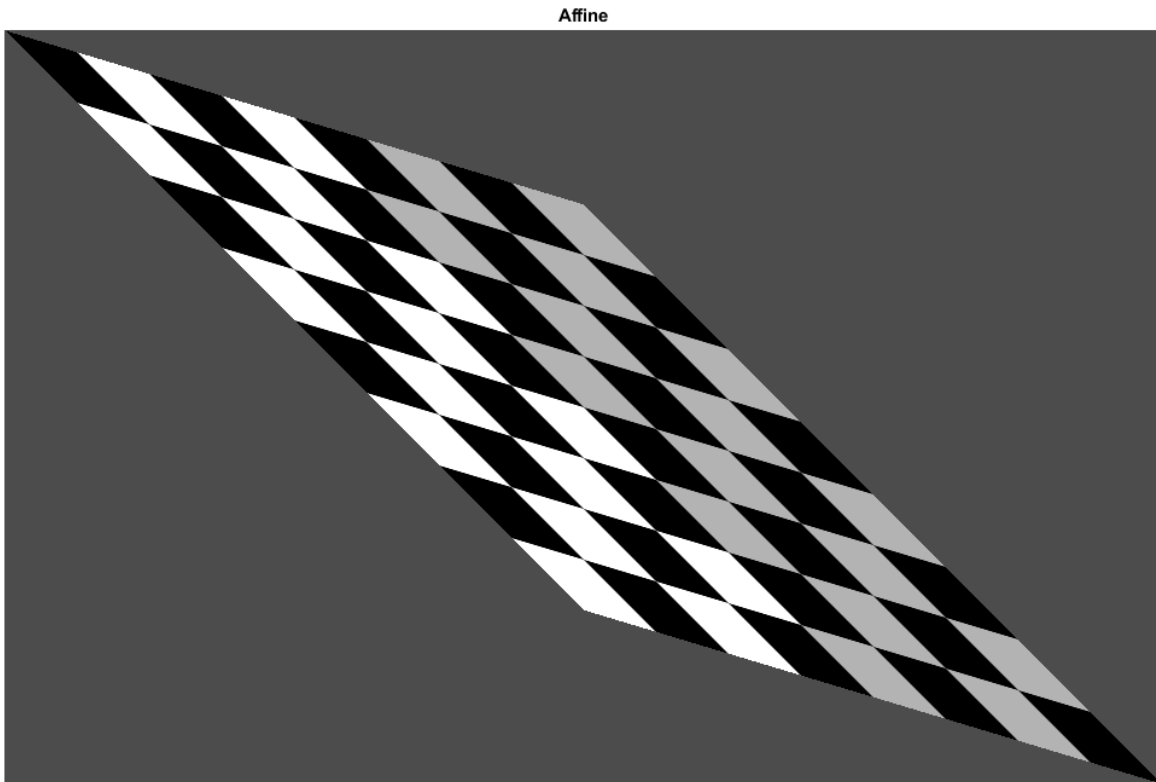
Affine

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:

$$[up\ vp\ wp] = [x\ y\ w]T$$

$$u = \frac{up}{wp}$$

$$v = \frac{vp}{wp}$$

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')
```

Projective
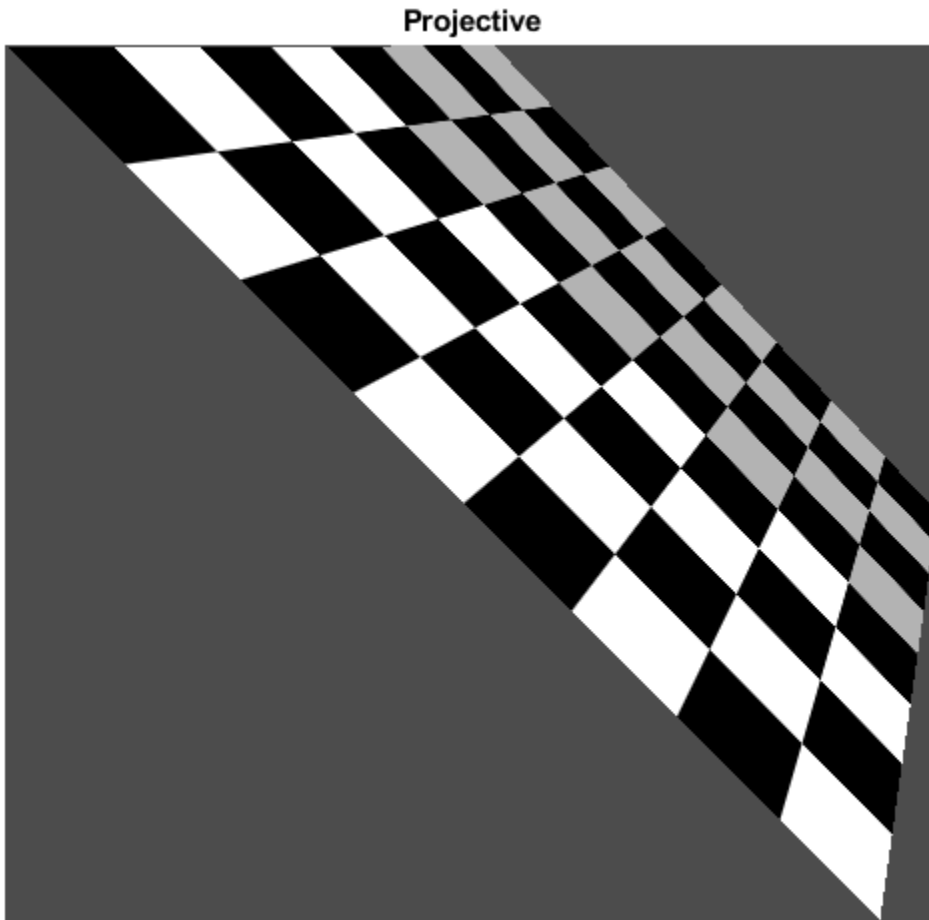
Image 6: Apply Piecewise Linear Transformation to Checkerboard

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')
```

Piecewise Linear
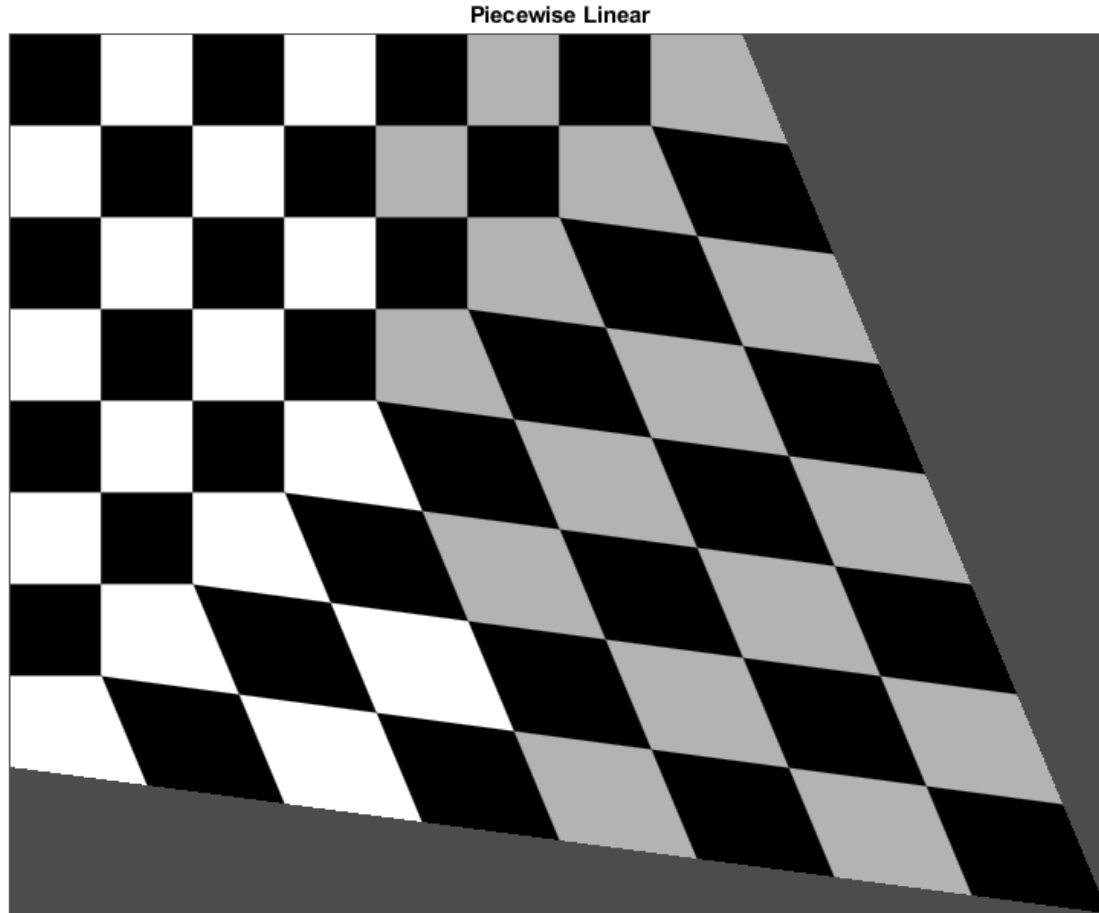
Image 7: Apply Sinusoidal Transformation to Checkerboard

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')
```

Sinusoid
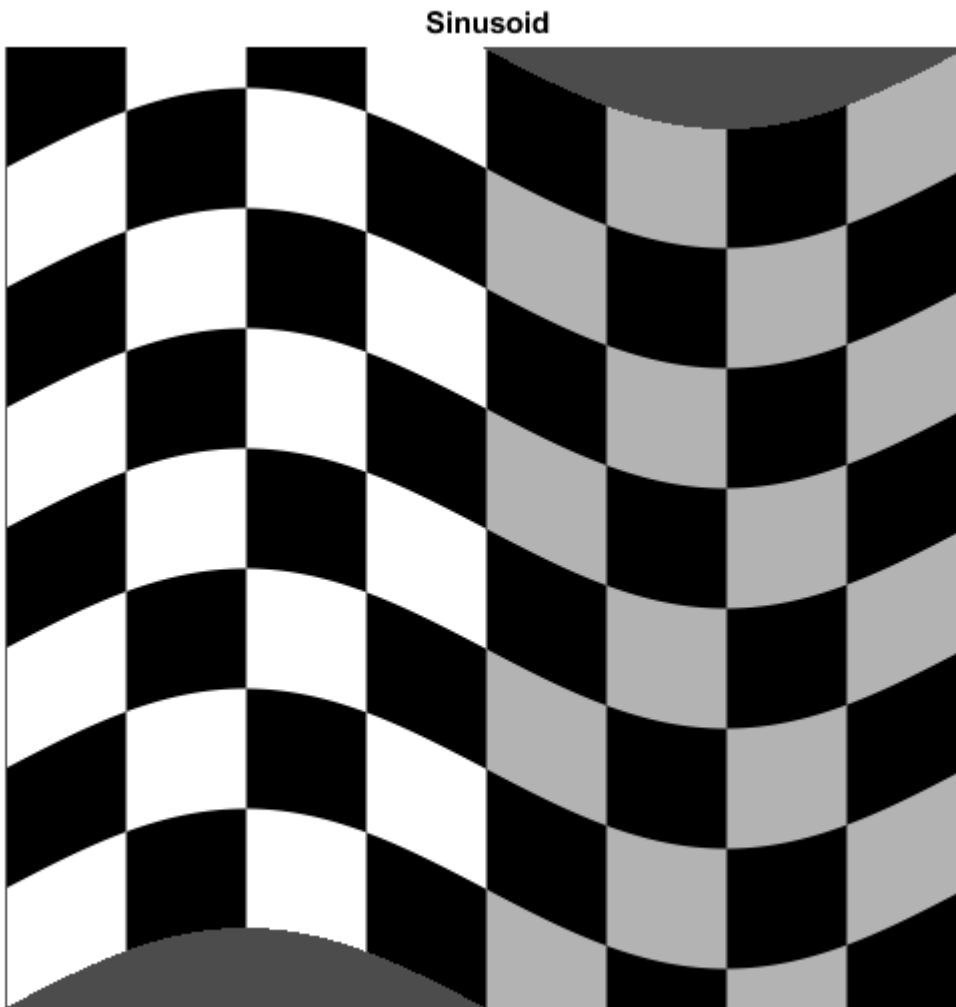
Image 8: Apply Barrel Transformation to Checkerboard

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')
```

Barrel
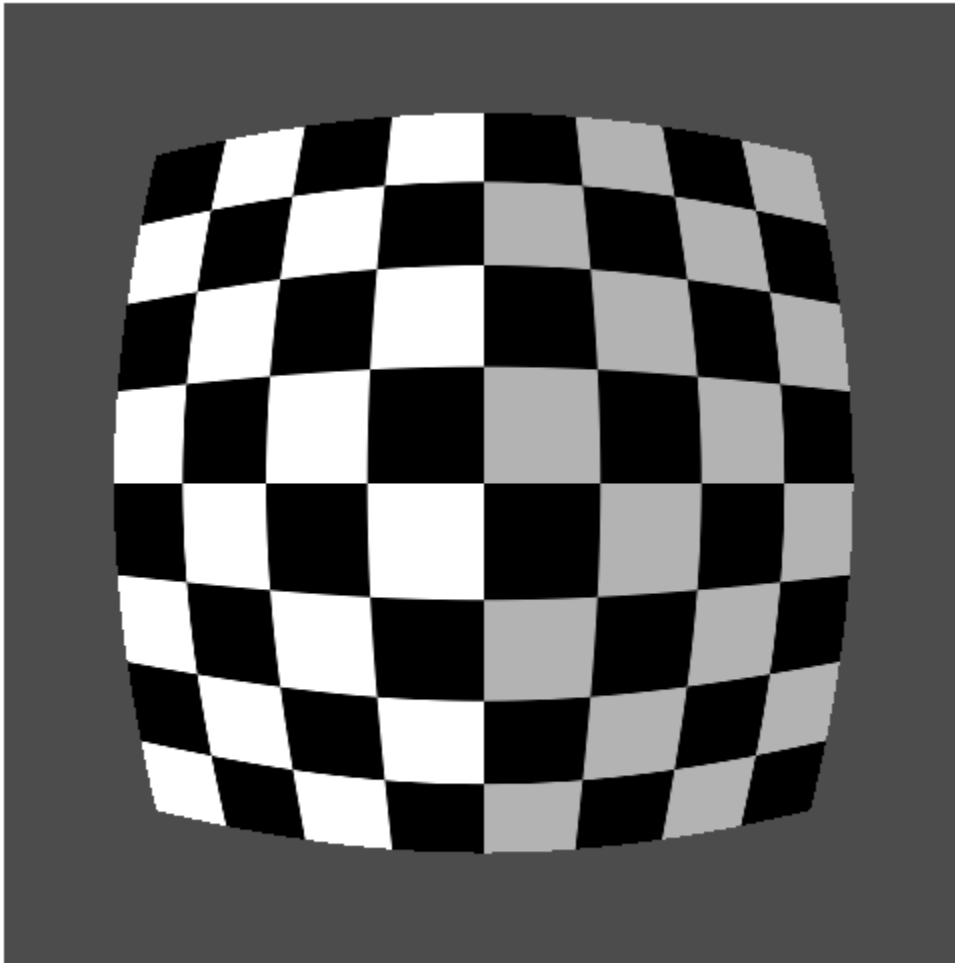
Image 9: Apply Pin Cushion Transformation to Checkerboard

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')
```


Pin Cushion
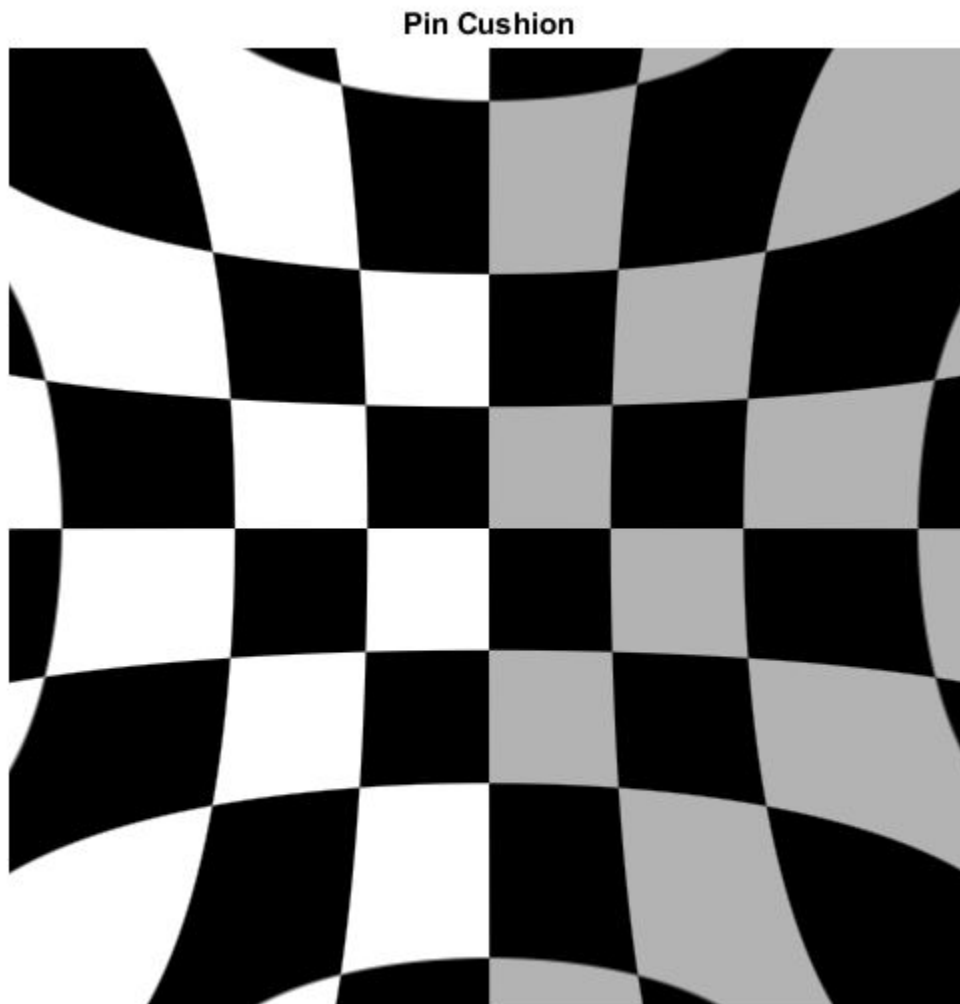
Summary: Display All Geometric Transformations of Checkerboard

```
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')
```
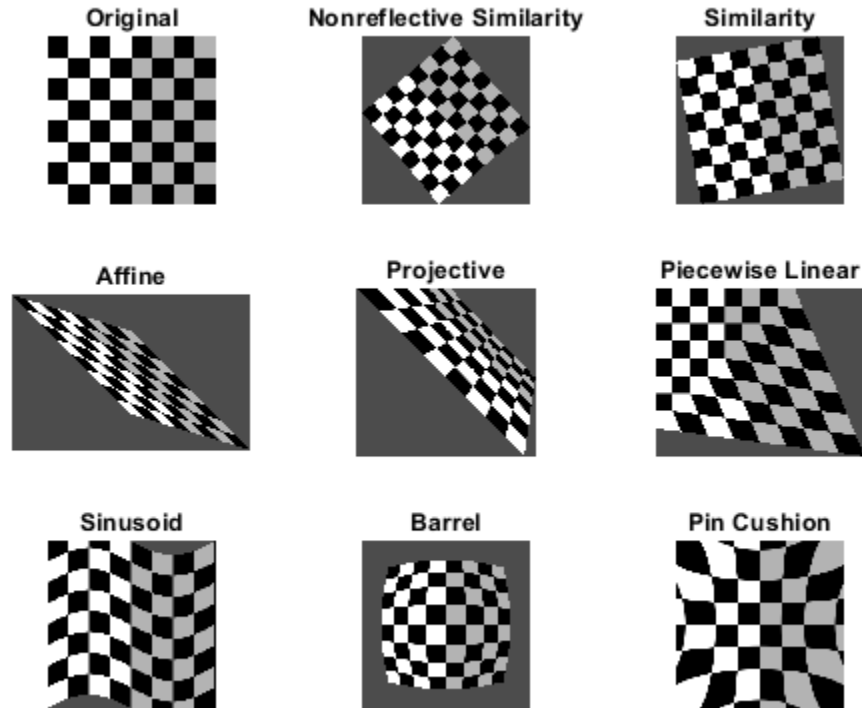
Note that `subplot` changes the scale of the images being displayed.

See Also

Functions
checkerboard | fitgeotrans | imwarp | makeresampler | tformarray

Objects
LocalWeightedMeanTransformation2D | PiecewiseLinearTransformation2D | PolynomialTransformation2D | affine2d | projective2d

More About

- "Matrix Representation of Geometric Transformations" on page 6-17
- "2-D and 3-D Geometric Transformation Process Overview" on page 6-13

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)
```

Original Image

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)
```
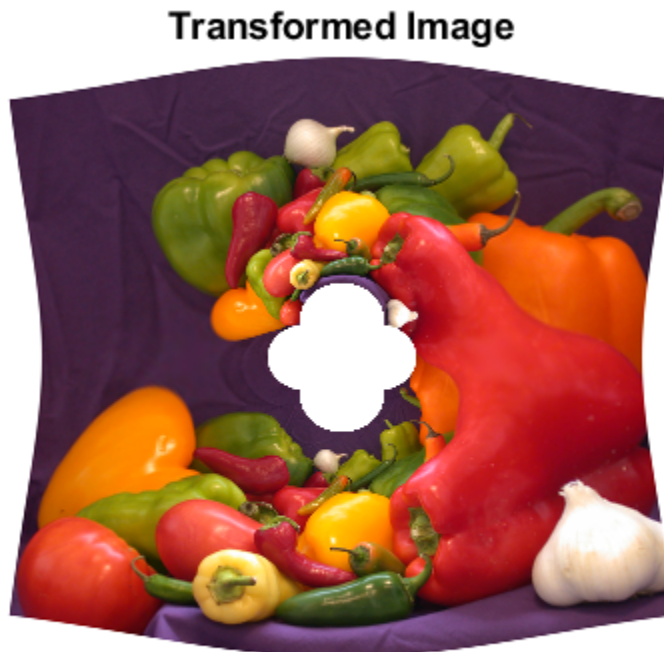

Transformed Image

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;
```
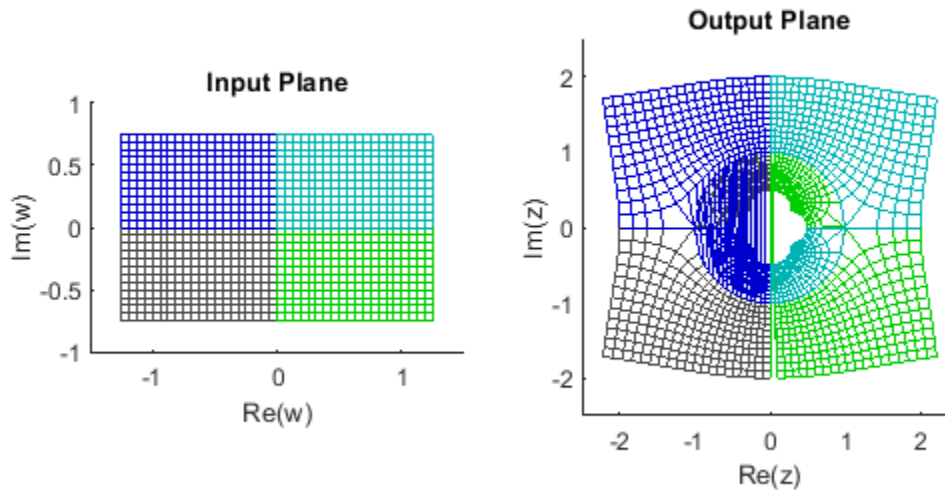

You can see that the grid lines are color-coded according to their quadrants in the input plane before and after the transformations. The colors also follow the transformed grids to the output planes. Note that each quadrant transforms to a region outside the unit circle and to a region inside the unit 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;
```
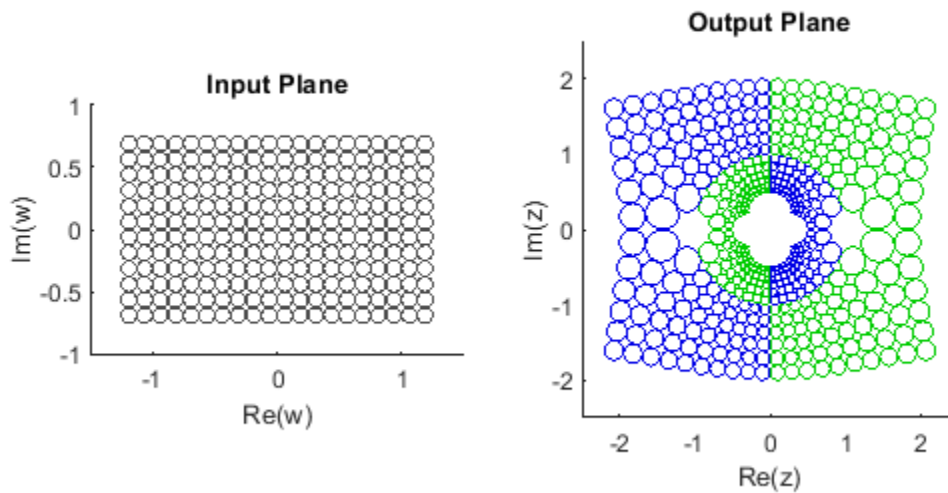
You can see that the transform to a circle packing where tangencies have been preserved. In this example, the color coding indicates use of the positive (green) or negative (blue) square root of $w^2 - 1$. Note that the circles change dramatically but that they remain circles (shape-preservation, once 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)
```

Original Image Superposed on Input Plane
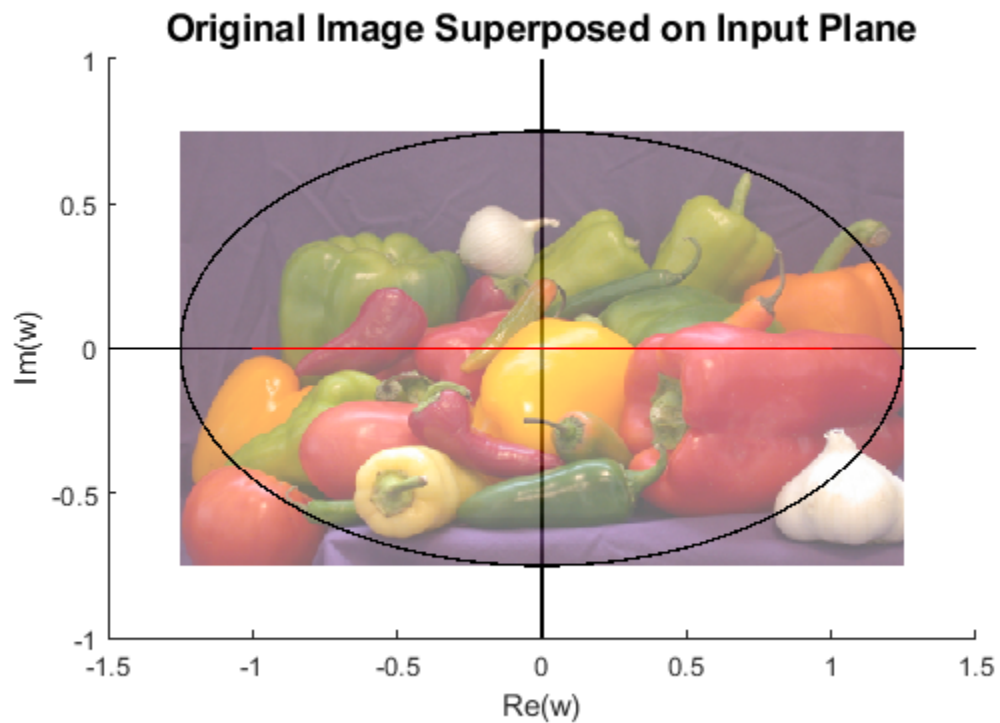
Next we display the output image over the output axes of the conformal map, along with two black circles and one red circle. Again, the image is semi-transparent.

```
figure
axOut = conformalSetupOutputAxes(axes);
conformalShowOutput(axOut, B, xData, yData)
title('Transformed Image Superposed on Output Plane','FontSize',14)
```

Transformed Image Superposed on Output Plane
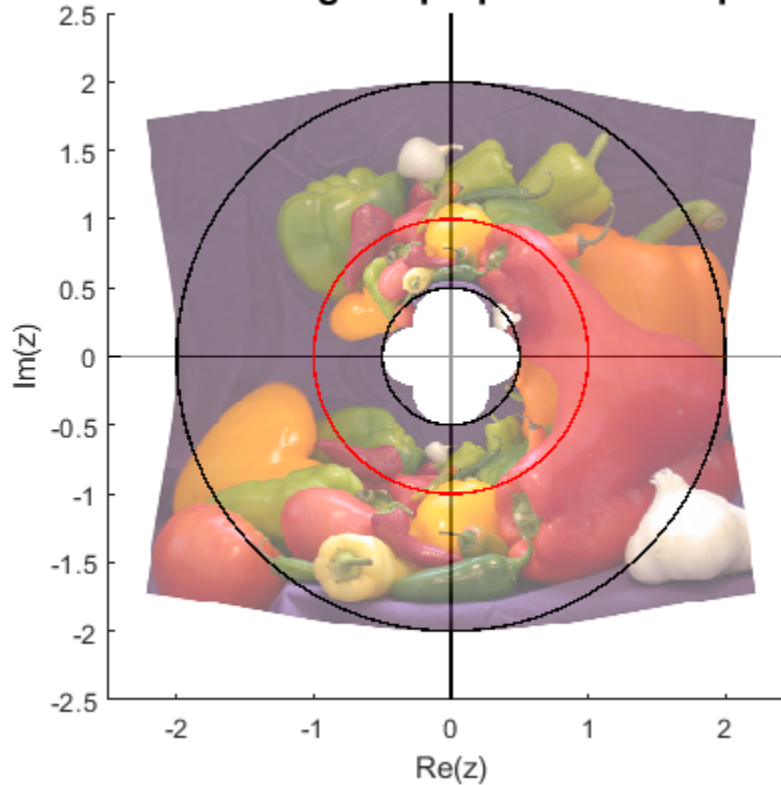
MATLAB® graphics made it easy to shift and scale the original and transformed images to superpose them on the input (`w-`) and output (`z-`) planes, respectively. The use of semi-transparency makes it easier to see the ellipse, line, and circles. The ellipse in the w-plane has intercepts at 5/4 and -5/4 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);
```

Transformed Image With Masking

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);
```

Winter Greens Image

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);
```

Transformed and Masked Winter Greens Image

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');
```
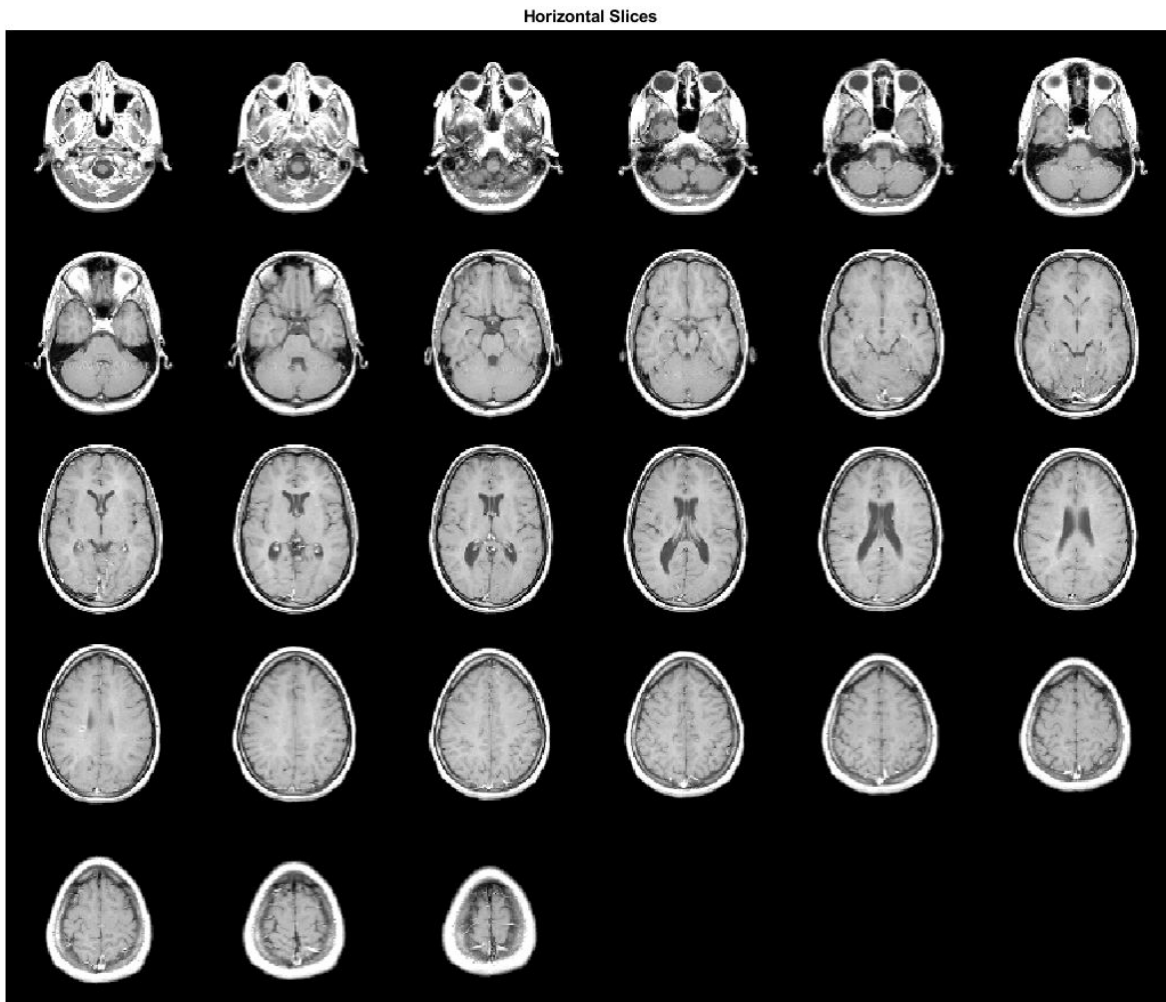
Horizontal Slices

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');
```


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')
```

Sagittal - IMTRANSFORM
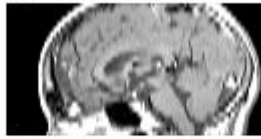
Step 3: Extract Sagittal Slice from the Horizontal Slices Using TFORMARRAY

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');
```

Sagittal - TFORMARRAY
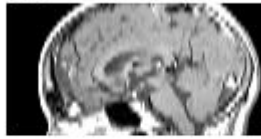
Step 4: Create and Display Sagittal Slices

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');
```
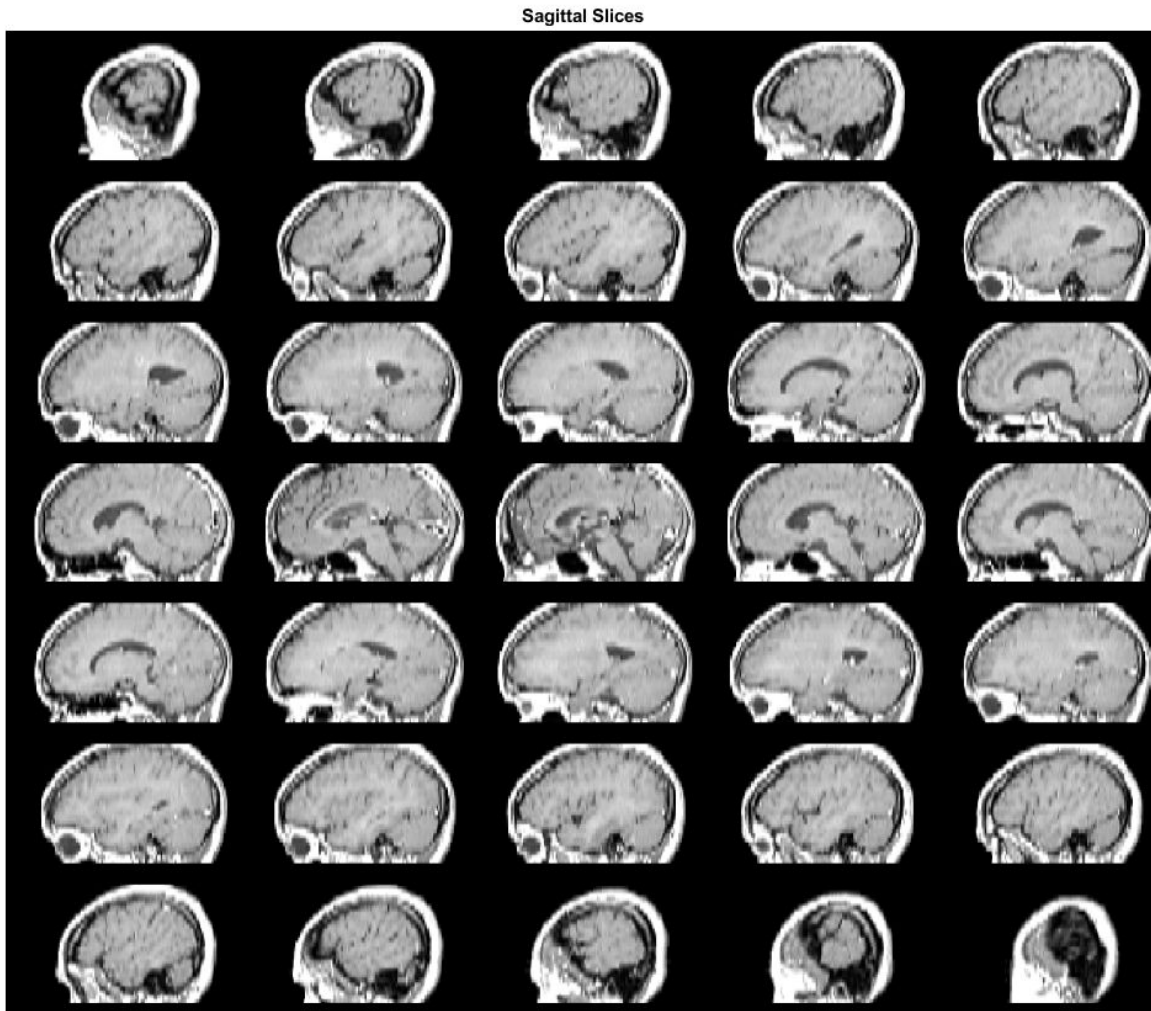
Sagittal Slices

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
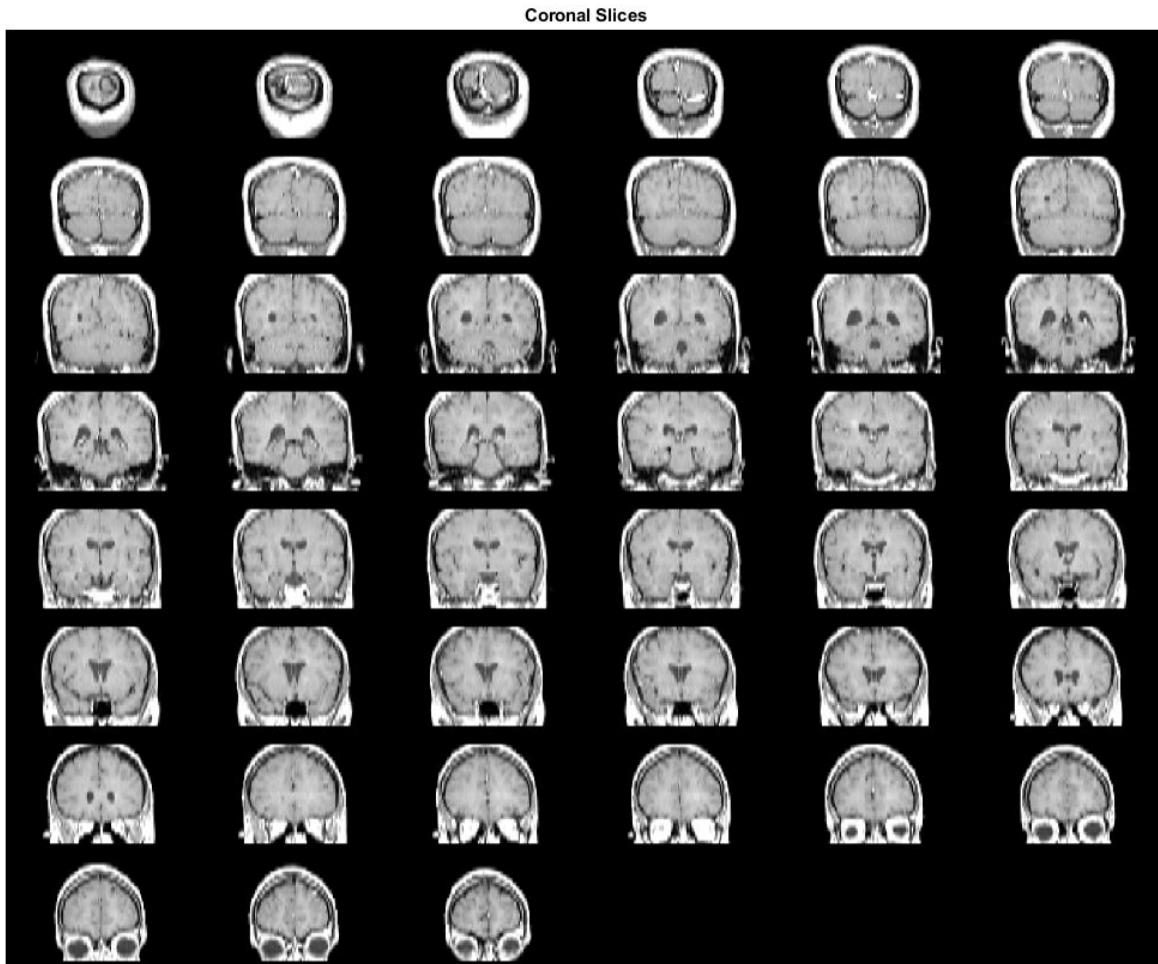
Padding and Shearing an Image Simultaneously

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');
```

Original Image
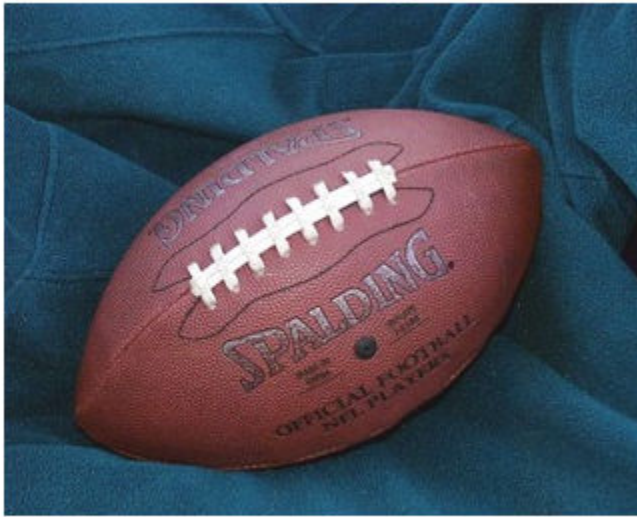
We choose a shade of orange as our fill value.

```
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');
```

Sheared Image
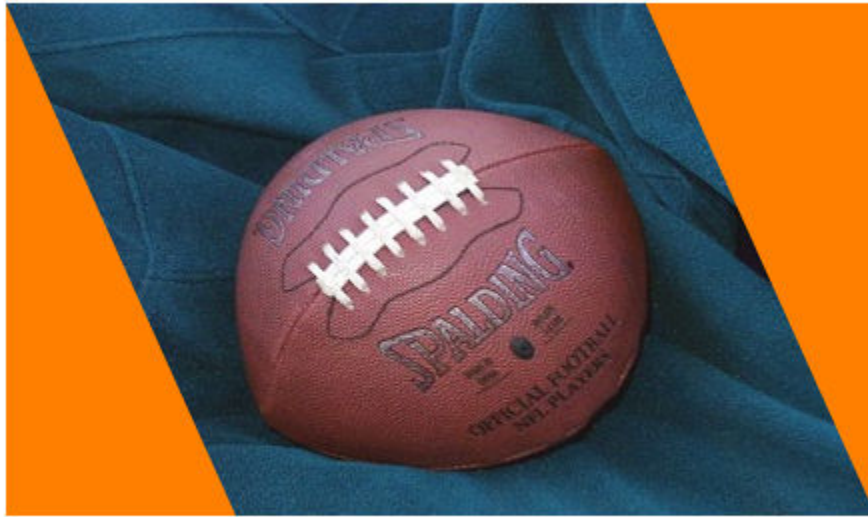
Step 2: Explore the Transformation

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);
```

Original Image
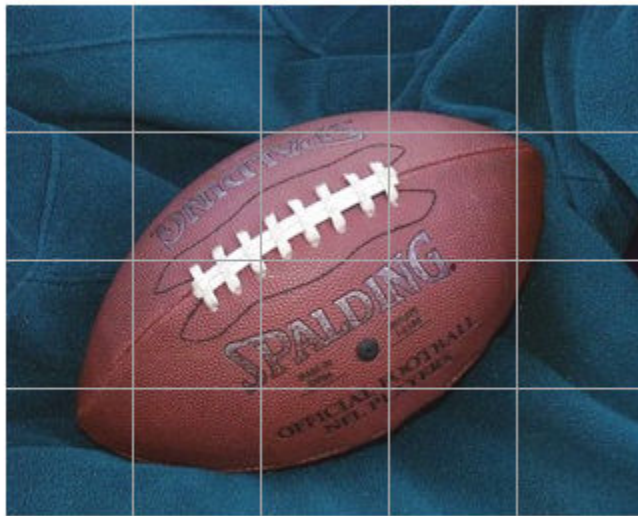

```
figure(h2);
hold on;
line(X, Y, 'Color',gray);
line(X',Y','Color',gray);
```

Sheared Image

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
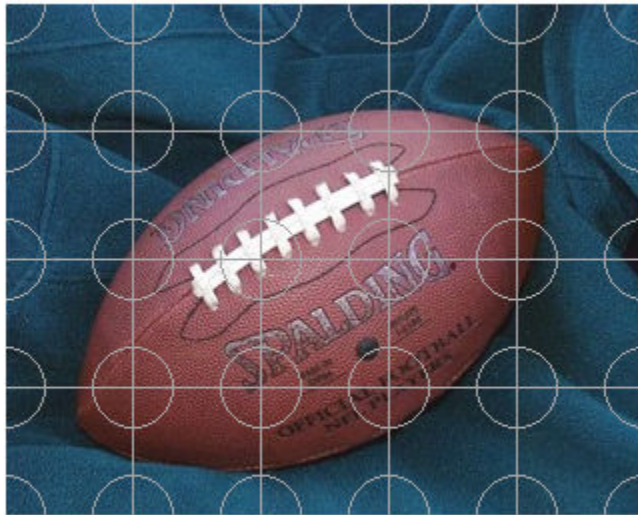
Sheared Image
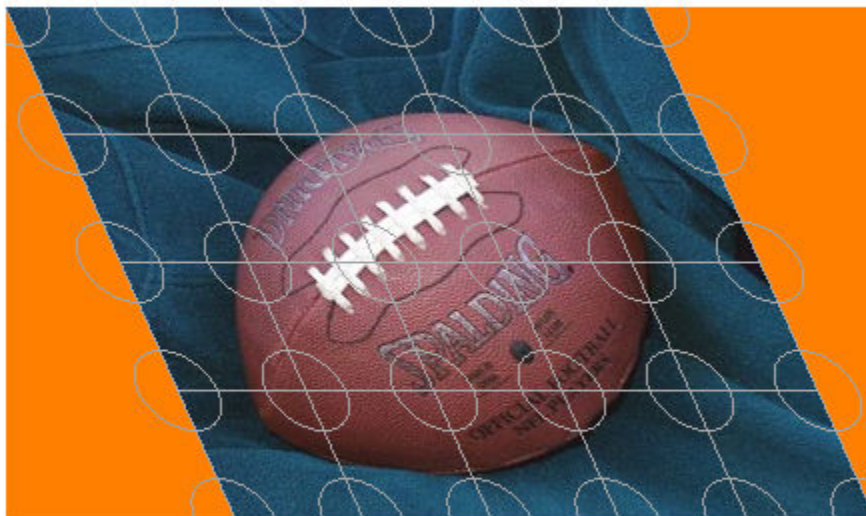
Step 3: Compare the 'fill', 'replicate', and 'bound' Pad Methods

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''');
```

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''');
```

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''');
```

Pad Method = 'bound'
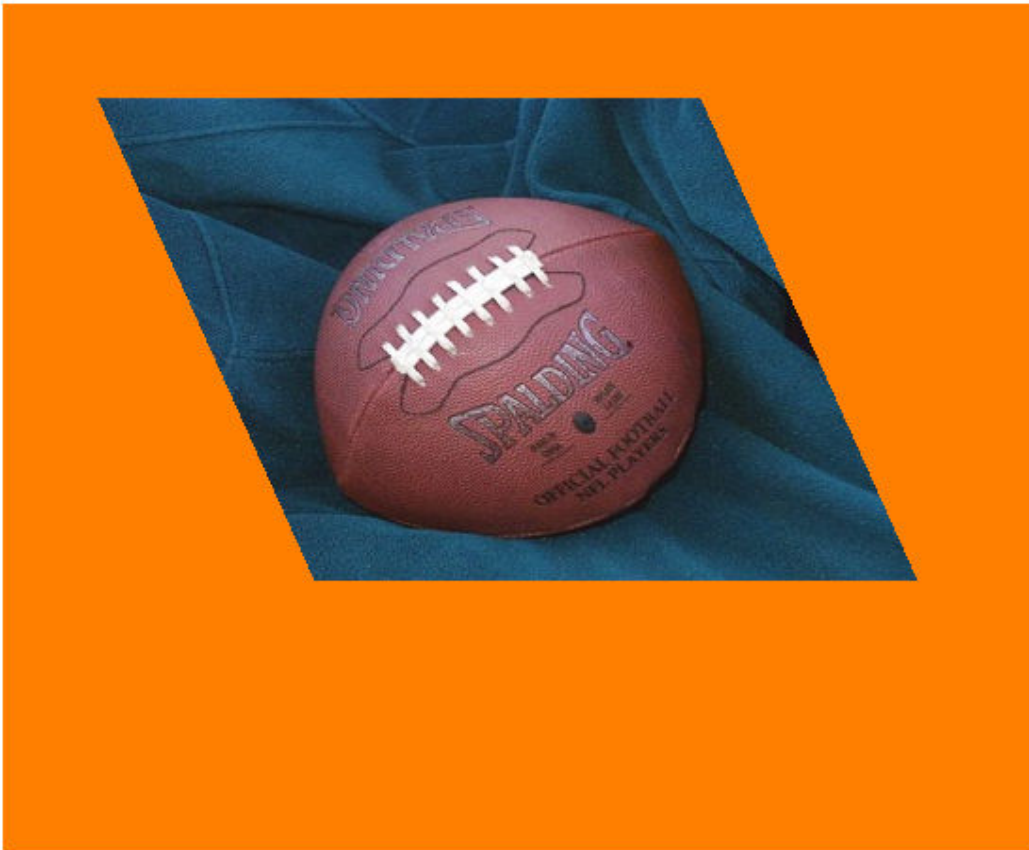
Results with `'fill'` and `'bound'` look very similar, but look closely and you'll see that the edges are smoother with `'fill'`. That's because the input image is padded with the fill values, then the cubic interpolation is applied across the edge, mixing fill and image values. In contrast, `'bound'` 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''');
```
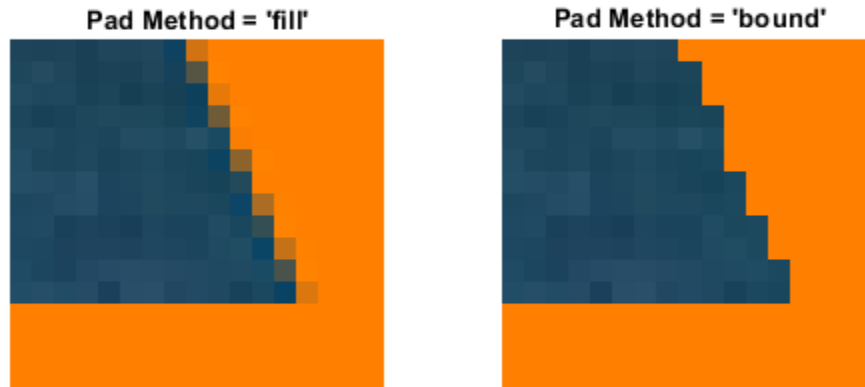

Step 4: Exercise the 'circular' and 'symmetric' Pad Methods

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''');
```

Pad Method = 'circular'
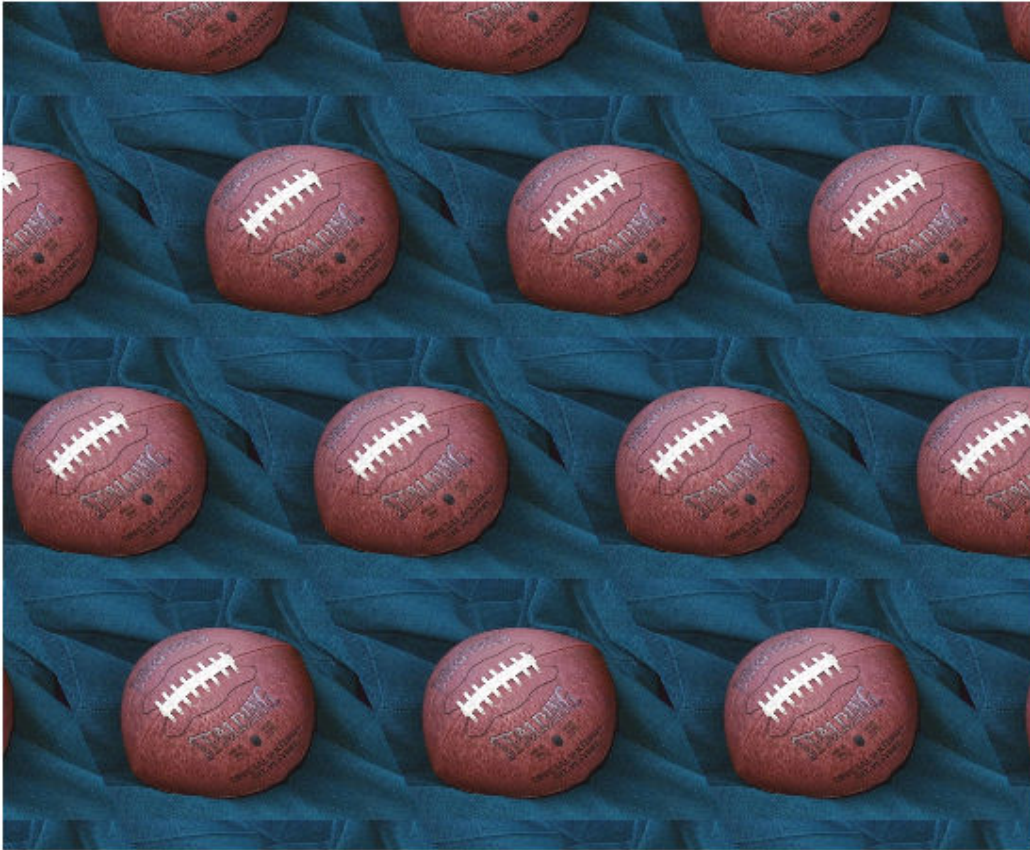

```matlab
R = makeresampler({'cubic','nearest'},'symmetric');
Bs = imtransform(A,Thalf,R,'XData',[-49 500],'YData',[-49 400],...
                  'FillValues',orange);
figure, imshow(Bs);
title('Pad Method = ''symmetric''');
```

Pad Method = 'symmetric'

7
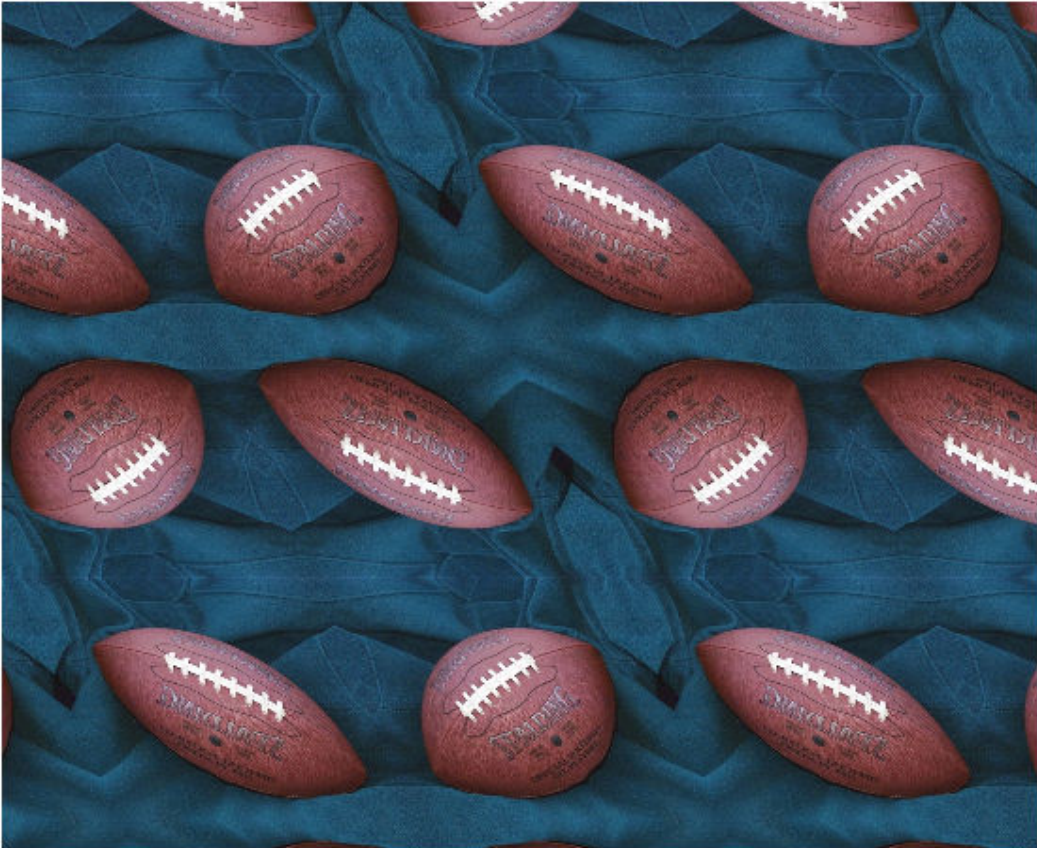
Image Registration

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

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 Toolbox offers automated feature detection and matching.

Capability	"Registration Estimator App" on page 7-2	"Intensity-Based Automatic Image Registration" on page 7-3	"Control Point Registration" on page 7-4	"Automated Feature Detection and Matching" on page 7-5 (requires Computer Vision Toolbox)
Interactive registration	X			
Automated intensity-based registration	X	X		
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 7-22.
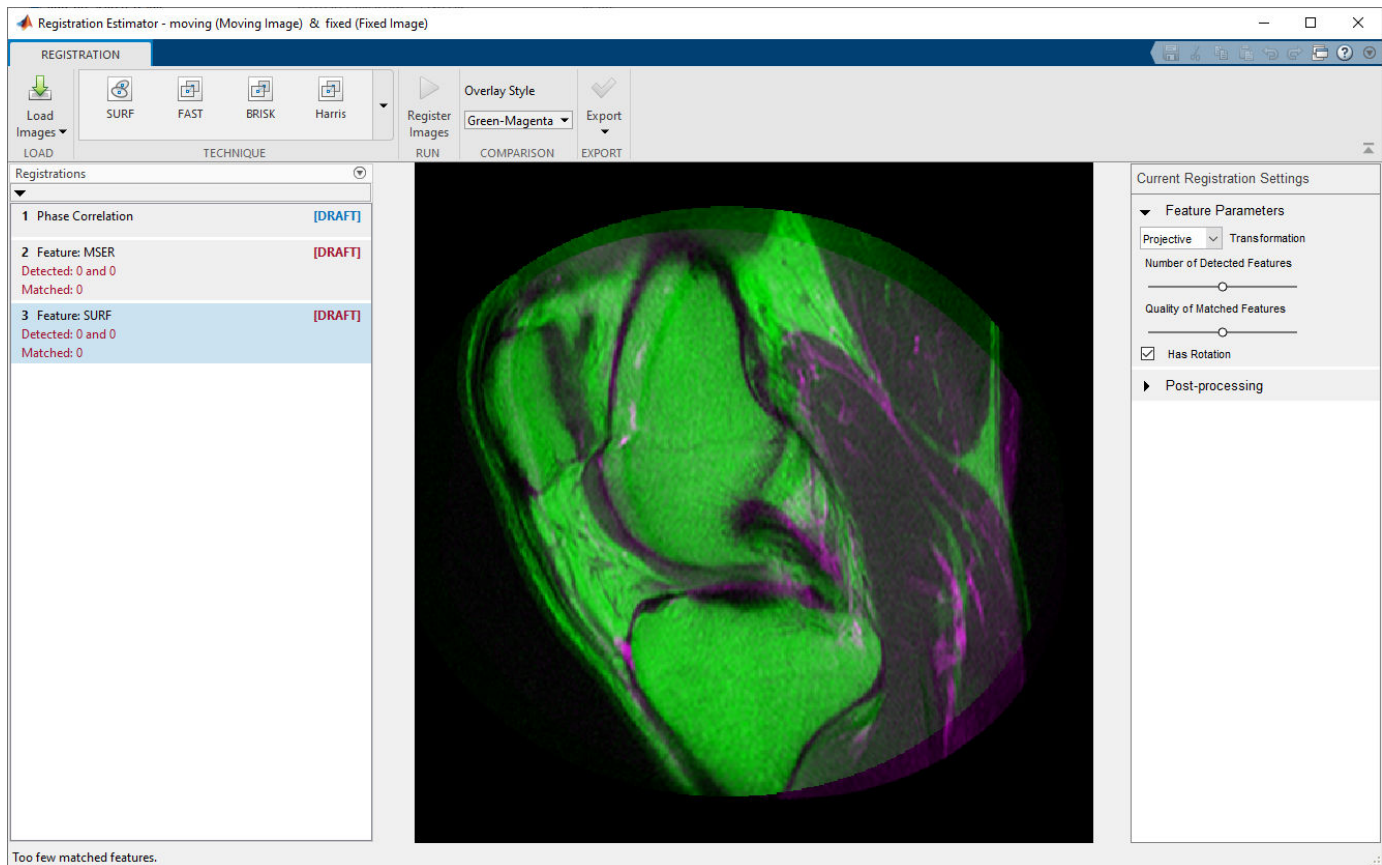
Intensity-Based Automatic Image Registration

"Intensity-Based Automatic Image Registration" on page 7-24 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 7-56 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`.
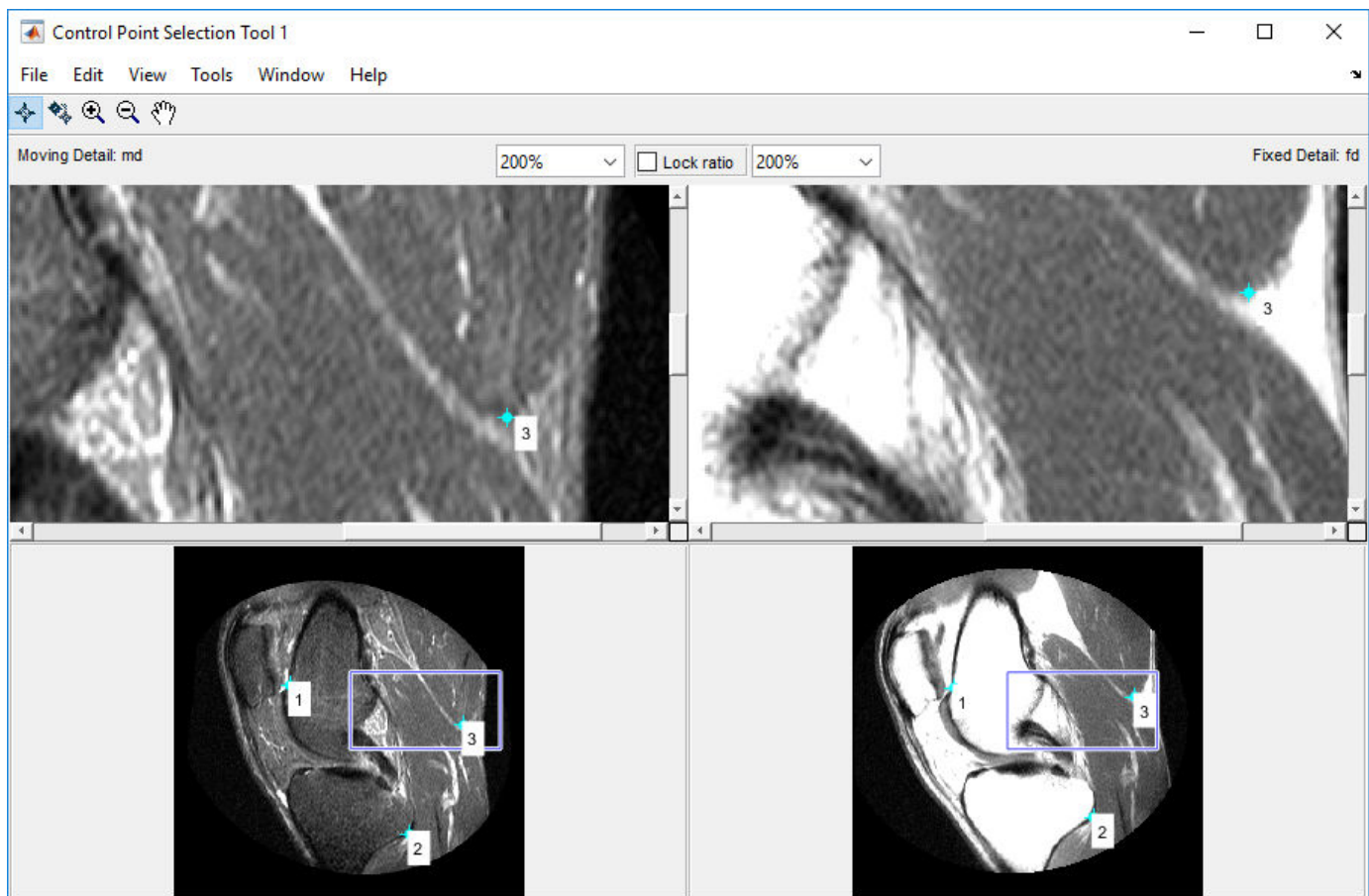
Automated Feature Detection and Matching

Automated "Feature Detection and Extraction" (Computer Vision 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" (Computer Vision Toolbox). You must have Computer Vision 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 Registration Estimator App" on page 7-6
- "Register Multimodal MRI Images" on page 7-32
- "Register Images with Projection Distortion Using Control Points" on page 7-80

Register Images Using Registration Estimator App

This example shows how to align a pair of images using the Registration Estimator app. **Registration Estimator** offers several registration techniques using feature-based, intensity-based, and nonrigid registration algorithms. For more information, see "Techniques Supported by Registration Estimator App" on page 7-22.

Create two misaligned images in the workspace. This example creates the moving image J by rotating the fixed image I clockwise by 30 degrees.

```
I = imread('cameraman.tif');
J = imrotate(I,-30);
```

Open Registration Estimator

In this example, you can open **Registration Estimator** from the command window because the images have no spatial referencing information or initial transformation estimate. Run this command in the command window, and specify the moving image and the fixed image as the two input arguments.

```
registrationEstimator(J,I)
```

If your images have spatial referencing information, or if you want to specify an initial transformation estimate, then you must load the images using a dialog window. For more information, see"Load Images, Spatial Referencing Information, and Initial Transformation" on page 7-14.

You can also open **Registration Estimator** from the MATLAB™ Toolstrip. Open the Apps tab and click **Registration Estimator** under Image Processing and Computer Vision. If you open the app from the toolstrip, you must load the images using a dialog window.

Obtain Initial Registration Estimate

After you load the images, the app displays an overlay of the images and creates three registration trials: Phase Correlation, Feature: MSER, and Feature: SURF. These trials appear as drafts in the history list. You can click on each trial to adjust the registration settings. To create a trial for a different registration technique, select a technique from the **Technique** menu.

The default Green-Magenta overlay style 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. Additional overlay styles assist with visualizing the results of the registration. When you click a feature-based technique in the history list, the image overlay displays a set of red and green dots connected by yellow lines. These points are the matched features used to align the images.
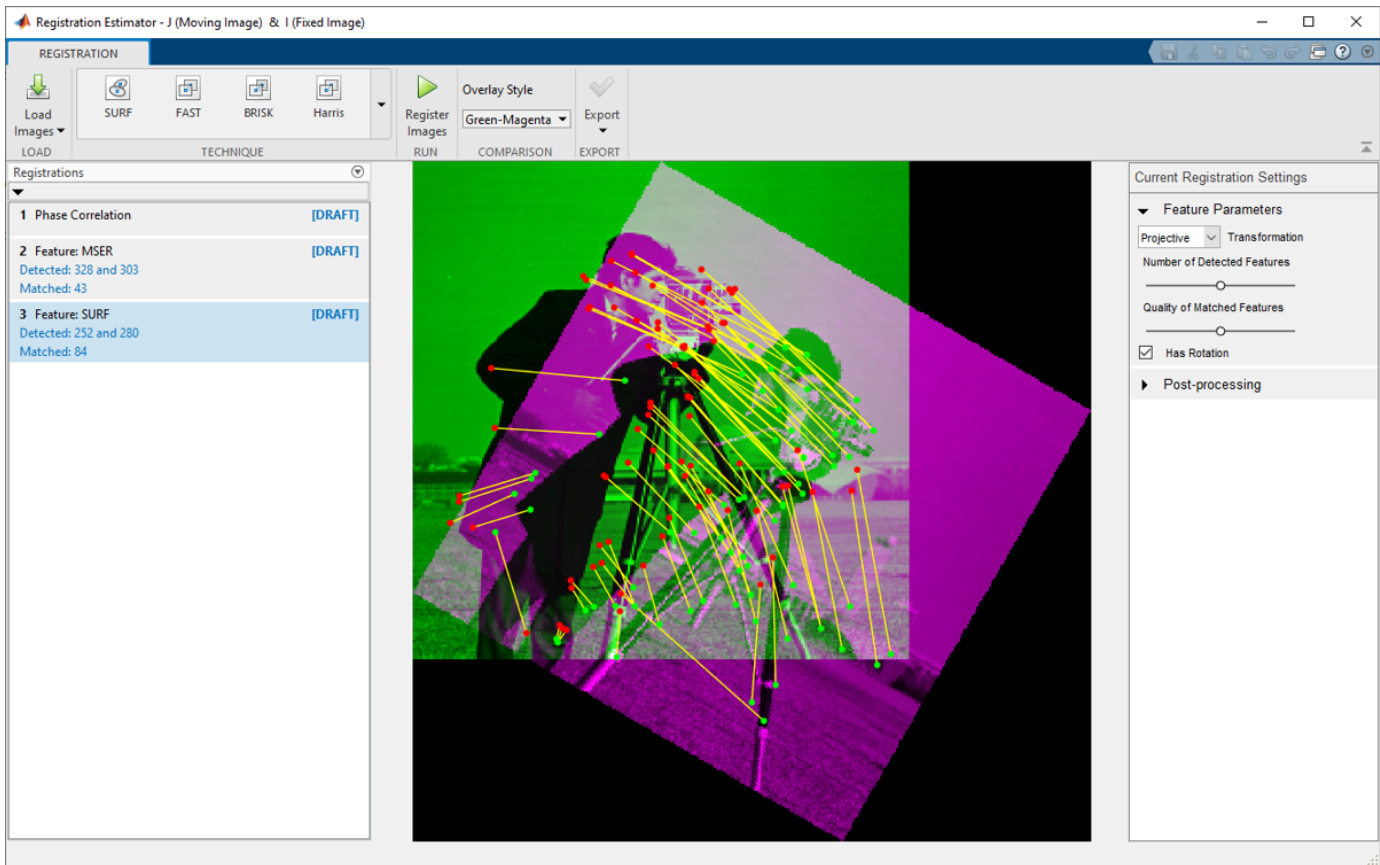
Run the three default registration trials with the default settings. Click each trial in the history list, then click **Register Images**.

After the registration finishes, the trial displays a quality score and computation time. The quality score is based loosely on the `ssim` function and provides an overall estimate of registration quality. A score closer to 1 indicates a higher quality registration. Different registration techniques and settings can yield similar quality scores but show error in different regions of the image. Inspect the image overlay to confirm which registration technique is the most acceptable. Colors in the image overlay indicate residual misalignment.

Note: due to randomness in the registration optimizer, the quality score, registered image, and geometric transformation can vary slightly between trials despite identical registration settings.
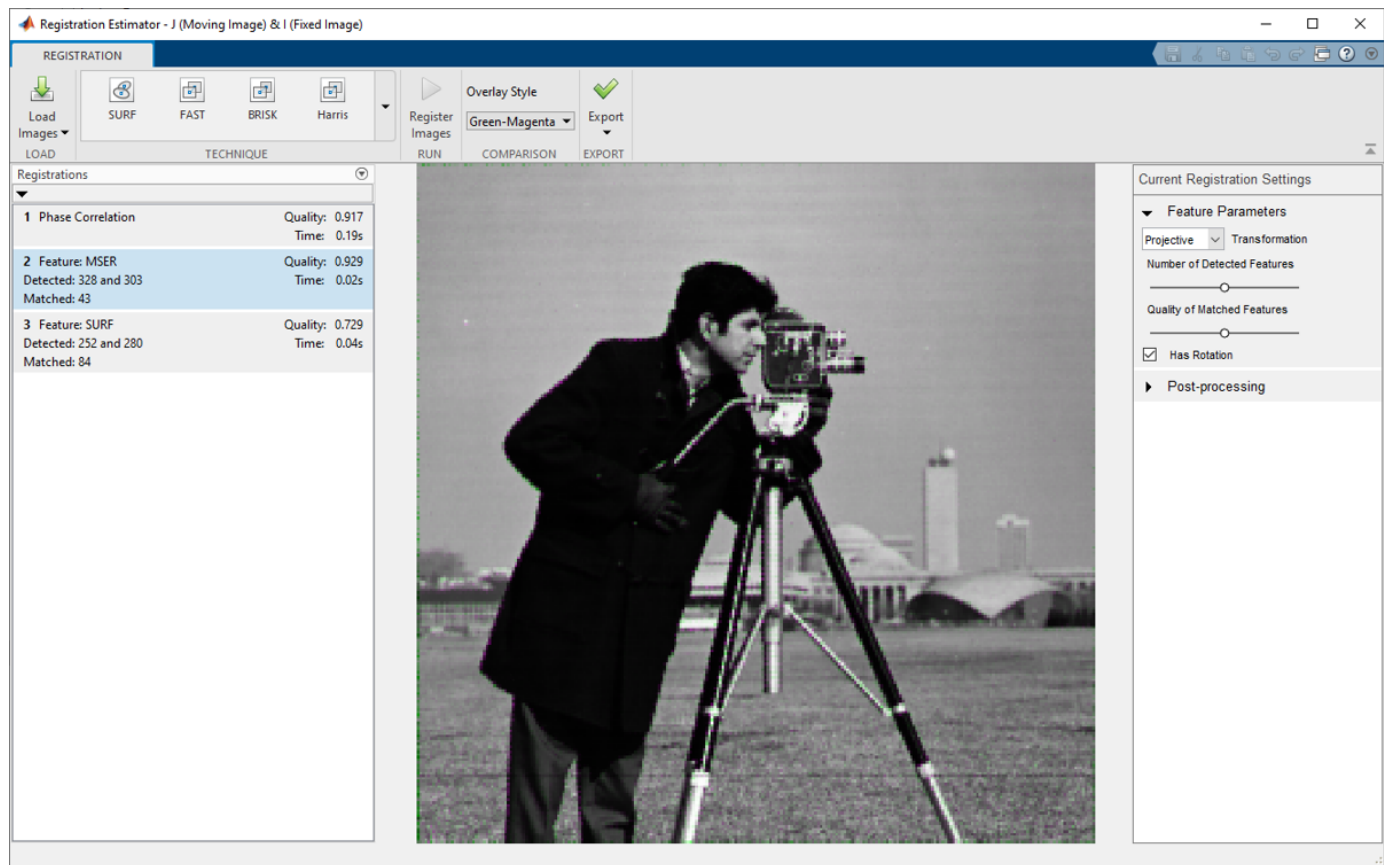
Refine Registration Settings

After 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 7-17. 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.

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 Current Registration Settings panel, drag the **Number of Detected Features** slider to the right. When you change the setting, the app creates a new trial, numbered 2.1, in the history list. The image overlay shows more matched features, as expected.
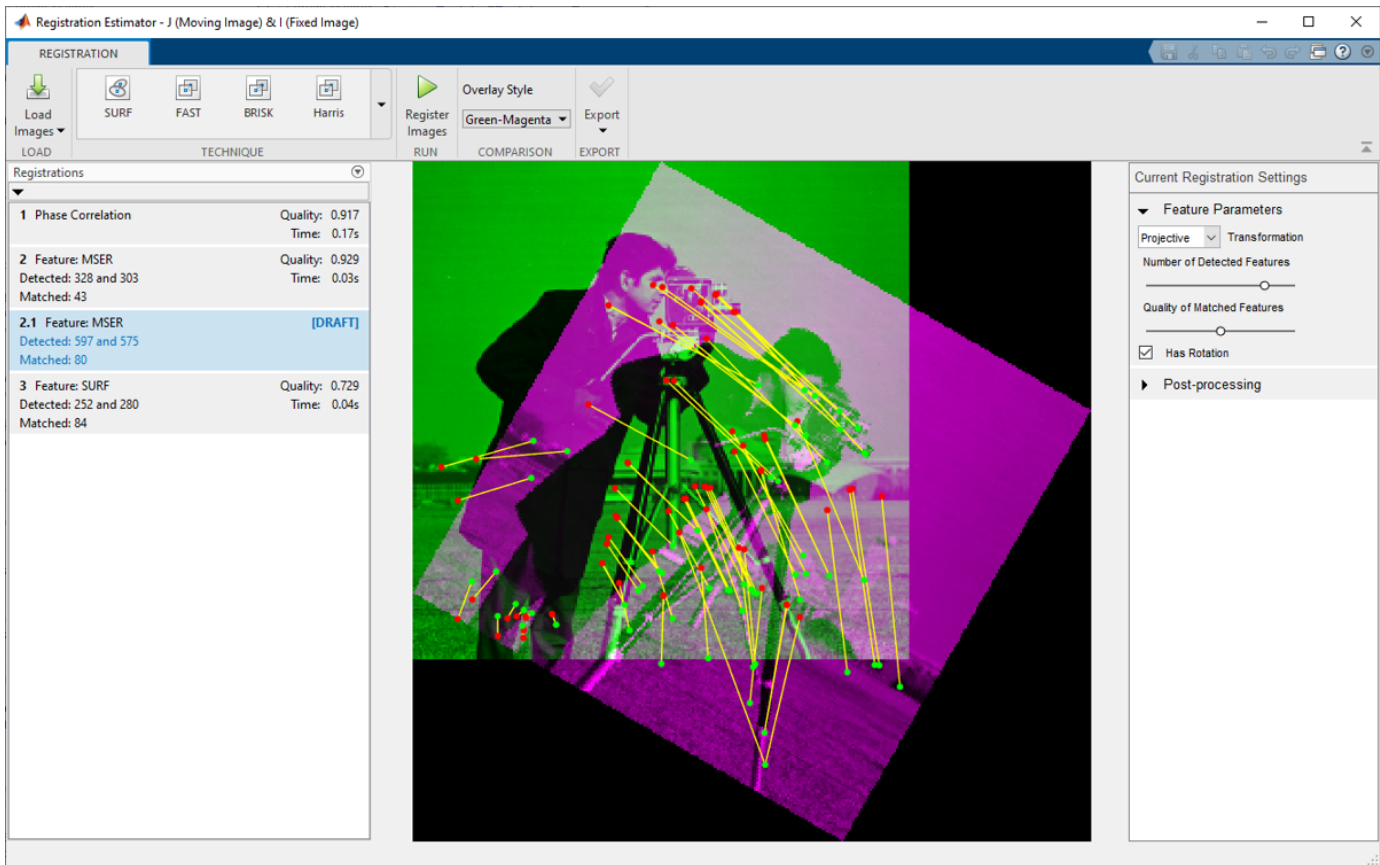
To run the registration with these settings, click **Register Images**. The quality metric of this trial is less than the quality of the original MSER trial with the default number of matched features. The image overlay of this trial has an overall magenta tint and a thick green strip along the top of the man's head. Therefore, increasing the number of detected features from 43 to 80 does not improved the quality of the registration.
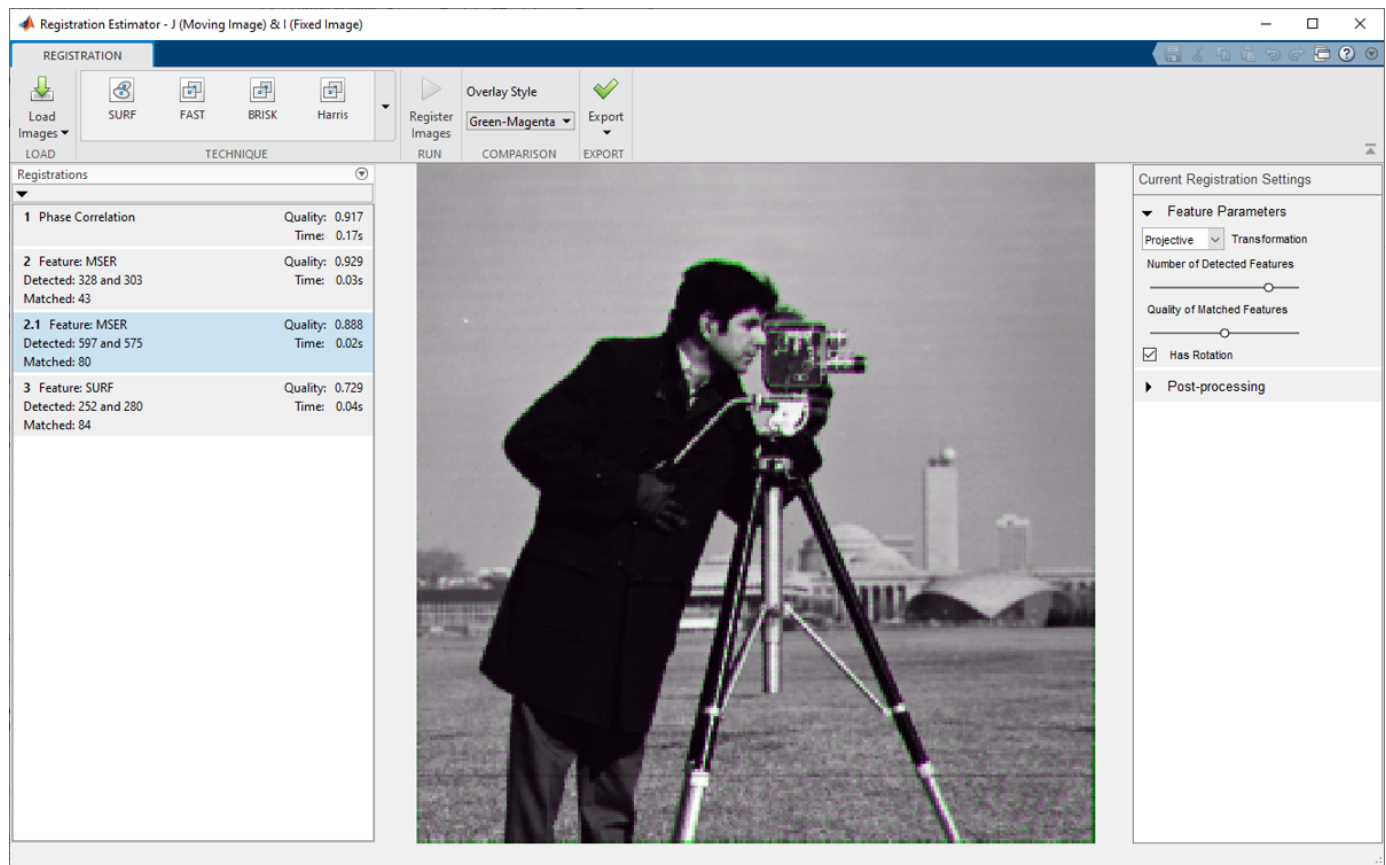
To see the effect of increasing the quality of matched features, click the `Feature: MSER` trial 2 (not 2.1) in the history list. In the **Current Registration Settings** panel, drag the **Quality of Matched Features** slider to the right. When you change the setting, the app creates a new trial, numbered 2.2, in the history list. The image overlay displays a smaller number of high quality matched points.
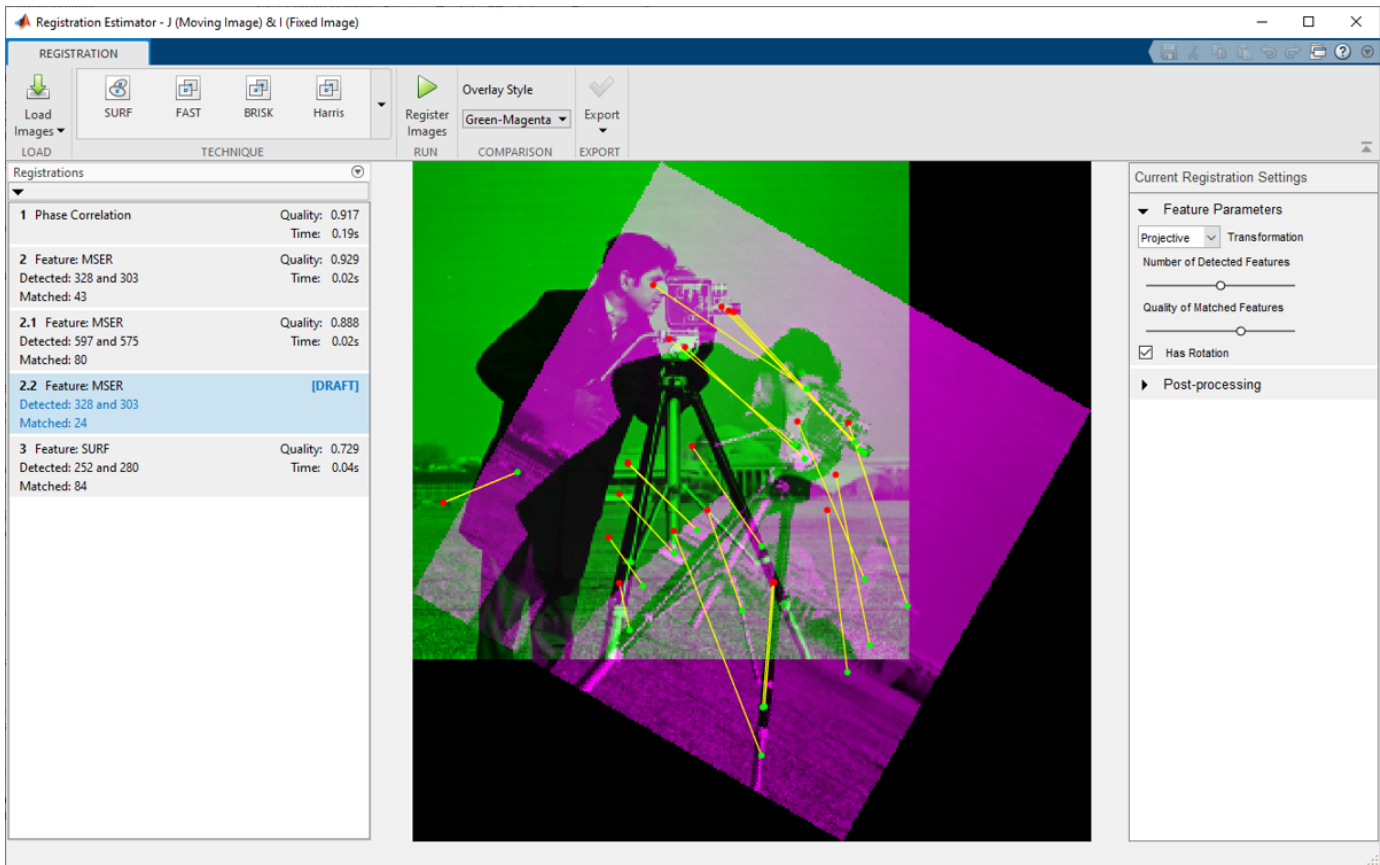
To see the registration with these settings, click **Register Images**. Compared to the other MSER trials, this trial has the best quality score. There is not a noticeable difference in the visual quality of the image compared to the original MSER trial with default settings. If you want to see which pixels differ between the default MSER trial and this trial, change the overlay style to Difference and toggle between the two trials.

Export Registration Results
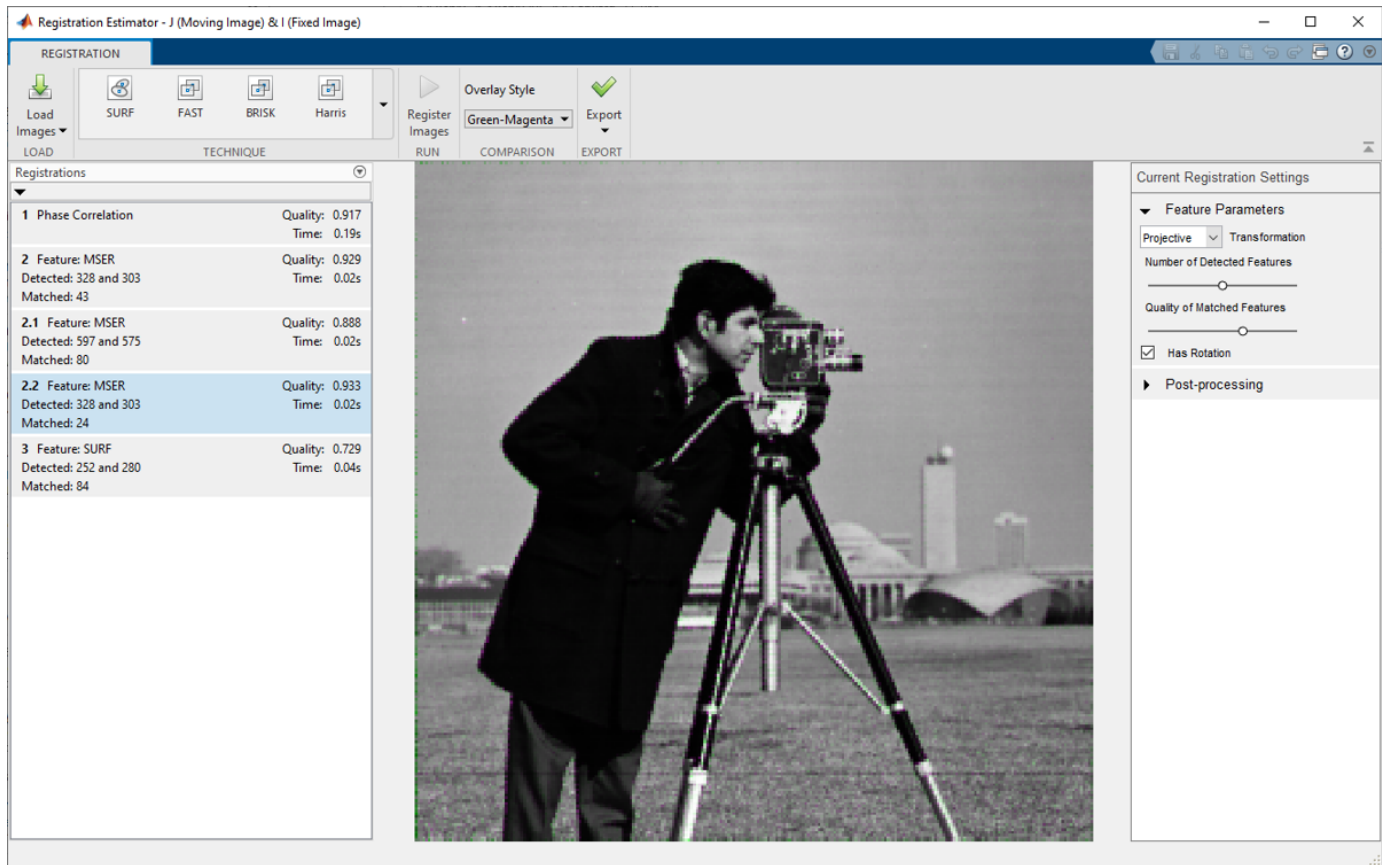
When you find an acceptable registration, export the registered image and the geometric transformation to the workspace. You can use the registration results to apply a similar registration to multiple frames in an image sequence. To learn more, see "Export Results from Registration Estimator App" on page 7-20.

This example exports trial 2.2 because it has the best quality score and no severe regions of misalignment. Click trial 2.2 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.
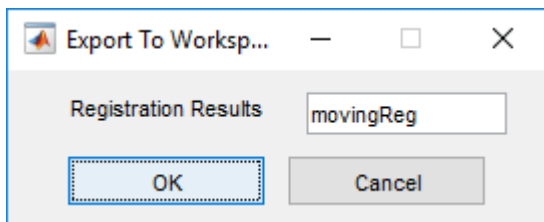
See Also
Registration Estimator

More About

- *"Approaches to Registering Images"* on page 7-2
- *"Techniques Supported by Registration Estimator App"* on page 7-22

Load Images, Spatial Referencing Information, and Initial Transformation

You can load images into **Registration Estimator** from file or from the workspace. You can also provide optional spatial referencing information and an optional initial geometric transformation.

Load Images from File or Workspace

You can load images into **Registration Estimator** from file or from the workspace. Although you can load grayscale or color images, the app converts all RGB images to grayscale by using the `rgb2gray` function. **Registration Estimator** supports only 2-D images.

Loading images from file supports only BMP, JPG, JPEG, TIF, TIFF, PNG, and DCM file types. To work with a wider range of file formats, load images from the workspace. **Registration Estimator** supports any image read into the workspace by the `imread` function and DICOM image read into the workspace using the `dicomread` function.

Load images into the app by clicking the **Load Images** icon.

- To load images from a file, select the `Load from file` option. In the dialog box, select the file path of the moving and fixed images.
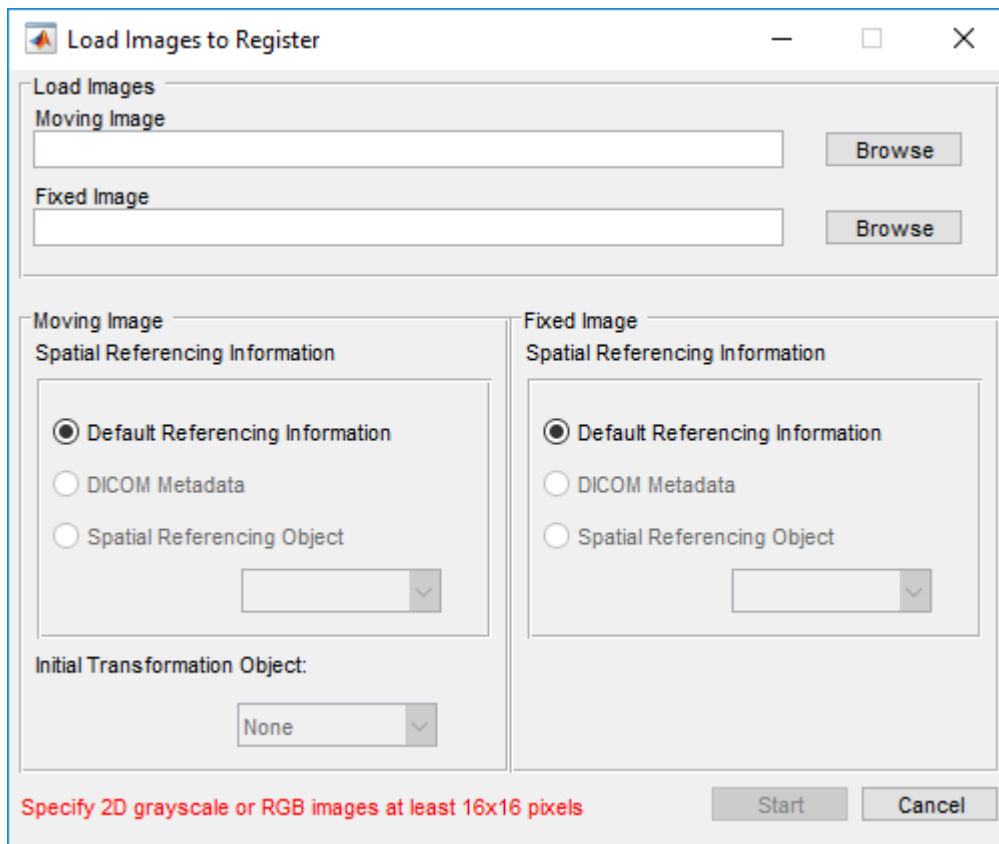
- To load variable from the workspace, select the `Load from workspace` option. In the dialog box, select two images from your workspace.
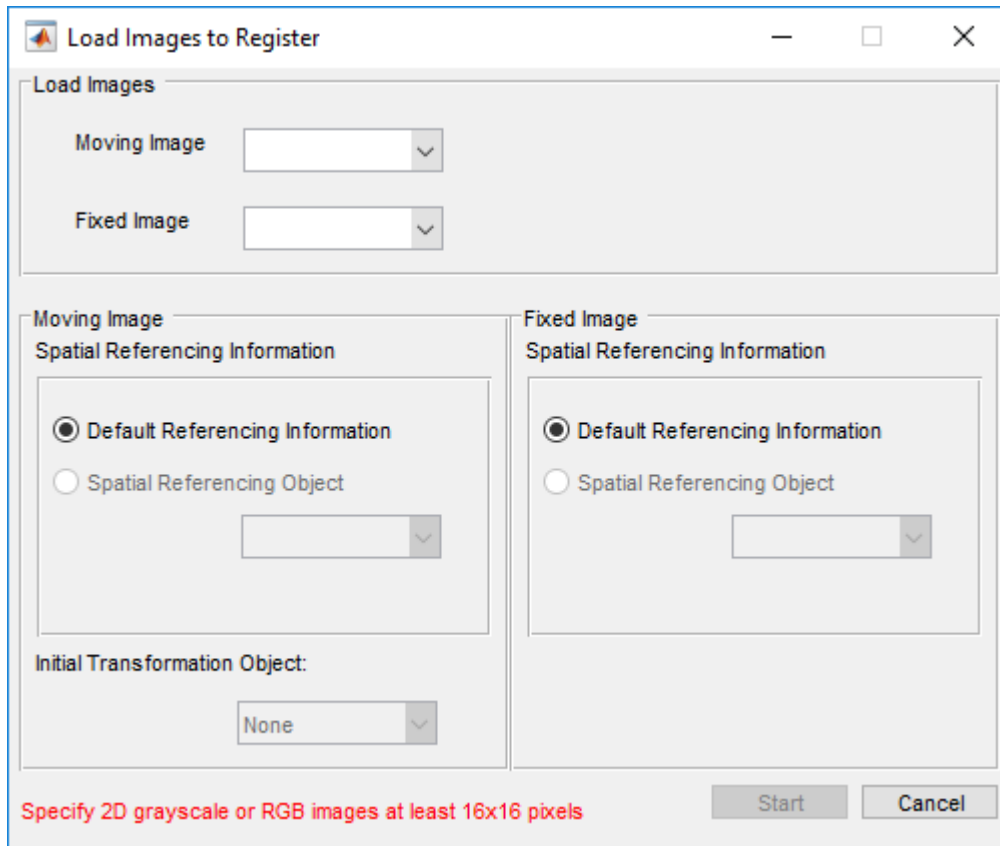
Provide Spatial Referencing Information

If you have 2-D spatial referencing objects in your workspace, or if you load DICOM images from a file, then 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 3-8.

If you do not have spatial referencing information, then 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 Results from Registration Estimator App" on page 7-20.

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 Registration Estimator App" on page 7-6

Tune Registration Settings in Registration Estimator App

Adjust settings in **Registration Estimator** 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.

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 6-17.

- *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 Registration Estimator App" on page 7-6

More About

- "Techniques Supported by Registration Estimator App" on page 7-22
- "Matrix Representation of Geometric Transformations" on page 6-17

Export Results from Registration Estimator App

When you find an acceptable registration from **Registration Estimator**, 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.
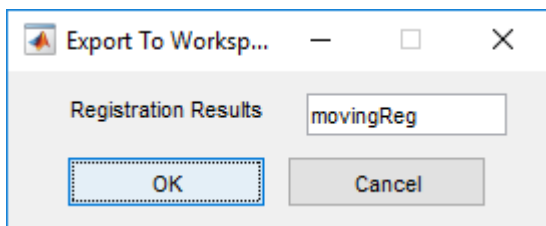
Generate a Function

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, then you must have Computer Vision Toolbox to run the function.

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. If you generate a function using a feature-based registration technique, then the output structure has two additional fields for the moving matched features and the fixed matched features.

See Also
`imwarp`

Related Examples
- "Register Images Using Registration Estimator App" on page 7-6

More About

- "2-D and 3-D Geometric Transformation Process Overview" on page 6-13

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.

In **Registration Estimator**, you can register images and generate functions for all feature-based techniques without a Computer Vision Toolbox license. However, to run an autogenerated function that uses a feature-based registration technique, you must have Computer Vision Toolbox. For more information, see "Export Results from Registration Estimator App" on page 7-20.

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 Registration Estimator App" on page 7-6
- "Approaches to Registering Images" on page 7-2

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.
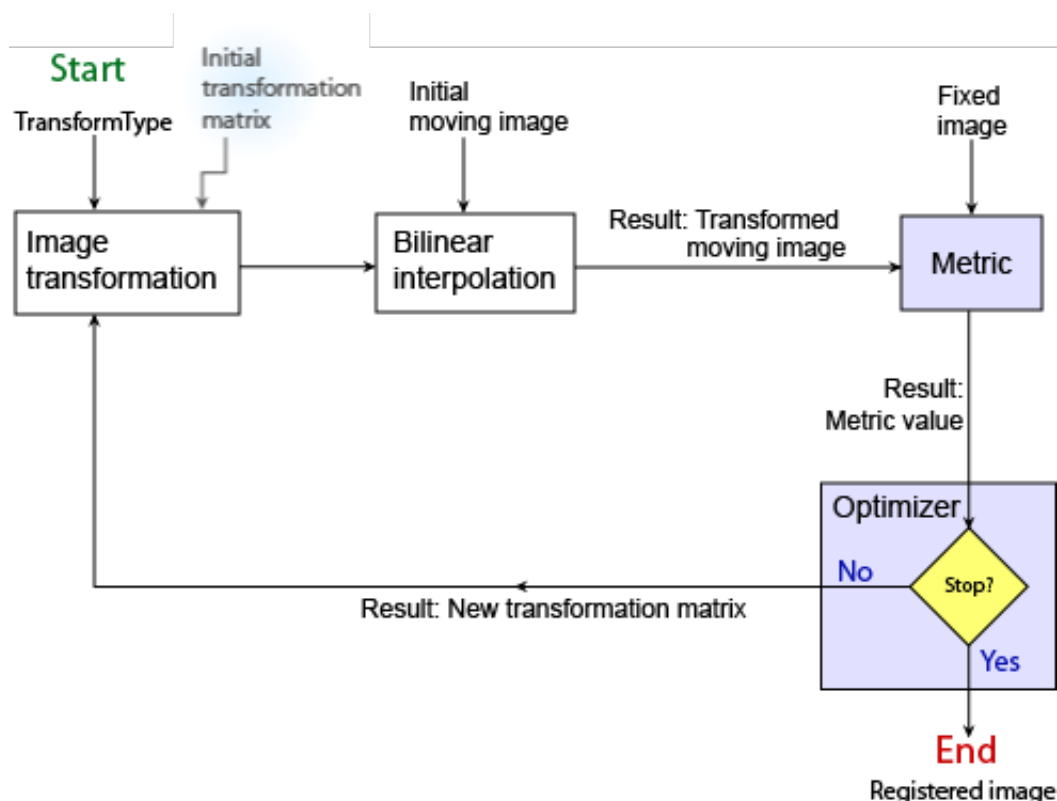
Perform intensity-based image registration with the following steps:

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 7-26.

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 7-32

More About

- "Use Phase Correlation as Preprocessing Step in Registration" on page 7-27
- "Approaches to Registering Images" on page 7-2

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 7-32.

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, the approach 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 do not always indicate misregistration, but in this example it is easy to use the color information to see where they do.

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

Unregistered
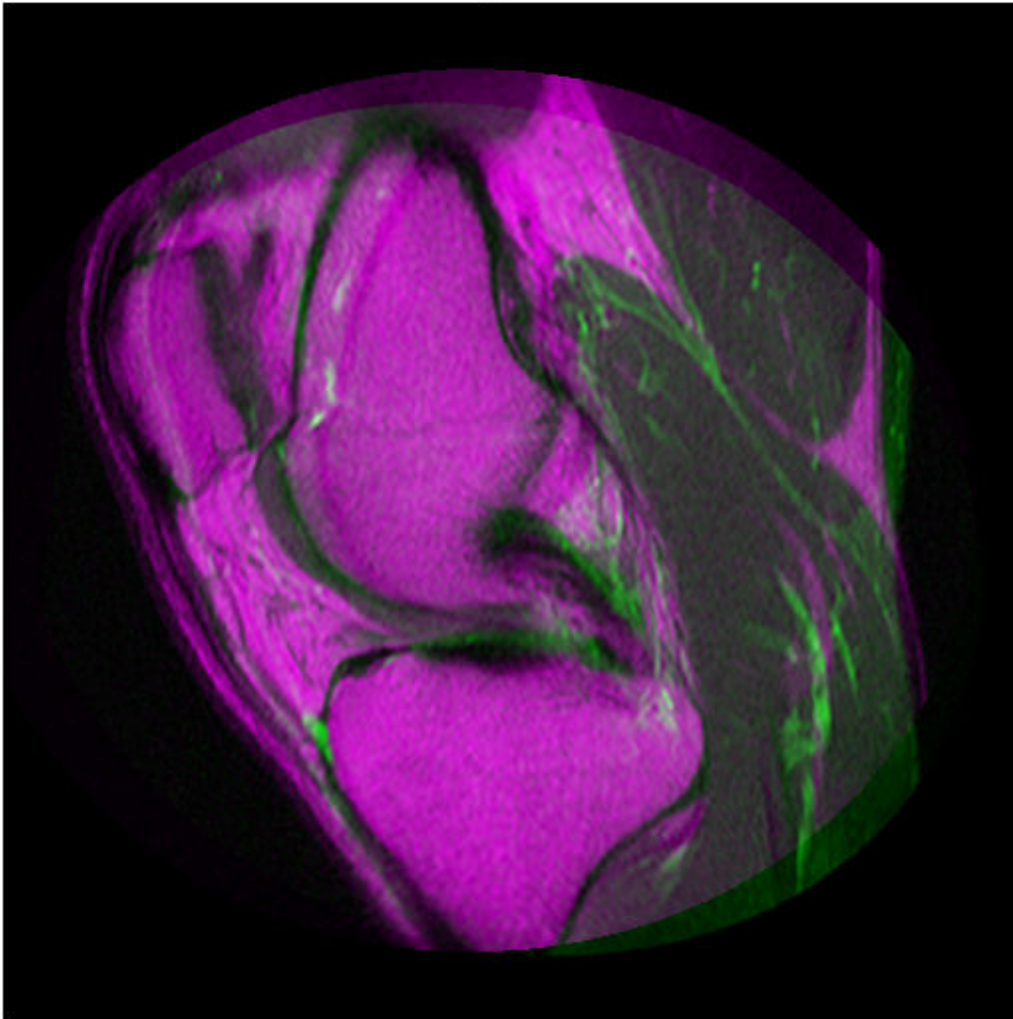
Step 2: Set up the Initial Registration

The `imregconfig` function makes it easy to pick the correct optimizer and metric configuration to use with `imregister`. The optimizer and metric variables are objects whose properties control the registration. For more information, see "Create an Optimizer and Metric for Intensity-Based Image Registration" on page 7-26.

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.

```
movingRegisteredDefault = imregister(moving,fixed,'affine',optimizer,metric);
```

Display the result. It is 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.

```
imshowpair(movingRegisteredDefault,fixed)
title('A: Default Registration')
```
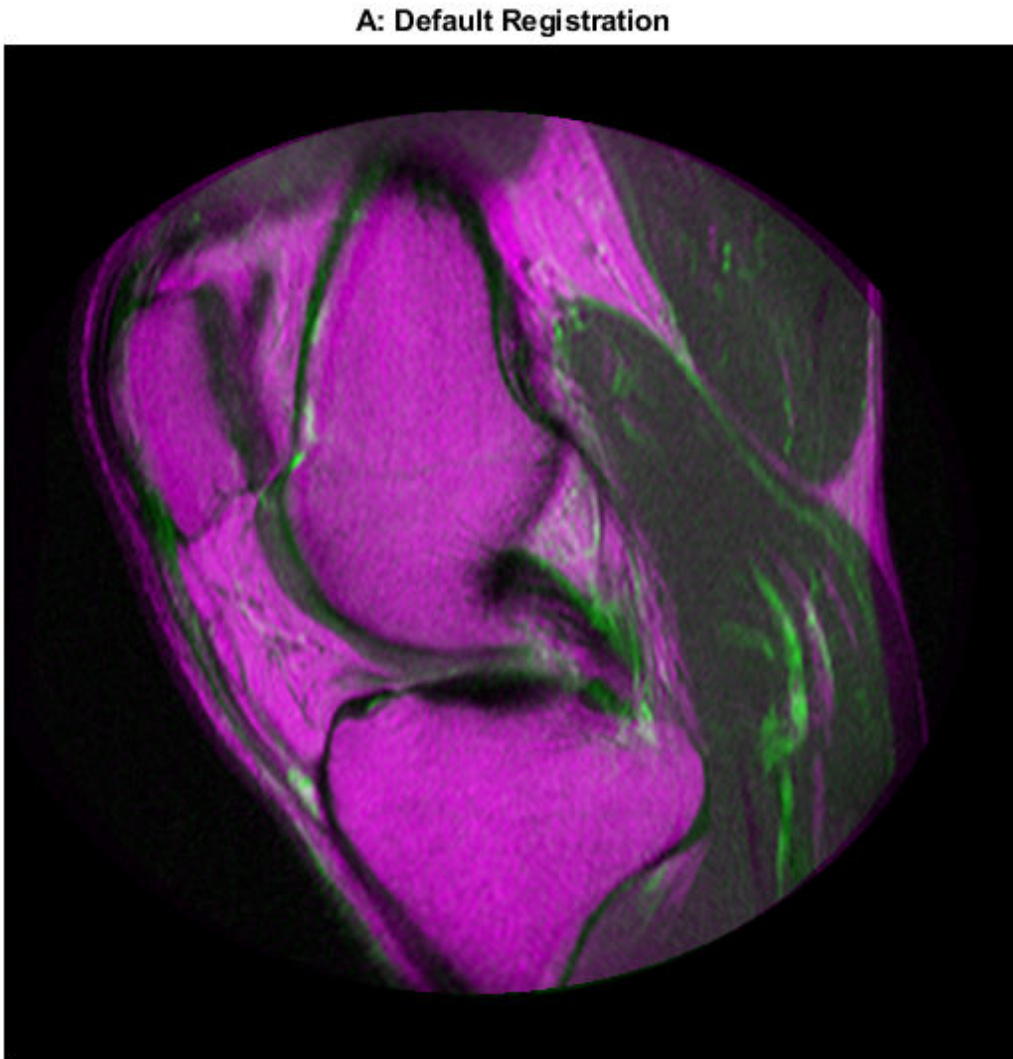

Step 3: Improve the Registration

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.

```
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 multimodal 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,metric);
```

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

```
imshowpair(movingRegisteredAdjustedInitialRadius,fixed)
title('B: Adjusted InitialRadius')
```

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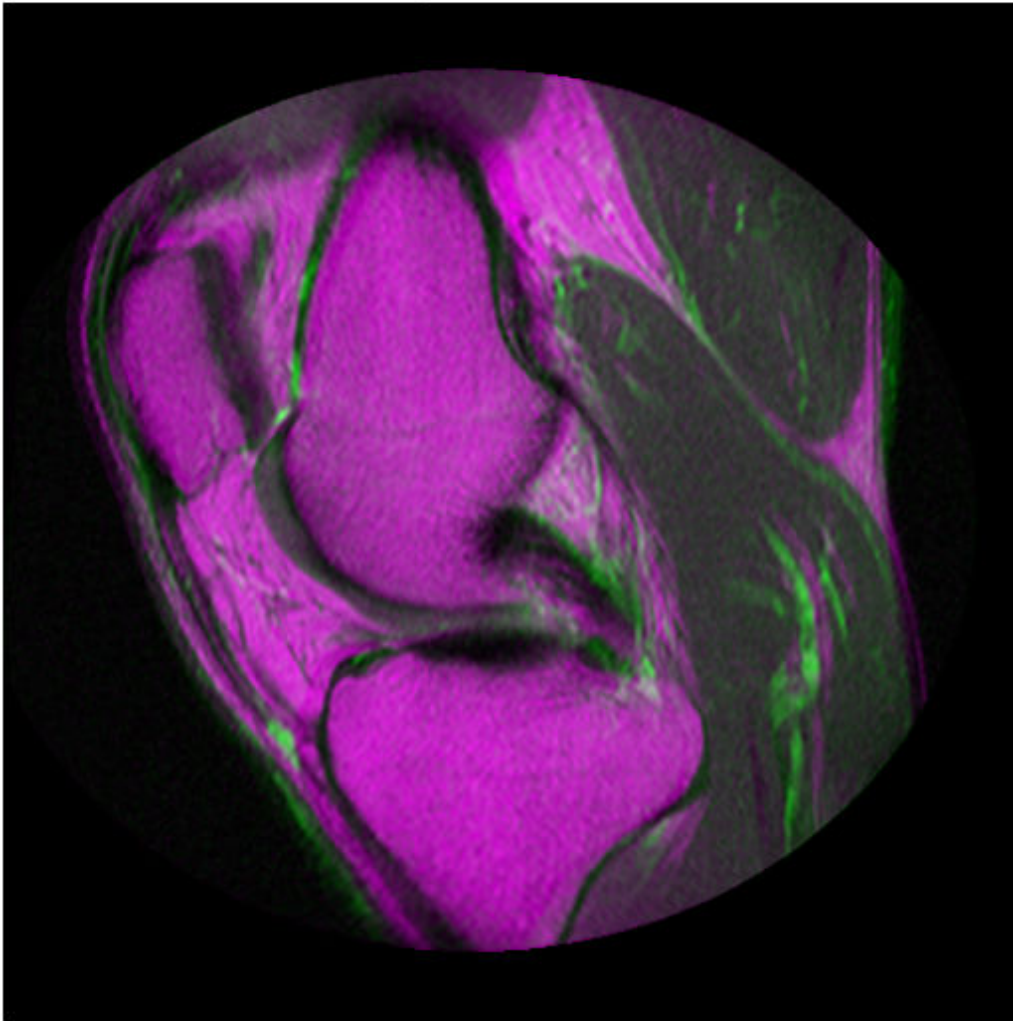
The `MaximumIterations` property of the optimizer controls the maximum number of iterations that the optimizer will be allowed to take. Increasing `MaximumIterations` allows the registration search to run longer and potentially find better registration results. Does the registration continue to improve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

```
optimizer.MaximumIterations = 300;
movingRegisteredAdjustedInitialRadius300 = imregister(moving,fixed,'affine',optimizer,metric);
```

Display the results. 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.

```
imshowpair(movingRegisteredAdjustedInitialRadius300,fixed)
title('C: Adjusted InitialRadius, MaximumIterations = 300')
```

C: Adjusted InitialRadius, MaximumIterations = 300
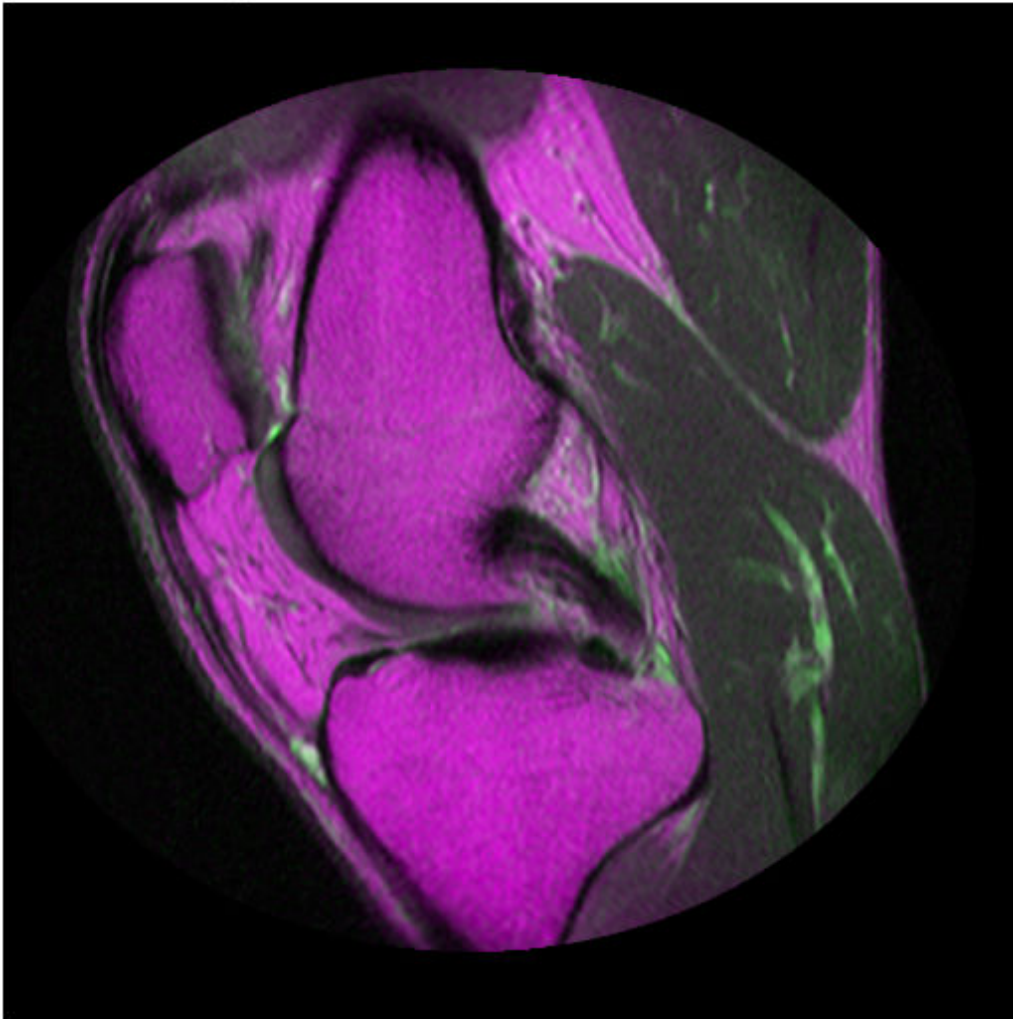
Step 4: Use Initial Conditions to Improve Registration

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);
```

Display the result.

```
imshowpair(movingRegisteredRigid, fixed)
title('D: Registration Based on Similarity Transformation Model')
```

D: Registration Based on Similarity Transformation Model
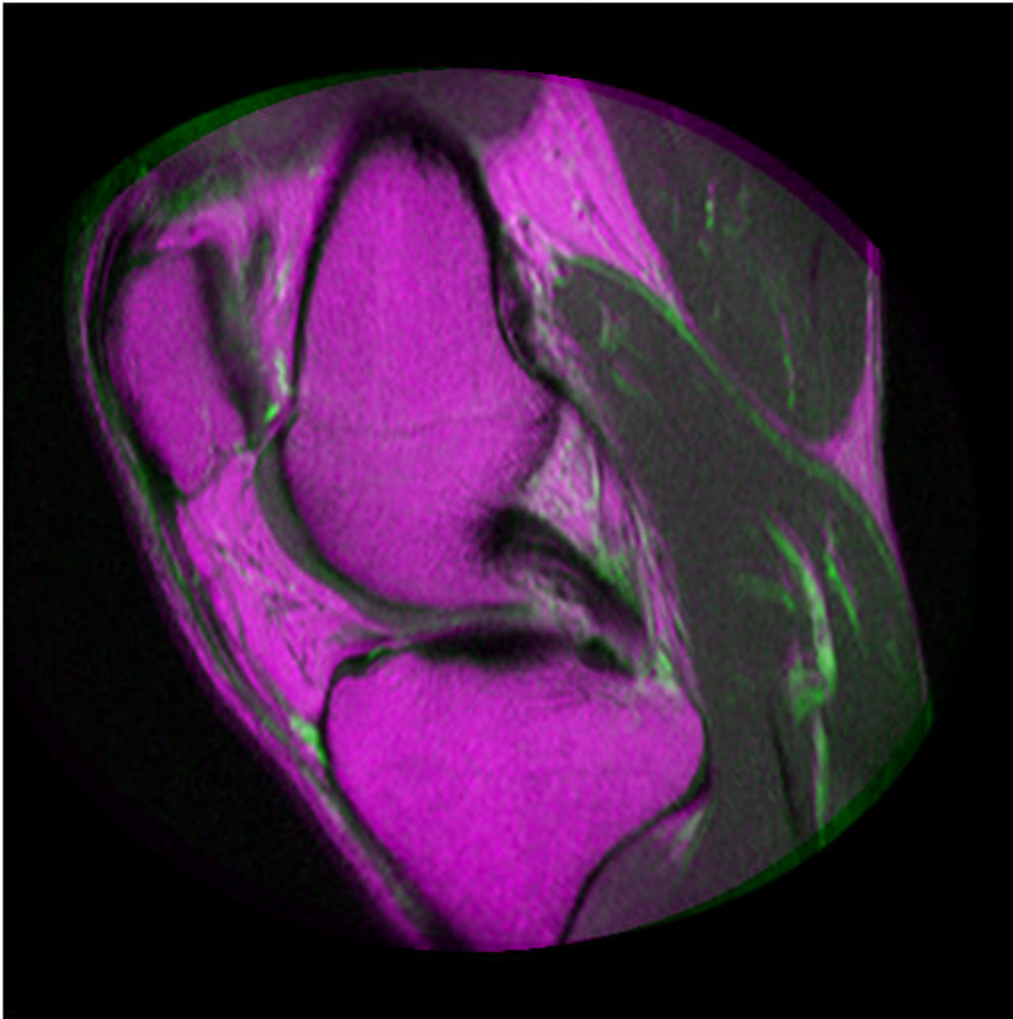
The `'T'` property of the output geometric transformation defines the transformation matrix that maps points in moving to corresponding points in fixed.

```
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);
```

Display the result. Using the `'InitialTransformation'` to refine the `'similarity'` result of `imregtform` with a full affine model yields a nice registration result.

```
imshowpair(movingRegisteredAffineWithIC,fixed)
title('E: Registration from Affine Model Based on Similarity Initial Condition')
```


E: Registration from Affine Model Based on Similarity Initial Condition
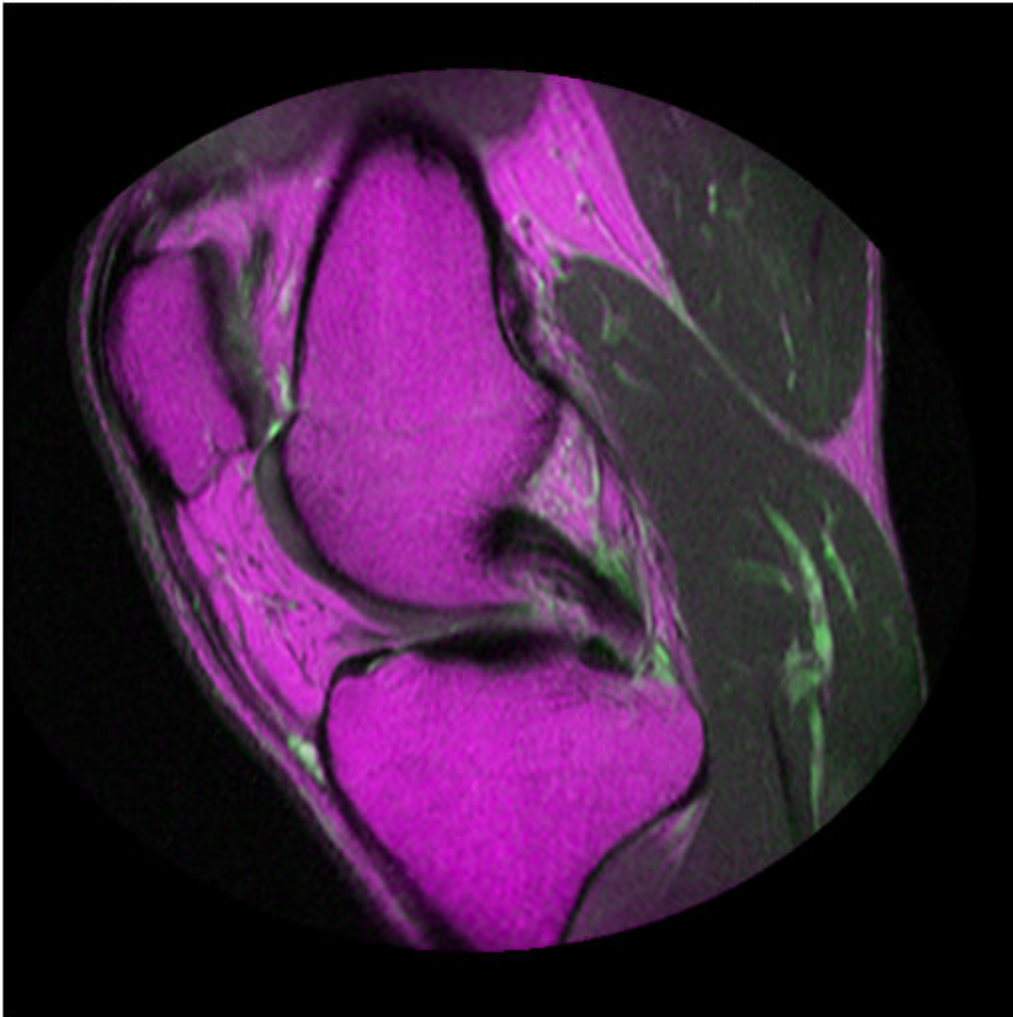
Step 5: Deciding When Enough is Enough

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.Slices],...
                            '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);
```
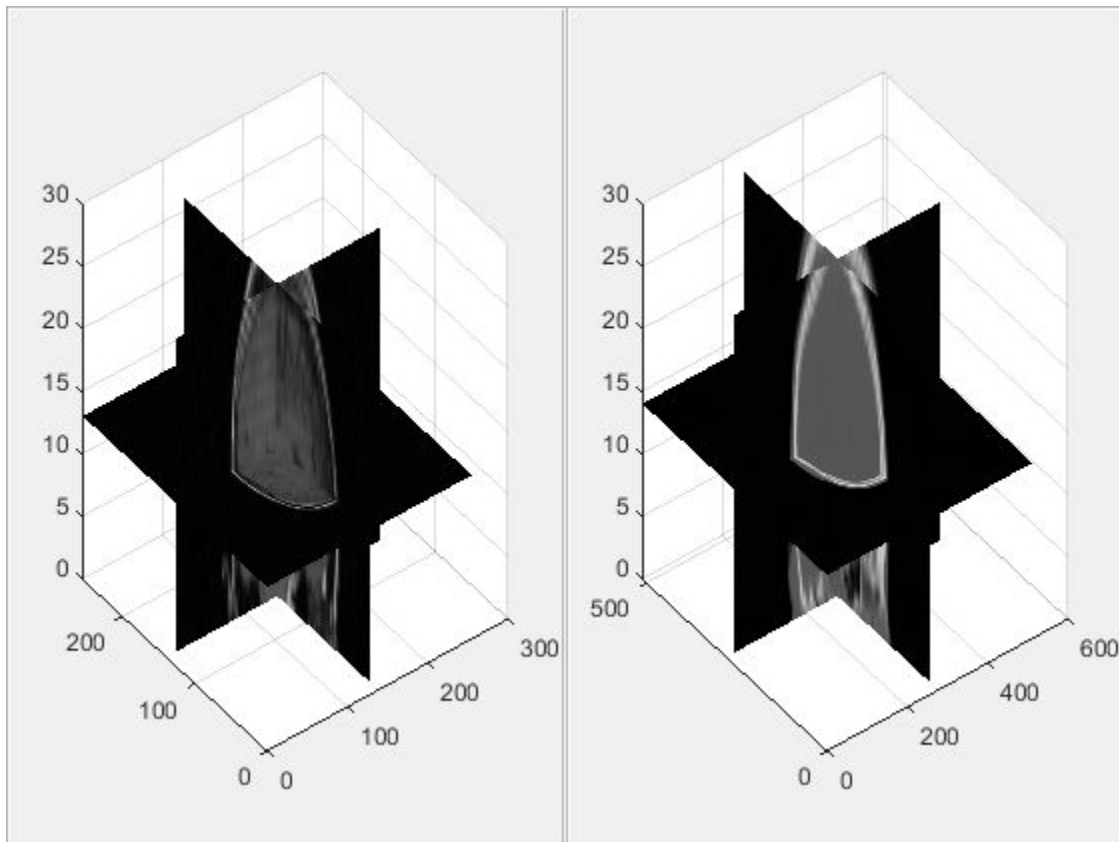
You can also use `imshowpair` to look at single planes from the fixed and moving volumes to get a sense of the overall alignment of the volumes. In the overlapping image from `imshowpair`, gray areas correspond to areas that have similar intensities, while magenta and green areas show places where one 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')
```

Unregistered Axial Slice
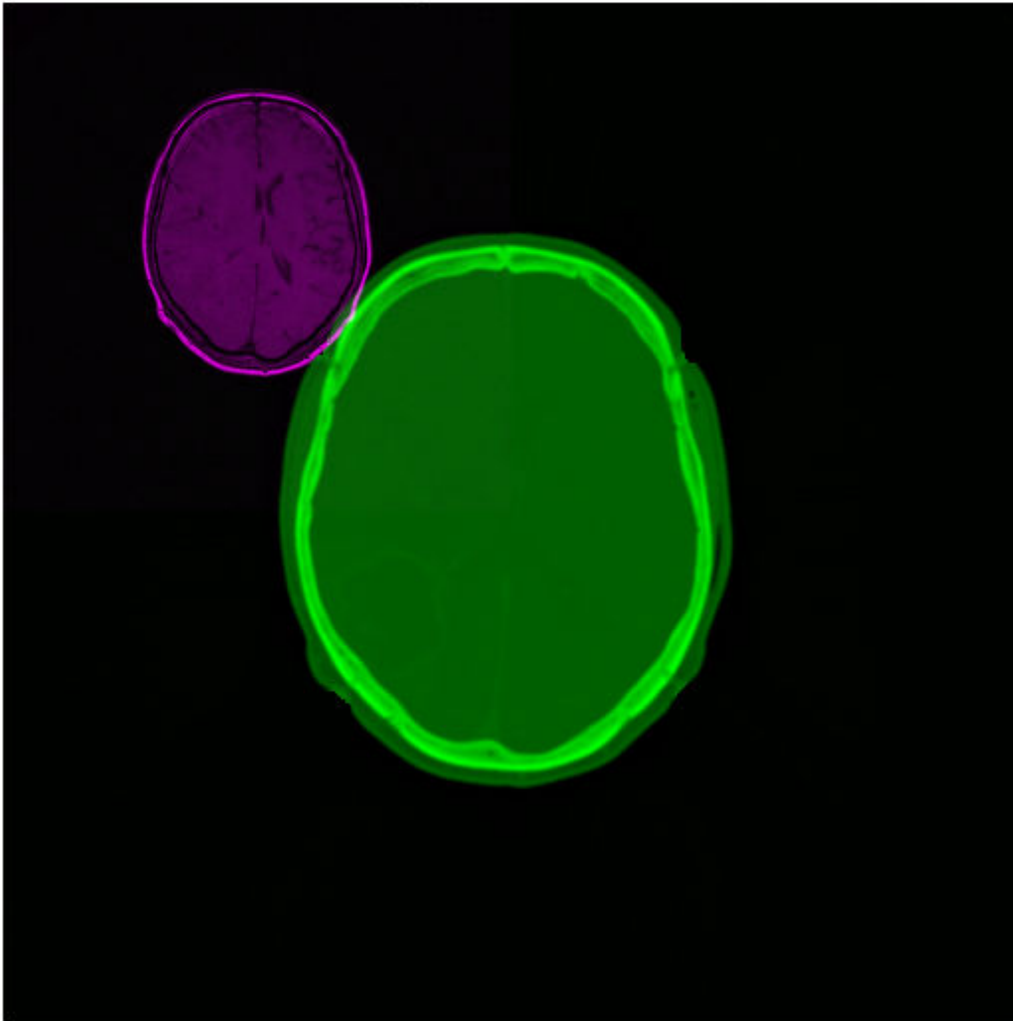
Step 2: Set up the Initial Registration

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),fixedHeader
Rmoving = imref3d(size(movingVolume),movingHeader.PixelSize(2),movingHeader.PixelSize(1),movingHe
```

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', optimizer
```

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')
```

Axial Slice of Registered Volume
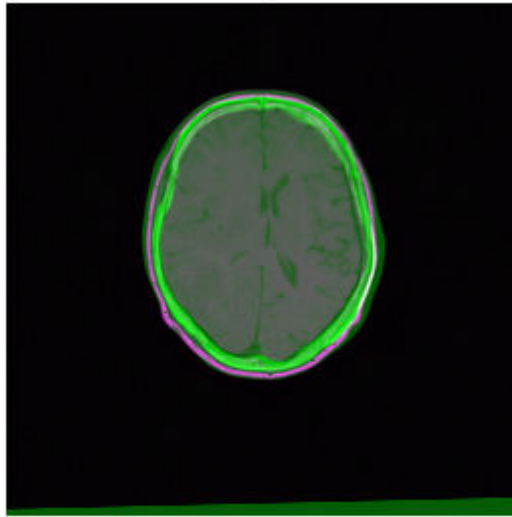
From the axial slice above, it looks like the registration was successful. Use `helperVolumeRegistration` again to view the registered volume to continue judging the success of registration.

```
helperVolumeRegistration(fixedVolume,movingRegisteredVolume);
```
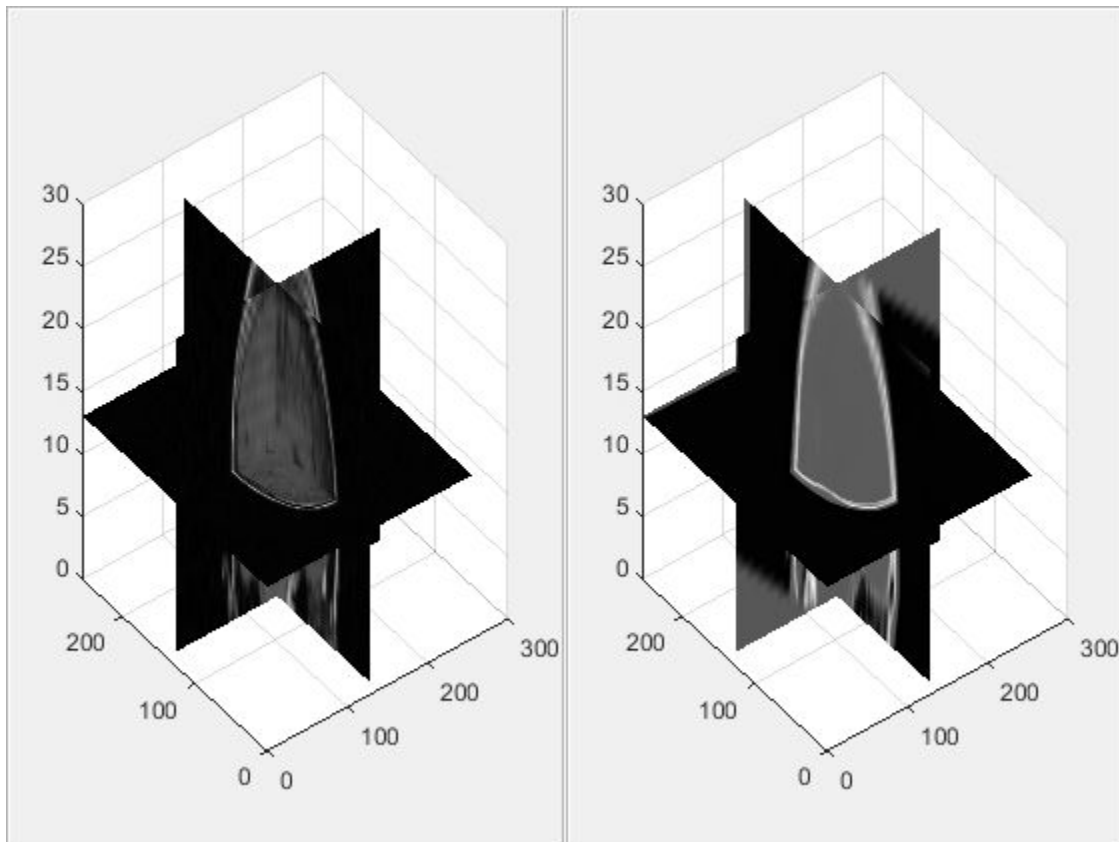
Step 3: Get 3-D Geometric Transformation That Aligns Moving With Fixed.

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, metric)

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,centerZWorld]
```

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',Rfixed);
```

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')
```

Axial Slice of Registered Volume
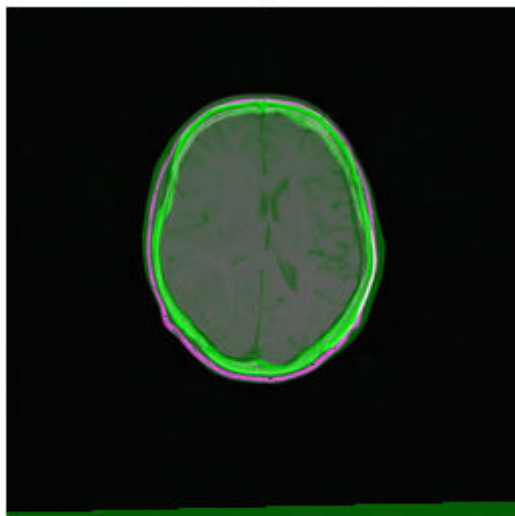
See Also

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
```
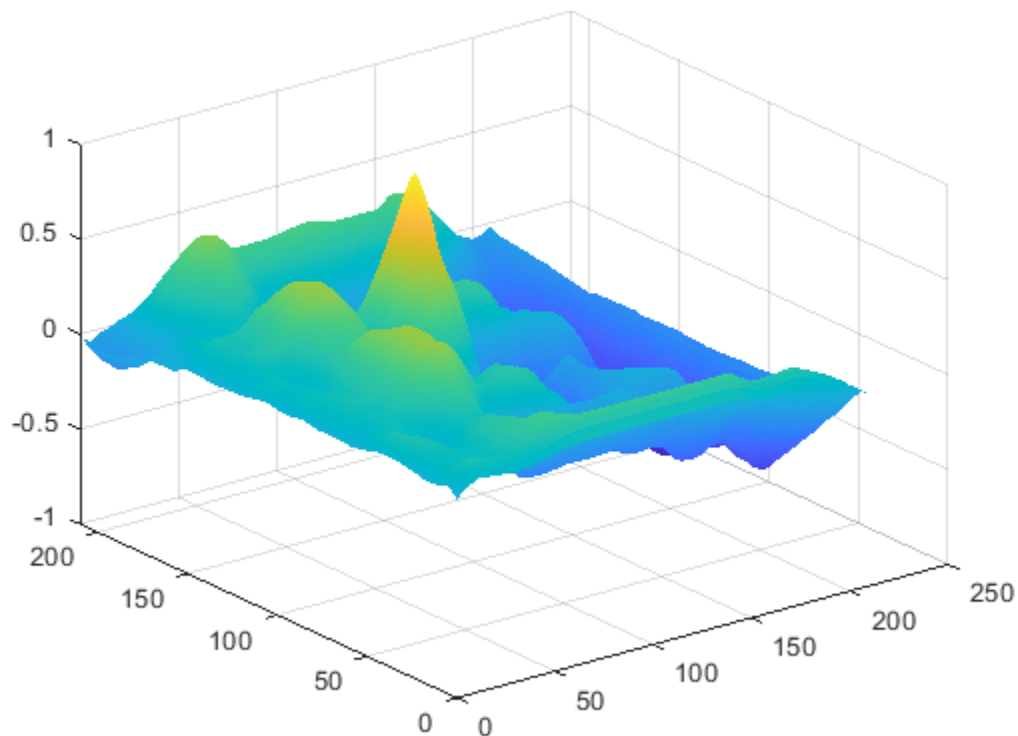
Step 4: Find the Total Offset Between the Images

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)
```
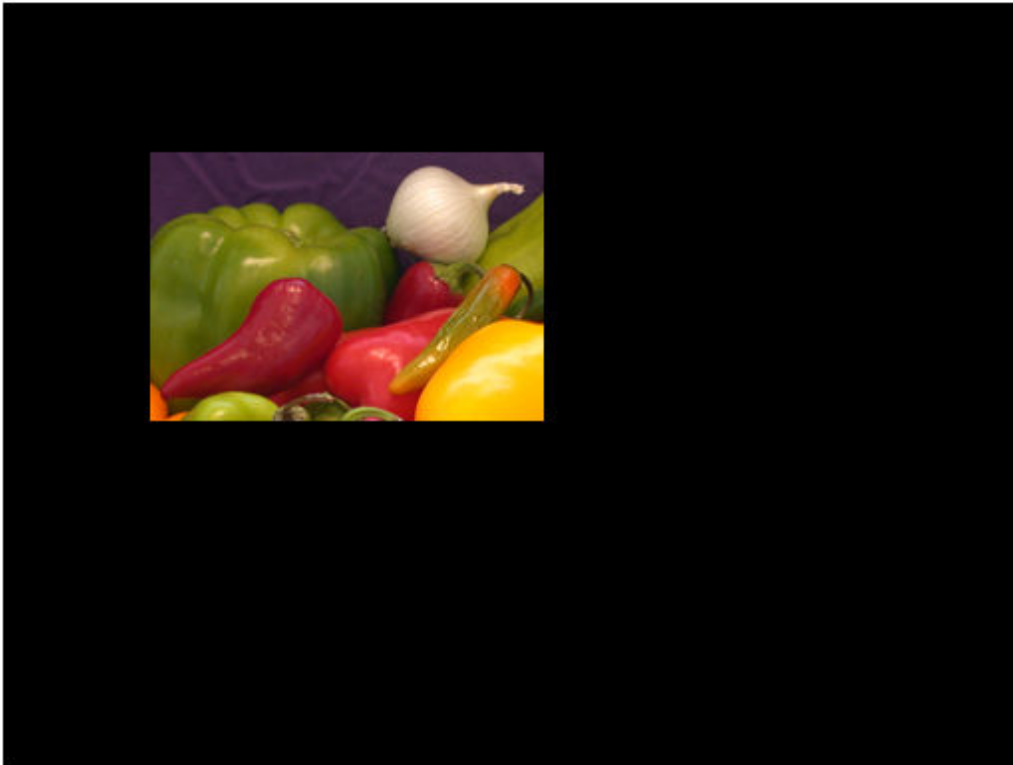

Step 7: Use Alpha Blending To Show Images Together

Display one plane of the `peppers` image with the `recovered_onion` image using alpha blending.

```
figure, imshowpair(peppers(:,:,1),recovered_onion,'blend')
```

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. See "Register Images with Projection Distortion Using Control Points" on page 7-80 for an extended example.
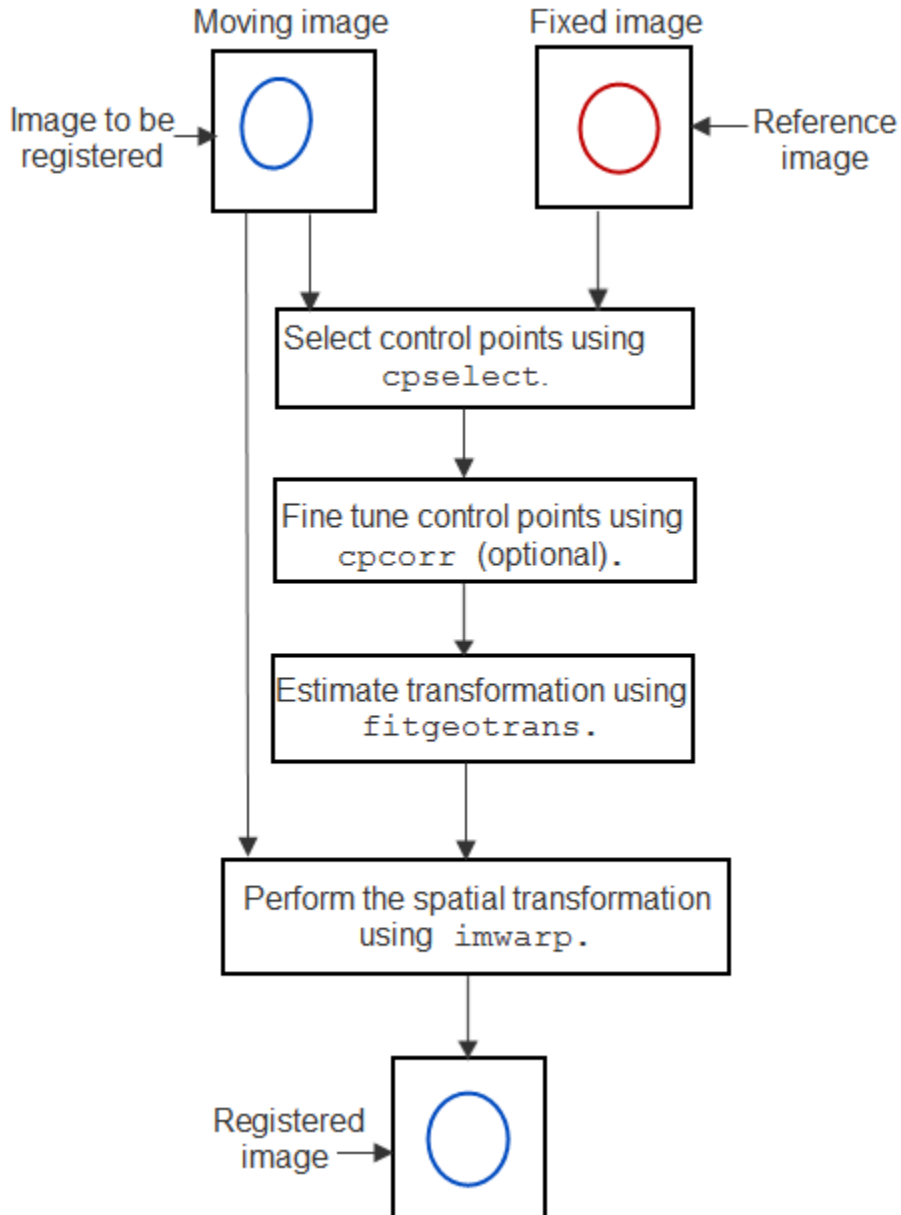
See Also

cpcorr | cpselect | fitgeotrans | imwarp

Related Examples

- "Register Images with Projection Distortion Using Control Points" on page 7-80
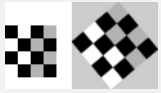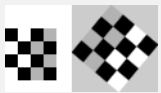
More About

- "Control Point Selection Procedure" on page 7-60
- "Approaches to Registering Images" on page 7-2

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	
`'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 7-62, specifying the moving image and the fixed image.

2   Use navigation aids to explore the image on page 7-64, 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 7-68 in the moving image and the fixed image.

4   Save the control points on page 7-73 in the workspace.

The following figure shows the default appearance of the tool when you first start it.

Point
Prediction

Zoom controls

Pan images

Magnification

Select
points

Detail
windows

Overview
windows

Detail
rectangles

Moving Detail: moon_moving 200% Lock ratio 200% Fixed Detail: moon_fixed
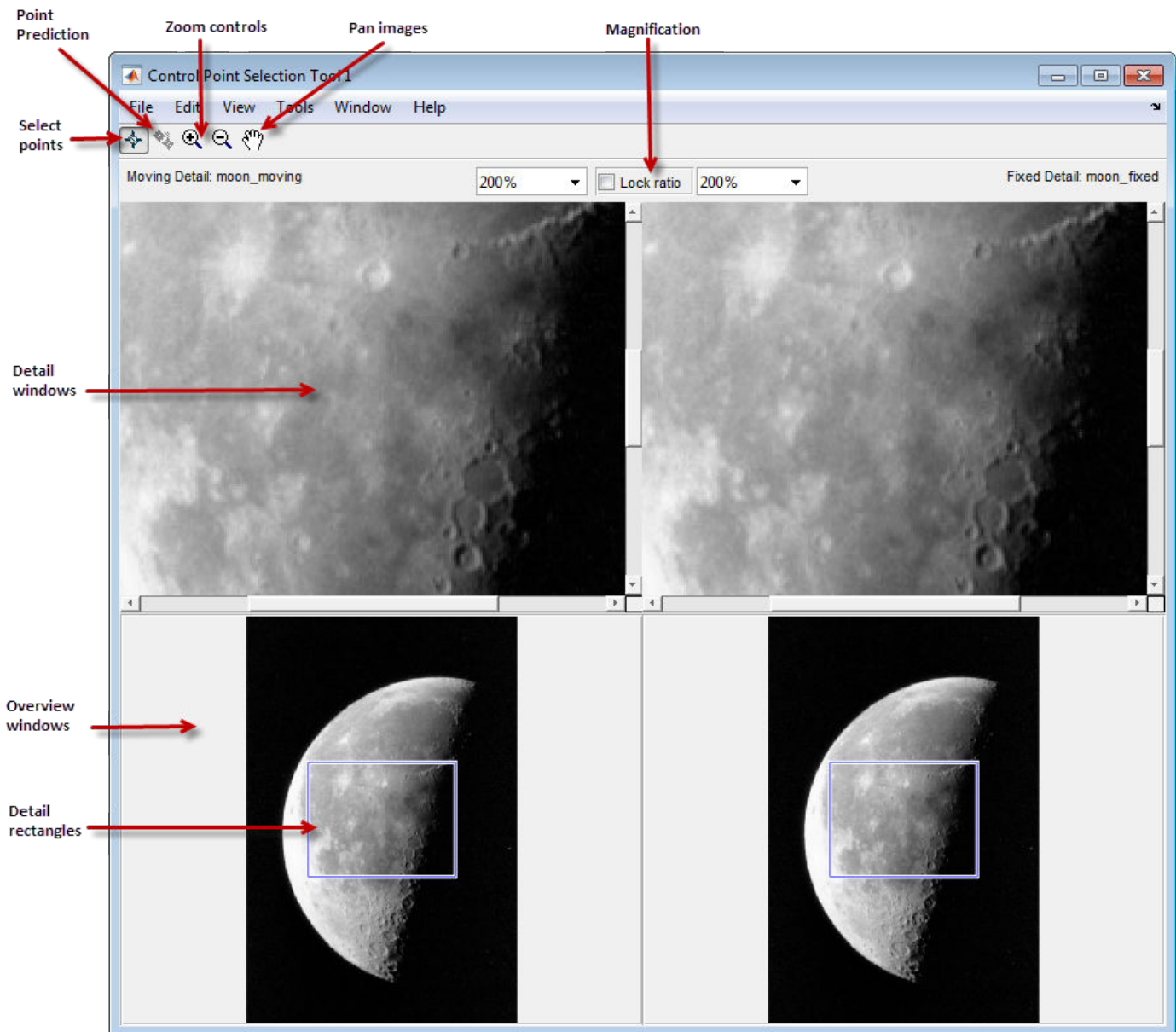
See Also
`cpselect`

More About

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 7-73.

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 Images with Projection Distortion Using Control Points" on page 7-80.

```
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.
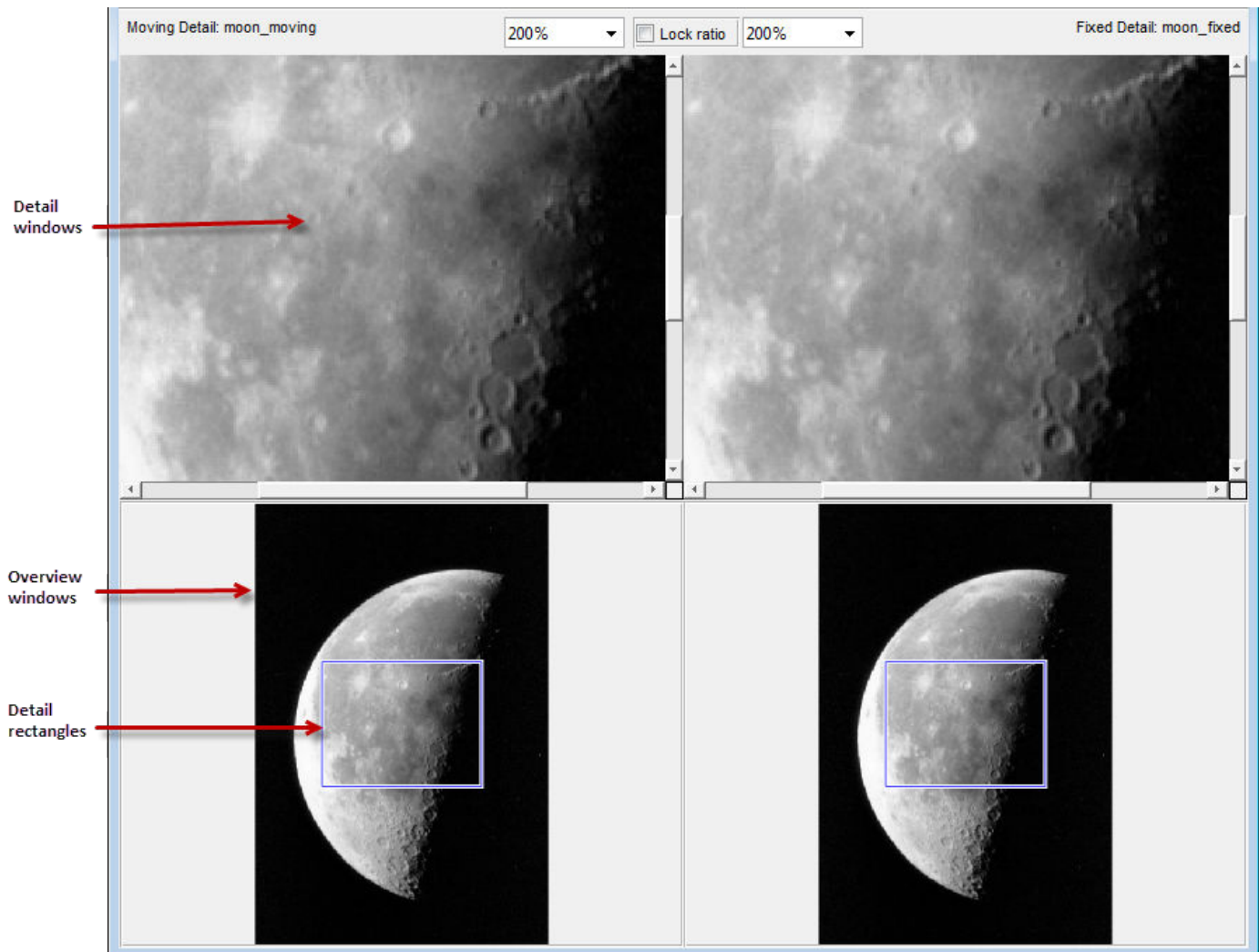
The next step is to use navigation aids to explore the image, looking for visual elements shared by both images. For more information, see "Find Visual Elements Common to Both Images" on page 7-64.

See Also

cpselect

More About

- "Find Visual Elements Common to Both Images" on page 7-64
- "Export Control Points to the Workspace" on page 7-73
- "Control Point Selection Procedure" on page 7-60

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 7-64
"Use the Detail Rectangle to Change the View" on page 7-64
"Pan the Image Displayed in the Detail Window" on page 7-64
"Zoom In and Out on an Image" on page 7-65
"Specify the Magnification of the Images" on page 7-65
"Lock the Relative Magnification of the Moving and Fixed Images" on page 7-66

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 7-65 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.
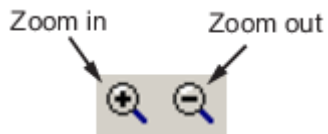
2 Move the pointer over the image in the Detail window that you want to zoom in or out on. The cursor changes to the appropriate magnifying glass shape, such as 🔍 . Position the cursor over a location in the image and click the mouse. With each click, the Control Point Selection Tool changes the magnification of the image by a preset amount. (See "Specify the Magnification of the Images" on page 7-65 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 7-66 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 7-65 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 7-66 for more information.
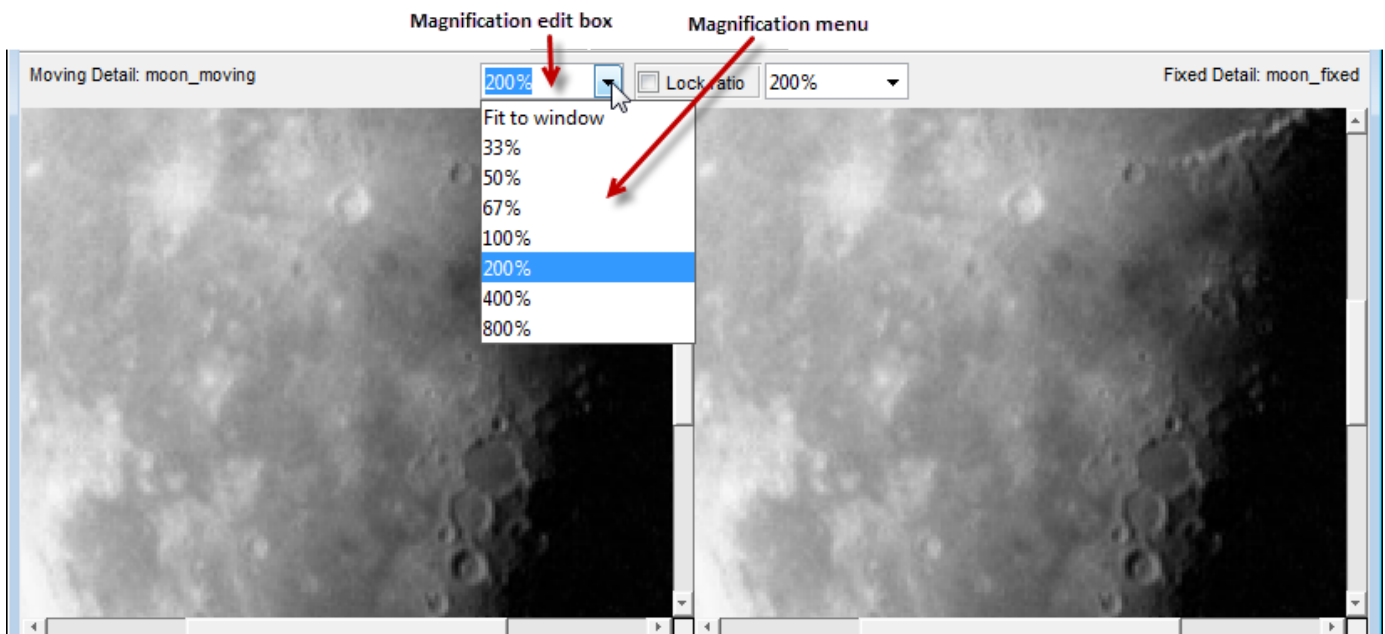
Lock the Relative Magnification of the Moving and Fixed Images

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 7-65 or specify a magnification value on page 7-65 for either of the images.
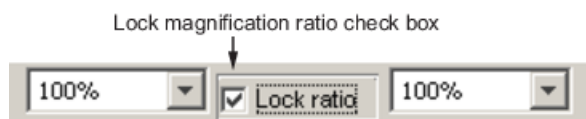
The next step is to specify matching control point pairs. For more information, see "Select Matching Control Point Pairs" on page 7-68.

See Also

More About

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 7-68
"Use Control Point Prediction" on page 7-69
"Move Control Points" on page 7-71
"Delete Control Points" on page 7-71

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, ⊕, 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 7-71 or delete on page 7-71 control points after you create them.

The following figure illustrates control points in several states.

Matched pair of points

Moving Detail: moon_moving

Fixed Detail: moon_fixed

Selected, unmatched point
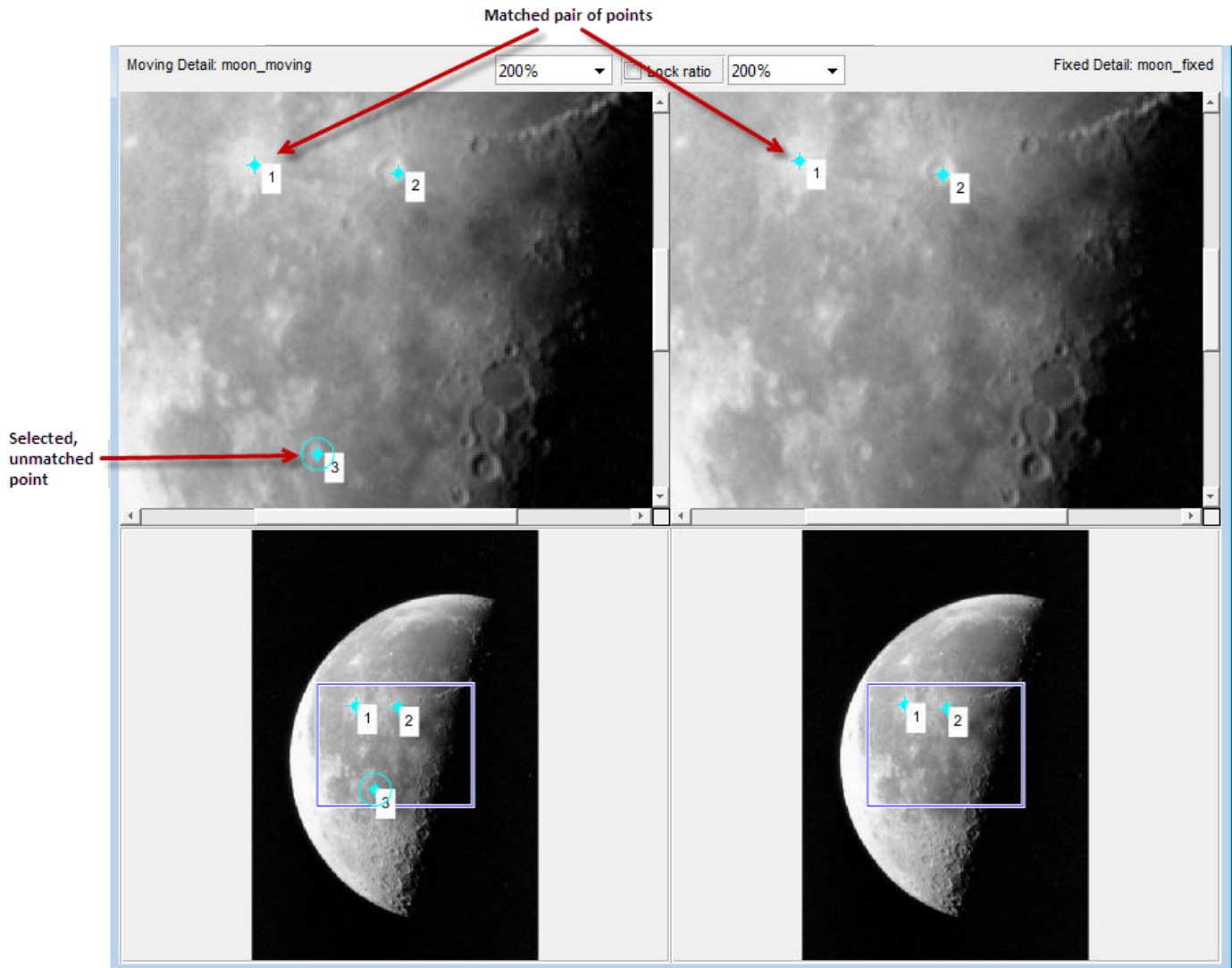
Use Control Point Prediction

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.
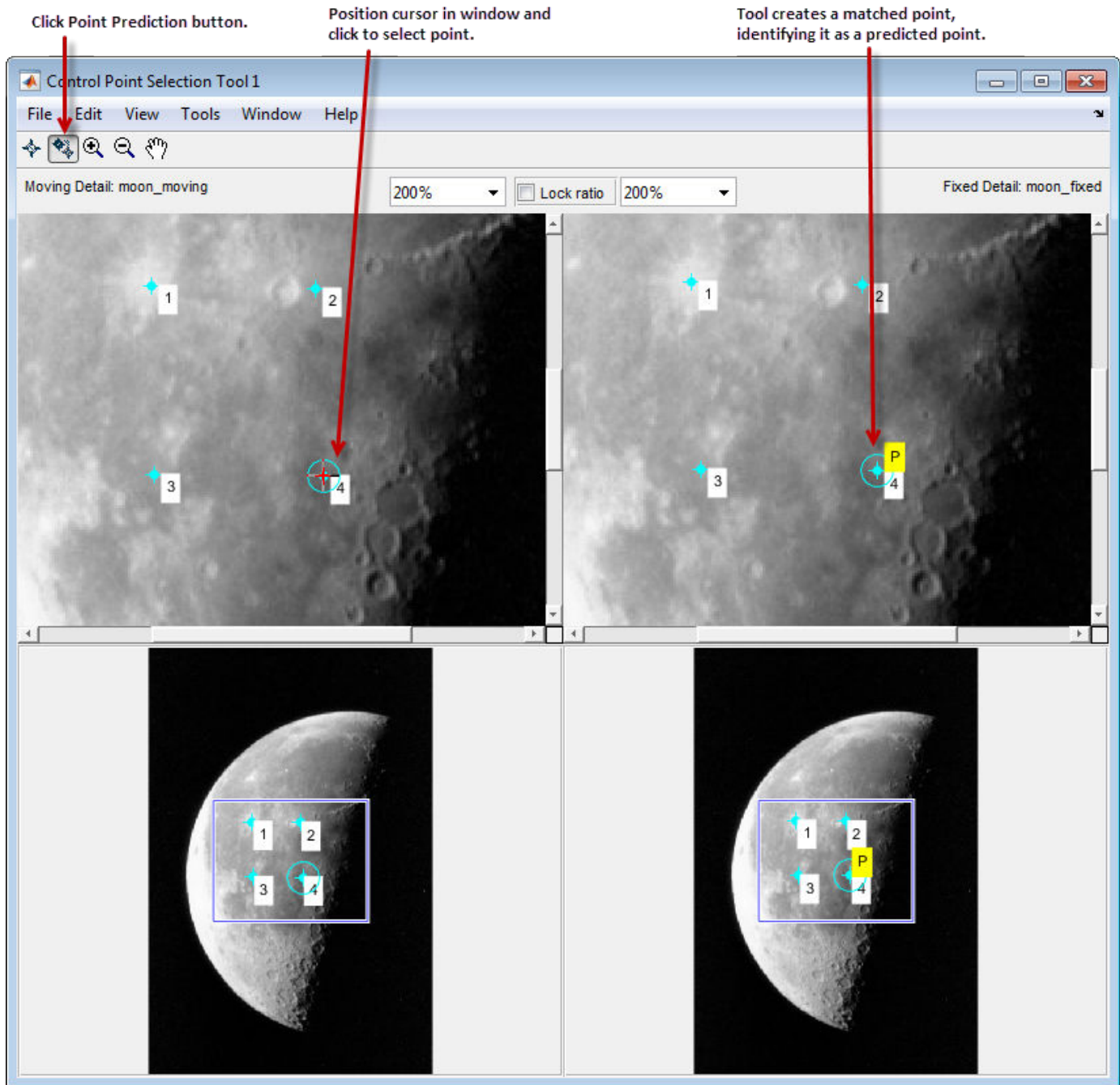
Note By default, the Control Point Selection Tool does not include predicted points in the set of valid control points returned in `movingPoints` or `fixedPoints`. To include predicted points, you must accept them by selecting the points and fine-tuning their position with the cursor. When you move a 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 7-71.

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.

Delete Active Pair
Delete Active Input Point
Delete Active Base Point

See Also

More About

- "Find Visual Elements Common to Both Images" on page 7-64
- "Export Control Points to the Workspace" on page 7-73
- "Control Point Selection Procedure" on page 7-60

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:
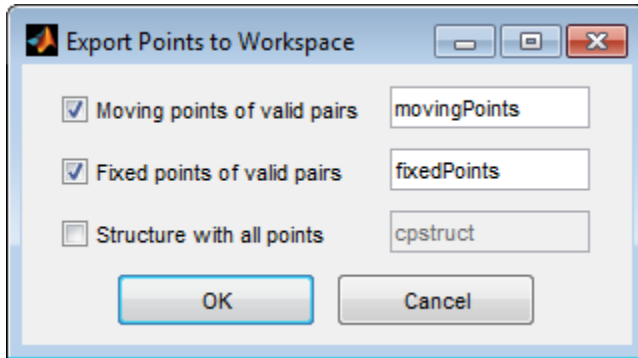
By default, the Control Point Selection Tool saves the coordinates of valid control points. The Control Point Selection Tool does not include unmatched and predicted points in the `movingPoints` and `fixedPoints` arrays. The arrays are *n*-by-2 arrays, where *n* is the number of valid control point pairs 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.
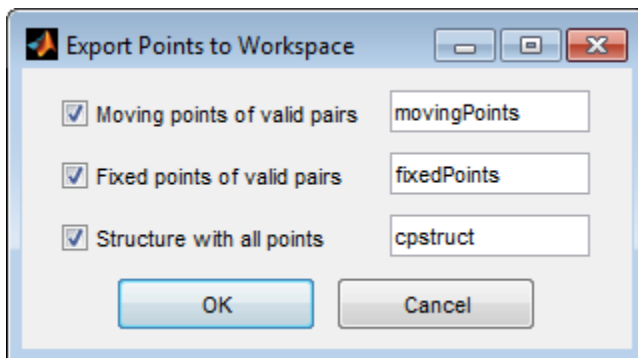
This option saves the positions of all control points and their current state in a `cpstruct` structure.

```
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 7-68
- "Control Point Selection Procedure" on page 7-60

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')
```


Image courtesy of Massachusetts Institute of Technology

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.
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 = scale * cos(theta)

Let ss = scale*sin(theta)

Then, `Tinv = invert(tform)`, and `Tinv.T` = $\begin{bmatrix} sc & -ss & 0 \\ ss & sc & 0 \\ tx & ty & 1 \end{bmatrix}$

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.

```
montage({original,recovered})
```

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 7-68
- "Control Point Selection Procedure" on page 7-60
- "Find Image Rotation and Scale Using Automated Feature Matching" (Computer Vision Toolbox)

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 unmodified values in `movingPoints`.

See Also
`cpcorr` | `cpselect` | `cpstruct2pairs`

More About
- "Select Matching Control Point Pairs" on page 7-68
- "Control Point Selection Procedure" on page 7-60

Register Images with Projection Distortion Using Control Points

This example shows how to register two images by selecting control points common to both images and inferring a geometric transformation that aligns the control points.

Read Images

Read the image `westconcordorthophoto.png` into the workspace. This image is an orthophoto that has already been registered to the ground.

```
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

Read the image `westconcordaerial.png` into the workspace. This image was taken from an airplane and is distorted relative to the orthophoto. Because the unregistered image was taken from a distance and the topography is relatively flat, it is likely that most of the distortion is projective.

```
unregistered = imread('westconcordaerial.png');
imshow(unregistered)
```

```
text(size(unregistered,2),size(unregistered,1)+15, ...
    'Image courtesy of mPower3/Emerge', ...
    'FontSize',7,'HorizontalAlignment','right');
```


Image courtesy of mPower3/Emerge

Select Control Point Pairs

To select control points interactively, open the Control Point Selection tool by using the `cpselect` function. Control points are landmarks that you can find in both images, such as a road intersection or a natural feature. Select at least four pairs of control points so that `cpselect` can fit a projective transformation to the control points. After you have selected corresponding moving and fixed points, close the tool to return to the workspace.

```
[mp,fp] = cpselect(unregistered,ortho,'Wait',true);
```

Infer Geometric Transformation

Find the parameters of the projective transformation that best aligns the moving and fixed points by using the `fitgeotrans` function.

```
t = fitgeotrans(mp,fp,'projective');
```

Transform Unregistered Image

To apply the transformation to the unregistered aerial image, use the `imwarp` function. Specify that the size and position of the transformed image match the size and position of the ortho image by using the OutputView name-value pair argument.

```
Rfixed = imref2d(size(ortho));
registered = imwarp(unregistered,t,'OutputView',Rfixed);
```

See the result of the registration by overlaying the transformed image over the original orthophoto.

```
imshowpair(ortho,registered,'blend')
```

See Also
cpcorr | cpselect | cpstruct2pairs | fitgeotrans

More About
- "Select Matching Control Point Pairs" on page 7-68
- "Control Point Selection Procedure" on page 7-60

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?

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 17-2 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.
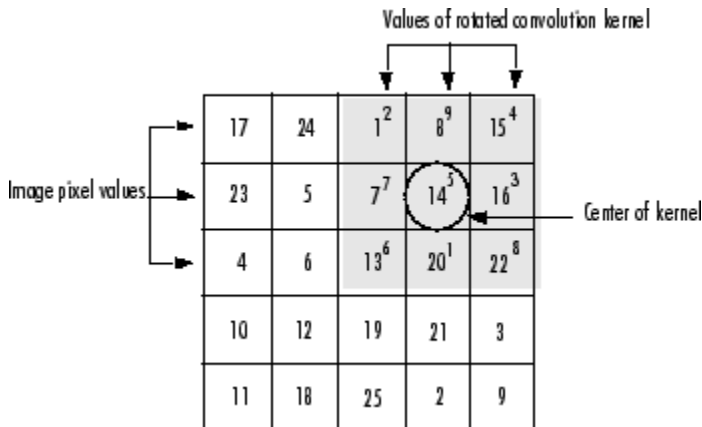
Computing the (2,4) Output of Convolution

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

$$1 \cdot 8 + 8 \cdot 1 + 15 \cdot 6 + 7 \cdot 3 + 14 \cdot 5 + 16 \cdot 7 + 13 \cdot 4 + 20 \cdot 9 + 22 \cdot 2 = 585$$
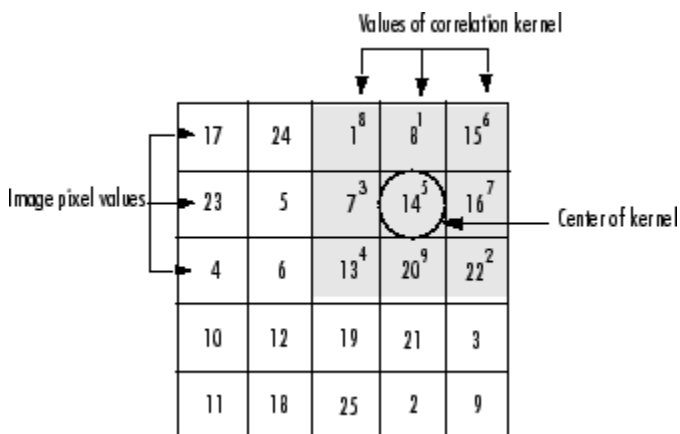
Computing the (2,4) Output of Correlation

See Also

conv2 | convn | imfilter

Related Examples

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).

input image   integral image
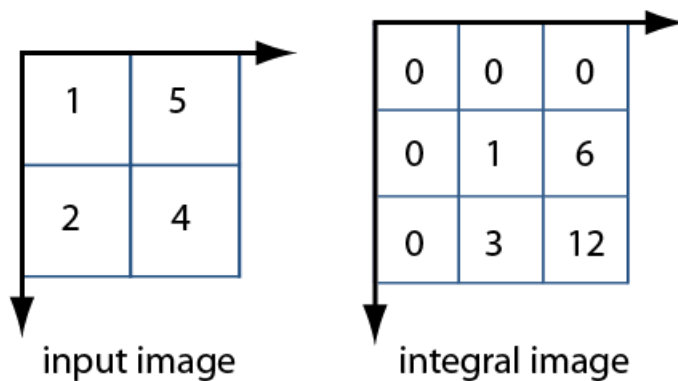
Using an integral image, you can rapidly calculate summations over image subregions. Integral images facilitate summation of pixels and can be performed in constant time, regardless of the neighborhood size. The following figure illustrates the summation of a subregion of an image, you can use the corresponding 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.

input image   integral image
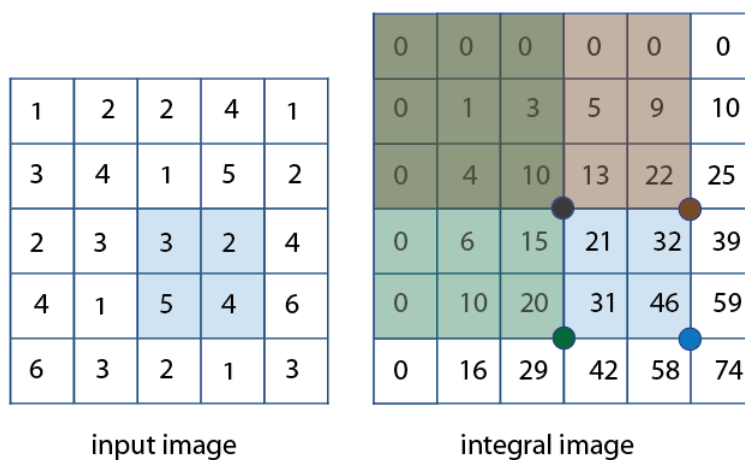
In this way, you can calculate summations in rectangular regions rapidly, irrespective of the filter size. Use of integral images was popularized by the Viola-Jones algorithm. To see the full citation for this algorithm and learn how to create an integral image, see `integralImage`.

See Also

integralBoxFilter | integralBoxFilter3 | integralImage | integralImage3

Related Examples

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')
```


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')
```
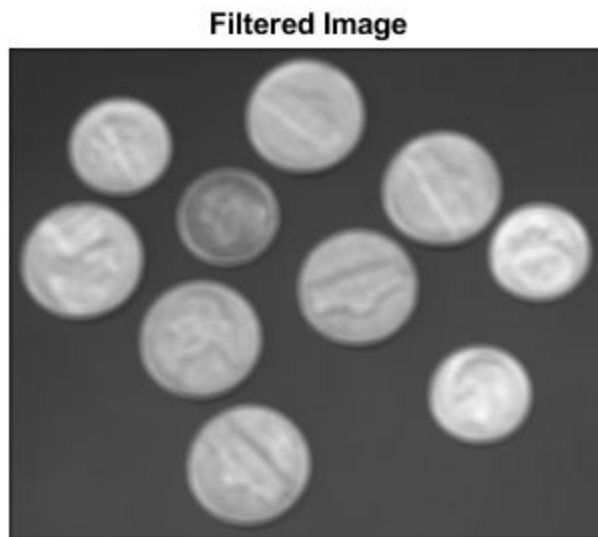

Filtered Image

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.
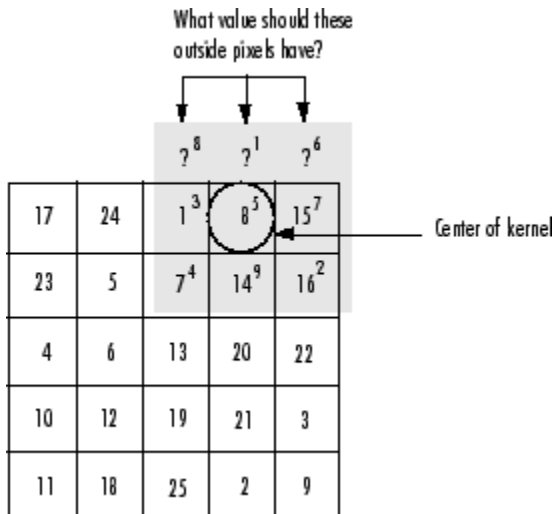
When the Values of the Kernel Fall Outside the Image

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.
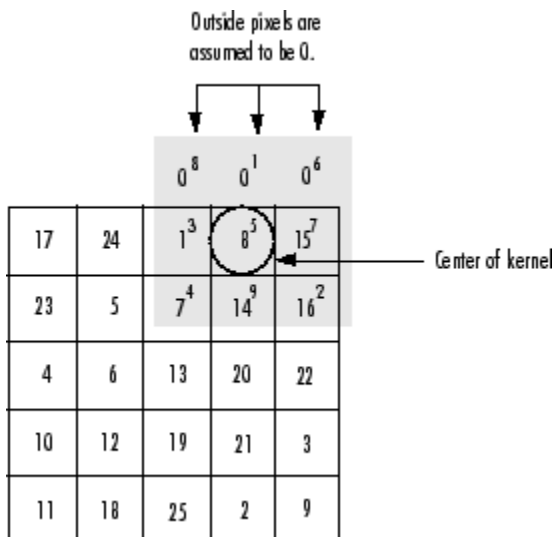
Zero Padding of Outside Pixels

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
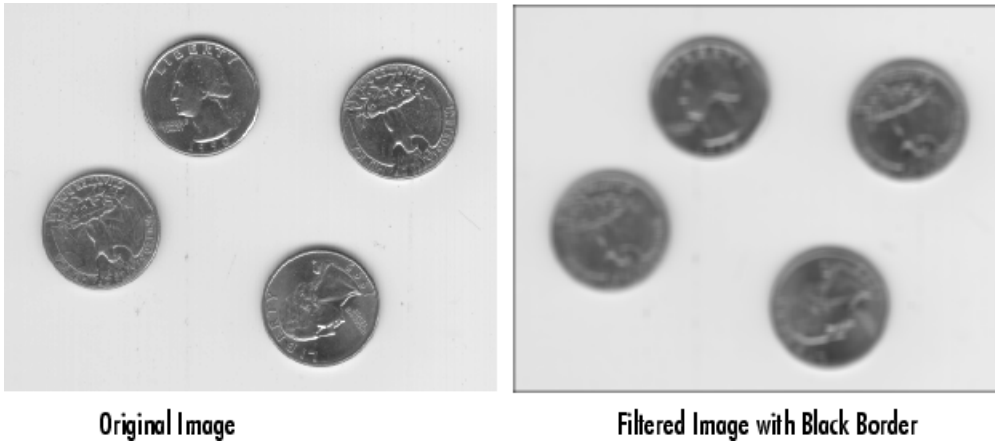
To eliminate the zero-padding artifacts around the edge of the image, `imfilter` offers an alternative boundary padding method called *border replication*. In border replication, the value of any pixel outside the image is determined by replicating the value from the nearest border pixel. This is illustrated in the following figure.
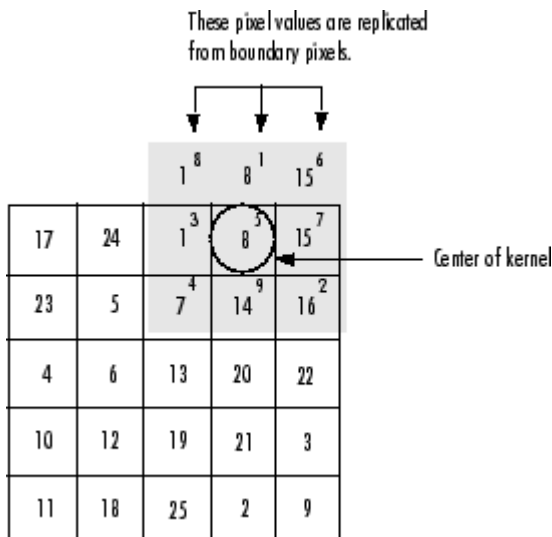
Replicated Boundary Pixels

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 8-46

Filter Images Using Predefined Filter

This example shows how to create a predefined Laplacian of Gaussian (LoG) filter using the `fspecial` function and apply the filter to an image using the `imfilter` function. A LoG filter highlights regions with rapidly varying intensities and reduces the impact of variations caused by noise. The `fspecial` function produces several additional types of predefined filters in the form of correlation kernels.

Read and display an image.

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

Create a 7-by-7 LoG filter with a standard deviation of `0.4` using `fspecial`.

```
h = fspecial('log',7,0.4)
```

h = *7×7*

```
    0.1263    0.1263    0.1263    0.1263    0.1263    0.1263    0.1263
    0.1263    0.1263    0.1263    0.1267    0.1263    0.1263    0.1263
    0.1263    0.1263    0.2333    1.1124    0.2333    0.1263    0.1263
    0.1263    0.1267    1.1124  -10.4357    1.1124    0.1267    0.1263
    0.1263    0.1263    0.2333    1.1124    0.2333    0.1263    0.1263
    0.1263    0.1263    0.1263    0.1267    0.1263    0.1263    0.1263
    0.1263    0.1263    0.1263    0.1263    0.1263    0.1263    0.1263
```

Apply the filter to the image using `imfilter`.

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

Display the filtered image.

```
imshow(I2)
```
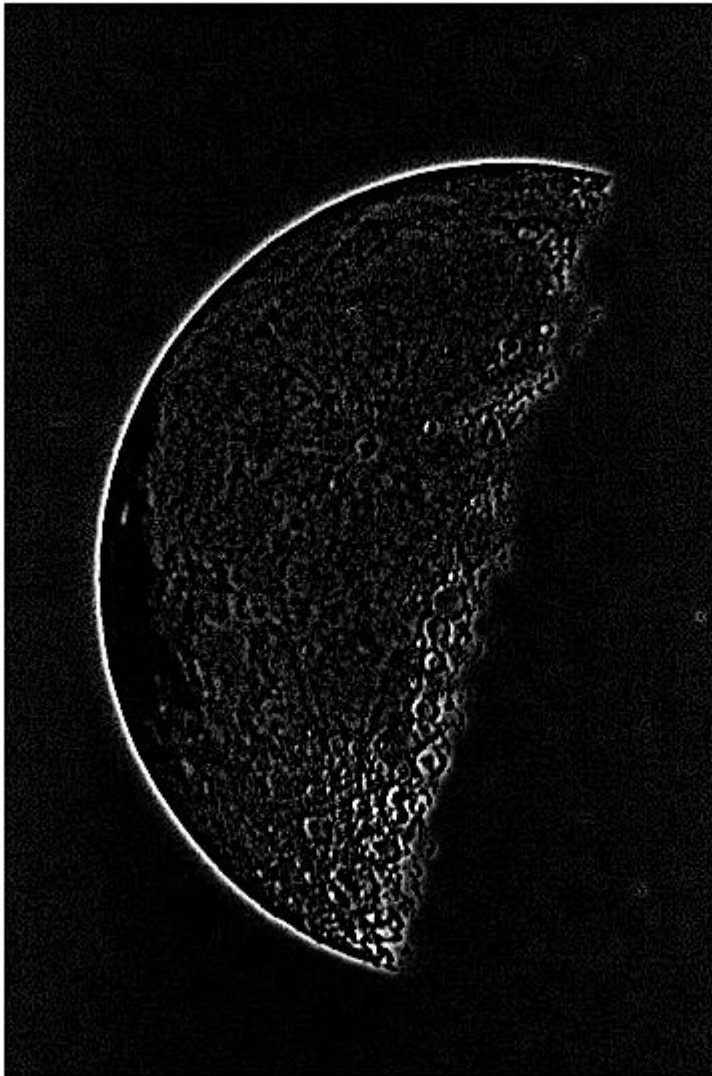
See Also
`fspecial` | `imfilter`

Related Examples

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 0.000557 seconds.
```

Blurred Image and Sharpened Image
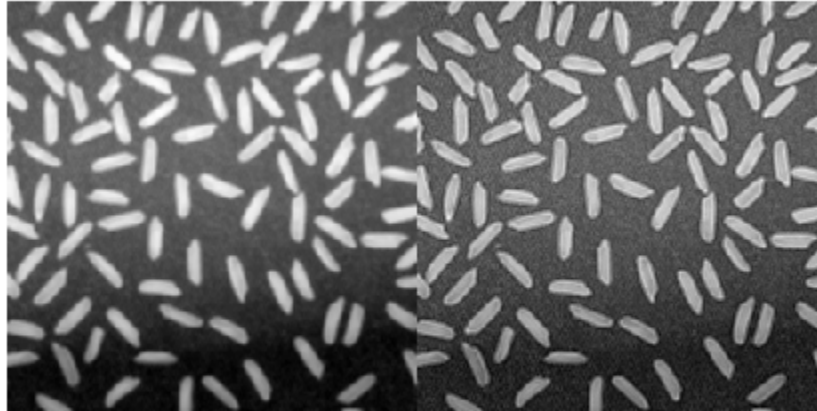
HDL-Optimized Design Considerations

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 "Functions" (Vision HDL Toolbox) and "Blocks" (Vision HDL Toolbox). 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" (Vision HDL Toolbox).

- **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`.

```matlab
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`.

```matlab
...
[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 107.544594 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" (Vision HDL Toolbox).

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
```

For more detail on how to create and configure MATLAB to HDL projects, see the "Getting Started with MATLAB to HDL Workflow" tutorial in the HDL Coder documentation.

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');
```

HDL Code Generation for Image Sharpening

Copyright 2015 The MathWorks, Inc.
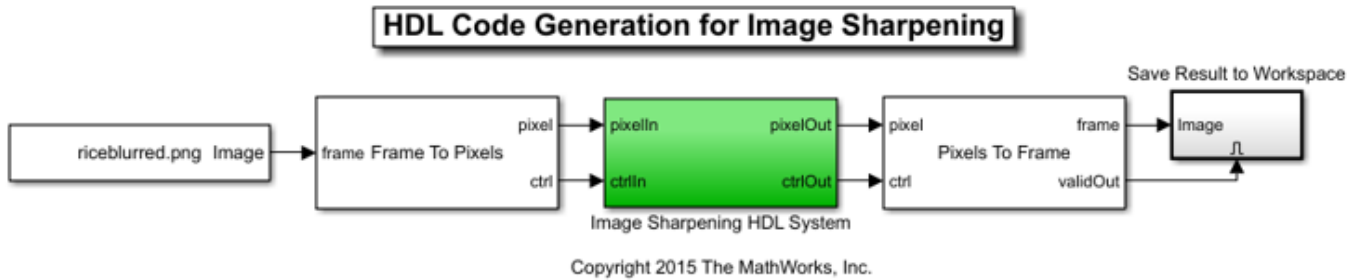
The model reads in the blurred image. The Frame To Pixels block converts a full-frame image to a pixel stream, and the Pixels To Frame block converts the pixel stream back to a full-frame image. The Image Sharpening HDL System contains an Image Filter block, which is the HDL-friendly counterpart in Vision 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 32.122551 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.
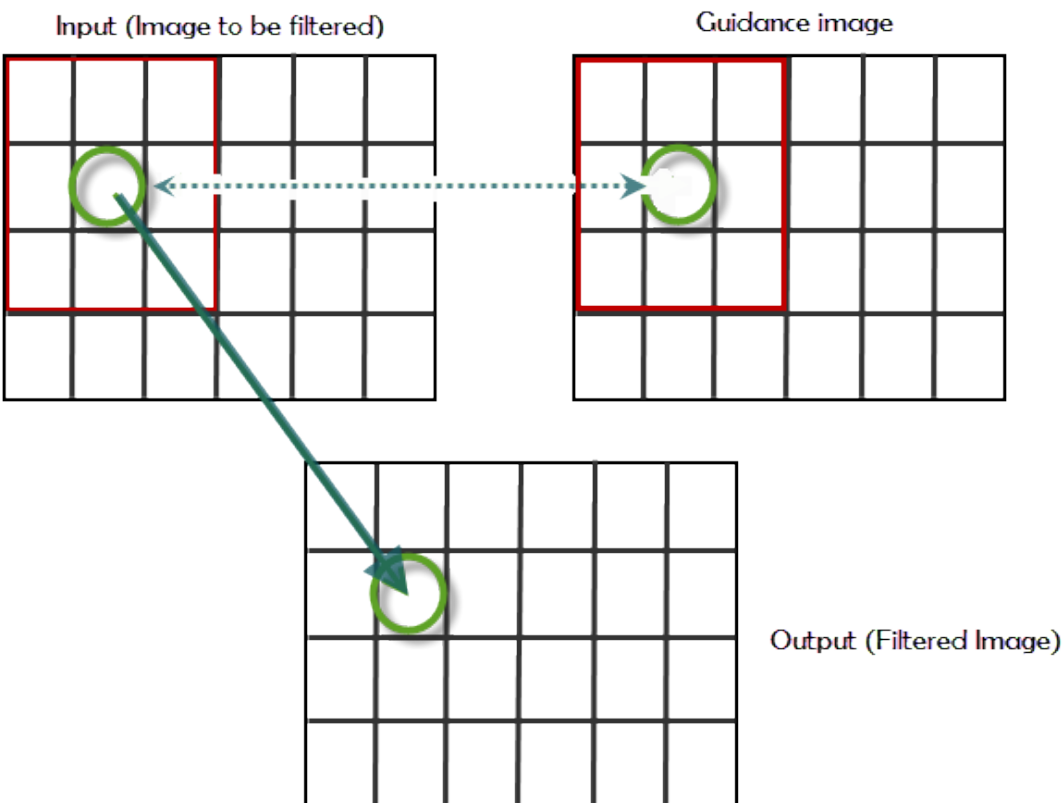
If the guidance is the same as the image to be filtered, the structures are the same—an edge in original image is the same in the guidance image. If the guidance image is different, structures in the guidance image will impact the filtered image, in effect, imprinting these structures on the original image. This effect is called structure transference.

See Also
`imguidedfilter`

Related Examples
*   "Perform Flash/No-flash Denoising with Guided Filter" on page 8-25

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')
```


Input Image - Camera Flash Off
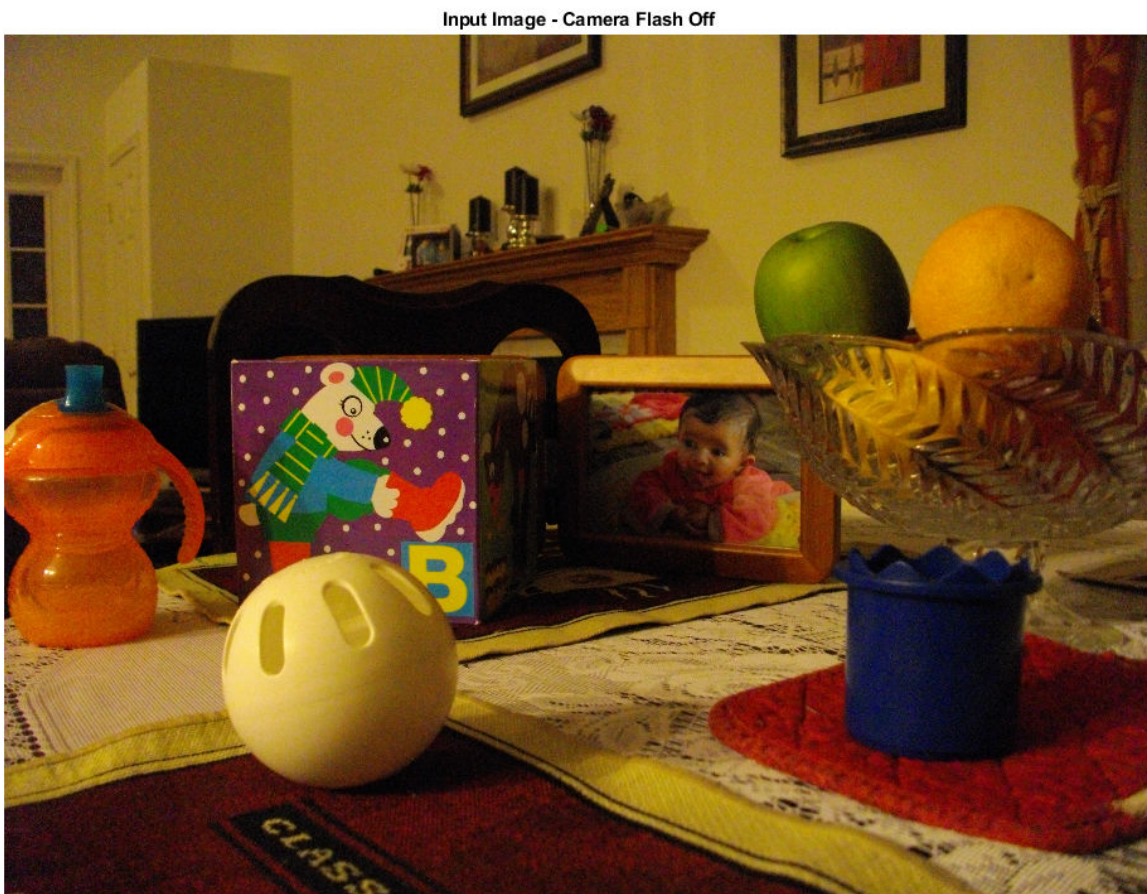
Read the image that you want to use as the guidance image into the workspace. In this example, the guidance image is a picture of the same scene taken with a flash.

```
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')
```

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])
```
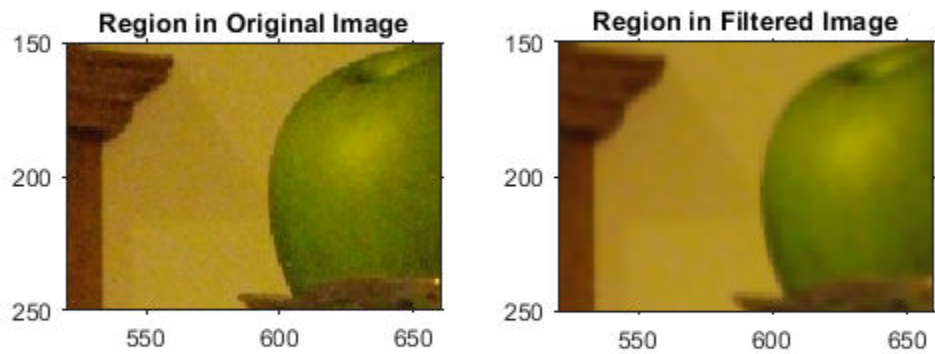
See Also
imguidedfilter

More About

- "What is Guided Image Filtering?" on page 8-24

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')
```


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')
```


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')
```

Label matrix from 3-level Otsu
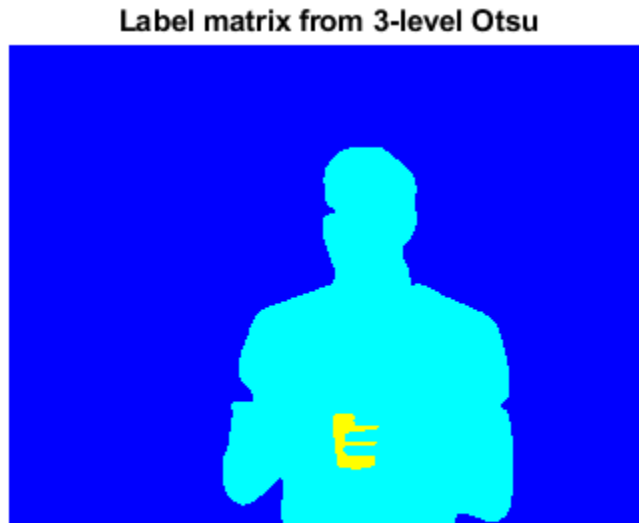
Draw a bounding box around the foreground regions in the image and put the mean temperature value of the region in the box. The example assumes that the largest region is the background. Use the `regionprops` function to get information about the regions in the segmented image.

```
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
```

Segmented regions with mean temperature

See Also

imfill | imguidedfilter | imquantize | multithresh

More About

- "What is Guided Image Filtering?" on page 8-24

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')
```

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')
```

Integral Image Representation
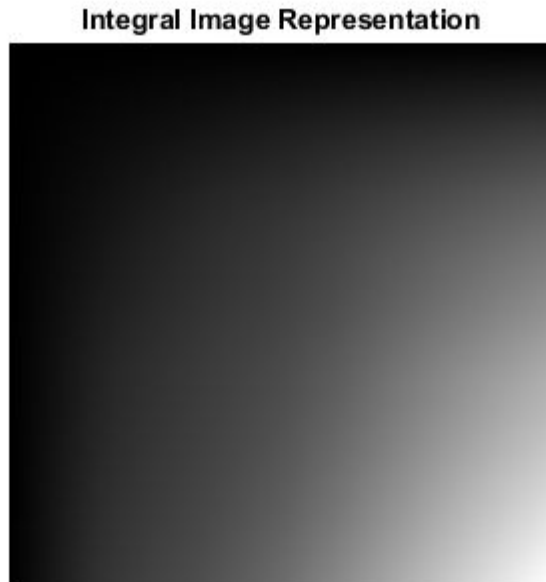
Apply three box filters of varying sizes to the integral image. The `integralBoxFilter` function can be used to apply a 2-D box filter to the integral image representation of an image.

```
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')
```


Image filtered with [7 7] box filter

```
figure
imshow(filteredImage2,[])
title('Image filtered with [11 11] box filter')
```

Image filtered with [11 11] box filter

```
figure
imshow(filteredImage3,[])
title('Image filtered with [15 15] box filter')
```


Image filtered with [15 15] box filter

See Also

integralBoxFilter | integralBoxFilter3 | integralImage | integralImage3

More About

- "Integral Image" on page 8-5

Reduce 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);

imshow(originalImage)
```

To simulate noise for this example, add some Gaussian noise to the image.

```
noisyImage = imnoise(originalImage,'gaussian');
imshow(noisyImage)
```

Compute the magnitude of the gradient by using the `imgradient` and `imgradientxy` functions. `imgradient` finds the gradient magnitude and direction, and `imgradientxy` finds directional image gradients.

```
sobelGradient = imgradient(noisyImage);
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');
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 8-46
"Transform 1-D FIR Filter to 2-D FIR Filter" on page 8-46
"Frequency Sampling Method" on page 8-48
"Windowing Method" on page 8-49
"Creating the Desired Frequency Response Matrix" on page 8-50
"Computing the Frequency Response of a Filter" on page 8-51

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   -0.0916

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

  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])
```
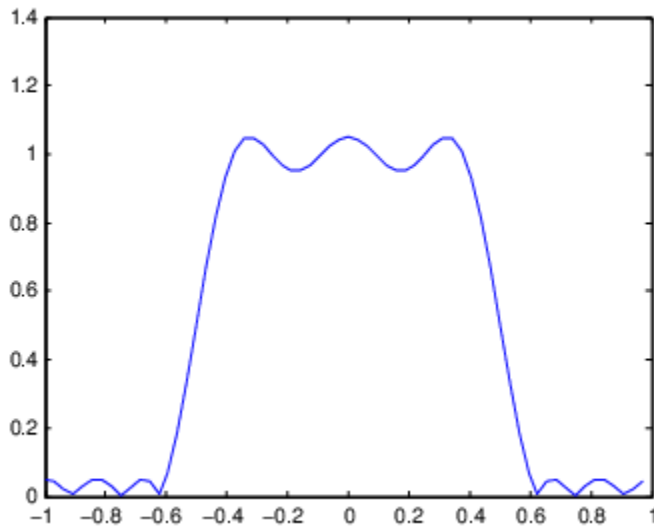
One-Dimensional Frequency Response
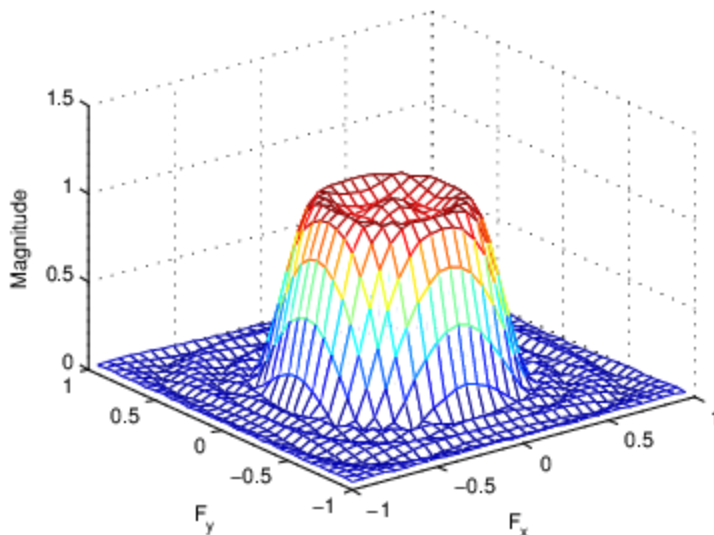
Corresponding Two-Dimensional Frequency Response

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 8-51 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])
```
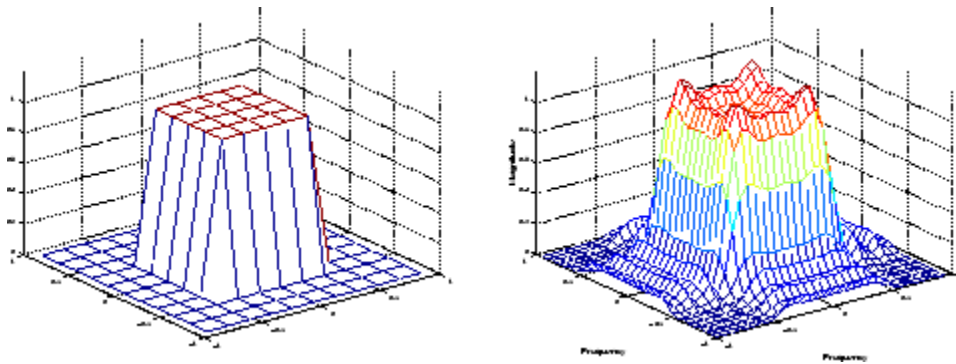

Desired Two-Dimensional Frequency Response (left) and Actual Two-Dimensional Frequency Response (right)

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])
```
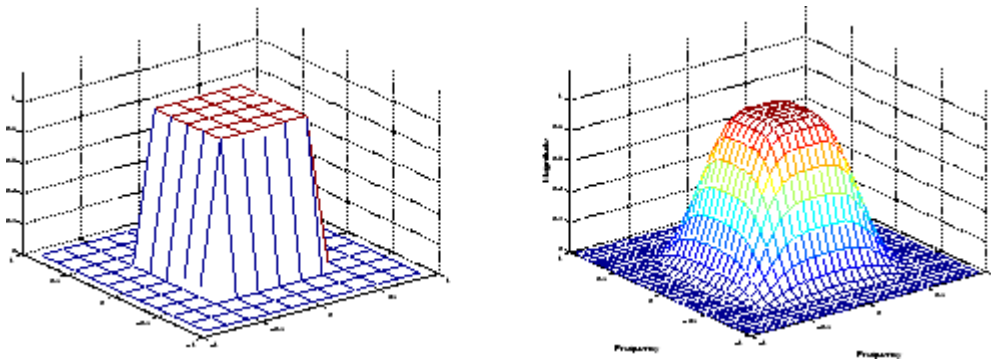

Desired Two-Dimensional Frequency Response (left) and Actual Two-Dimensional Frequency Response (right)

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)
```
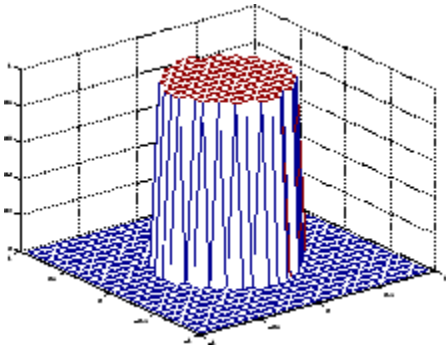
Ideal Circular Low-pass Frequency Response

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)
```
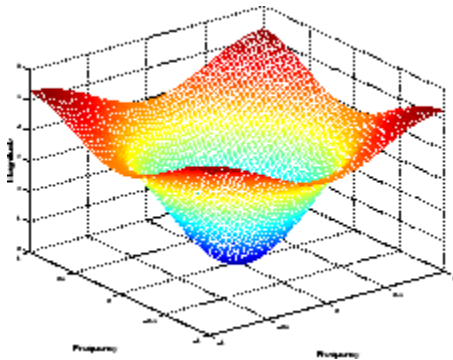

Frequency Response of a Two-Dimensional Filter

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

In this section...
"Definition of Fourier Transform" on page 9-2
"Discrete Fourier Transform" on page 9-5
"Applications of the Fourier Transform" on page 9-8

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.
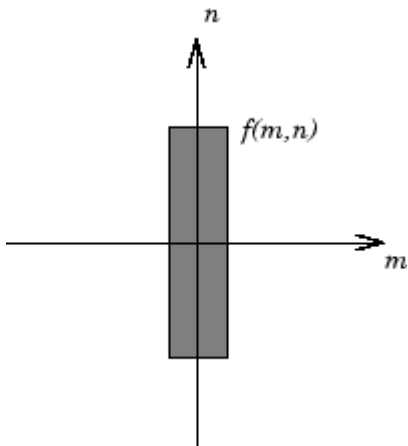
Rectangular Function

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.
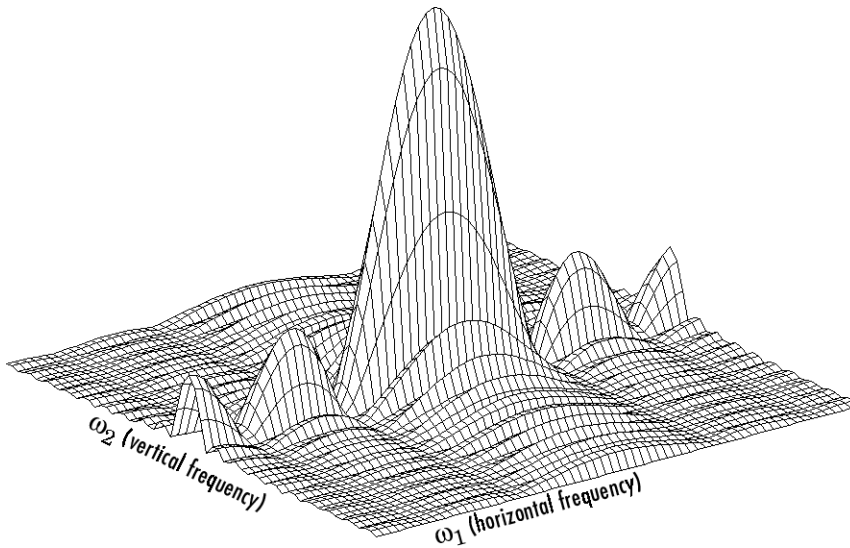
Magnitude Image of a Rectangular Function

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

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

as an image, as shown.
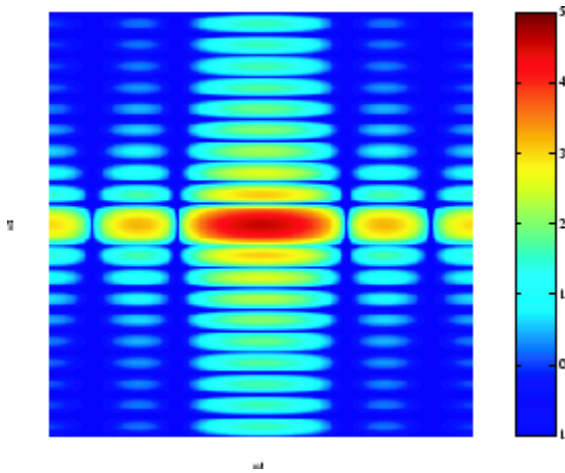
Log of the Fourier Transform of a Rectangular Function

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

Examples of the Fourier transform for other simple shapes are shown below.
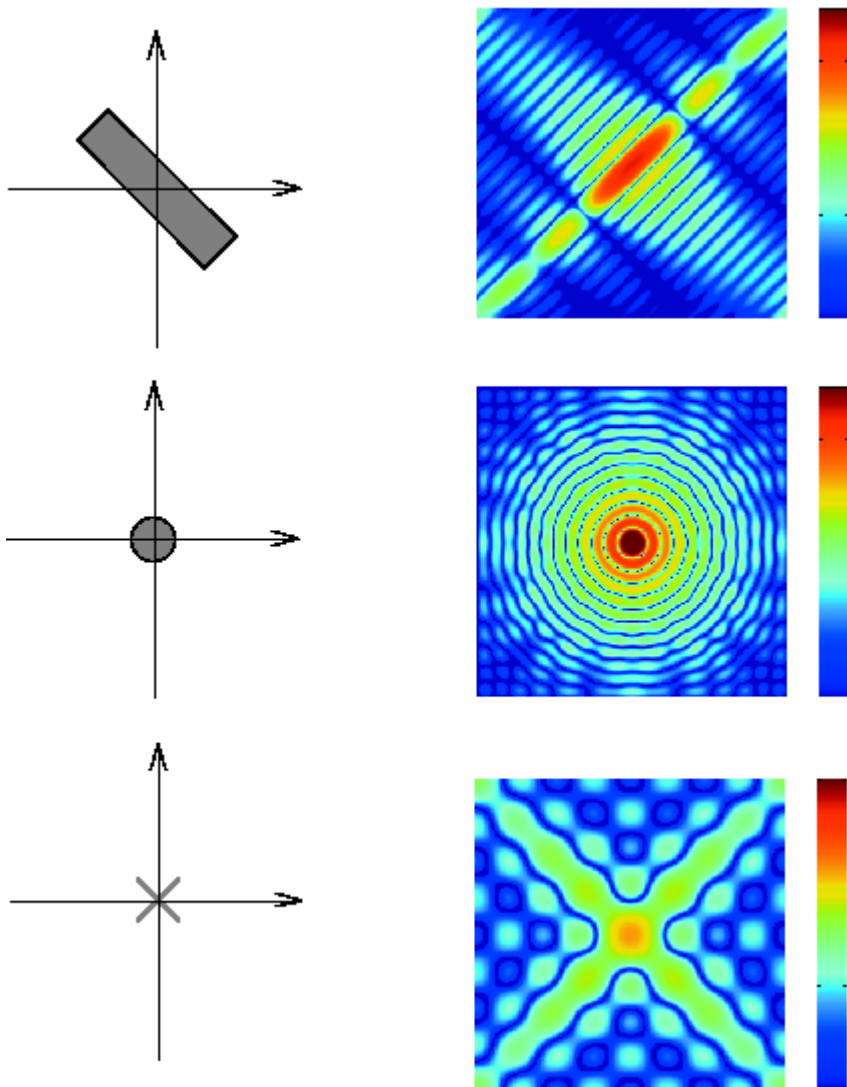
Fourier Transforms of Some Simple Shapes

Discrete Fourier Transform

Working with the Fourier transform on a computer usually involves a form of the transform known as the discrete Fourier transform (DFT). A discrete transform is a transform whose input and output values are discrete samples, making it convenient for computer manipulation. There are two principal reasons for using this form of the transform:

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

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

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

and

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

The values $F(p,q)$ are the DFT coefficients of $f(m,n)$. The zero-frequency coefficient, $F(0,0)$, is often called the "DC component." DC is an electrical engineering term that stands for direct current. (Note that matrix indices in MATLAB always start at 1 rather than 0; therefore, the matrix elements $f(1,1)$ and $F(1,1)$ correspond to the mathematical quantities $f(0,0)$ and $F(0,0)$, respectively.)

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

Relationship to the Fourier Transform

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

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

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')
```
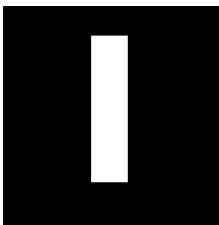

2   Compute and visualize the 30-by-30 DFT of f with these commands.

```
F = fft2(f);
F2 = log(abs(F));
imshow(F2,[-1 5],'InitialMagnification','fit');
colormap(jet); colorbar
```
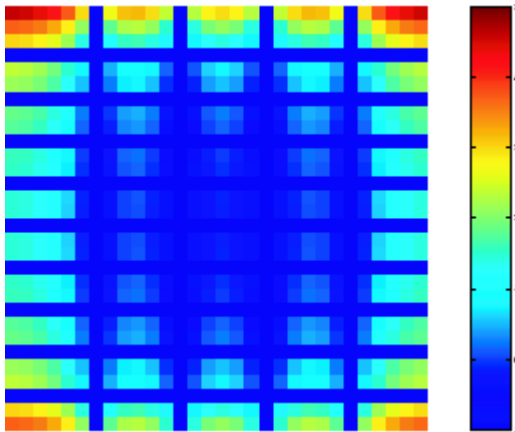
Discrete Fourier Transform Computed Without Padding

This plot differs from the Fourier transform displayed in "Visualizing the Fourier Transform" on page 9-2. 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
```
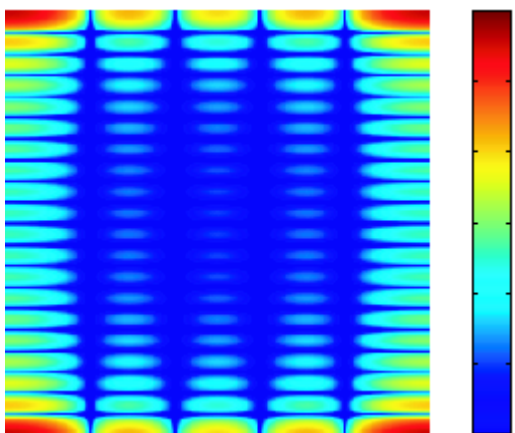

Discrete Fourier Transform Computed with Padding

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 9-2.

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)
```
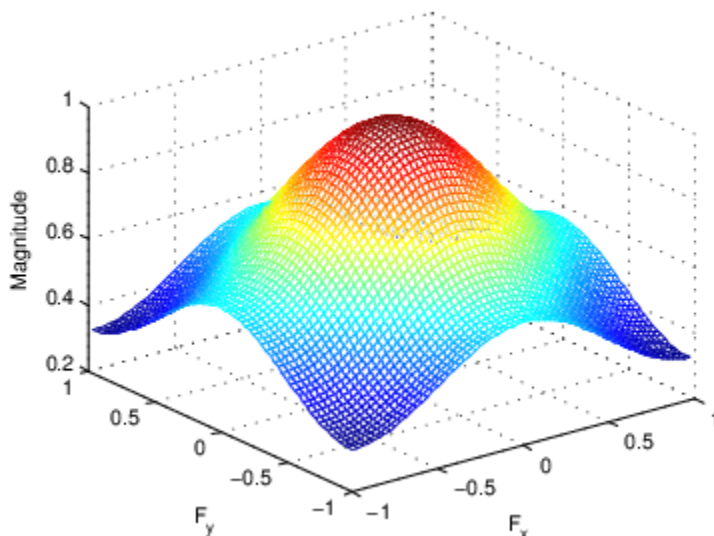

Frequency Response of a Gaussian Filter

See "Design Linear Filters in the Frequency Domain" on page 8-46 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.
```
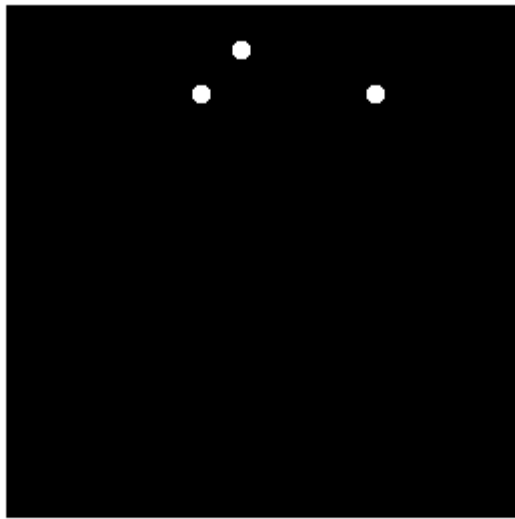
Discrete Cosine Transform

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.

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

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

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.
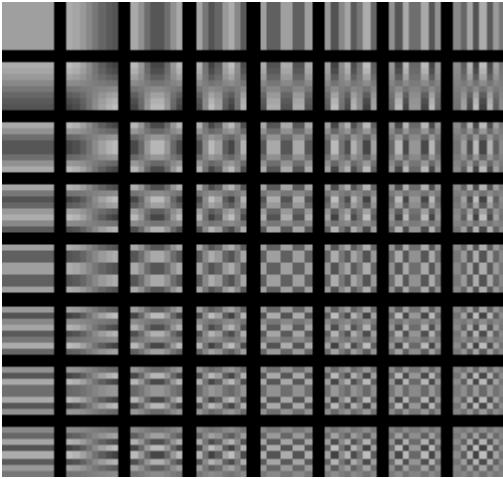
The 64 Basis Functions of an 8-by-8 Matrix

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

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

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)
```
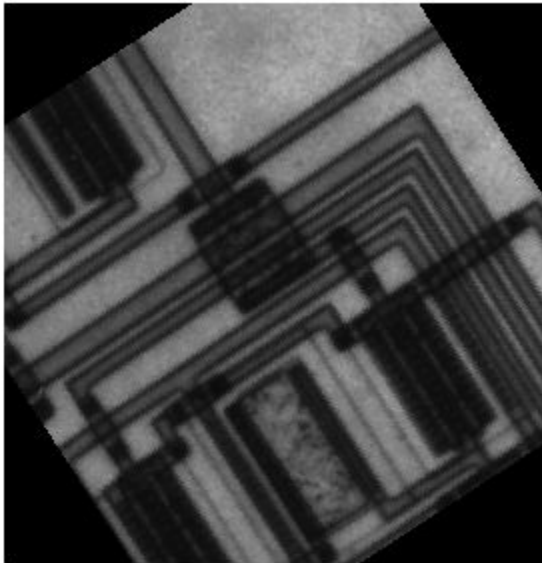
Find the edges in the image using the edge function.

```
BW = edge(rotI,'canny');
imshow(BW);
```
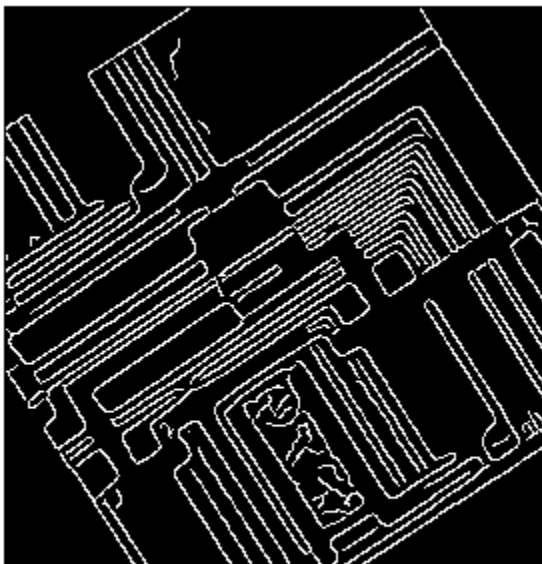
Compute the Hough transform of the binary image returned by `edge`.

```
[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)
```
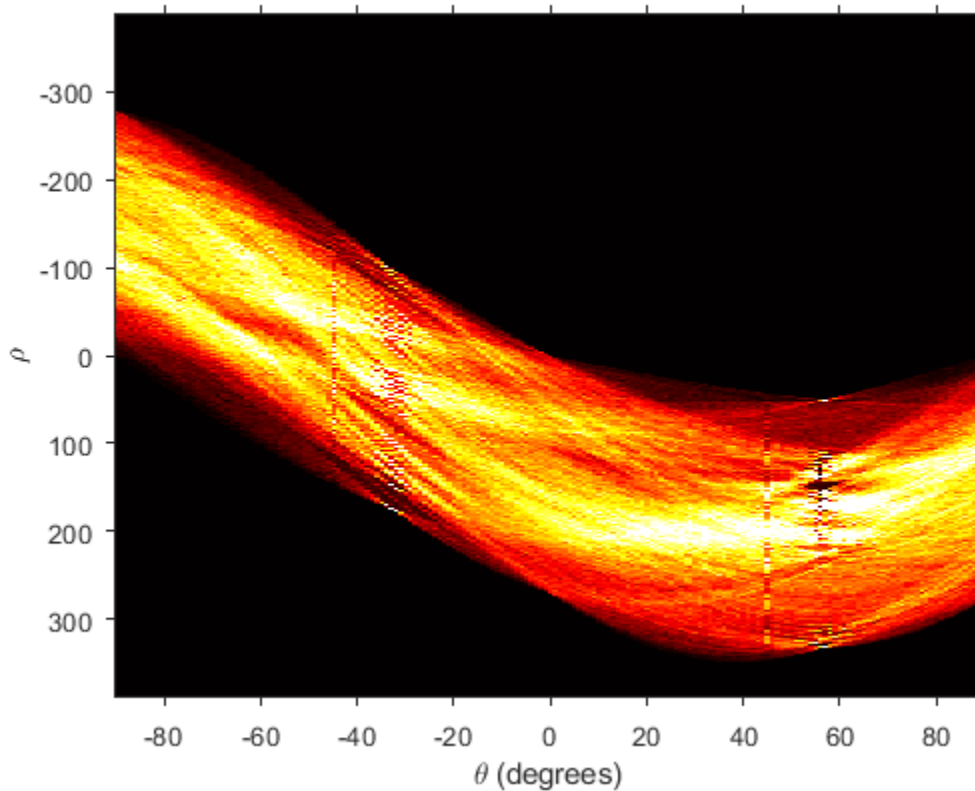

Find the peaks in the Hough transform matrix, `H`, using the `houghpeaks` function.

```
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');
```
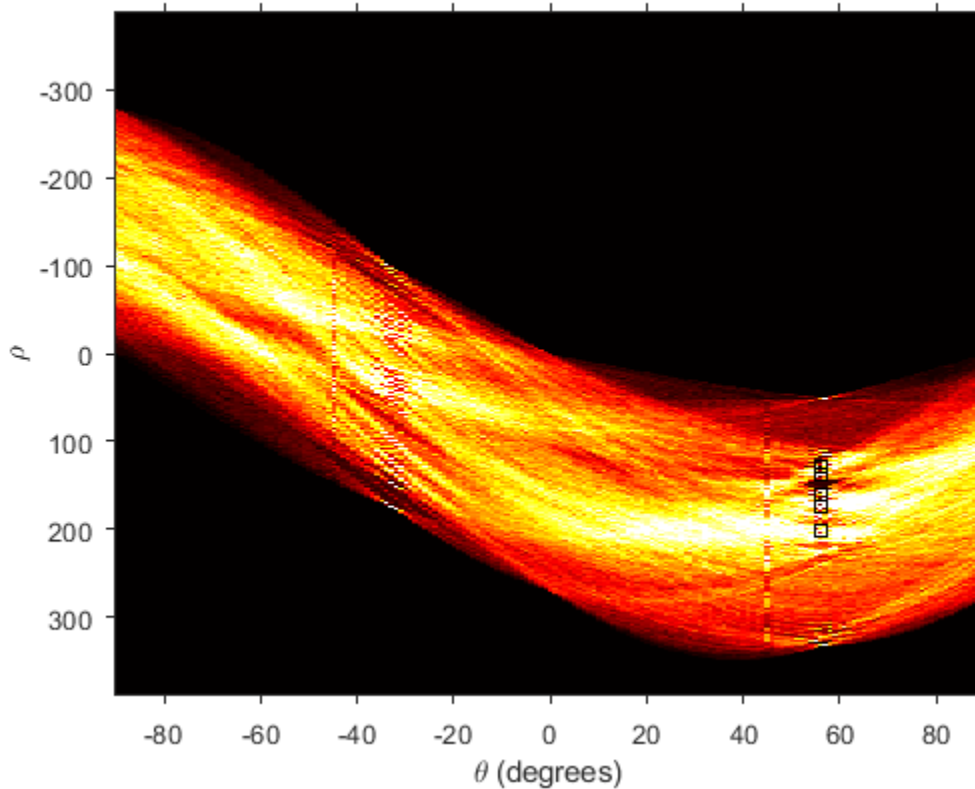
Find lines in the image using the `houghlines` function.

```
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');
```
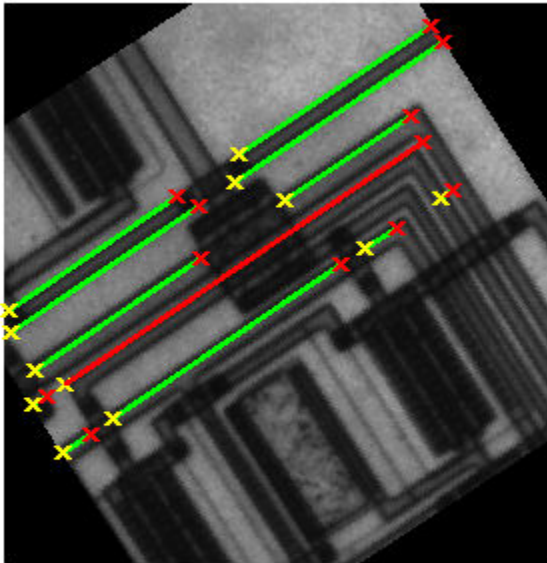
Radon Transform

In this section...

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 9-37. 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.
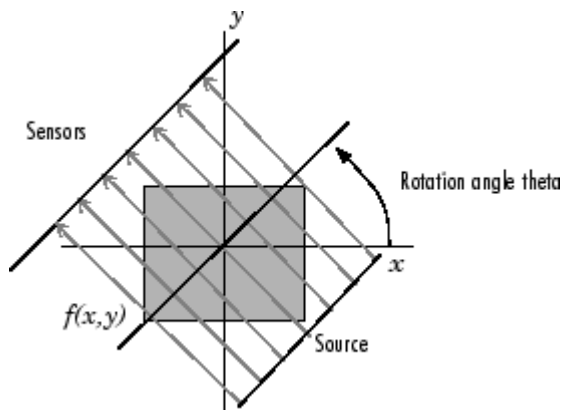
Parallel-Beam Projection at Rotation Angle Theta

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.
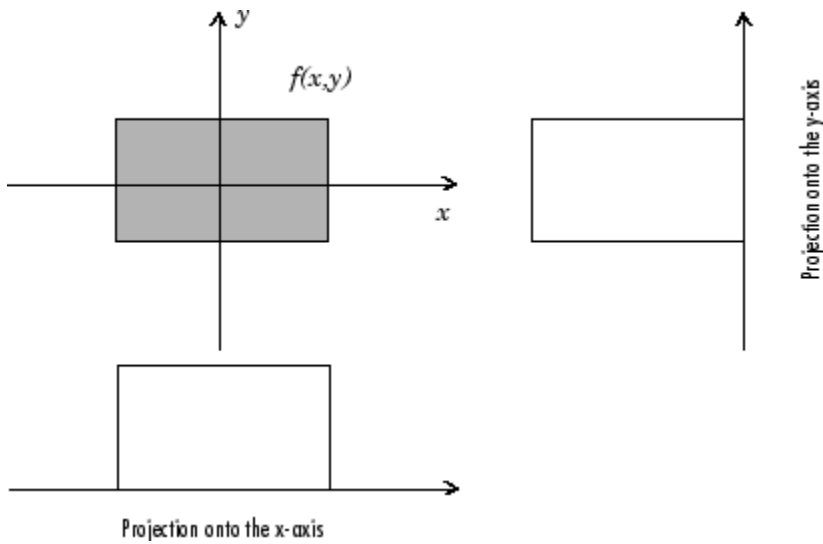
Horizontal and Vertical Projections of a Simple Function

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

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

where

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

The following figure illustrates the geometry of the Radon transform.
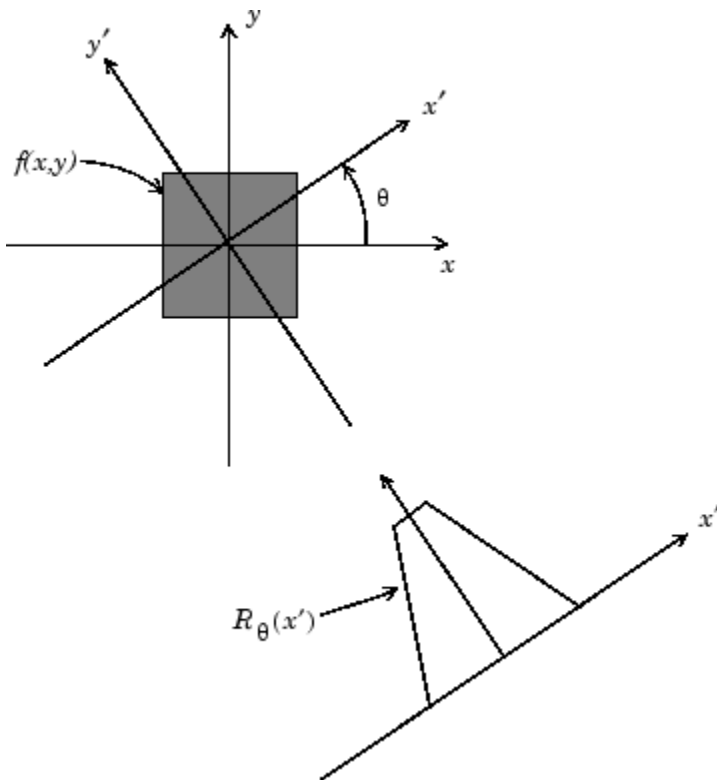
Geometry of the Radon Transform

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')
```
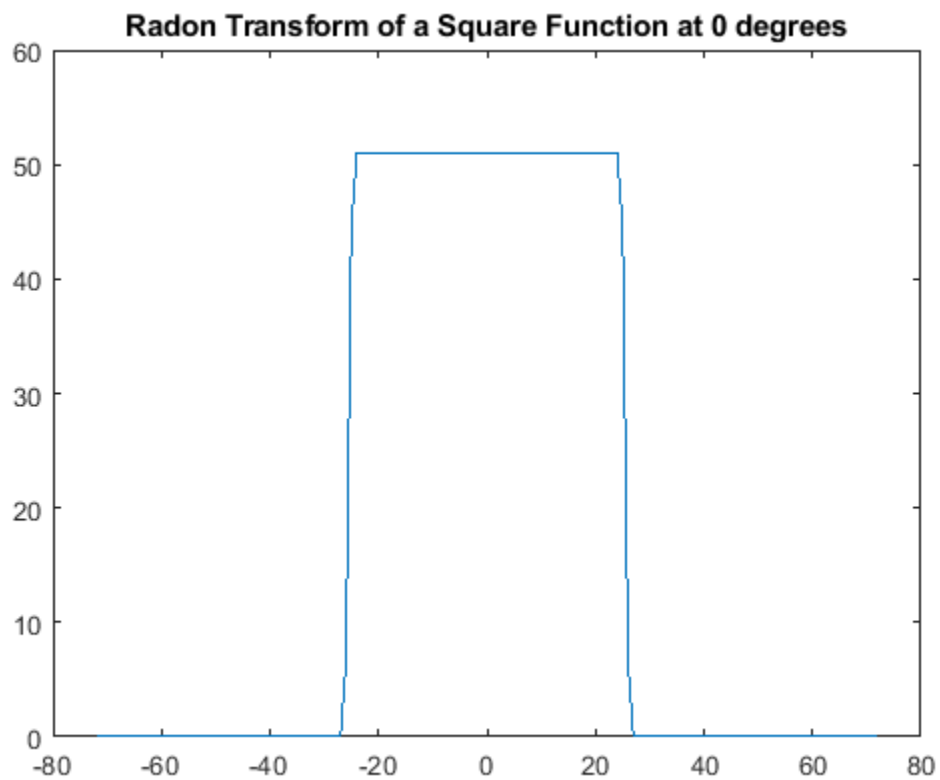

Plot the transform for 45 degrees.

```
figure
plot(xp,R(:,2));
title('Radon Transform of a Square Function at 45 degrees')
```
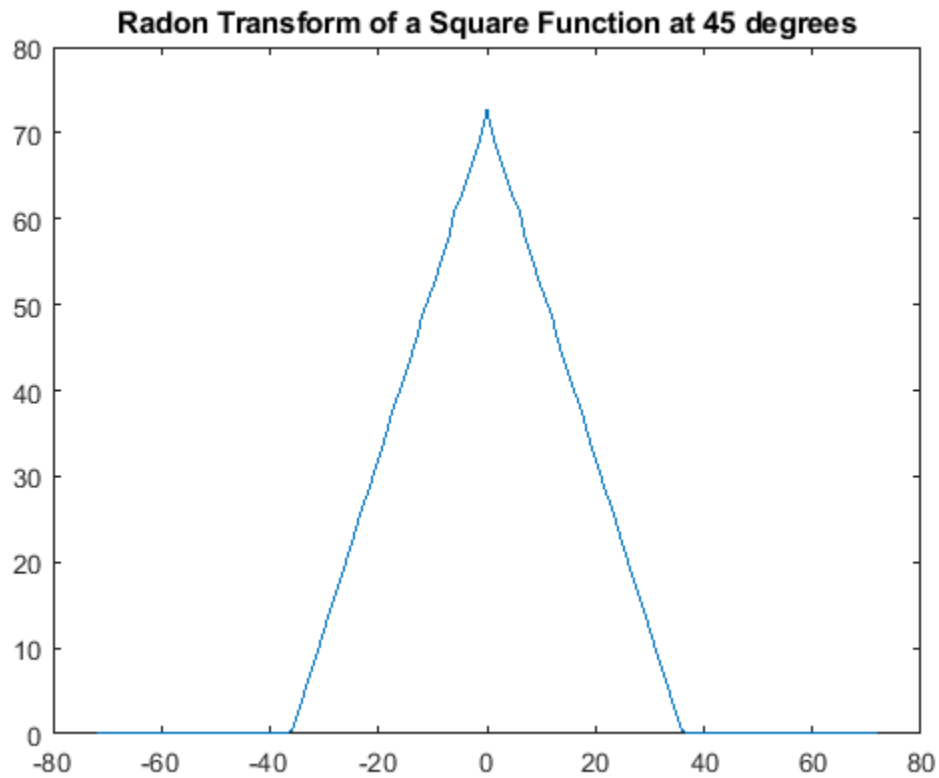

Viewing the Radon Transform as an Image

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
```
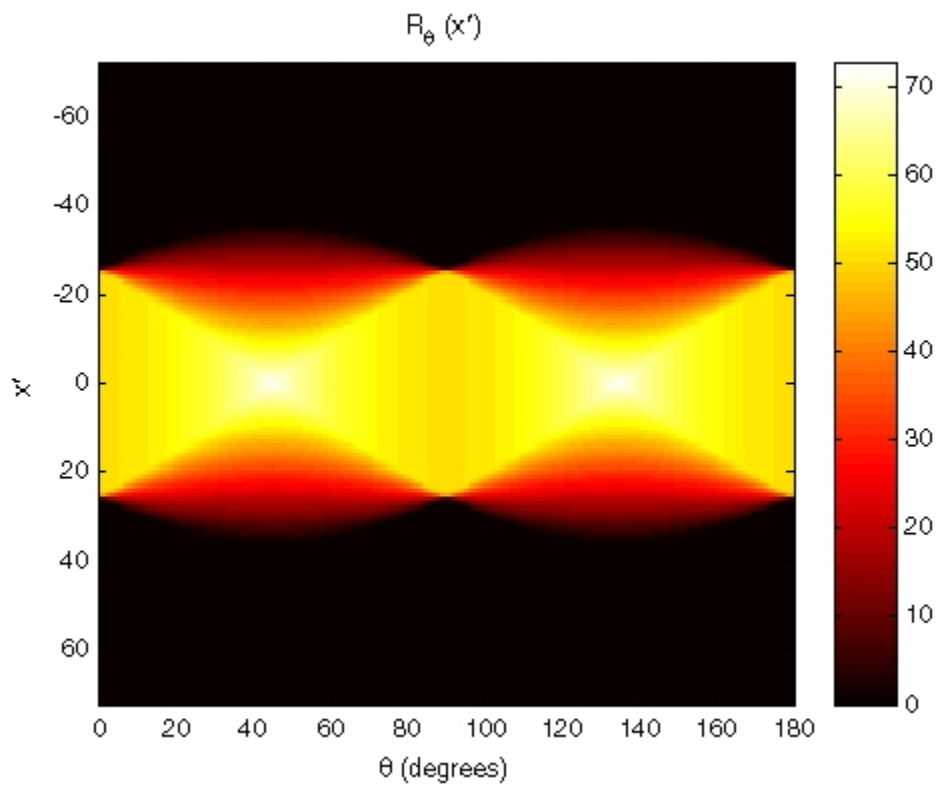
Radon Transform Using 180 Projections

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')
```
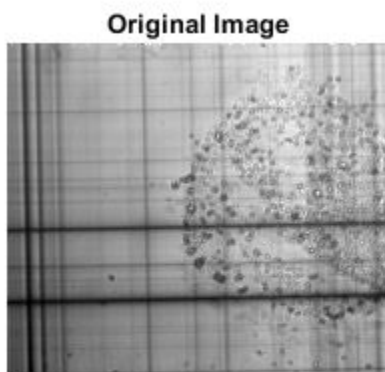

Original Image

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')
```

Edges of Original Image
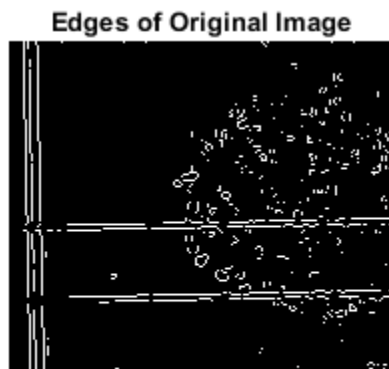
Calculate the radon transform of the image, using the `radon` function, and display the transform. The locations of peaks in the transform correspond to the locations of straight lines in the original image.

```
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
```

$R_\theta (x')$
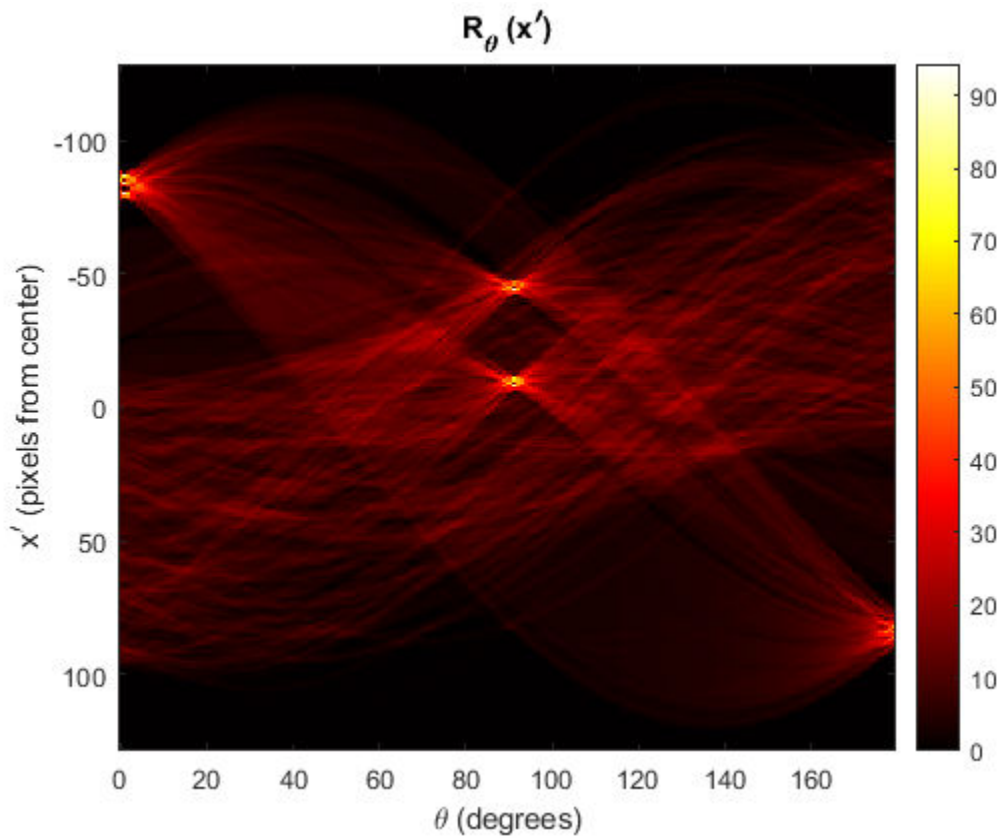
The strongest peak in R corresponds to θ = 1 degree and x' = -80 pixels from center.

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 θ = 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.
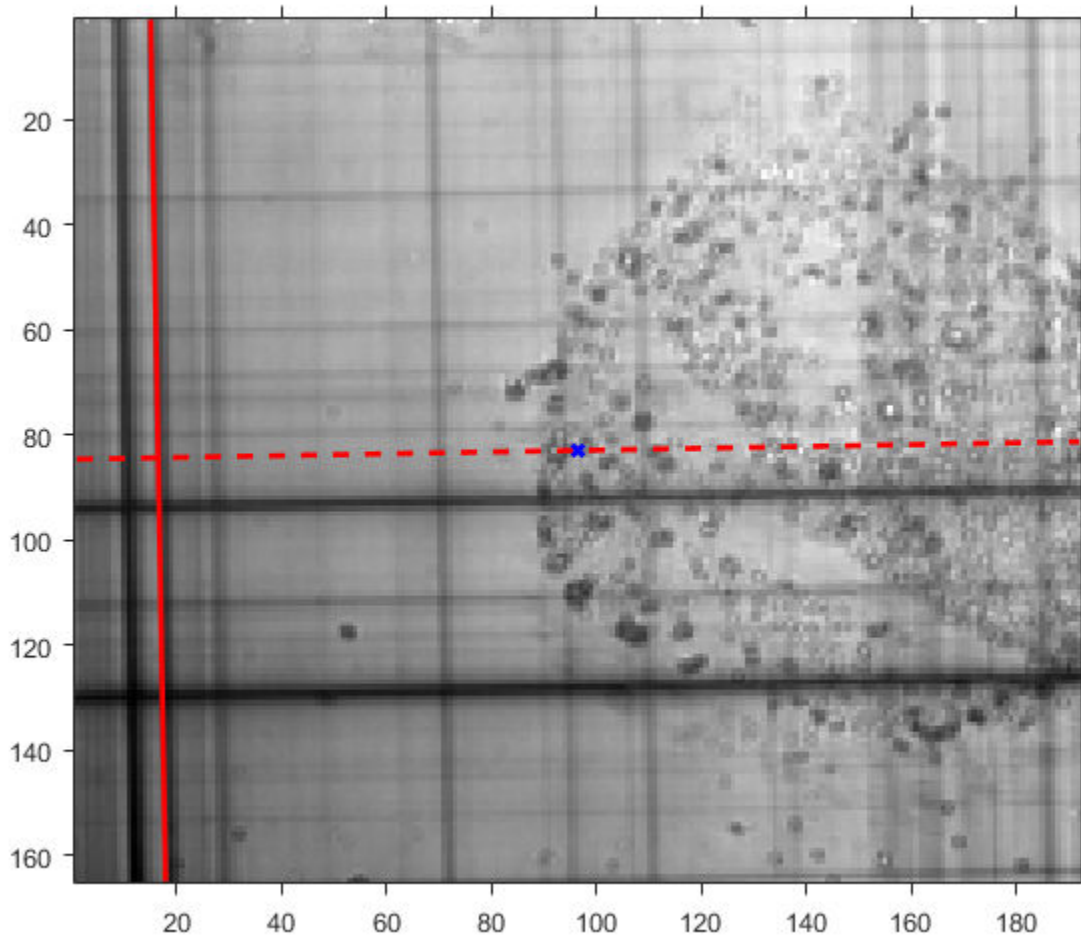
To interpret the Radon transform further, examine the next four strongest peaks in R.

Two strong peaks in R are found at θ = 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 θ = 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.
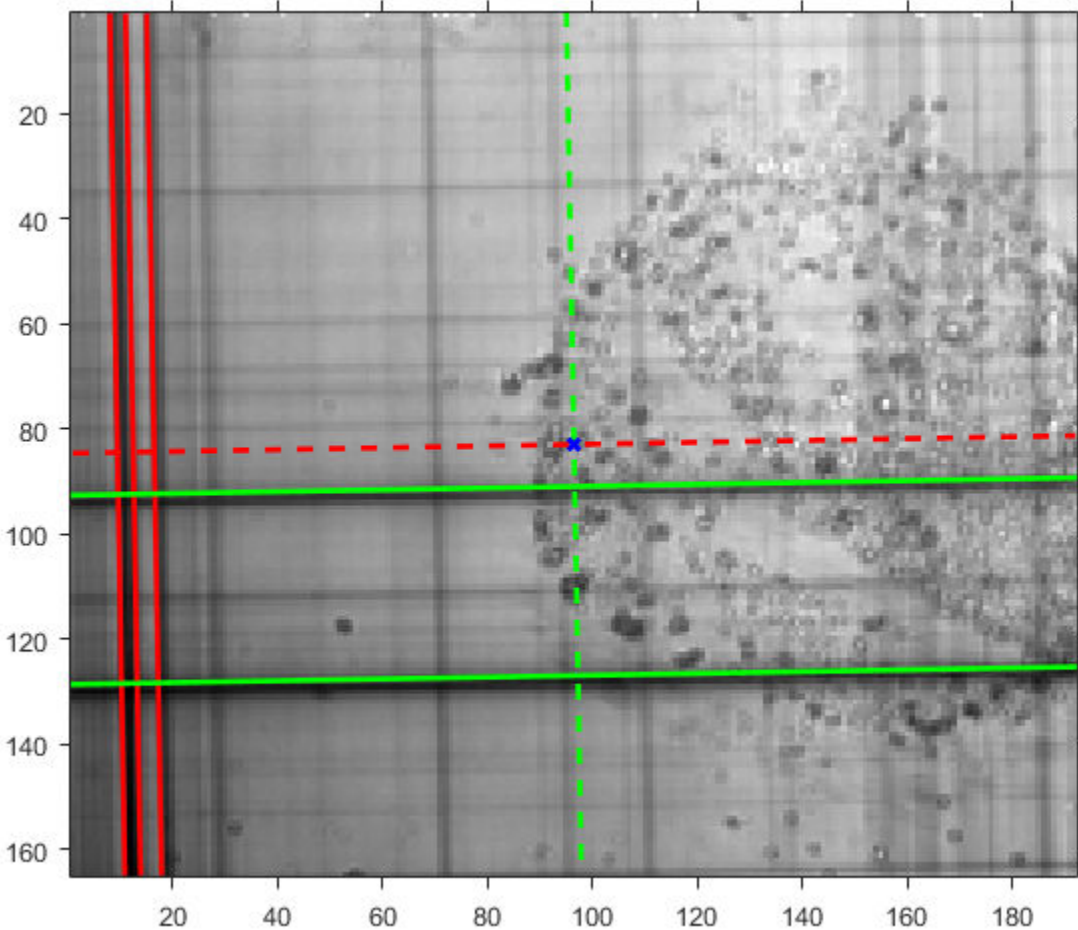
The fainter lines in the image relate to the weaker peaks in R.

The Inverse Radon Transformation

In this section...
"Inverse Radon Transform Definition" on page 9-32
"Reconstructing an Image from Parallel Projection Data" on page 9-34

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 9-21, 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 9-21. $f(x,y)$ denotes the brightness of the image and $R_\theta(x')$ is the projection at angle theta.
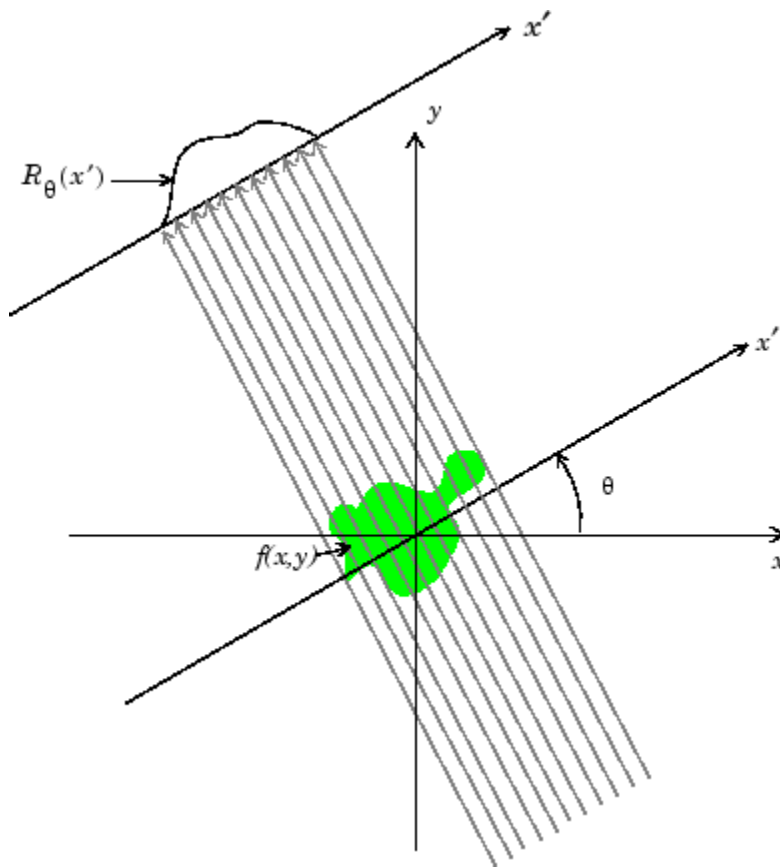
Parallel-Beam Projections Through an Object

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 9-37. 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');
```
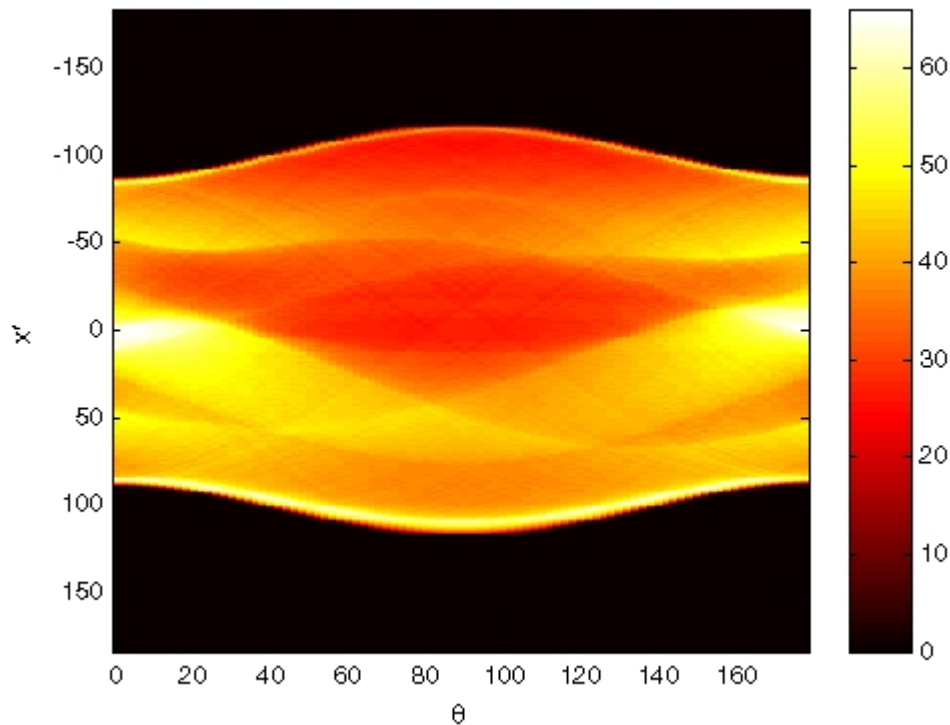
Radon Transform of Head Phantom Using 90 Projections

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.
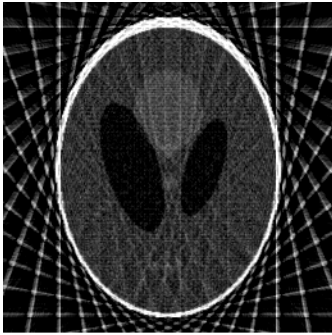
I1

I2

I3

Inverse Radon Transforms of the Shepp-Logan Head Phantom

Fan-Beam Projection

In this section...

Note For information about creating projection data from line integrals along parallel paths, see "Radon Transform" on page 9-21. 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.
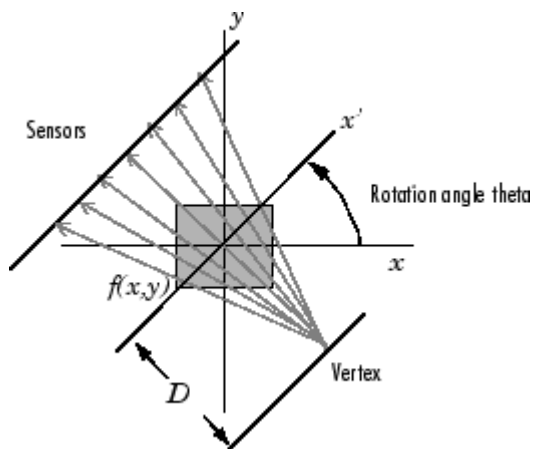
Fan-Beam Projection at Rotation Angle Theta

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.
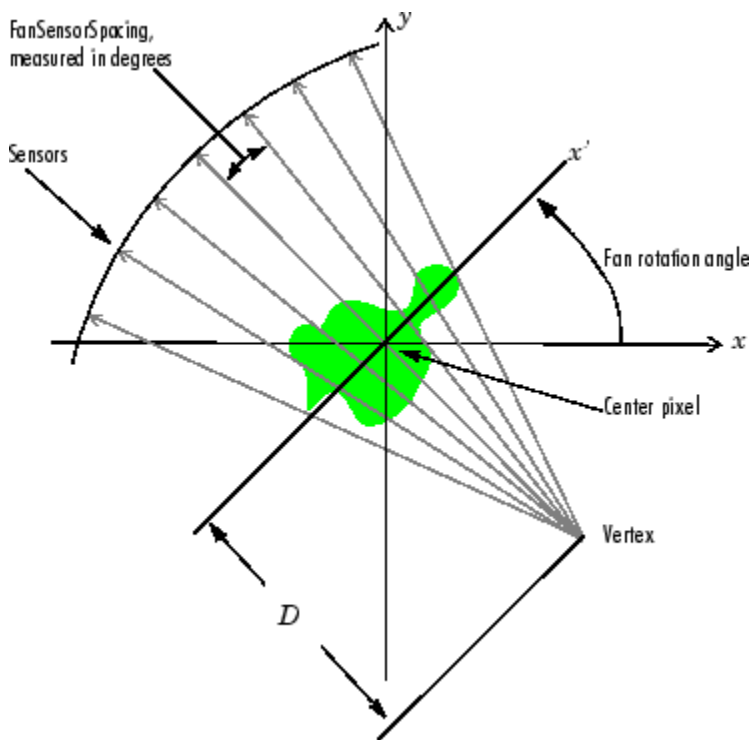
Fan-Beam Projection with Arc Geometry

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.
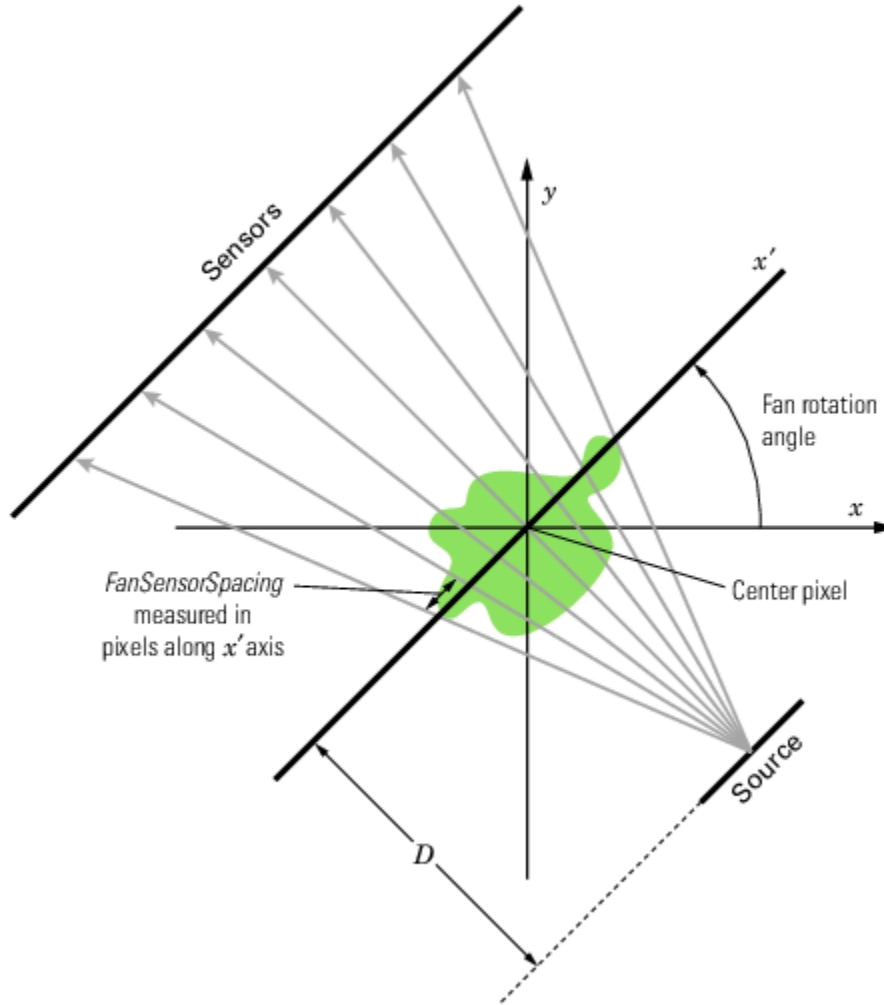
Fan-Beam Projection with Line Geometry

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 9-32 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)')
```
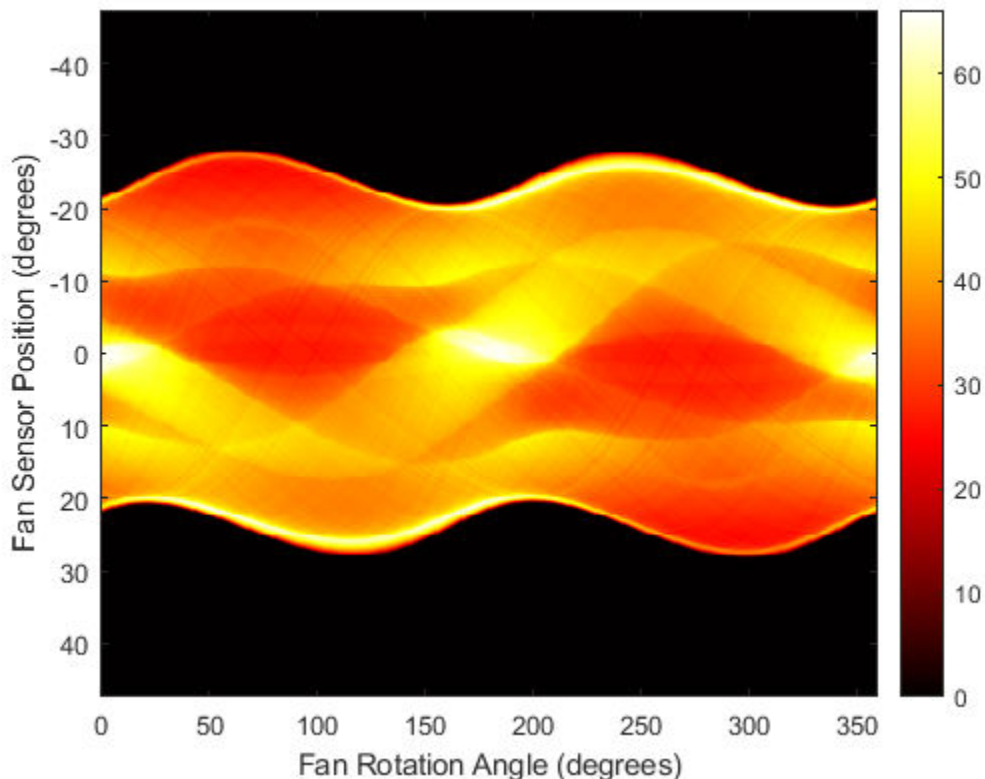

Reconstruct the image from the fan-beam projection data using ifanbeam . In each reconstruction, match the fan sensor spacing with the spacing used when the projection data was created previously. The example uses the OutputSize parameter to constrain the output size of each reconstruction to be the 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')
```
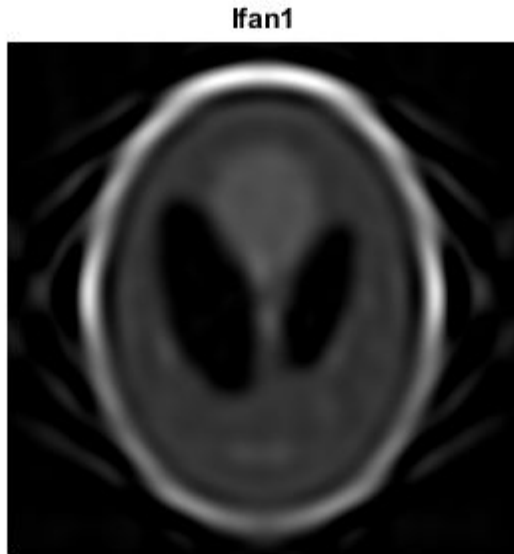

Ifan1

```
Ifan2 = ifanbeam(F2,D, ...
        'FanSensorSpacing',dsensor2,'OutputSize',output_size);
figure, imshow(Ifan2)
title('Ifan2')
```

Ifan2
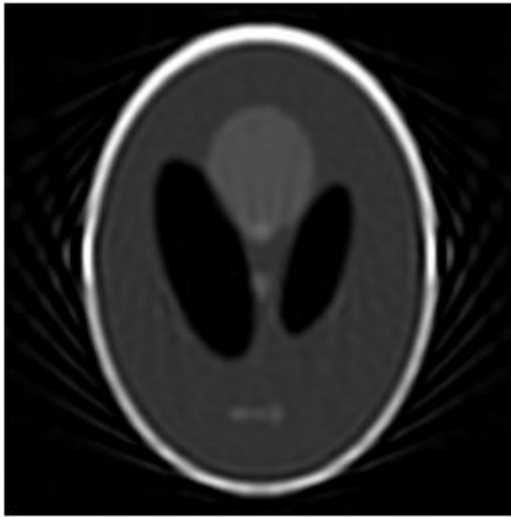
```
Ifan3 = ifanbeam(F3,D, ...
        'FanSensorSpacing',dsensor3,'OutputSize',output_size);
figure, imshow(Ifan3)
title('Ifan3')
```


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)');
```
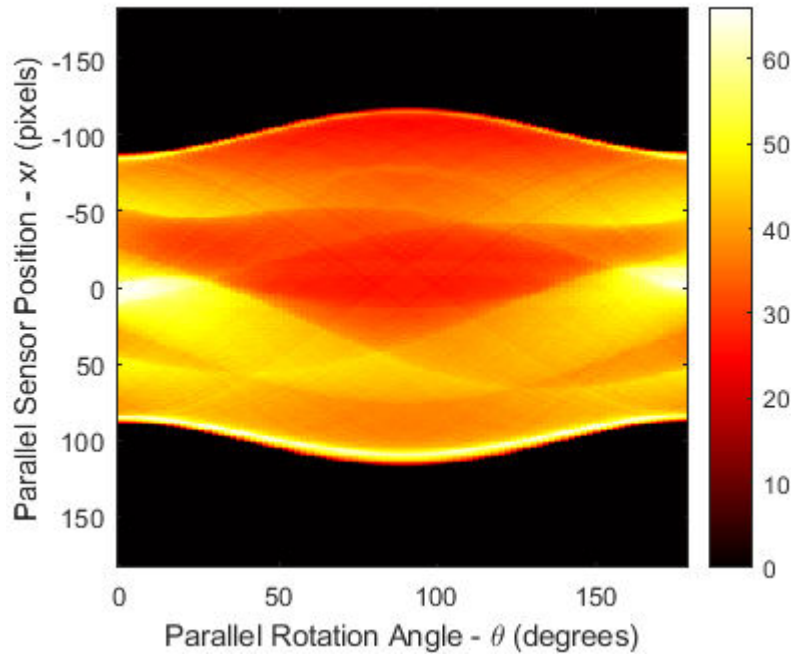
Parallel Beam - Reconstruct Head Phantom from Projection Data

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 Angles')
```

Reconstruction from Parallel Beam Projection with 18, 24, and 90 Projection Angles

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)')
```
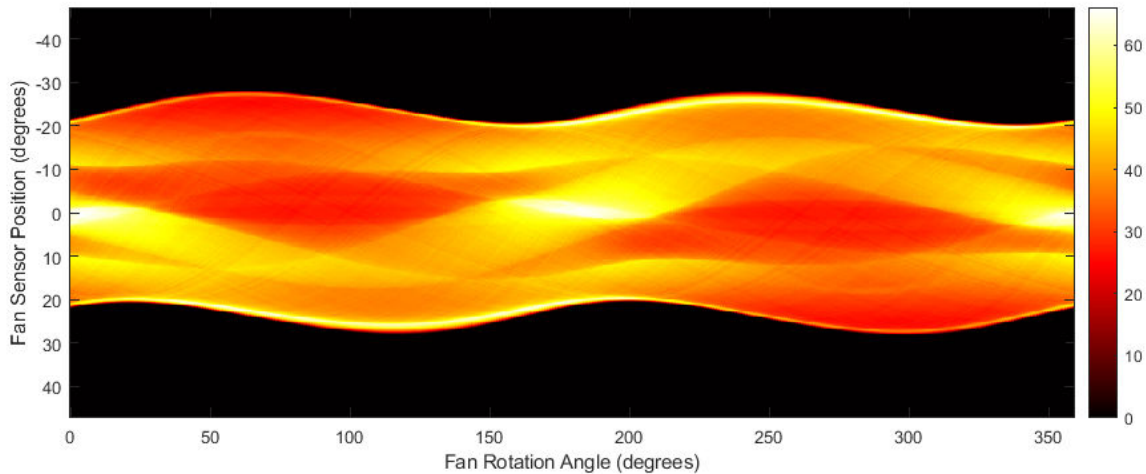
Fan Beam - Reconstruct Head Phantom from Projection Data

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')
```

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.

Types of Morphological Operations

In this section...
"Morphological Dilation and Erosion" on page 10-2
"Operations Based on Dilation and Erosion" on page 10-4

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.

Morphological Dilation and Erosion

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	Example (Original and Processed Image)
Dilation	The value of the output pixel [is the] *maximum* value of all pixels [in the] neighborhood. In a binary im[age, a] pixel is set to 1 if any of the [neighboring pixels have the [value 1.] Morphological dilation make[s objects] more visible and fills in smal[l holes in] objects.	
Erosion	The value of the output pixel [is the] *minimum* value of all pixels [in the] neighborhood. In a binary im[age, a] pixel is set to 0 if any of the [neighboring pixels have the [value 0.] Morphological erosion remo[ves] islands and small objects so [that only] substantive objects remain.	

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. 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. For more information, see "Structuring Elements" on page 10-9.
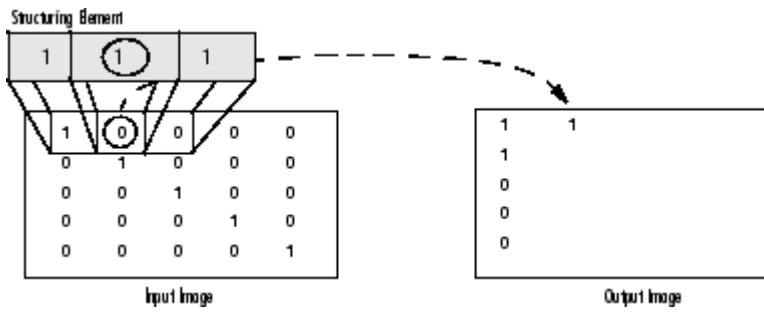
Morphological Dilation of a Binary Image

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.
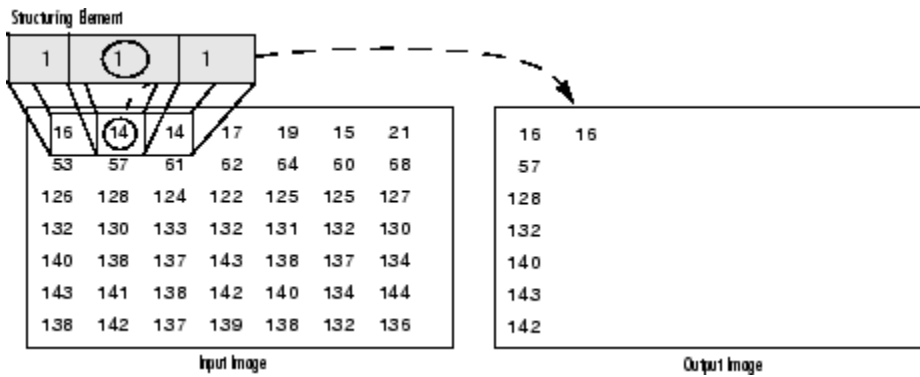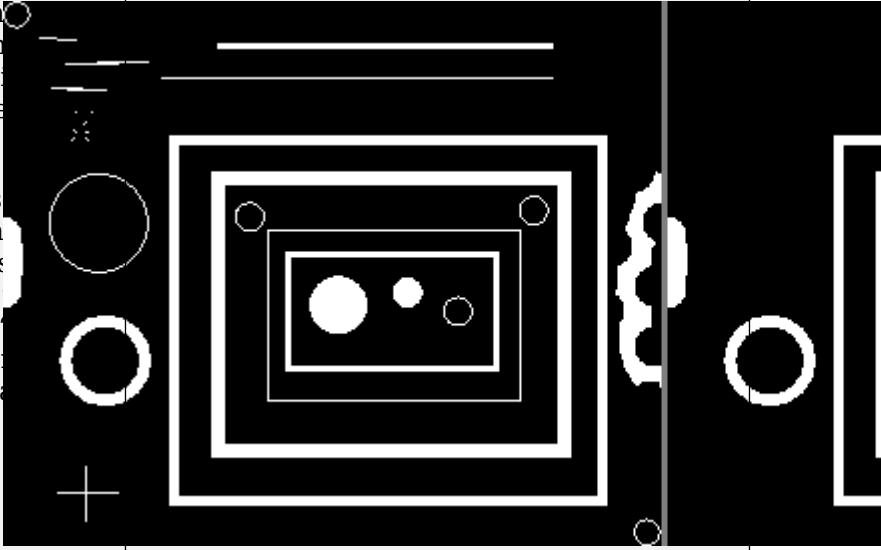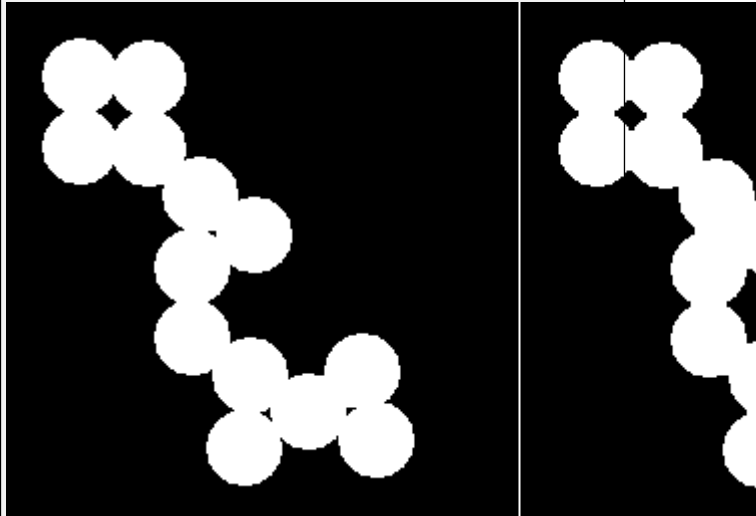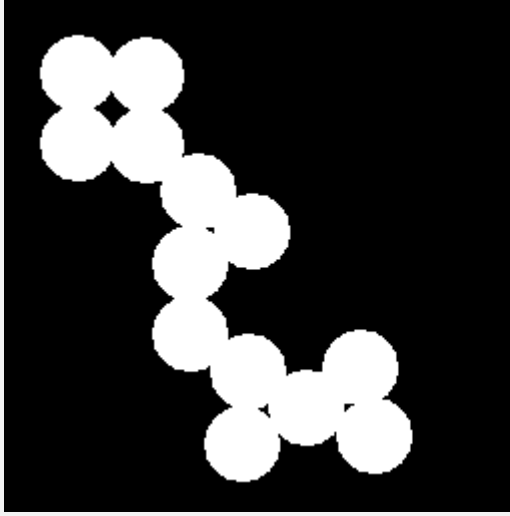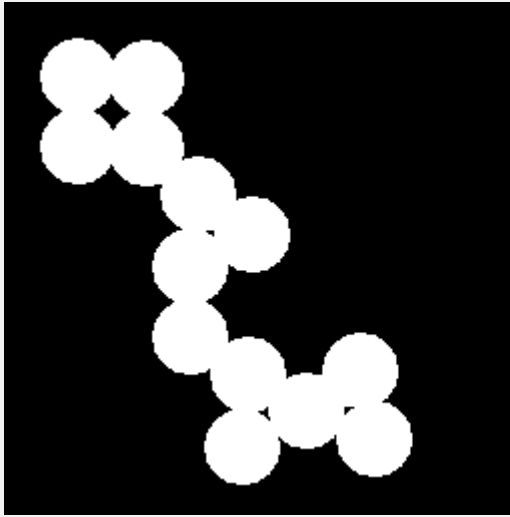
Morphological Dilation of a Grayscale Image
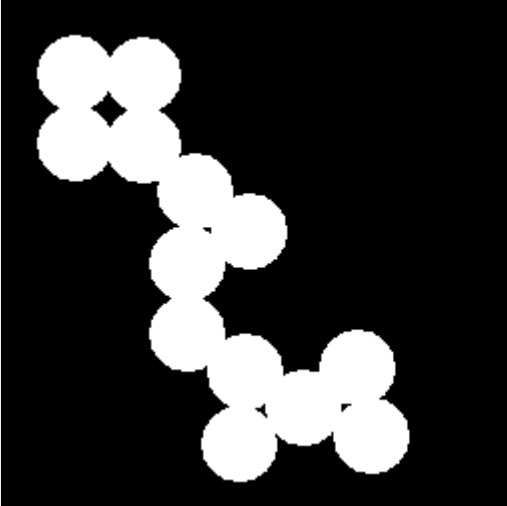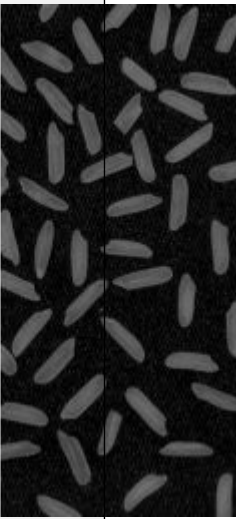
Operations Based on Dilation and Erosion

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. You can combine dilation and erosion to remove small objects from an image and smooth the border of large objects.

This table lists functions in the toolbox that perform common morphological operations that are based on dilation and erosion.

Function	Morphological Definition	Example (Original and Processed Image)
imopen	Perform morphological open opening operation erodes an and then dilates the eroded using the same structuring e for both operations. Morphological opening is us removing small objects from image while preserving the s and size of larger objects in image. For an example, see Morphological Opening to E Large Image Features" on pa 33.	
imclose	Perform morphological closing. The closing operation dilates an image and then erodes the dilated image, using the same structuring element for both operations. Morphological closing is useful for filling small holes from an image while preserving the shape and size of the objects in the image.	

Function	Morphological Definition	Example (Original and Processed Image)
bwskel	*Skeletonize* objects in a binary image. The process of skeletonization erodes all objects to centerlines without changing the essential structure of the objects, such as the existence holes and branches.	
bwperim	Find perimeter of objects in a binary image. A pixel is part of the perimeter if it is nonzero and it is connected to at least one zero-valued pixel.	

Function	Morphological Definition	Example (Original and Processed Image)
bwhitmiss	Perform binary hit-miss transform. The hit-miss transform preserves pixels in a binary image whose neighborhoods match the shape of one structuring element and do not match the shape of a second disjoint structuring element. The hit-miss transforms can be used to detect patterns in an image.	 This example uses one structuring element with a neighborhood above and to the right of center, and a second structuring element with a neighborhood below and to the left of center. The transform preserves pixels with neighbors only to the top and right.
imtophat	Perform a morphological top-hat transform. The top-hat transform opens an image, then subtracts the opened image from the original image. The top-hat transform can be used to enhance contrast in a grayscale image with nonuniform illumination. The transform can also isolate small bright objects in an image.	

Function	Morphological Definition	Example (Original and Processed Image)
imbothat	Perform morphological bottom-hat transform. The bottom-hat transform closes an image, then subtracts the original image form the closed image. The bottom-hat transform can be used to find intensity troughs in a grayscale image.	

See Also
imclose | imdilate | imerode | imopen

More About
- "Structuring Elements" on page 10-9
- "Pixel Connectivity" on page 10-22
- "Border Padding for Morphology" on page 10-13

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.
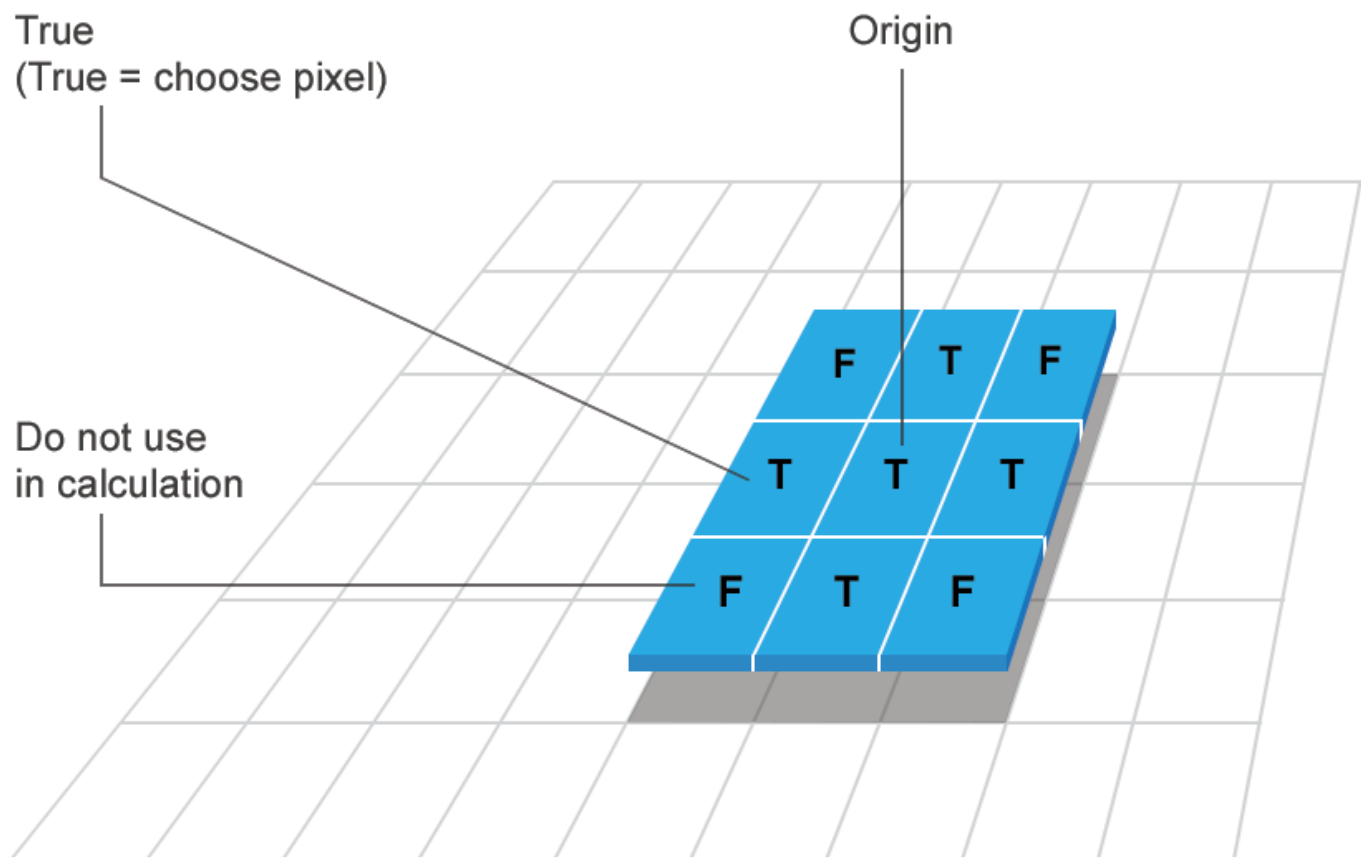
FLAT (binary valued)

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.
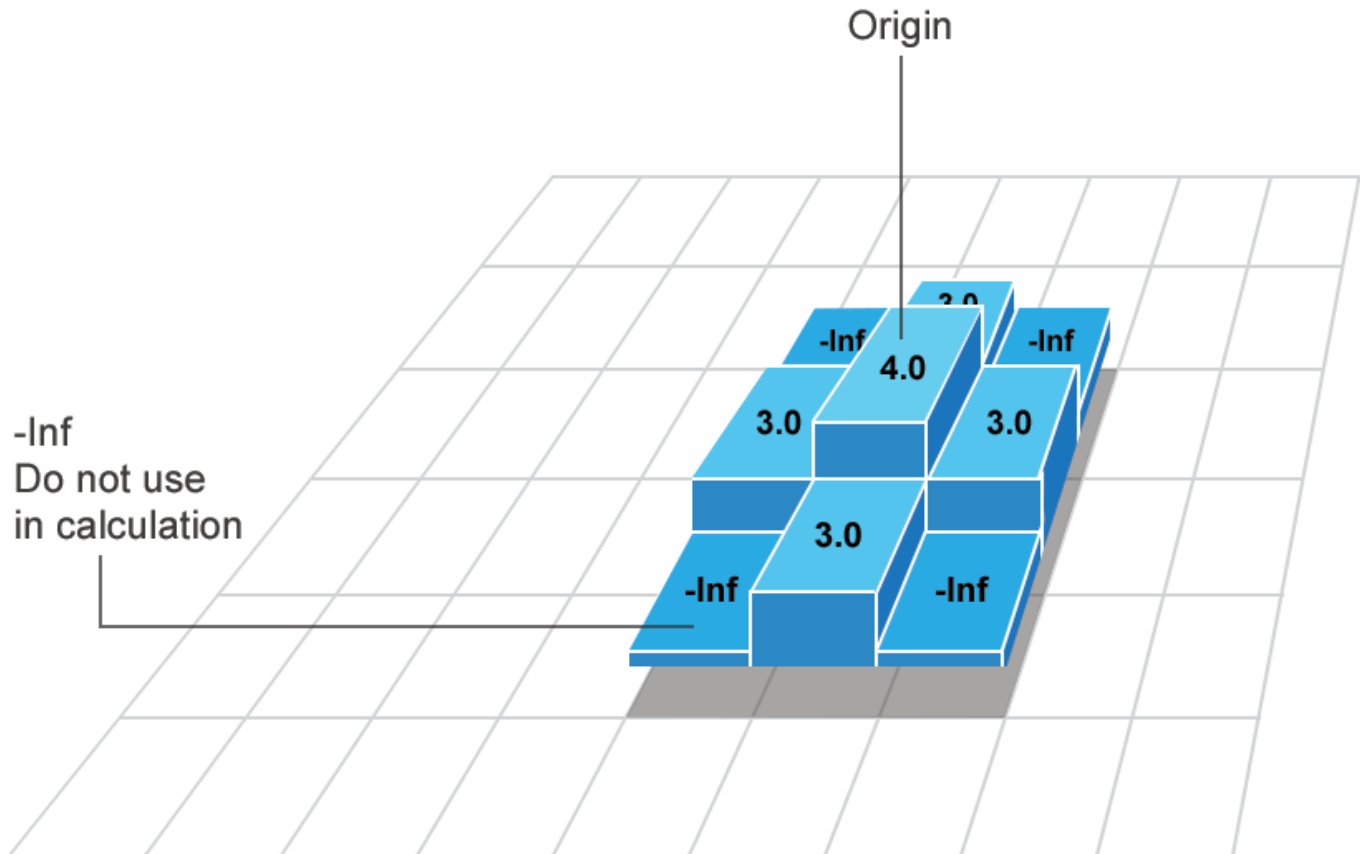
NON-FLAT (real valued)

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.
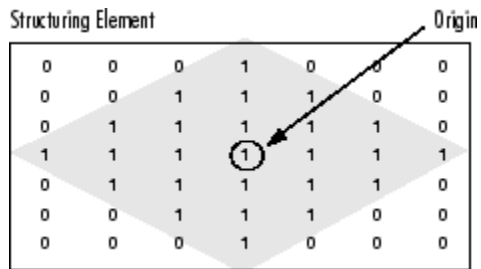
Structuring Element Decomposition

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
```

See Also
`offsetstrel` | `strel`

More About

Border Padding for Morphology

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 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.

See Also

imclose | imdilate | imerode | imopen | offsetstrel | strel

More About

- "Types of Morphological Operations" on page 10-2
- "Structuring Elements" on page 10-9

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.
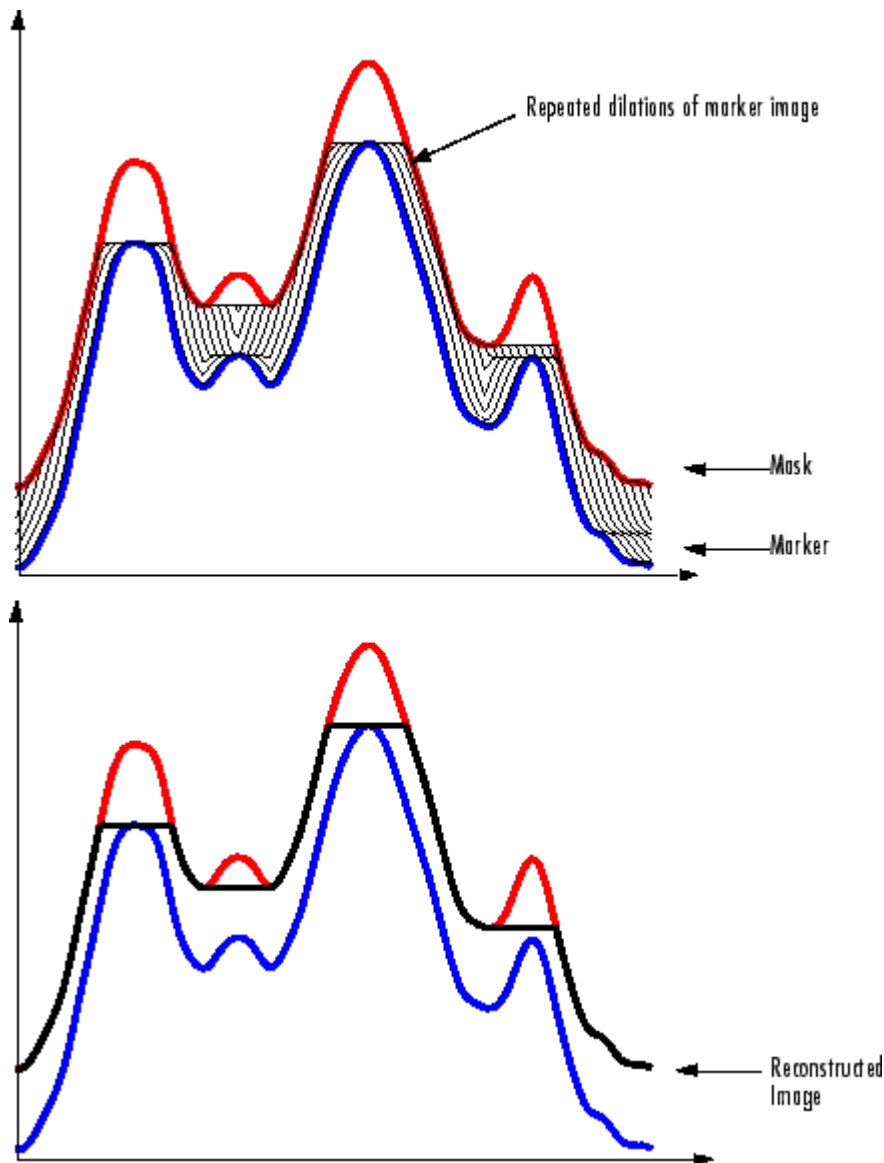
Repeated Dilations of Marker Image, Constrained by Mask

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 10-22, rather than a structuring element.
- Processing repeats until the image is stable and 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 10-16
- "Understanding the Maxima and Minima Functions" on page 10-17
- "Finding Areas of High or Low Intensity" on page 10-17
- "Suppressing Minima and Maxima" on page 10-18
- "Imposing a Minimum" on page 10-19

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*.

Term	Definition
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.
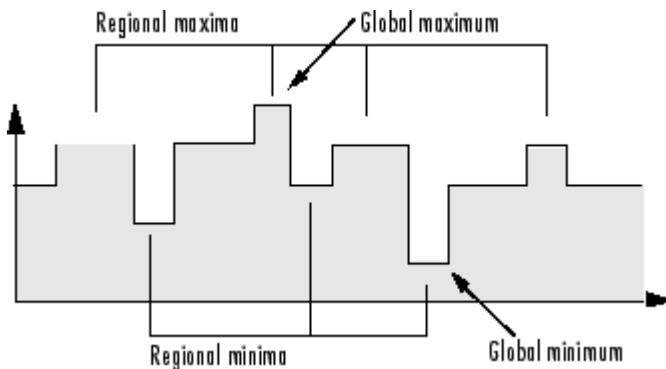
Finding Areas of High or Low Intensity

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.
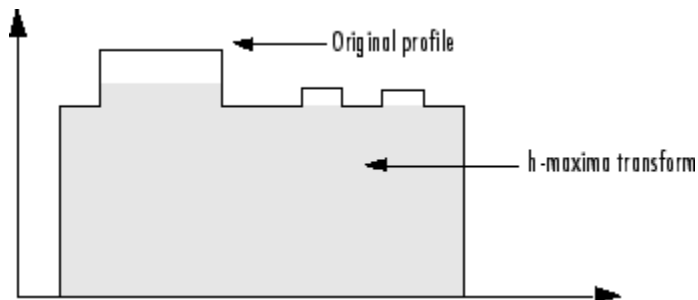
Imposing a Minimum

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.
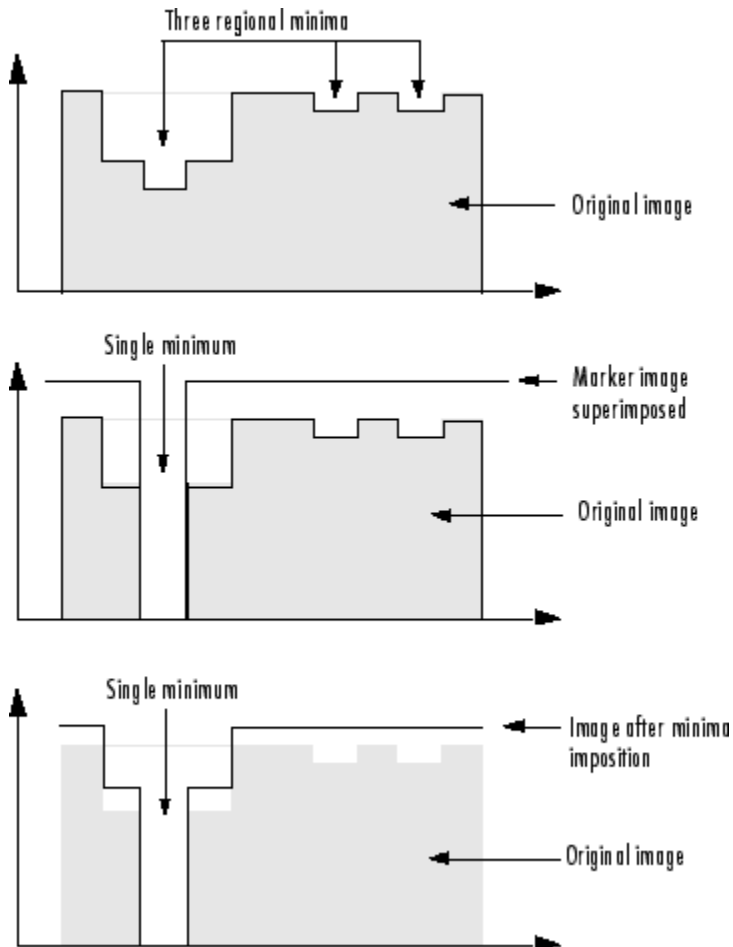
Imposing a Minimum

See Also

More About

- "Pixel Connectivity" on page 10-22
- "Flood-Fill Operations" on page 10-38

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*.
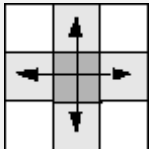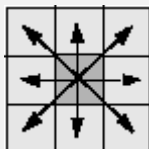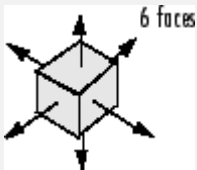
Determining which pixels create a connected component depends on how pixel connectivity is defined. For example, this binary image contains one foreground object or two, depending on the connectivity. If the foreground is 4-connected, the image is all one object — there is no distinction between a foreground 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.

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.	
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	

Value	Meaning	
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	6 faces + 12 edges
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	6 faces + 12 edges + 8 corners

Choosing a Connectivity

The type of neighborhood you choose affects the number of objects found in an image and the boundaries of those objects. For this reason, the results of many morphology operations often differ depending upon the type of connectivity you specify.

For example, if you specify a 4-connected neighborhood, this binary image contains two objects; if you specify an 8-connected neighborhood, the image has one object.

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

Specifying Custom Connectivities

You can also define custom neighborhoods by specifying a 3-by-3-by-...-by-3 array of 0s and 1s. 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
```

10-23

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

*   "Morphological Reconstruction" on page 10-14

Lookup Table Operations

In this section...
"Creating a Lookup Table" on page 10-25
"Using a Lookup Table" on page 10-25

Creating a Lookup Table

Certain binary image operations can be implemented most easily through lookup tables. A lookup table is a column vector in which each element represents the value to return for one possible combination of pixels in a neighborhood. To create lookup tables for various operations, use the `makelut` function. `makelut` creates lookup tables for 2-by-2 and 3-by-3 neighborhoods. The following figure illustrates these types of neighborhoods. Each neighborhood pixel is indicated by an x, and the center pixel is the one with a circle.

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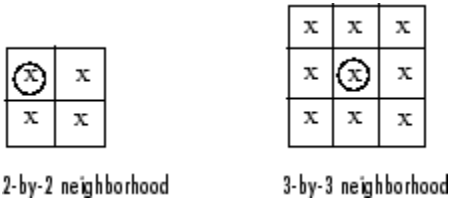
For a 2-by-2 neighborhood, there are 16 possible permutations of the pixels in the neighborhood. Therefore, the lookup table for this operation is a 16-element vector. For a 3-by-3 neighborhood, there are 512 permutations, so the lookup table is a 512-element vector.

Note `makelut` and `applylut` support only 2-by-2 and 3-by-3 neighborhoods. Lookup tables larger than 3-by-3 neighborhoods are not practical. For example, a lookup table for a 4-by-4 neighborhood would have 65,536 entries.

Using a Lookup Table

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

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

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

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

You then perform the operation using `applylut`.

```
BW1 = imread('text.png');
BW2 = applylut(BW1,lut);
```

10-25

```
imshow(BW1)
figure, imshow(BW2)
```
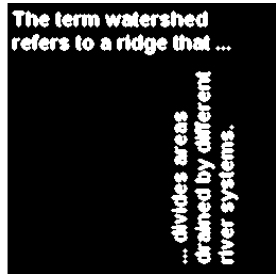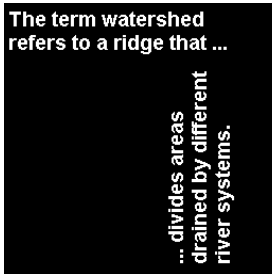

Image Before and After Applying Lookup Table Operation

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

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'))
```
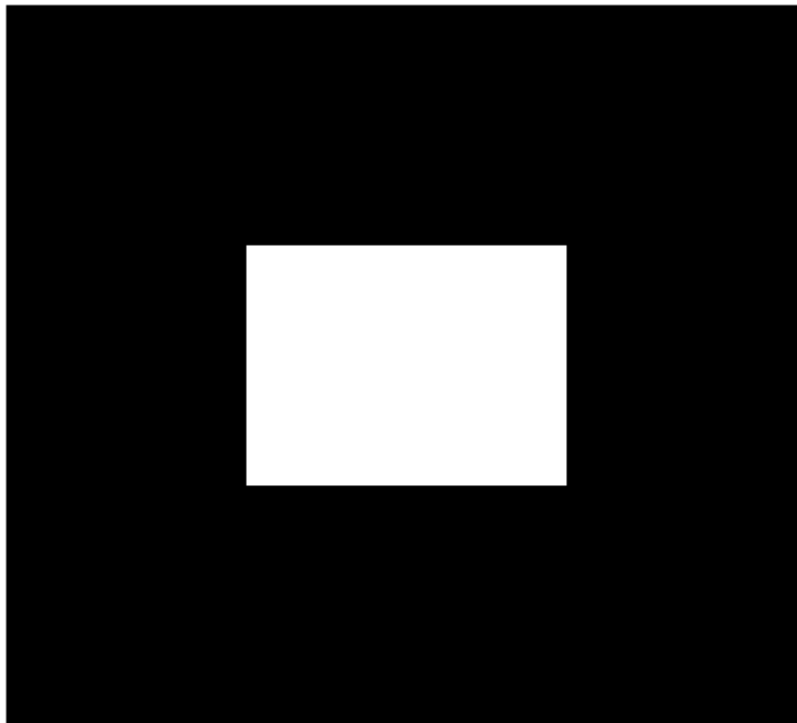
Create a structuring element to use with `imdilate`. To dilate a geometric object, you typically create a structuring element that is the same shape as the object.

```
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'))
```
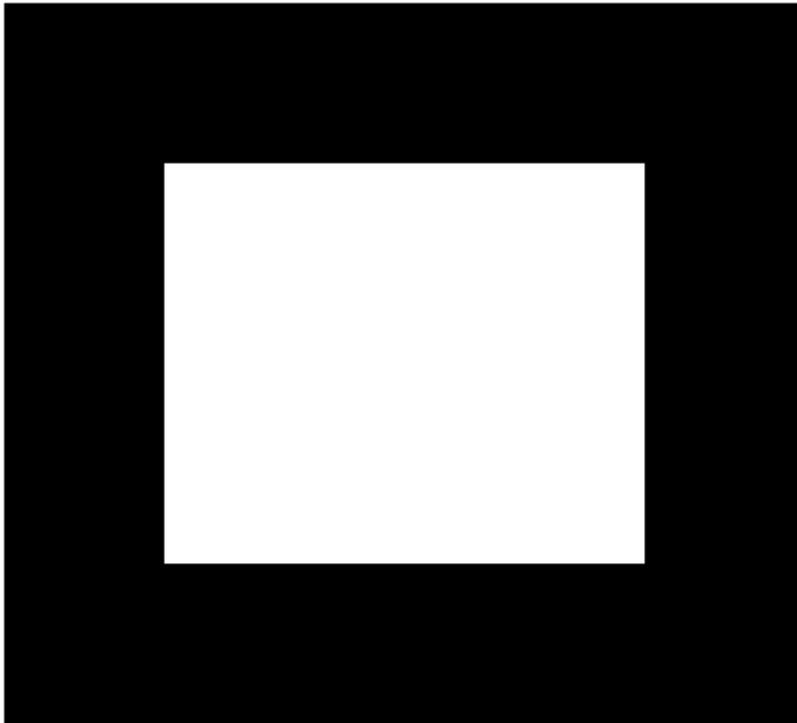
For comparison, create a structuring element that is a different shape. Dilate the original image using the new structuring element.

```
SE2 = strel('diamond',1);
BW3 = imdilate(BW,SE2);
imshow(imresize(BW3,40,'nearest'))
```
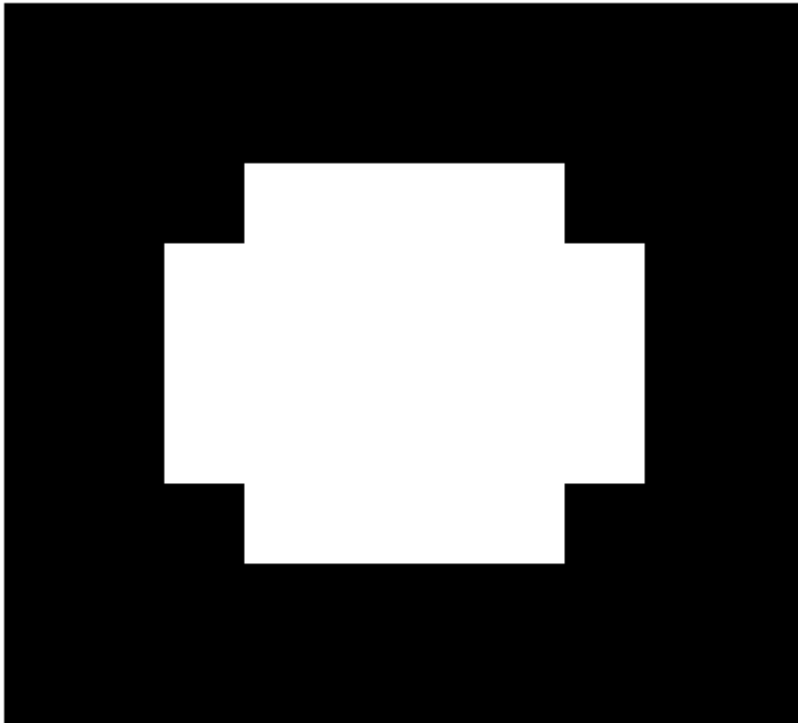
Remove Thin Lines Using Erosion

This example shows how to remove thin lines in a binary image by using morphological erosion with a neighborhood larger than the width of the lines.

Read and display a binary image. The white lines that represent wires are approximately four or five pixels wide. In some places, the wires are touching and the overall width is closer to ten or eleven pixels.

```
BW1 = imread('circbw.tif');
imshow(BW1)
```


Define a neighborhood larger than the width of the lines. This example uses a disk-shaped structuring element with a radius of 7 pixels so that the overall neighborhood size is 13-by-13 pixels.

```
SE = strel("disk",7)

SE =
strel is a disk shaped structuring element with properties:

      Neighborhood: [13x13 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 eroded image.

```
imshow(BW2)
```

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 circuit board 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 1s, 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);
```
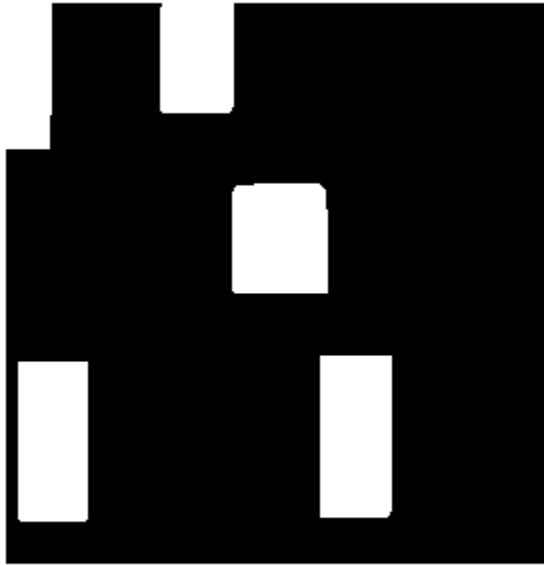
Open an Image By Performing Erosion Then Dilation

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)
```
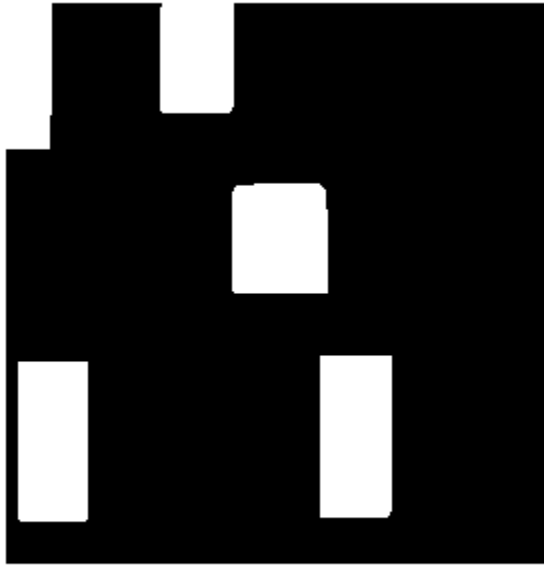
By performing the operations sequentially, you have the flexibility to change the structuring element. Create a different structuring element, and dilate the eroded image using the new structuring element.

```
SE = strel('diamond',15);
BW5 = imdilate(BW3,SE);
imshow(BW5)
```
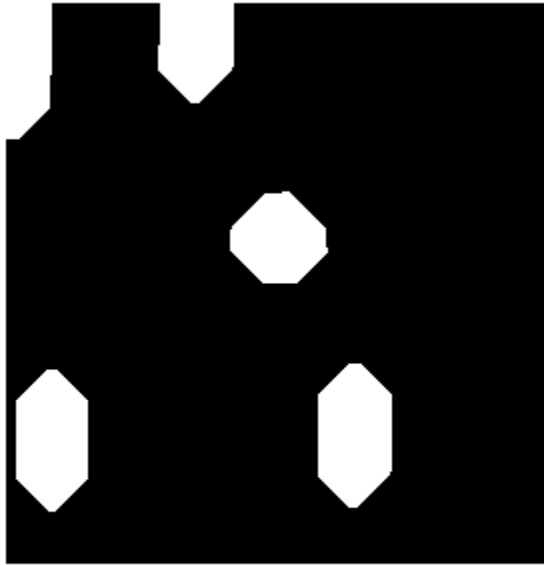
See Also
imclose | imdilate | imerode | imopen | strel

More About
- "Types of Morphological Operations" on page 10-2

Flood-Fill Operations

The imfill function performs a *flood-fill* operation on binary and grayscale images. This operation can be useful in removing irrelevant artifacts from images.

- For binary images, imfill changes connected background pixels (0s) to foreground pixels (1s), 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. For more information, see "Finding Areas of High or Low Intensity" on page 10-17 for more information.

Specifying Connectivity

For both binary and grayscale images, the boundary of the fill operation is determined by the pixel connectivity on page 10-22 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]);
```
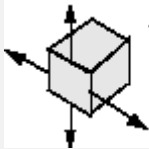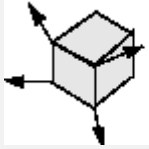
If the background is 4-connected, this binary image contains two separate background elements (the part inside the loop and the part outside). If the background is 8-connected, the pixels connect diagonally, and there is only one background element.

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)
```
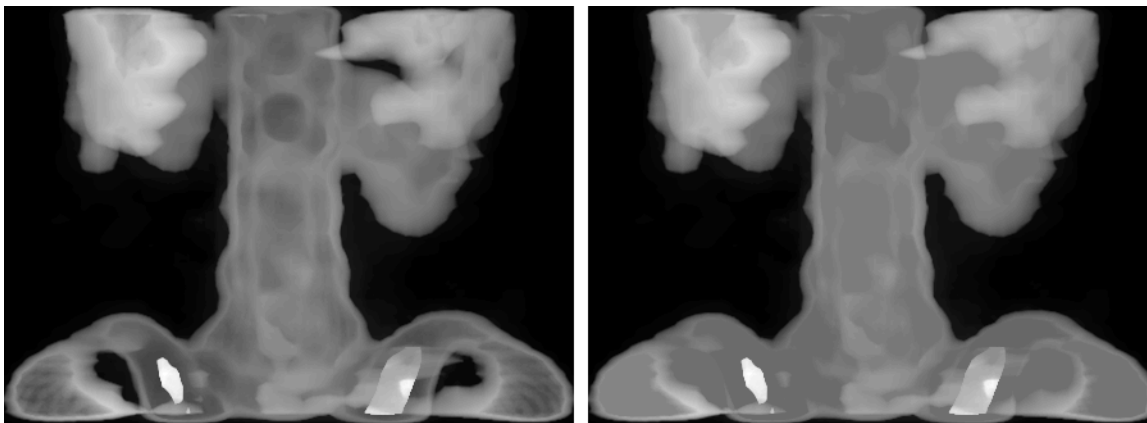

Original   After Filling Holes

See Also

`imfill`

More About

* "Morphological Reconstruction" on page 10-14

Detect Cell Using Edge Detection and Morphology

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.

Step 1: Read Image

Read in the `cell.tif` image, which is an image of a prostate cancer cell. Two cells are present in this image, but only one cell can be seen in its entirety. The goal is to detect, or segment, the cell that is completely visible.

```matlab
I = imread('cell.tif');
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');
```


Original Image

Image courtesy of Alan Partin
Johns Hopkins University

Step 2: Detect Entire Cell

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. To create a binary mask containing the segmented cell, calculate the gradient image and apply a threshold.

Use `edge` and the Sobel operator to calculate the threshold value. Tune the threshold value and use `edge` again to obtain a binary mask that contains the segmented cell.

```matlab
[~,threshold] = edge(I,'sobel');
fudgeFactor = 0.5;
BWs = edge(I,'sobel',threshold * fudgeFactor);
```

Display the resulting binary gradient mask.

```matlab
imshow(BWs)
title('Binary Gradient Mask')
```

10-41

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, there are 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. Create two perpindicular linear structuring elements by using `strel` function.

```
se90 = strel('line',3,90);
se0 = strel('line',3,0);
```

Dilate the binary gradient mask using the vertical structuring element followed by the horizontal structuring element. The `imdilate` function dilates the image.

```
BWsdil = imdilate(BWs,[se90 se0]);
imshow(BWsdil)
title('Dilated Gradient Mask')
```

Dilated Gradient Mask
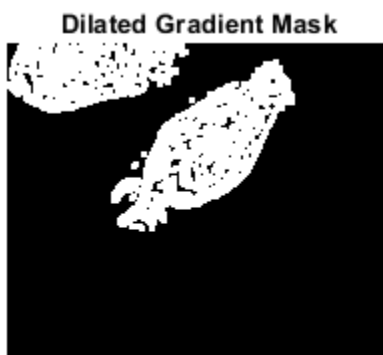
Step 4: Fill Interior Gaps

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, use the `imfill` function.

```
BWdfill = imfill(BWsdil,'holes');
imshow(BWdfill)
title('Binary Image with Filled Holes')
```

Binary Image with Filled Holes
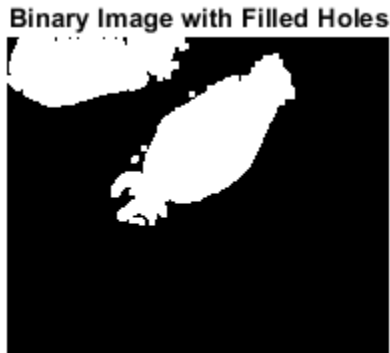
Step 5: Remove Connected Objects on Border

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. To remove diagonal connections, set the connectivity in the `imclearborder` function to 4.

```
BWnobord = imclearborder(BWdfill,4);
imshow(BWnobord)
title('Cleared Border Image')
```

Cleared Border Image

10-43
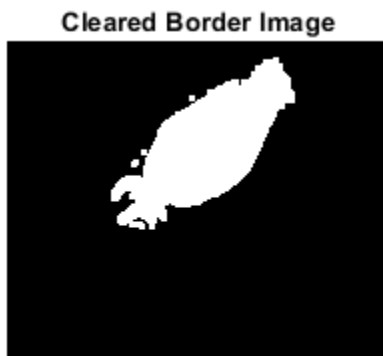
Step 6: Smooth the Object

Finally, in order to make the segmented object look natural, smooth the object by eroding the image twice with a diamond structuring element. Create the diamond structuring element using the `strel` function.

```
seD = strel('diamond',1);
BWfinal = imerode(BWnobord,seD);
BWfinal = imerode(BWfinal,seD);
imshow(BWfinal)
title('Segmented Image');
```
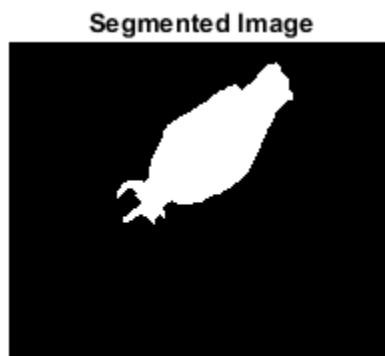

Segmented Image

Step 7: Visualize the Segmentation

You can use the `labeloverlay` function to display the mask over the original image.

```
imshow(labeloverlay(I,BWfinal))
title('Mask Over Original Image')
```


Mask Over Original Image
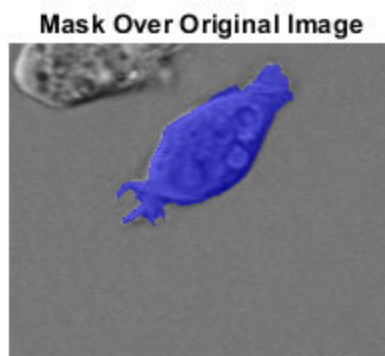
An alternate method to display the segmented object is to draw an outline around the segmented cell. Draw an outline by using the `bwperim` function.

```
BWoutline = bwperim(BWfinal);
Segout = I;
Segout(BWoutline) = 255;
imshow(Segout)
title('Outlined Original Image')
```

Outlined Original Image
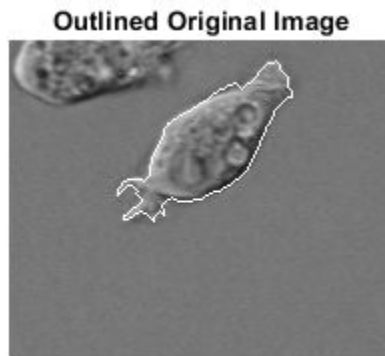
See Also

bwperim | edge | imclearborder | imdilate | imerode | imfill | strel

More About

*   "Types of Morphological Operations" on page 10-2

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)
```
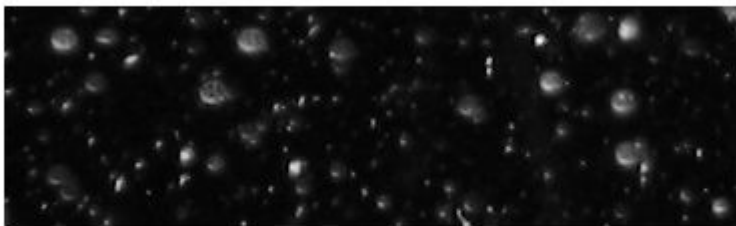

Enhance Contrast

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)
```
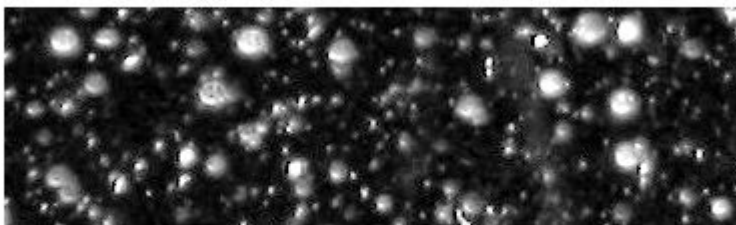

Determine Intensity Surface Area Distribution in Enhanced Image

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.

```matlab
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)')
```
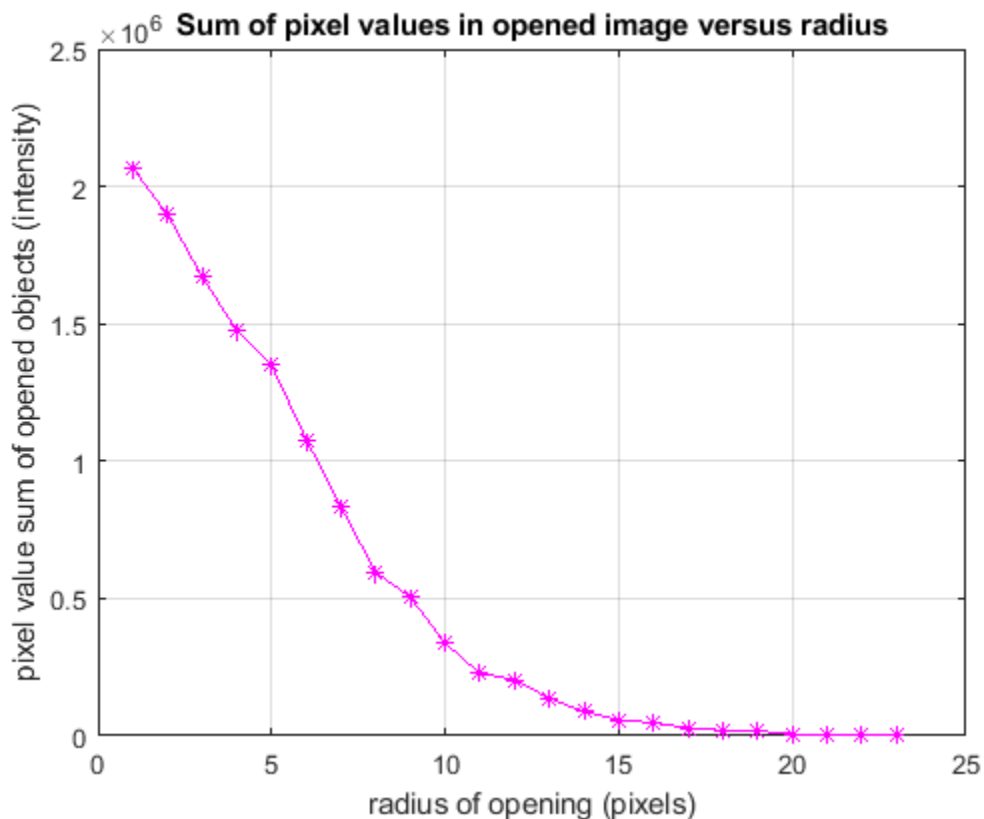

Calculate First Derivative of Distribution

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')
```
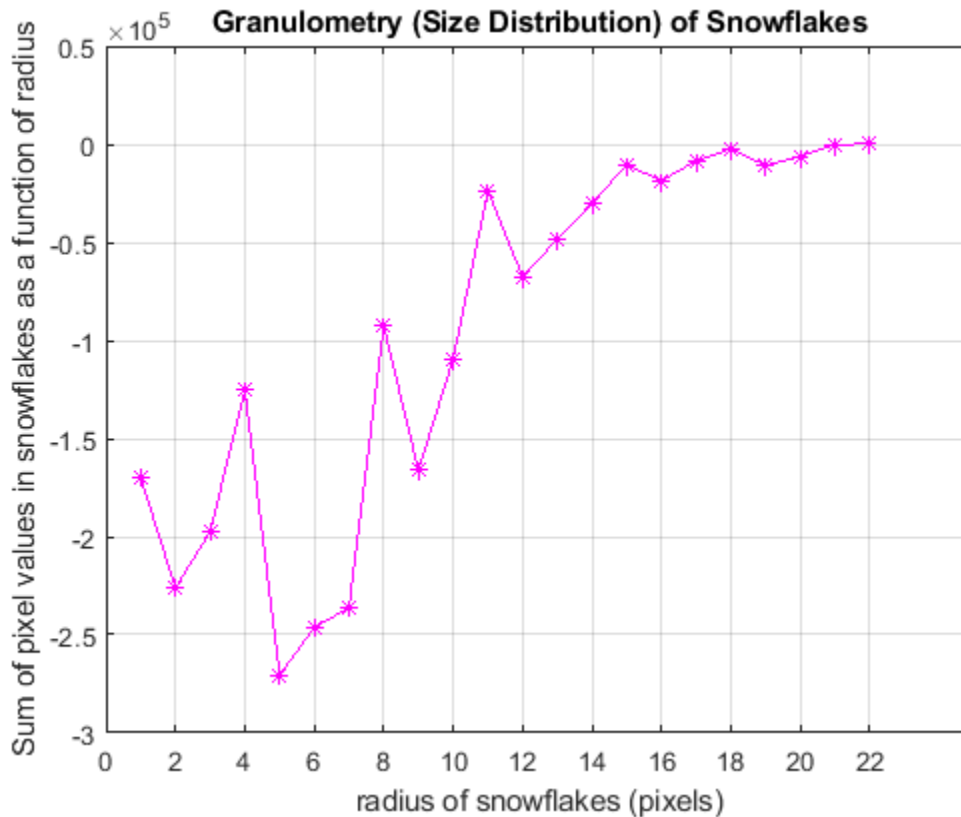

Extract Snowflakes Having a Particular Radius

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,[])
```
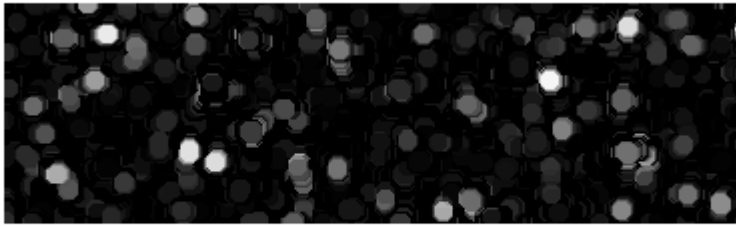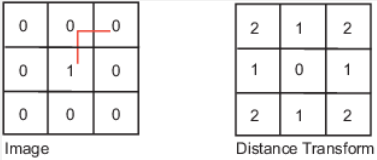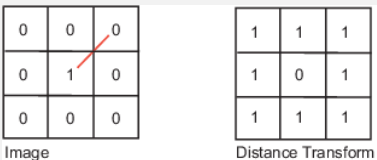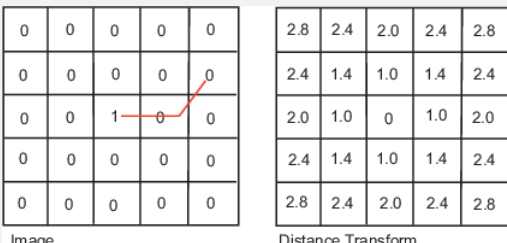
See Also
adapthisteq | imadjust | imopen | imsubtract | strel

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.

Distance Metrics

Distance Metric	Description	Illustration
Euclidean	The Euclidean distance is the straight-line distance between two pixels.	
City Block	The city block distance metric measures the path between the pixels based on a 4-connected neighborhood. Pixels whose edges touch are 1 unit apart; pixels diagonally touching are 2 units apart.	
Chessboard	The chessboard distance metric measures the path between the pixels based on an 8-connected neighborhood. Pixels whose edges or corners touch are 1 unit apart.	
Quasi-Euclidean	The quasi-Euclidean metric measures the total Euclidean distance along a set of horizontal, vertical, and diagonal line segments.	

This example creates a binary image containing two intersecting circular objects.

```
center1 = -10;
center2 = -center1;
dist = sqrt(2*(2*center1)^2);
radius = dist/2 * 1.4;
lims = [floor(center1-1.2*radius) ceil(center2+1.2*radius)];
[x,y] = meshgrid(lims(1):lims(2));
bw1 = sqrt((x-center1).^2 + (y-center1).^2) <= radius;
bw2 = sqrt((x-center2).^2 + (y-center2).^2) <= radius;
bw = bw1 | bw2;
figure
imshow(bw)
```

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,[])
```

Label and Measure Connected Components in a Binary Image

In this section...
"Detect Connected Components" on page 10-52
"Label Connected Components" on page 10-53
"Select Objects in a Binary Image" on page 10-54
"Measure Properties of Connected Components" on page 10-54

Detect Connected Components

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.
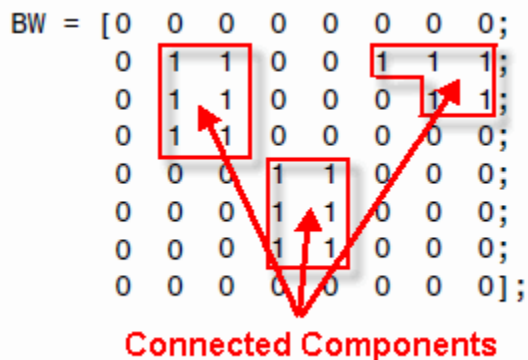
Calculate connected components by using `bwconncomp`. In this sample code, `BW` is the binary matrix shown in the above image. Specify a connectivity of 4 so that two adjoining pixels are part of the same object if they are both on and are connected along the horizontal or vertical direction. The `PixelIdxList` field identifies the list of pixels belonging to each connected component.

```
BW = zeros(8,9);
BW(2:4,2:3) = 1;
BW(5:7,4:5) = 1;
BW(2,7:9) = 1;
BW(3,8:9) = 1;
BW

BW =

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

cc = bwconncomp(BW,4)

cc =
```

```
   Connectivity: 8
      ImageSize: [8 9]
     NumObjects: 3
   PixelIdxList: {[6x1 double]  [6x1 double]  [5x1 double]}
```

Label Connected Components

Connected component labeling is the process of identifying the connected components in an image and assigning each one a unique label. The resulting matrix is called a *label matrix*.

Labeled Connected Components

Create a label matrix by using the `labelmatrix` function. This sample code continues with the connected component structure, `cc`, defined in the preceding section.

```
labeled = labelmatrix(cc)

labeled =

  8×9 uint8 matrix

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

To visualize connected components, display the label matrix as a pseudo-color image by using the `label2rgb` function. The label identifying each object in the label matrix maps to a different color in the associated colormap. You can specify the colormap, 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)
```

10-53
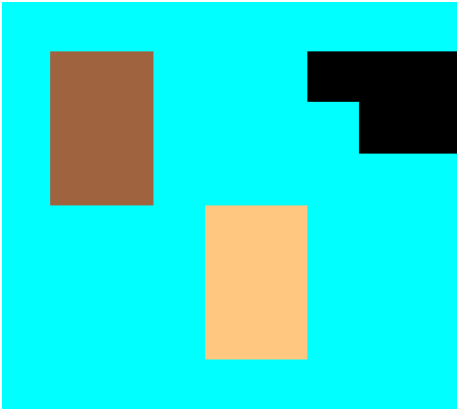
Select Objects in a Binary Image

You can use the `bwselect` function to select individual objects in a binary image. Specify pixels in the input image programmatically or interactively with a mouse. `bwselect` returns a binary image that includes only those objects from the input image that contain one of the specified pixels.

For example, use this command to select objects in the image displayed in the current axes.

```
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.

Measure Properties of Connected Components

The `regionprops` function can return measurements for several properties of connected components. Other functions measure a single property. For example, the `bwarea` function returns the area of a binary image.

This example uses `bwarea` to determine the percentage area increase in `circbw.tif` that results from a dilation operation. The area is a measure of the size of the foreground of the image and is roughly equal to the number of `on` pixels in the image. However, `bwarea` does not simply count the number of pixels set to `on`. 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.

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


increase =

    0.3456
```

See Also

bwconncomp | bwselect | label2rgb | labelmatrix | regionprops

Related Examples

- "Calculate Properties of Image Regions Using Image Region Analyzer" on page 12-28
- "Correct Nonuniform Illumination and Analyze Foreground Objects" on page 1-9

More About

- "Pixel Connectivity" on page 10-22

Analyzing and Enhancing Images

This topic describes functions that support a range of standard image processing operations for analyzing and enhancing images.

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 4-26.

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
```
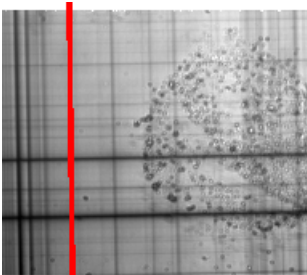

Image Courtesy of Ann Walker

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.
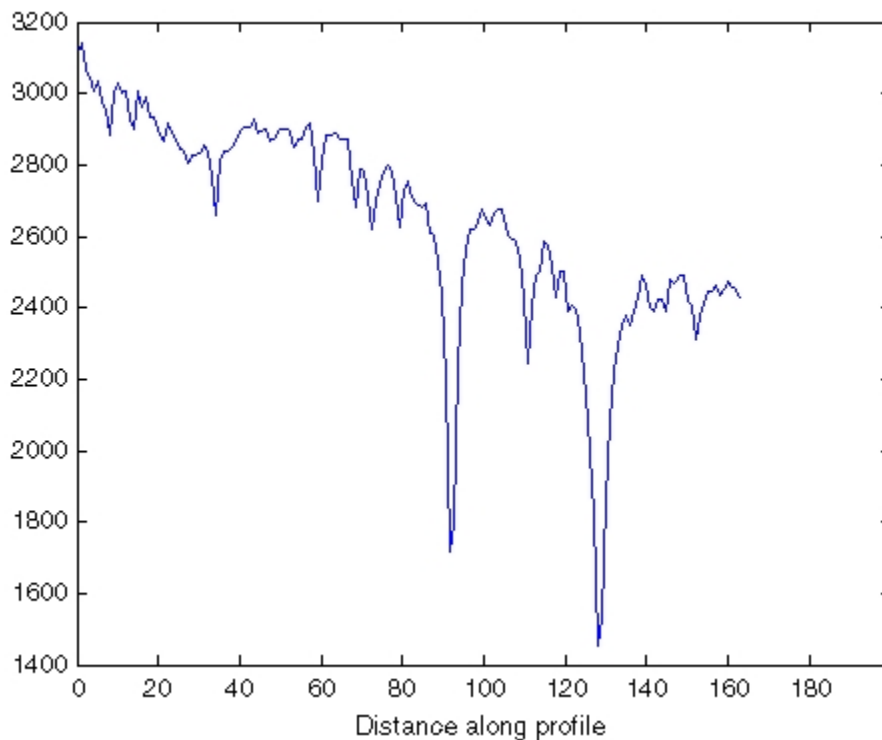
Create Intensity Profile of an RGB Image

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.
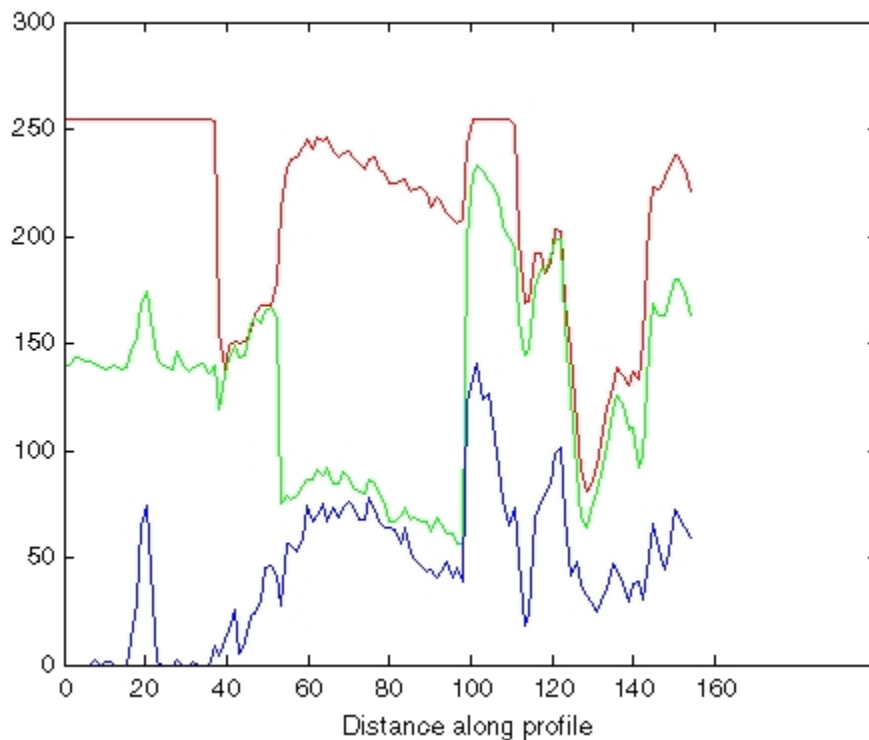
Plot of Intensity Values Along a Line Segment in an RGB Image

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)
```
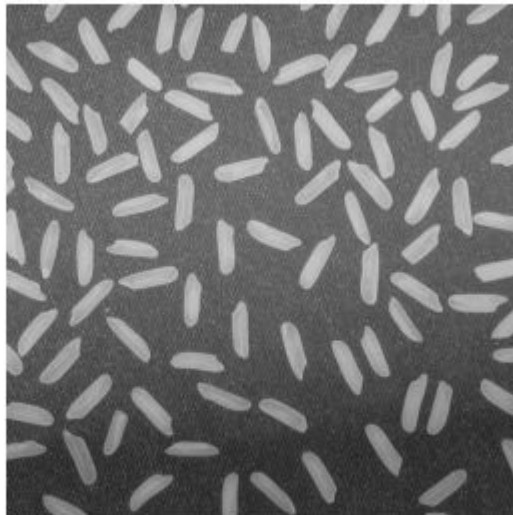

Create a contour plot of the image using `imcontour` .

```
figure;
imcontour(I,3)
```
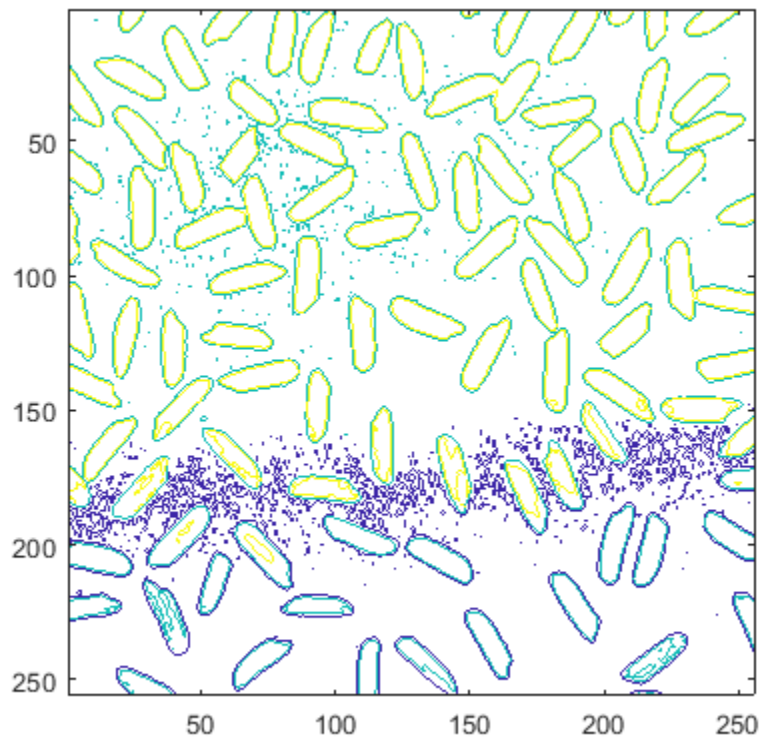
Measuring Regions in Grayscale Images

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')
```


Synthetic Image
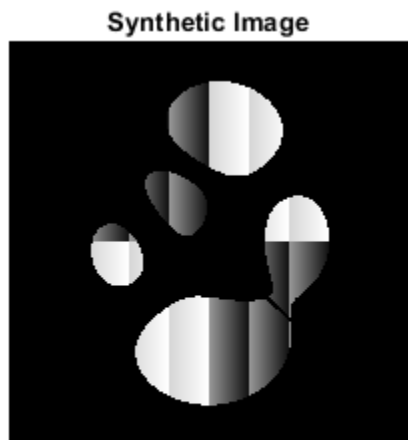
Step 2: Create a Binary Image

Segment the grayscale image by creating a binary image containing the objects in the image.

```
BW = I > 0;
imshow(BW)
title('Binary Image')
```

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
```

11-11

Weighted (red) and Unweighted (blue) Centroids
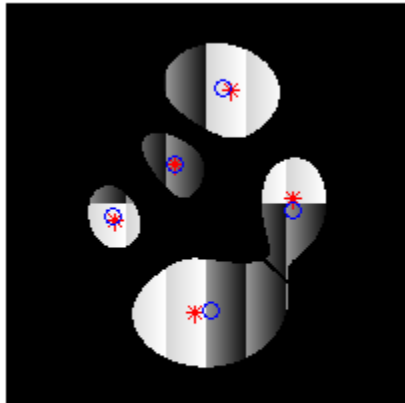
Step 4: Calculate Custom Pixel Value-Based Properties

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
```

Standard Deviation of Regions
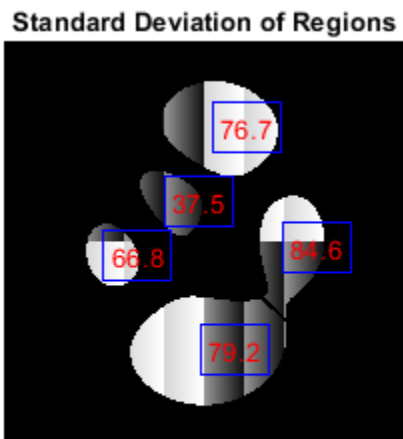
This figure shows the standard deviation measurement superimposed on each object in the image. You can also view the results in other ways, for example as a bar plot showing the standard deviation by label number.

```
figure
bar(1:numObj,[s.StandardDeviation])
xlabel('Region Label Number')
ylabel('Standard Deviation')
```
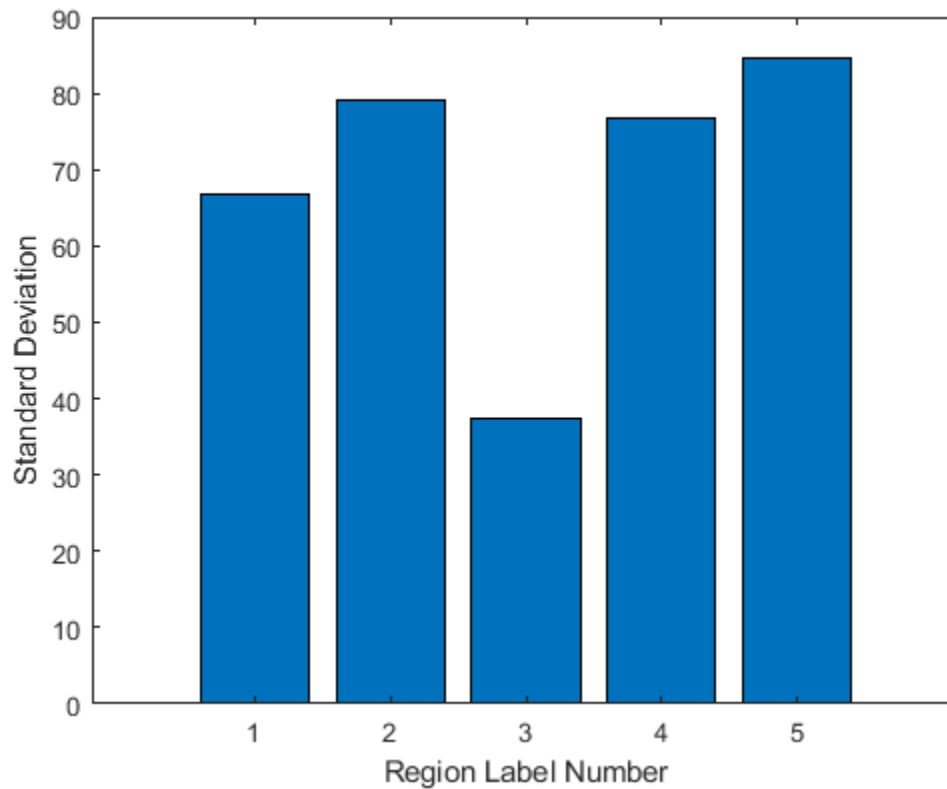
You can use the plot to determine how to partition the data. For example, the following code identifies objects with a standard deviation lower than 50.

```
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
```

Objects Having Standard Deviation < 50
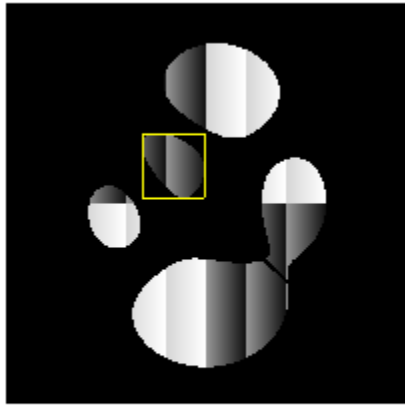
See Also

`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);
```
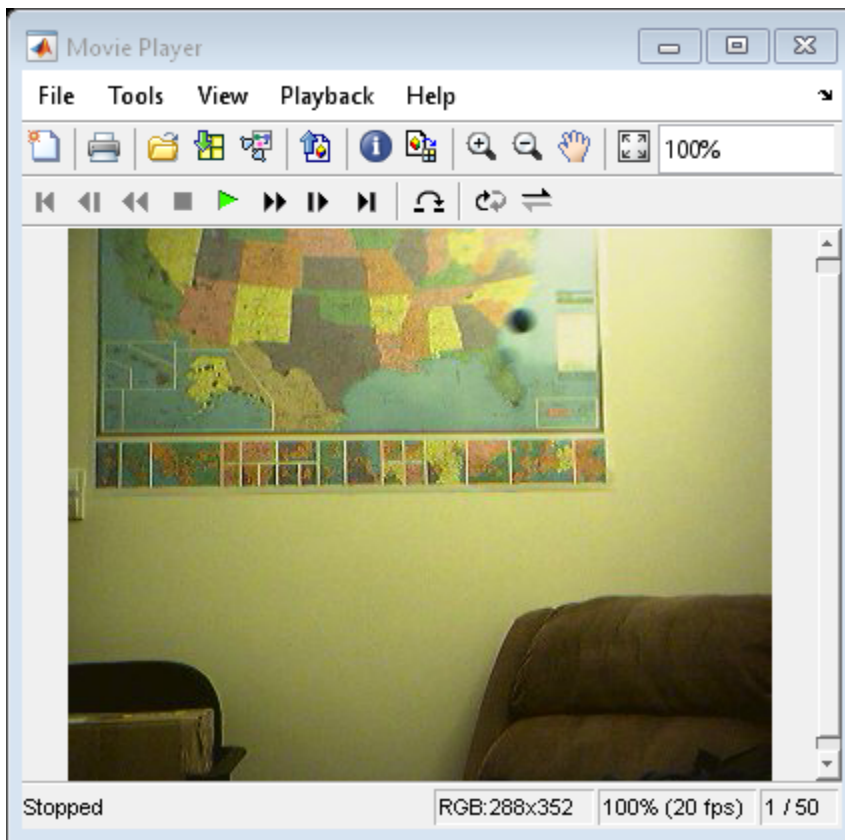
Step 3: Select Region where Pendulum is Swinging

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.

```matlab
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
```
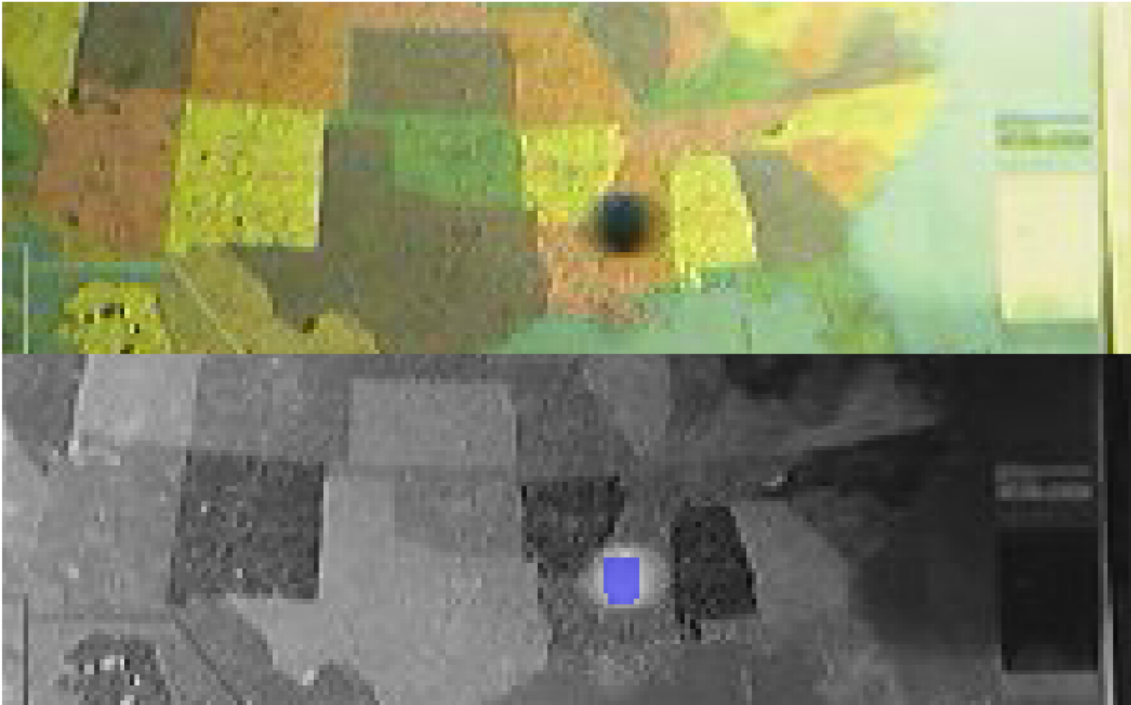
Step 5: Find the Center of the Segmented Pendulum in Each Frame

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');
```
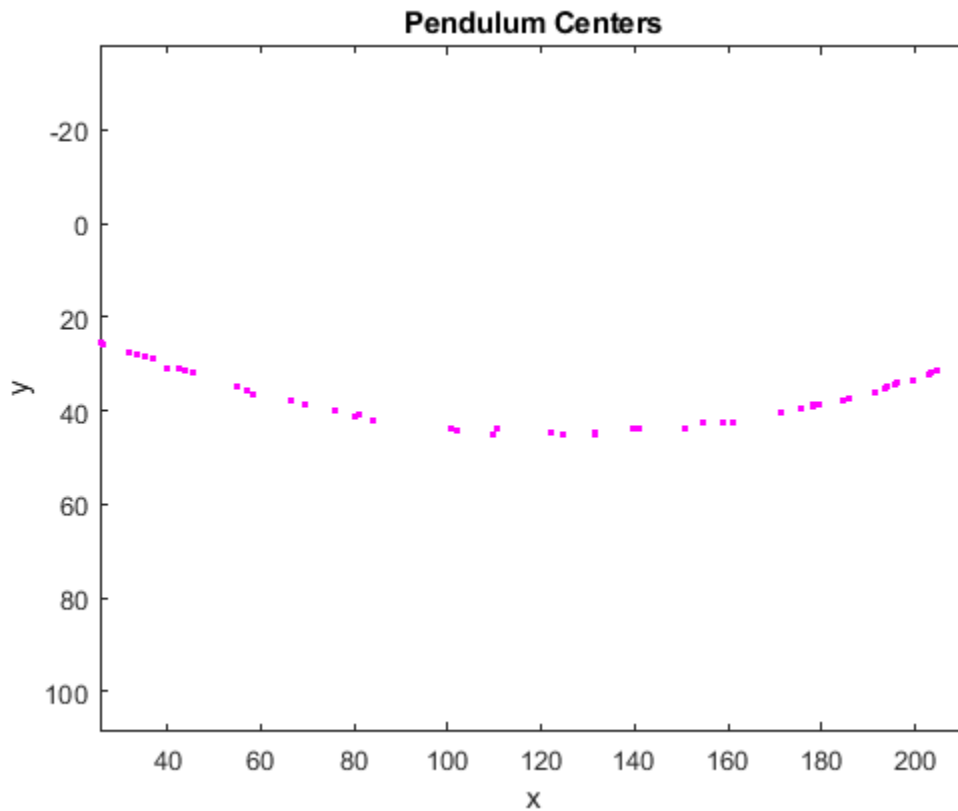
Pendulum Centers

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));
```
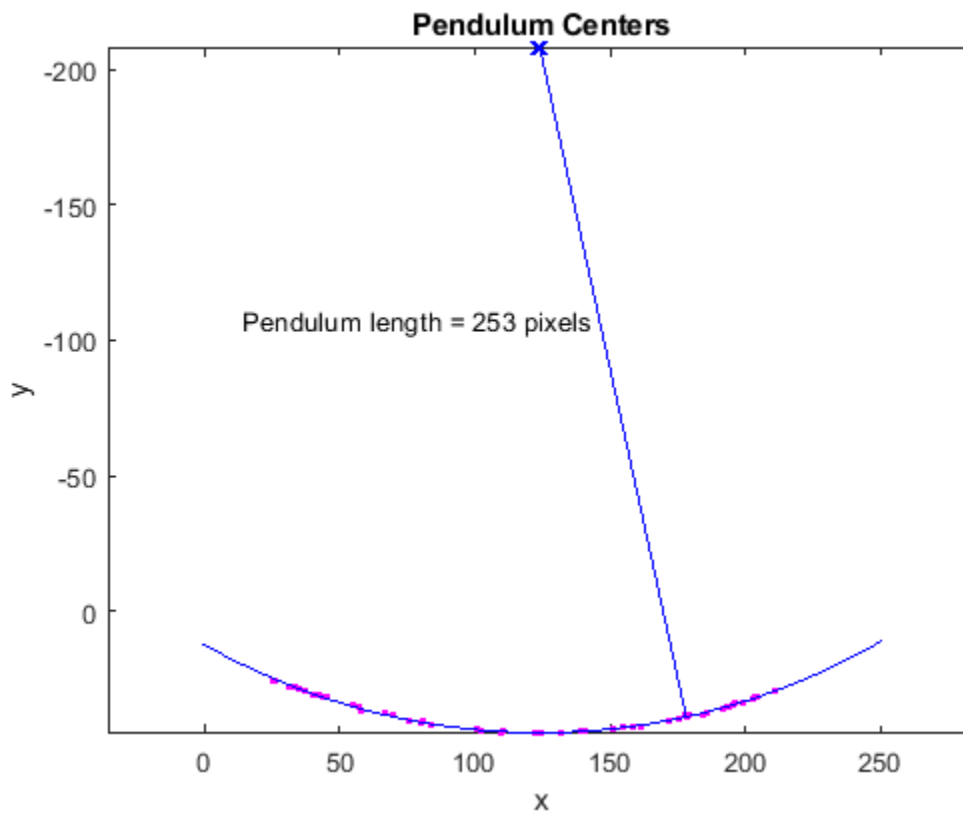

See Also

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)
```
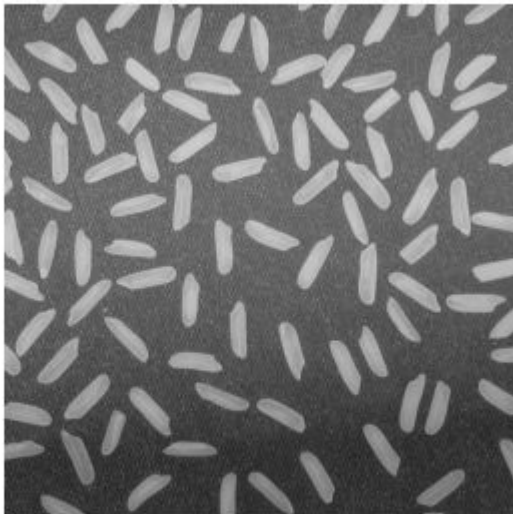
Create the histogram. For the example image, showing grains of rice, `imhist` creates a histogram with 64 bins. The `imhist` function displays the histogram, by default. The histogram shows a peak at around 100, corresponding to the dark gray background in the image.

```
figure;
imhist(I);
```
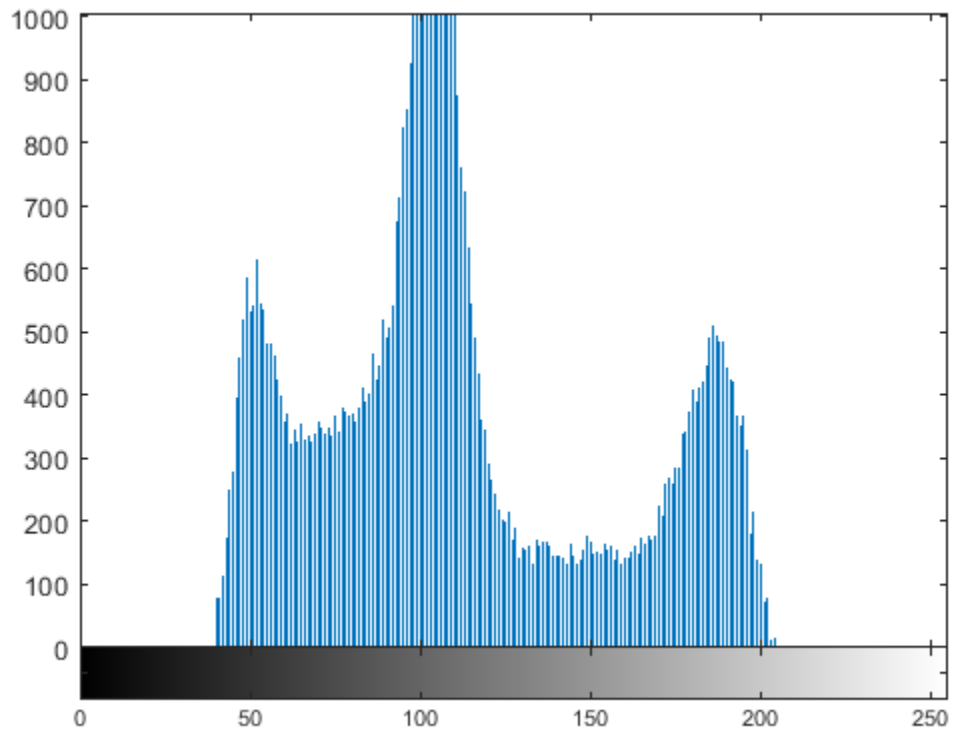
Image Mean, Standard Deviation, and Correlation Coefficient

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');
```
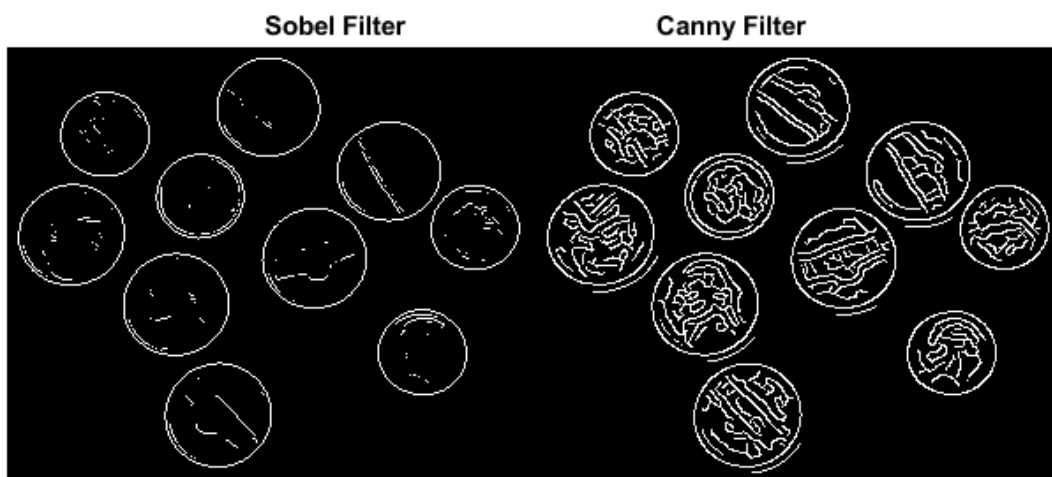
Boundary Tracing in Images

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)
```
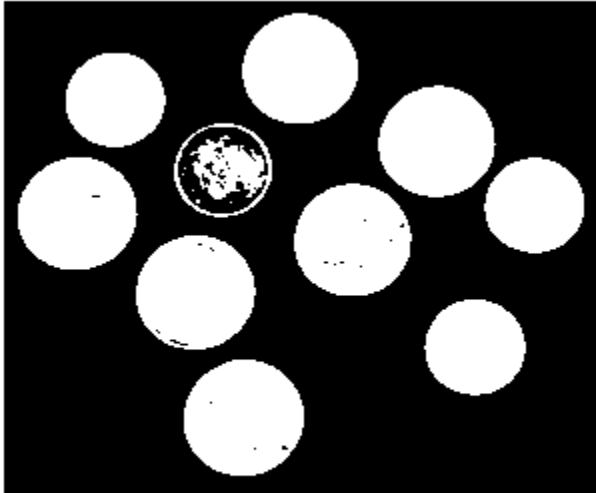

Determine the row and column coordinates of a pixel on the border of the object you want to trace. `bwboundary` uses this point as the starting location for the boundary tracing.

```
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
```
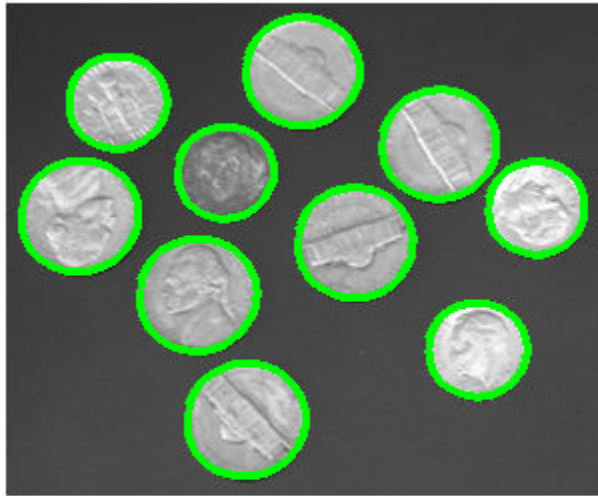
Select First Step and Direction for Tracing

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).
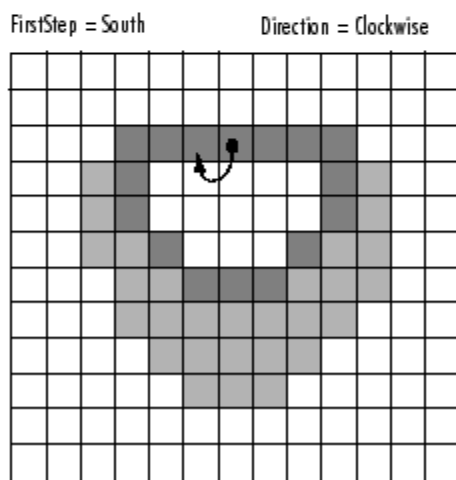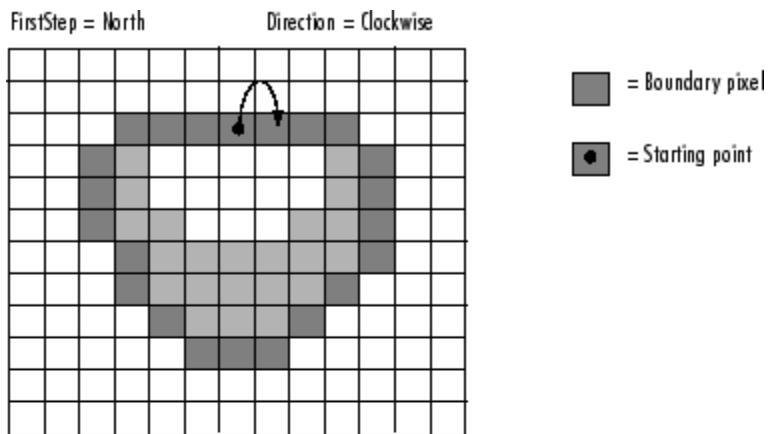
Impact of First Step and Direction Parameters on Boundary Tracing

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 (for example, 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,[])
```
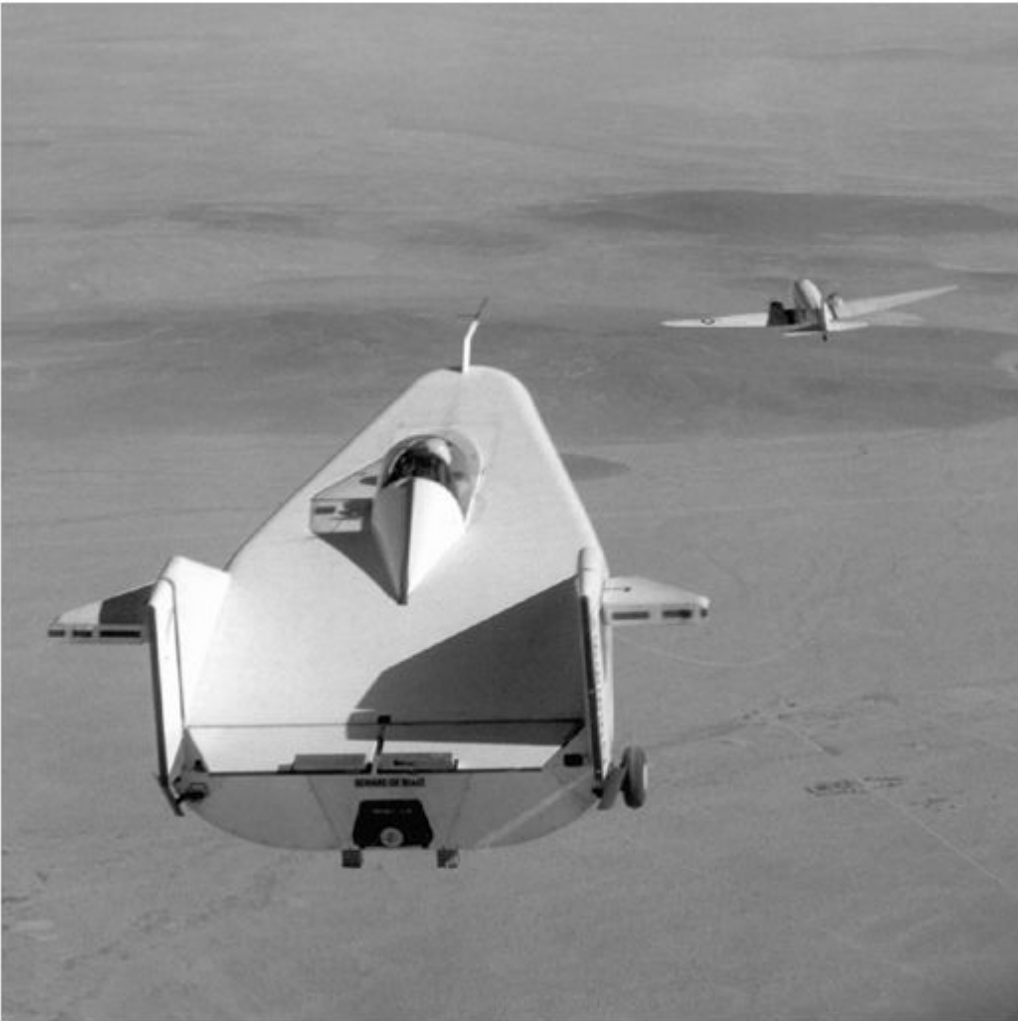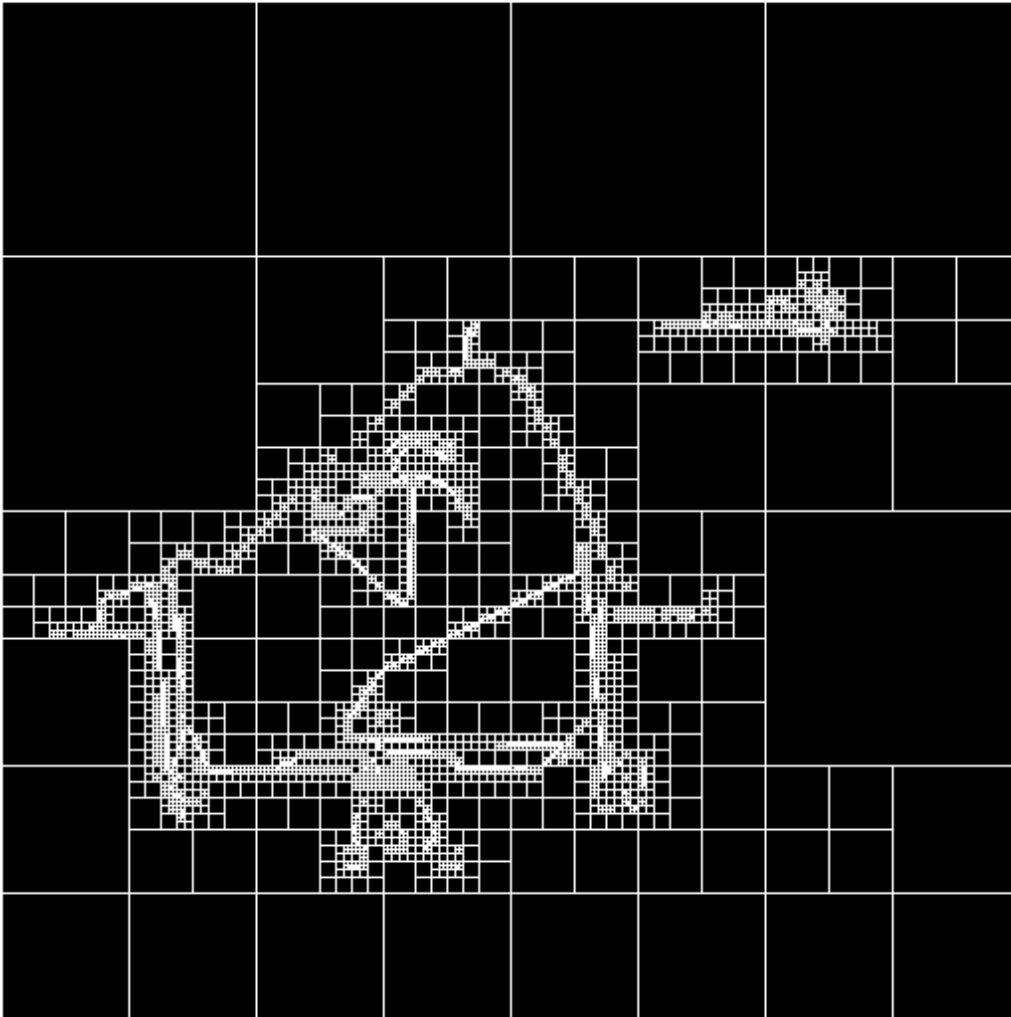
Detect and Measure Circular Objects in an Image

This example shows how to automatically detect circles or circular objects in an image and visualize the detected circles.

Step 1: Load Image

Read and display an image of round plastic chips of various colors. 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.

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

Step 2: Determine Radius Range for Searching Circles

Find the appropriate radius range of the circles using the `drawline` function. Draw a line over the approximate diameter of a chip.

```
d = drawline;
```


The length of the line ROI is the diameter of the chip. Typical chips have diameters in the range 40 to 50 pixels.

```
pos = d.Position;
diffPos = diff(pos);
diameter = hypot(diffPos(1),diffPos(2))
```

```
diameter = 45.3448
```

Step 3: Initial Attempt to Find Circles

The `imfindcircles` function searches for circles with a range of radii. Search for circles with radii in the range of 20 to 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)
```
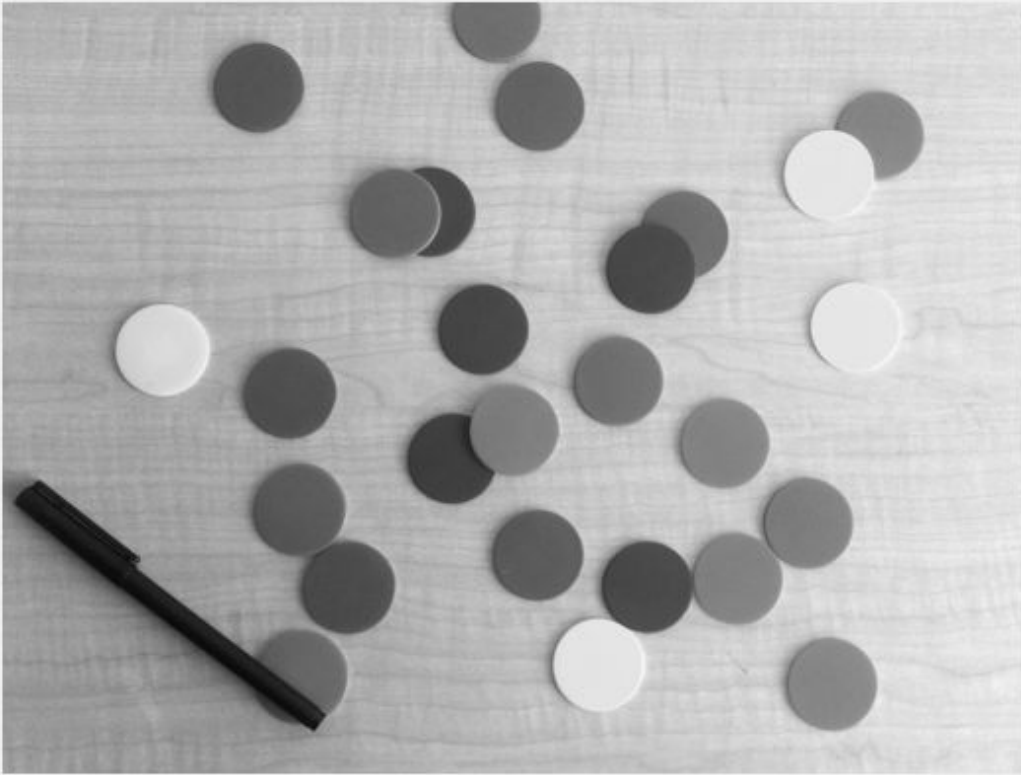
The background is quite bright and most of the chips are darker than the background. But, by default, `imfindcircles` finds circular objects that are brighter than the background. So, set the parameter 'ObjectPolarity' to 'dark' in `imfindcircles` to search for dark circles.

```
[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
  327.1316   297.0077
  243.9893   166.4538
  271.5873   280.8920
```

radii = *8×1*

```
   23.1604
   22.5710
   22.9576
   23.7356
   22.9551
   22.9995
   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);
```
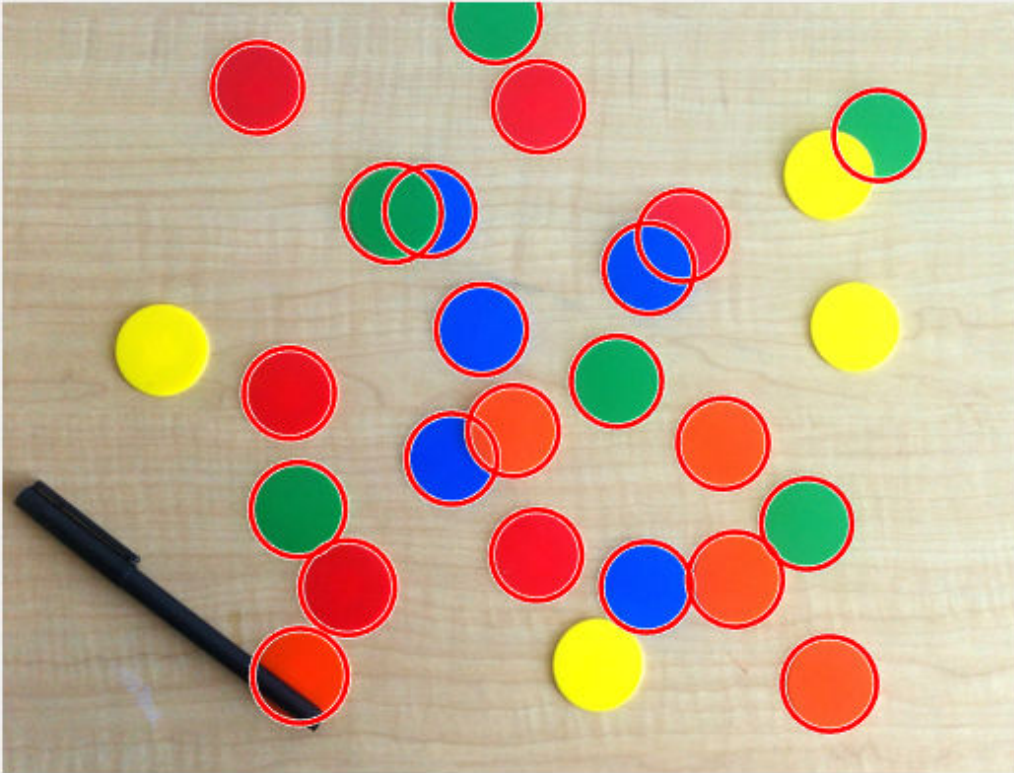
The two-stage method is detecting more circles, at the Sensitivity of 0.92. In general, these two method are complementary in that have they have different strengths. The Phase coding method is typically faster and slightly more robust to noise than the two-stage method. But it may also need higher 'Sensitivity' 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);
```
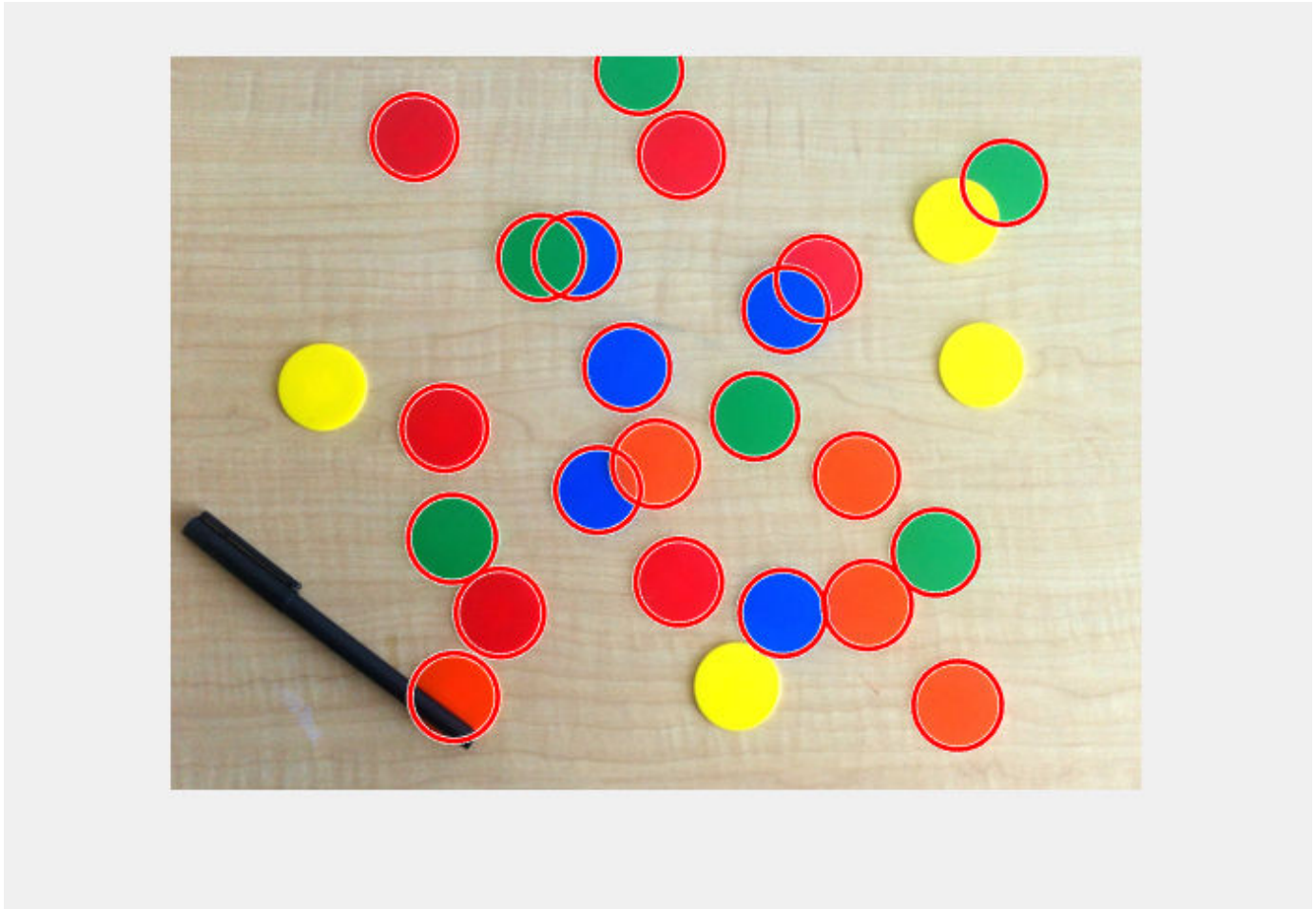
Note that both the methods in `imfindcircles` find the centers and radii of the partially visible (occluded) chips accurately.

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)
```
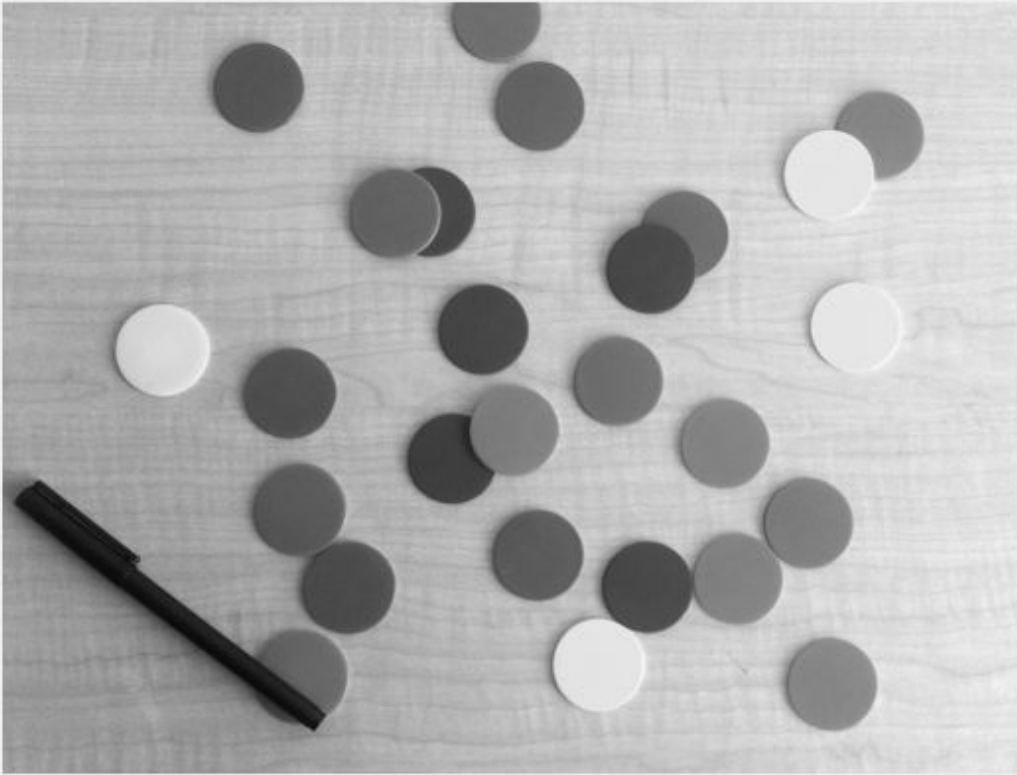
Step 8: Find 'Bright' Circles in the Image

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');
```
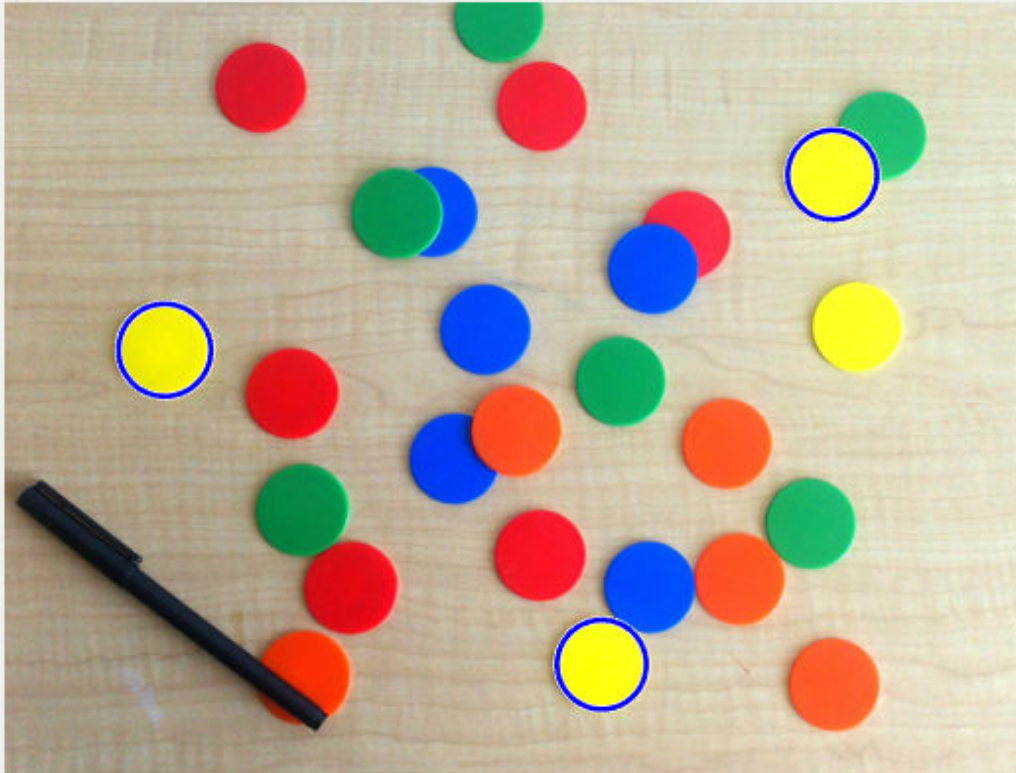
Note that three of the missing yellow chips were found, but one yellow chip is still missing. These yellow chips are hard to find because they don't *stand out* as well as others on this background.

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 11-47
- "Measuring the Radius of a Roll of Tape" on page 11-61

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)
```
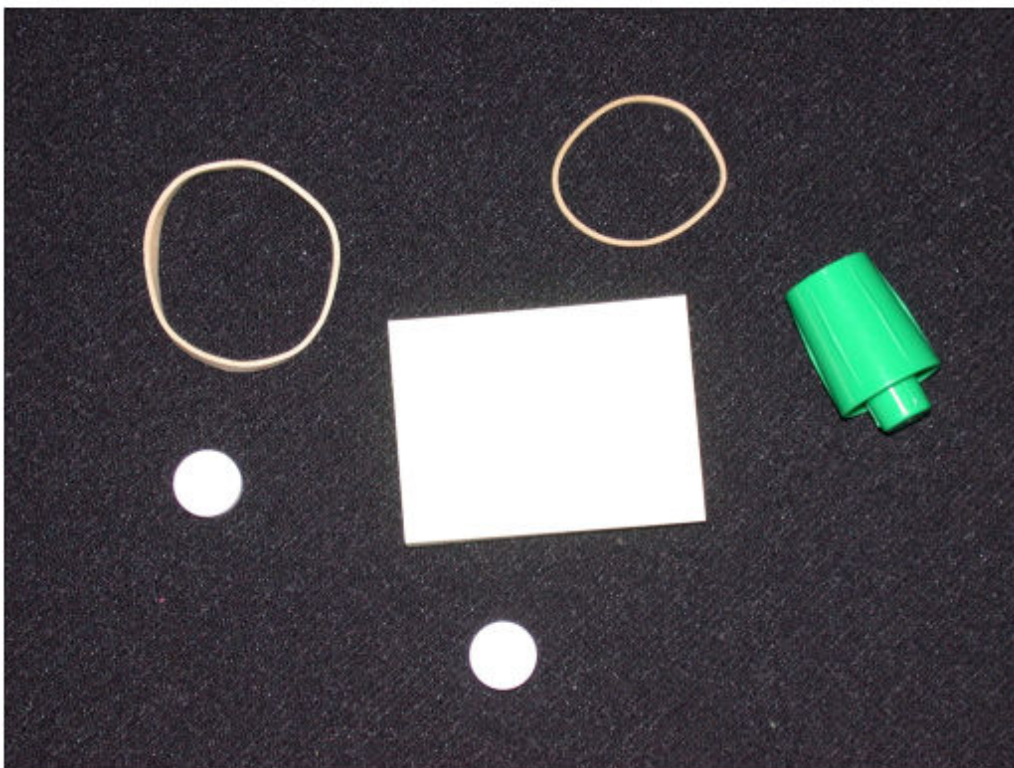

Step 2: Threshold the Image

Convert the image to black and white in order to prepare for boundary tracing using bwboundaries.

```
I = rgb2gray(RGB);
bw = imbinarize(I);
imshow(bw)
```
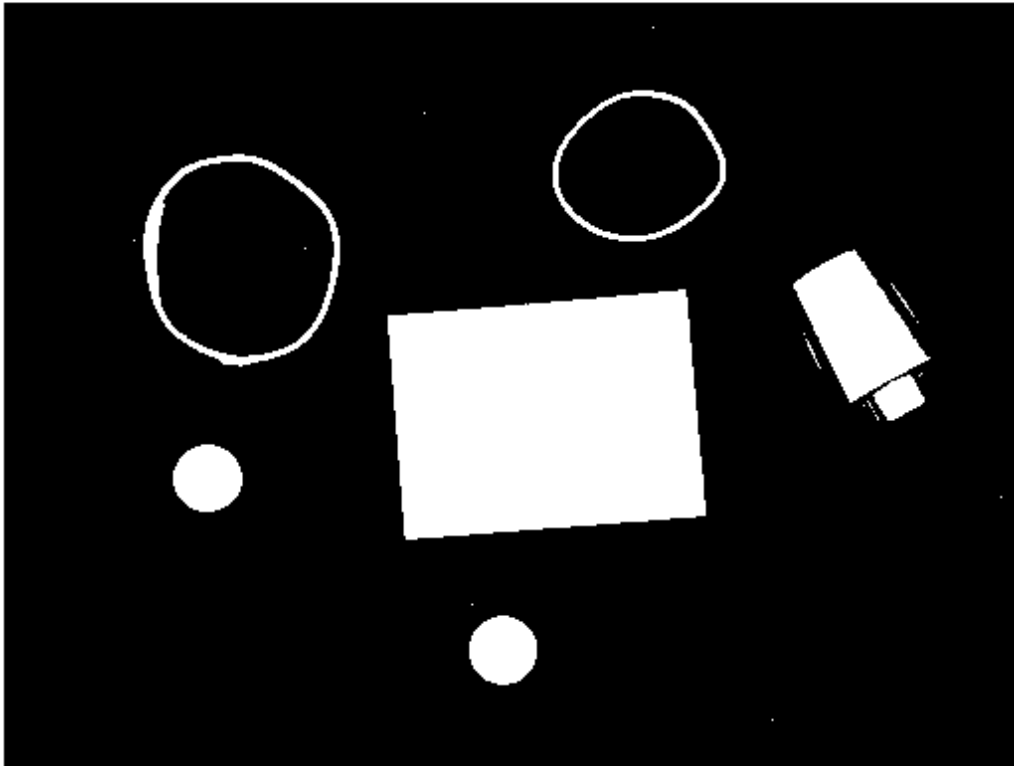
Step 3: Remove the Noise

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);
imshow(bw)
```
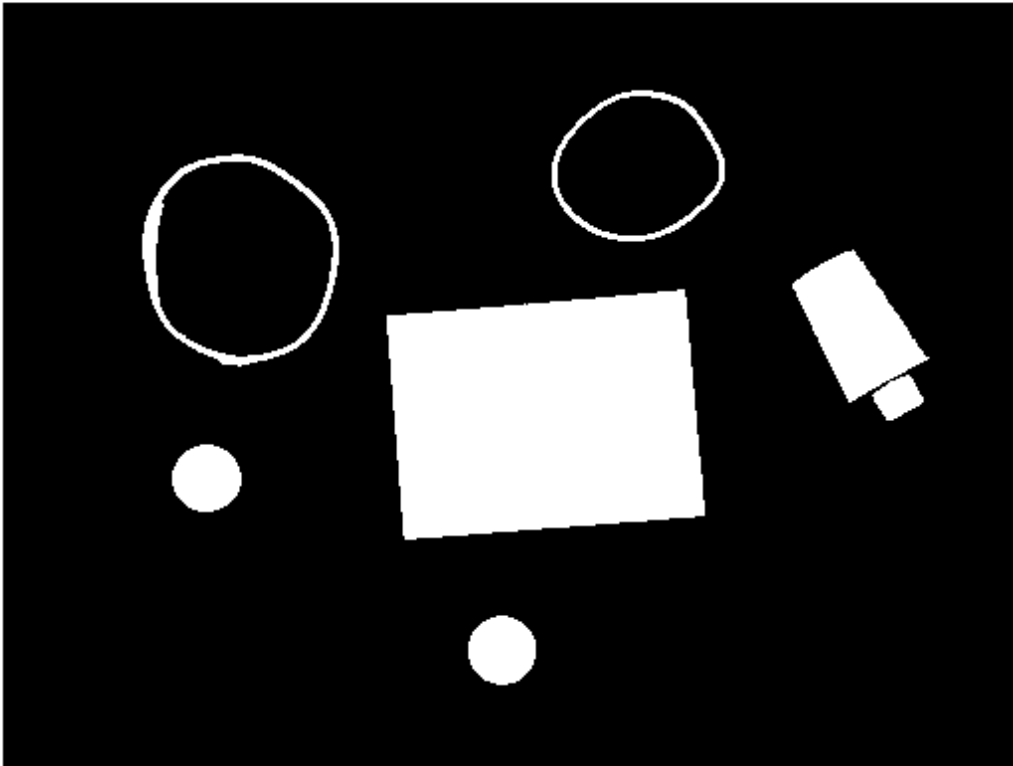
Fill a gap in the pen's cap.

```
se = strel('disk',2);
bw = imclose(bw,se);
imshow(bw)
```
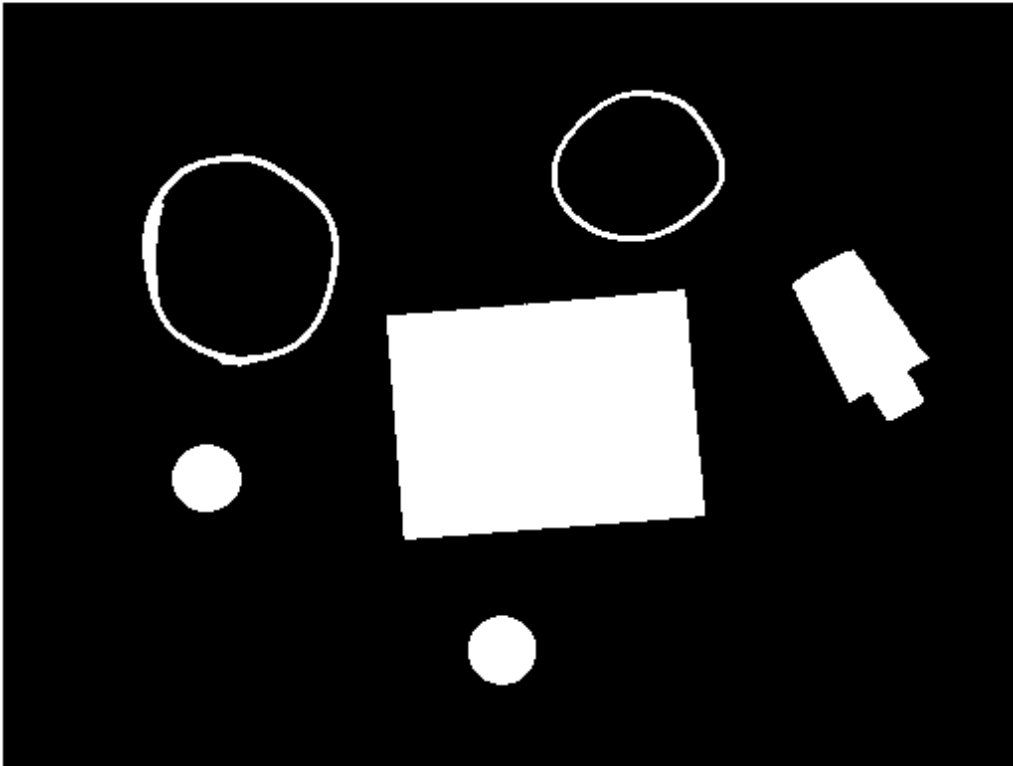
Fill any holes, so that regionprops can be used to estimate the area enclosed by each of the boundaries

```
bw = imfill(bw,'holes');
imshow(bw)
```
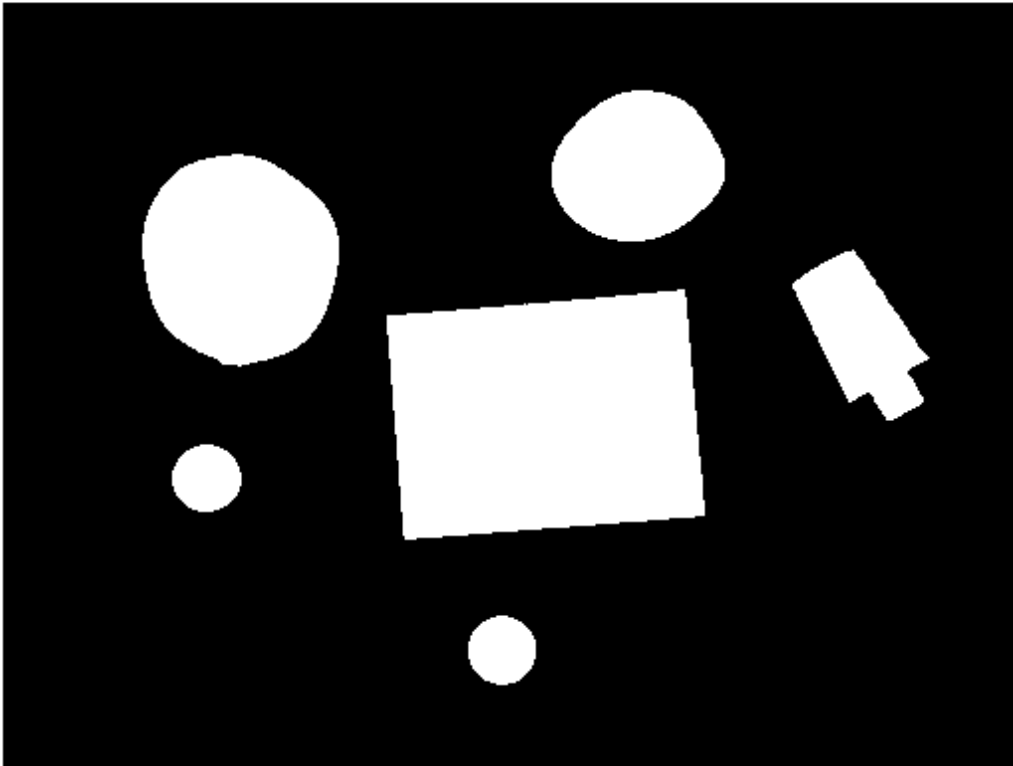
Step 4: Find the Boundaries

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
```

11-51
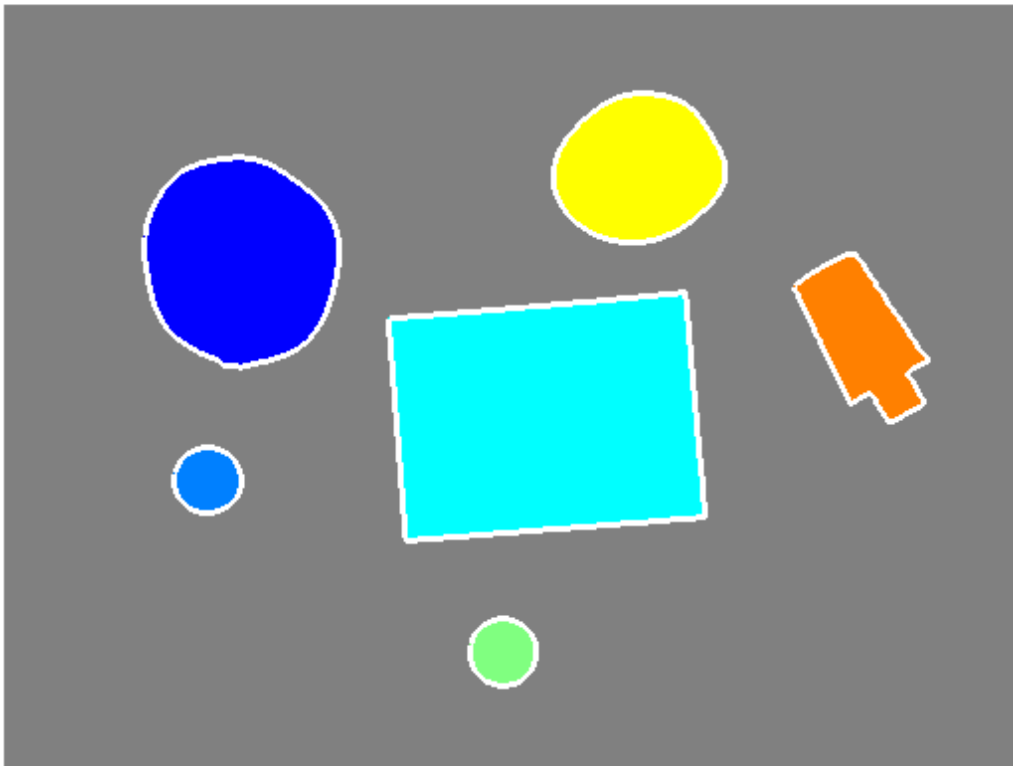
Step 5: Determine which Objects are Round

Estimate each object's area and perimeter. Use these results to form a simple metric indicating the roundness of an object:

$$\text{metric} = \frac{4\pi * \text{area}}{\text{perimeter}^2}$$

This metric is equal to 1 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'
```

```matlab
  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'])
```

Metrics closer to 1 indicate that the object is approximately round
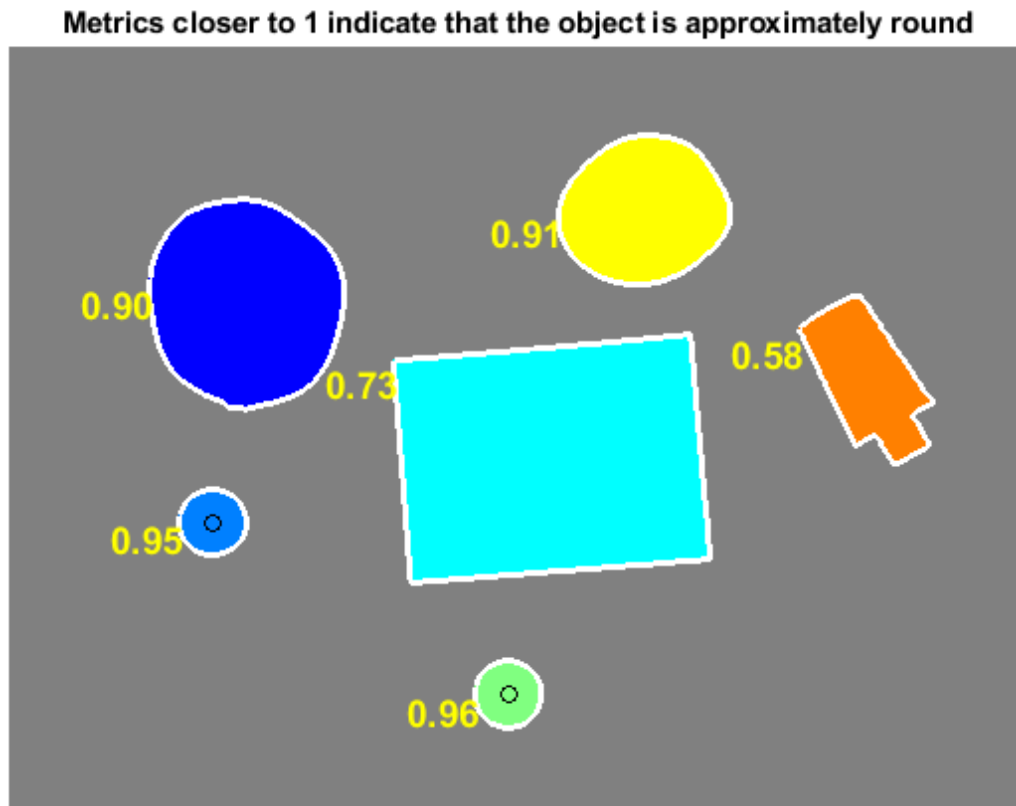
See Also
bwareaopen | bwboundaries | imbinarize | imclose | imfill | label2rgb | regionprops | strel

Related Examples

- "Detect and Measure Circular Objects in an Image" on page 11-35
- "Measuring the Radius of a Roll of Tape" on page 11-61

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');
```
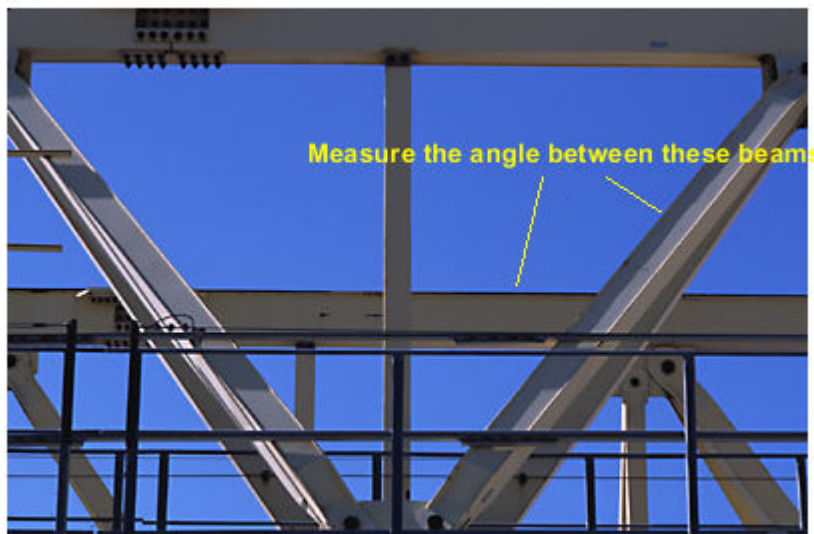

Image courtesy of Jeff Mather

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

More About

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

```matlab
% 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 11-47
- "Detect and Measure Circular Objects in an Image" on page 11-35

Calculate Statistical Measures of Texture

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 pixel intensities 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.
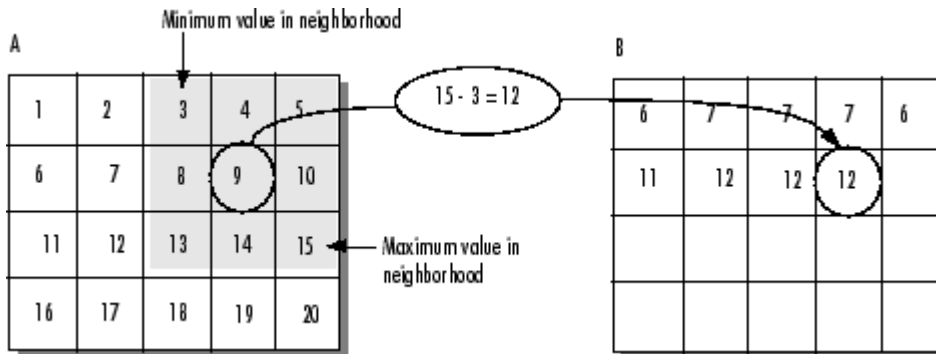
Determining Pixel Values in Range Filtered Output Image

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.

See Also
entropyfilt | rangefilt | stdfilt

More About
- "Texture Segmentation Using Texture Filters" on page 13-7

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 "Calculate Statistical Measures of Texture" on page 11-64 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 11-69

More About
- "Create a Gray-Level Co-Occurrence Matrix" on page 11-67

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 11-69 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.
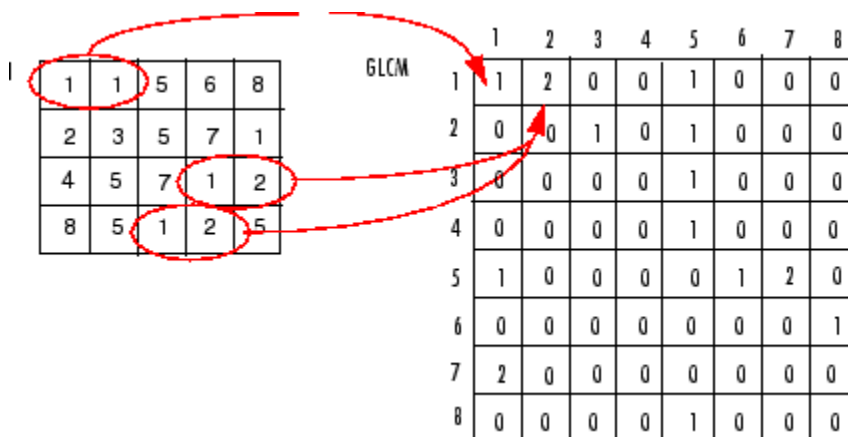
Process Used to Create the GLCM

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.
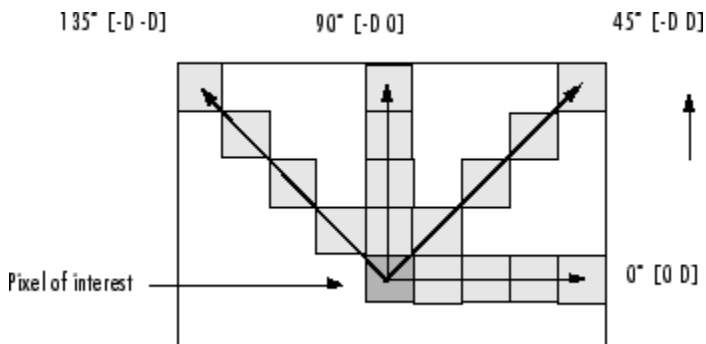
Derive Statistics from GLCM and Plot Correlation

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)
```
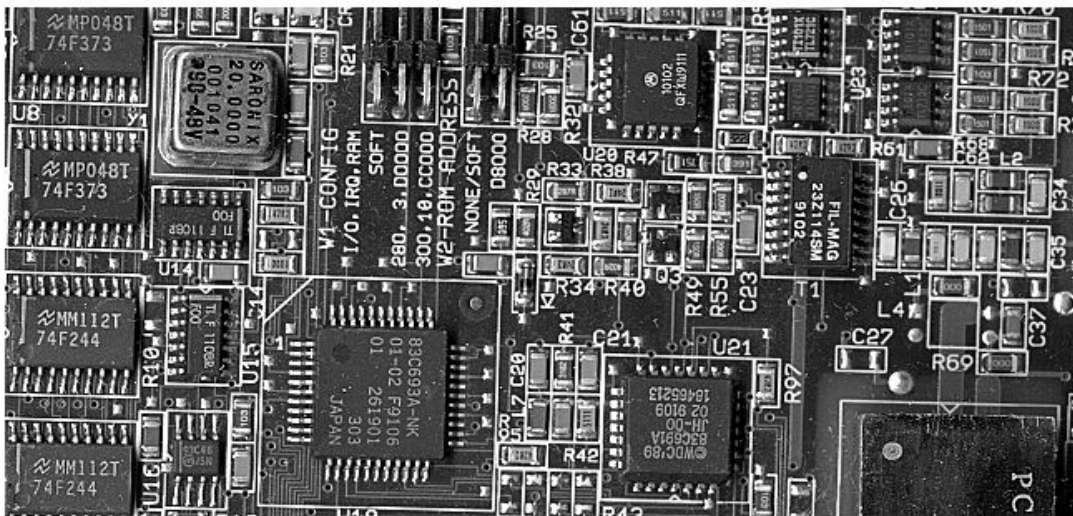

Define offsets of varying direction and distance. Because the image contains objects of a variety of shapes and sizes that are arranged in horizontal and vertical directions, the example specifies a set of horizontal offsets that only vary in distance.

```
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')
```

Texture Correlation as a function of offset
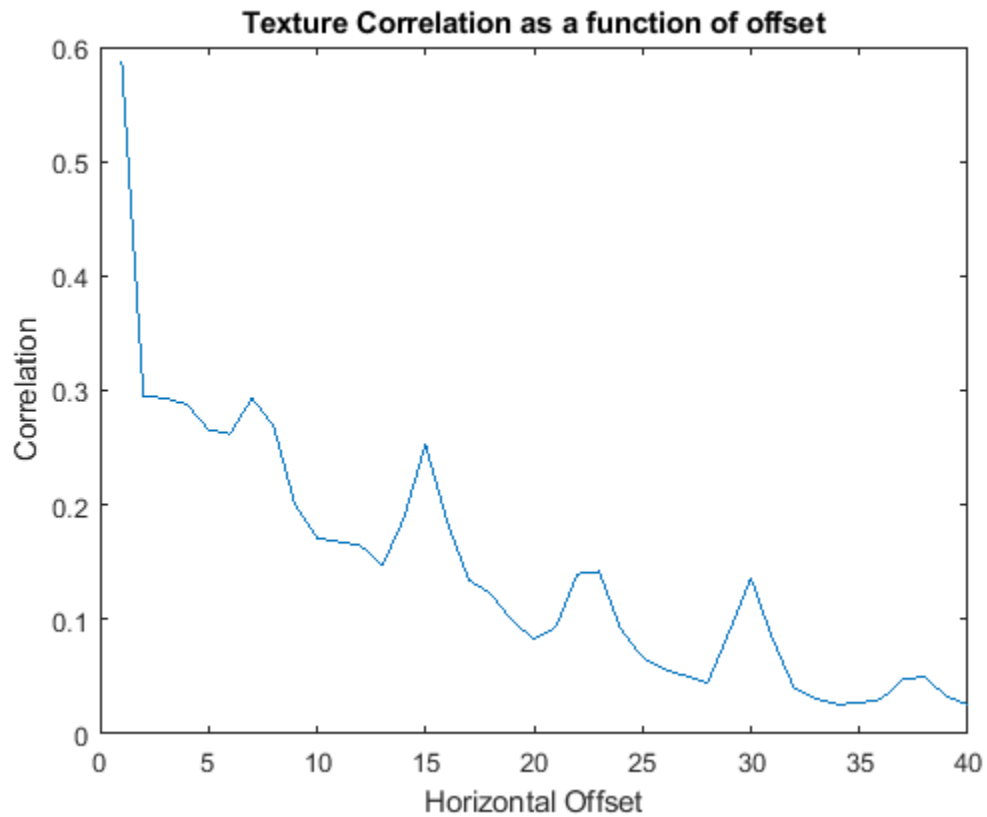
The plot contains peaks at offsets 7, 15, 23, and 30. If you examine the input image closely, you can see that certain vertical elements in the image have a periodic pattern that repeats every seven pixels.

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)
```
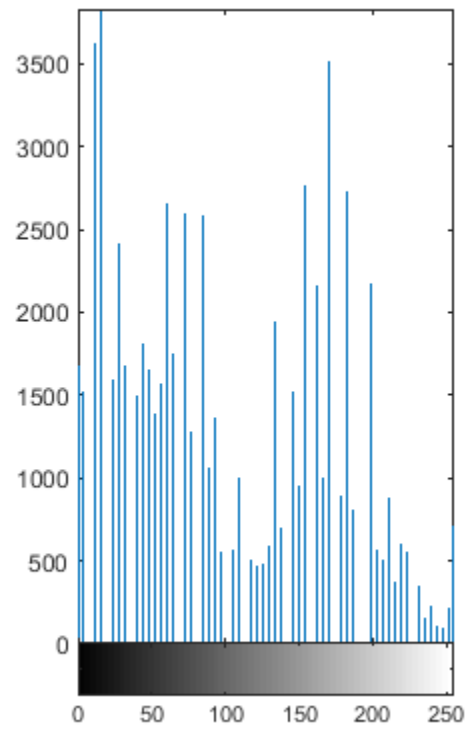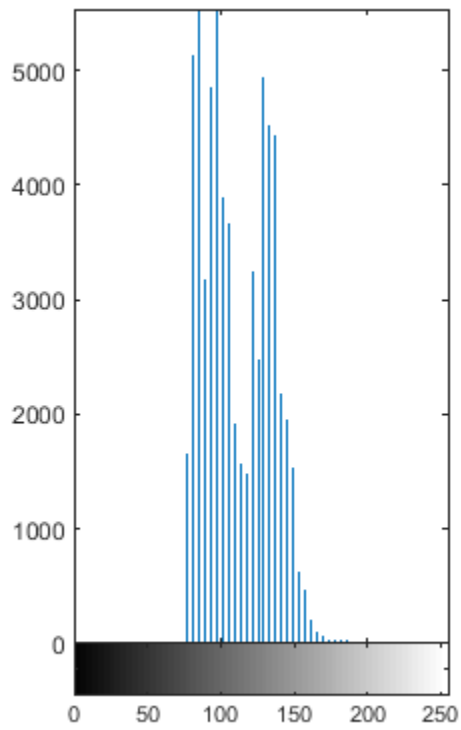
Gamma Correction

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.)
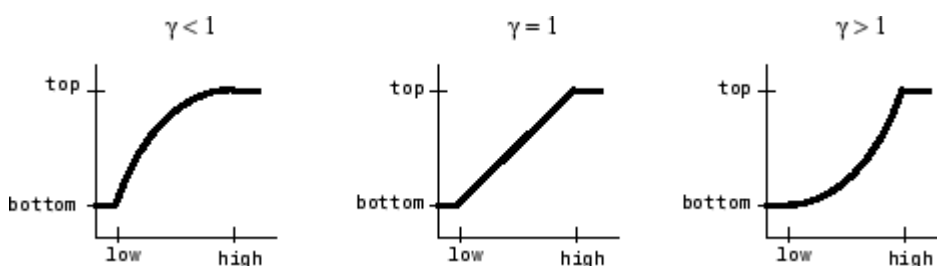
Plots Showing Three Different Gamma Correction Settings

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")
```
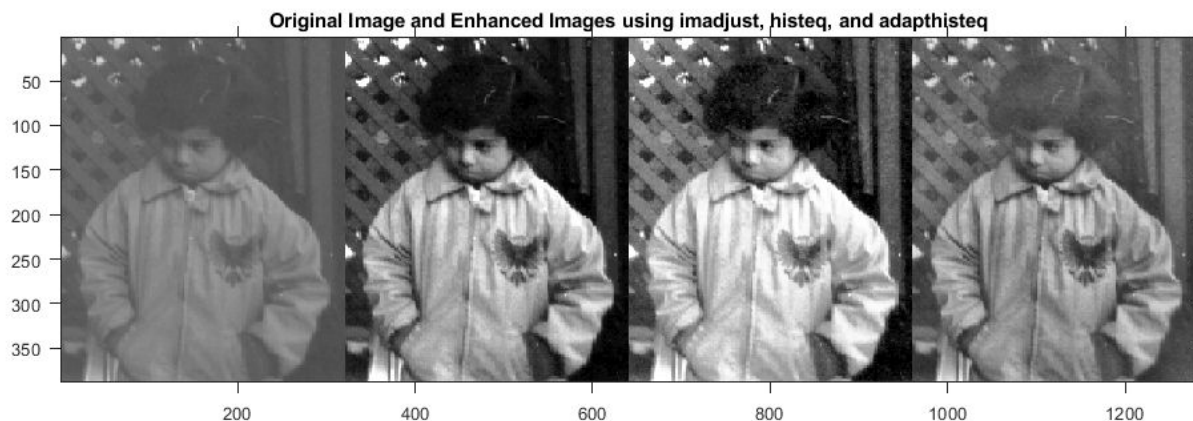

Read a second grayscale image into the workspace and enhance the image using the three contrast adjustment techniques.

```
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")
```
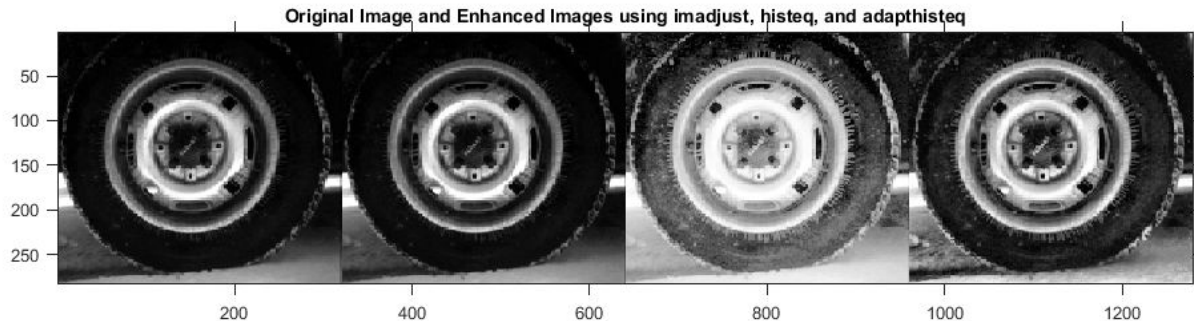

Notice that `imadjust` had little effect on the image of the tire, but it caused a drastic change in the case of pout. Plotting the histograms of `pout.tif` and `tire.tif` reveals that most of the pixels in the first image are concentrated in the center of the histogram, while in the case of `tire.tif`, 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');
```
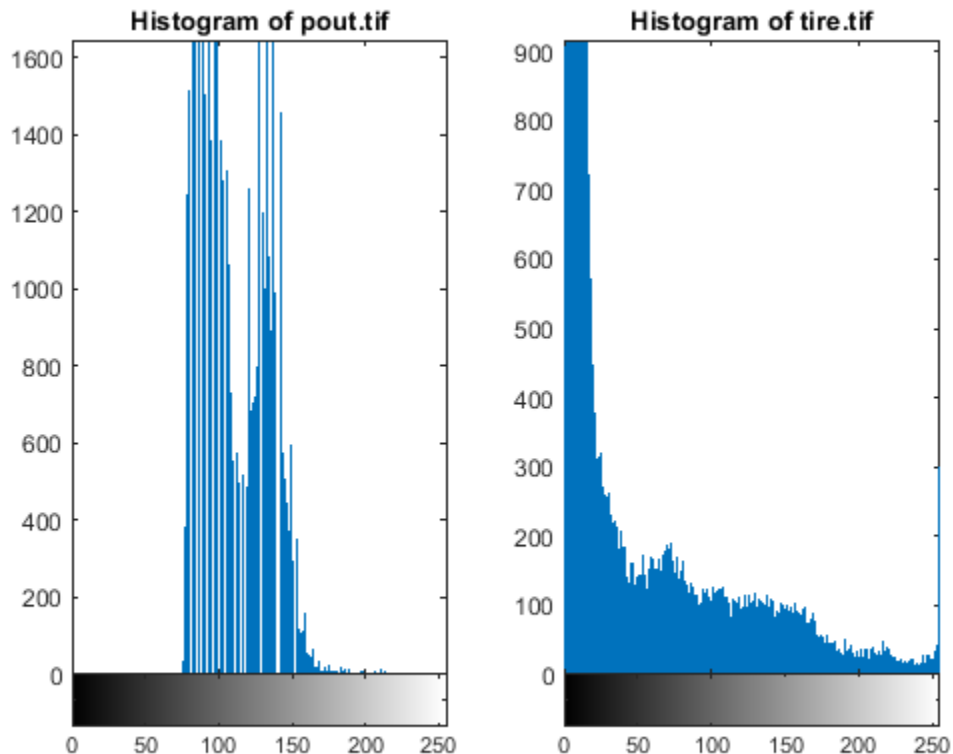
Histogram equalization, on the other hand, substantially changes both images. Many of the previously hidden features are exposed, especially the debris particles on the tire. Unfortunately, at the same time, the enhancement over-saturates several areas of both images. Notice how the center of the tire, 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")
```
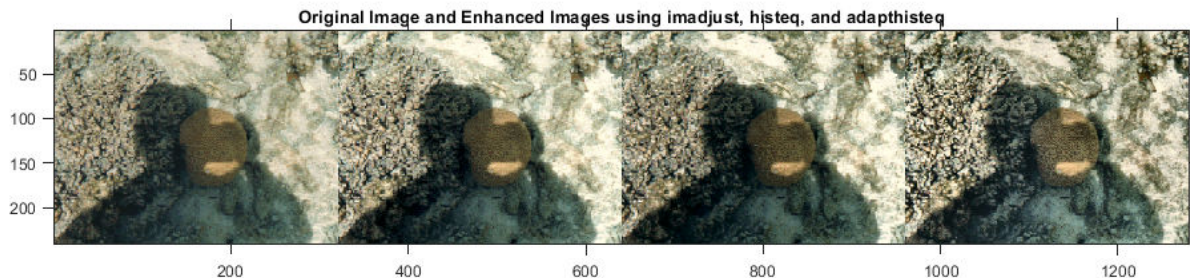

See Also
adapthisteq | histeq | imadjust

More About
- "Adaptive Histogram Equalization" on page 11-85
- "Histogram Equalization" on page 11-81

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 11-80.

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)
```
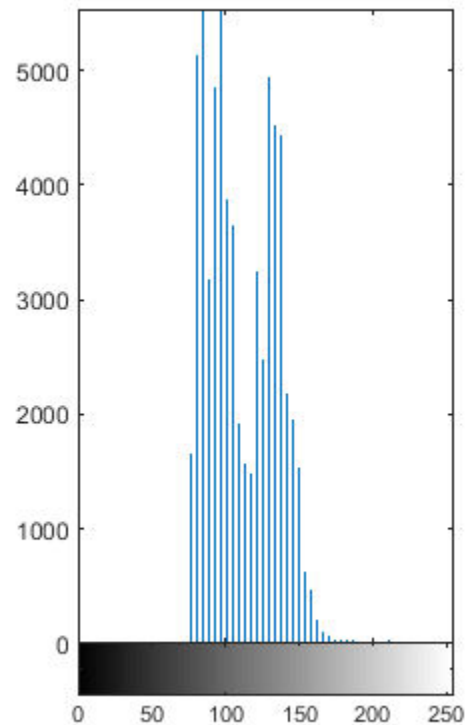
Adjust the contrast using histogram equalization. In this example, the histogram equalization function, `histeq`, tries to match a flat histogram with 64 bins, which is the default behavior. You can specify a different histogram instead.

```
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)
```
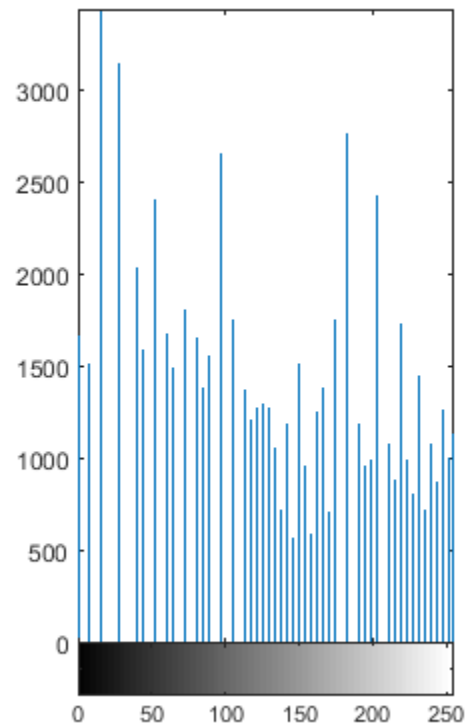
Plot Transformation Curve for Histogram Equalization

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);
```
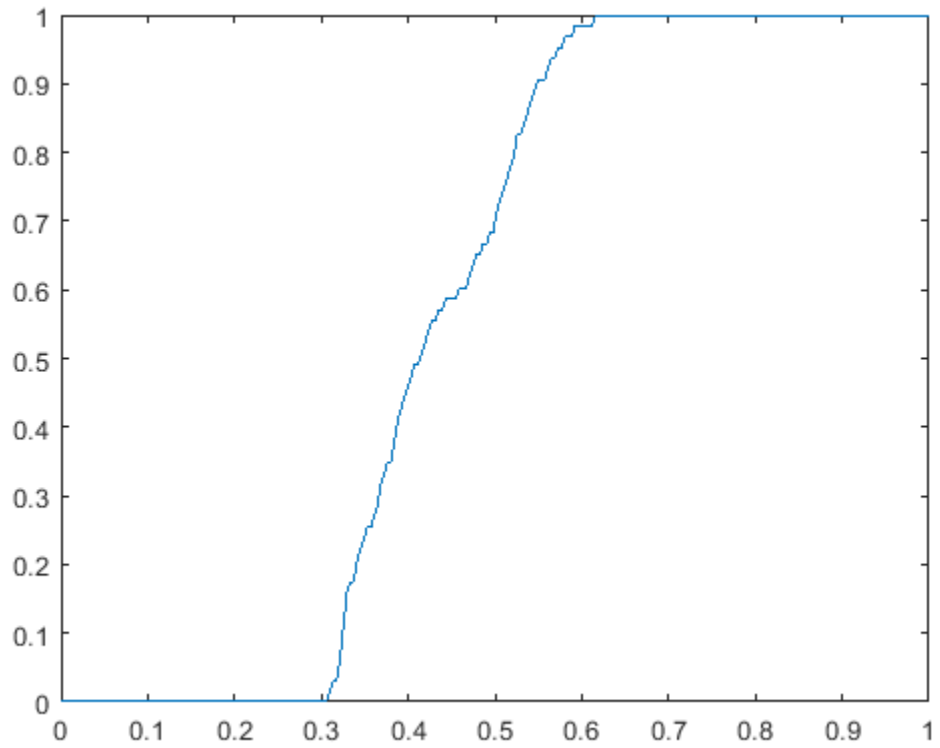
Adaptive Histogram Equalization

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)
```
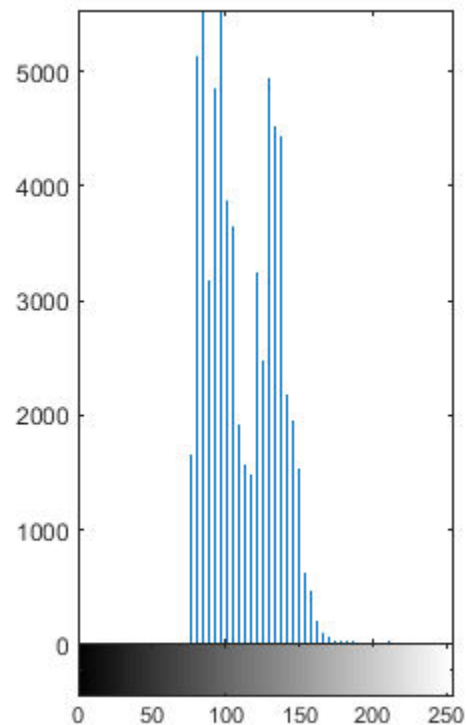
Adjust the contrast of the image using adaptive histogram equalization.

```
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)
```
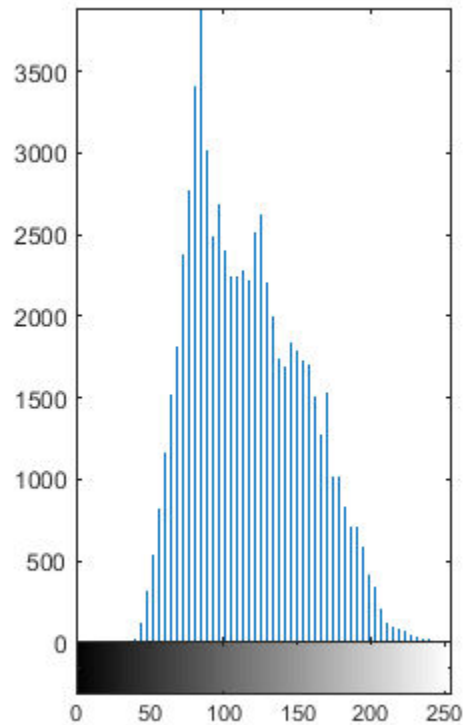

See Also

More About

*   "Histogram Equalization" on page 11-81

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 11-91 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 example 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.

```
imshow(A)
title('Little Colorado River Image')
```

Little Colorado River Image
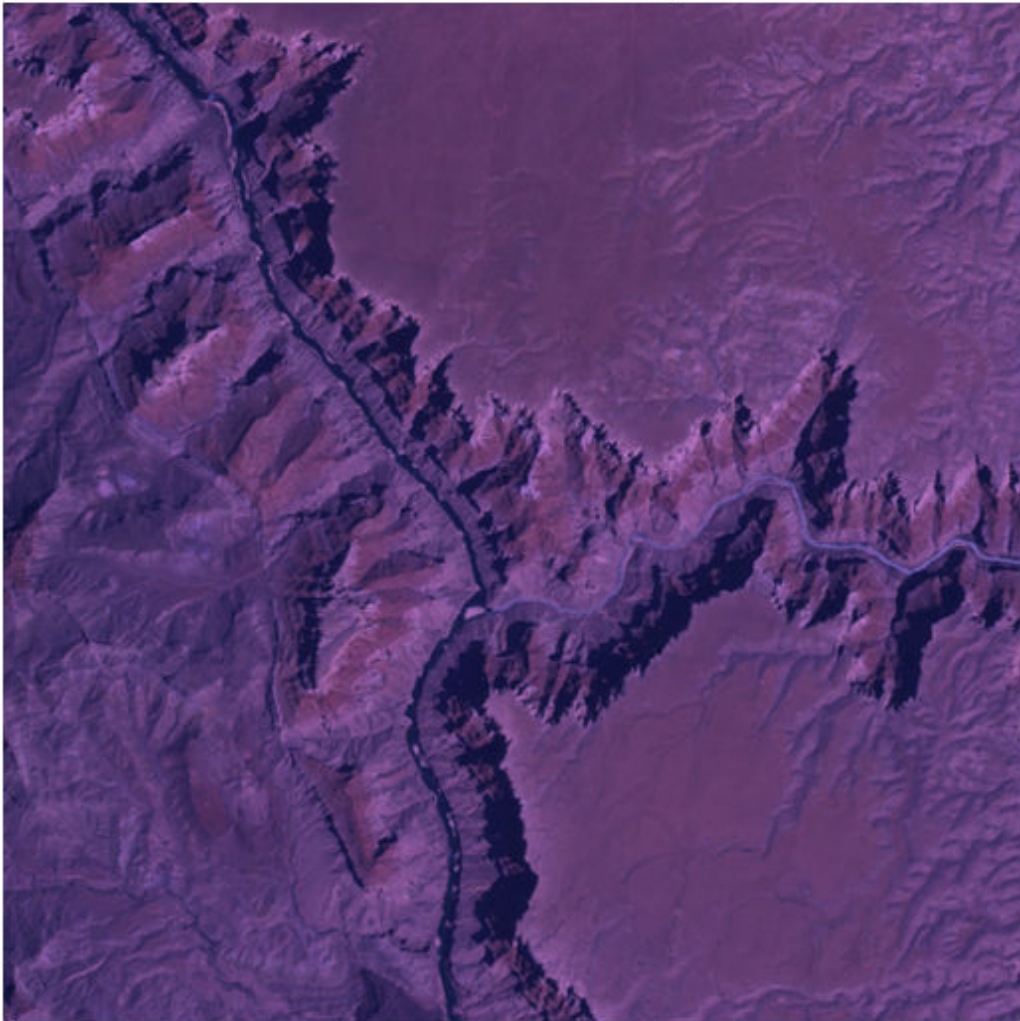

```
imshow(B)
title('Little Colorado River Image After Decorrelation Stretch')
```

Little Colorado River Image After Decorrelation Stretch
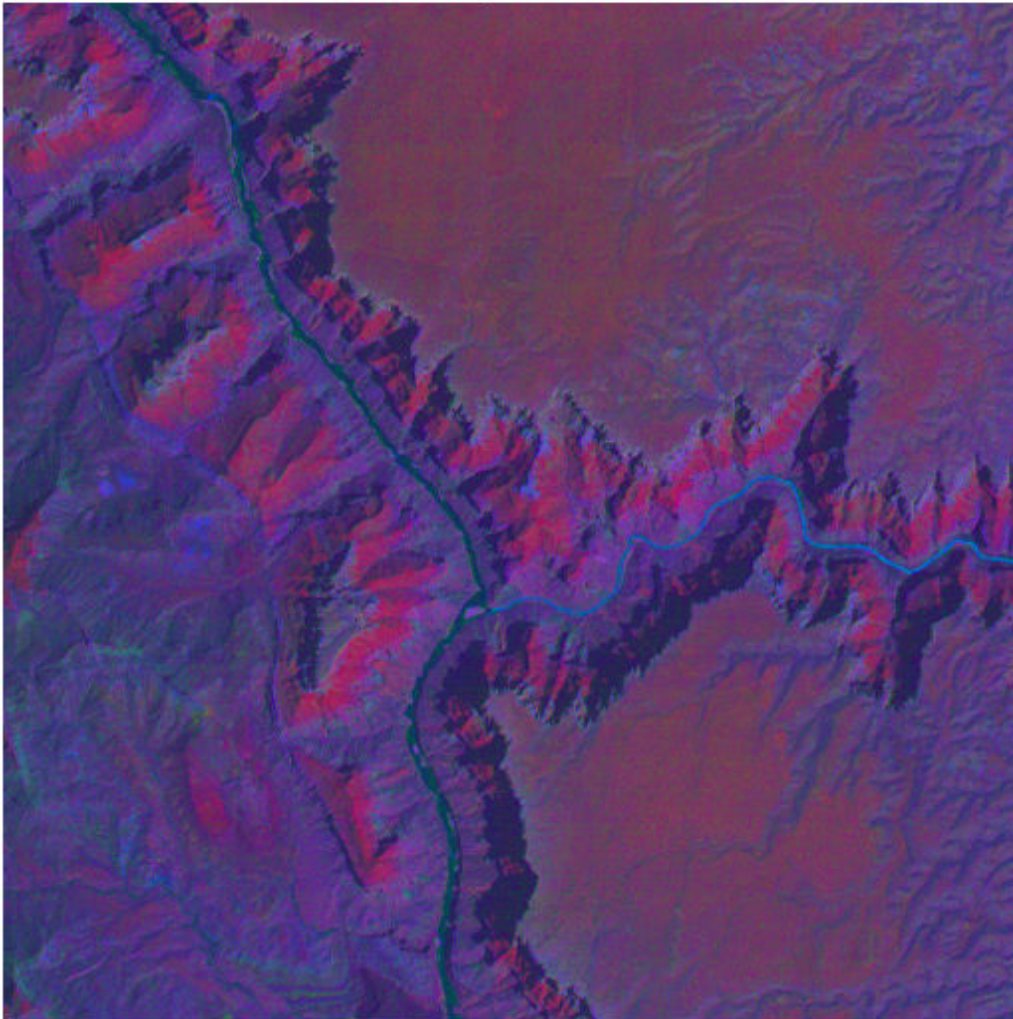
Create a Color Band Scatterplot

First separate the three color channels of the original image.

```
[rA,gA,bA] = imsplit(A);
```

Separate the three color channels of the image after decorrelation stretching.

```
[rB,gB,bB] = imsplit(B);
```

Display the color scatterplot of the original image. Then display the color scatterplot of the image after decorrelation stretching.

```
figure
plot3(rA(:),gA(:),bA(:),'.')
```

11-89

```
grid on
xlabel('Red (Band 3)')
ylabel('Green (Band 2)')
zlabel('Blue (Band 1)')
title('Color Scatterplot Before Decorrelation Stretch')
```


Color Scatterplot Before Decorrelation Stretch
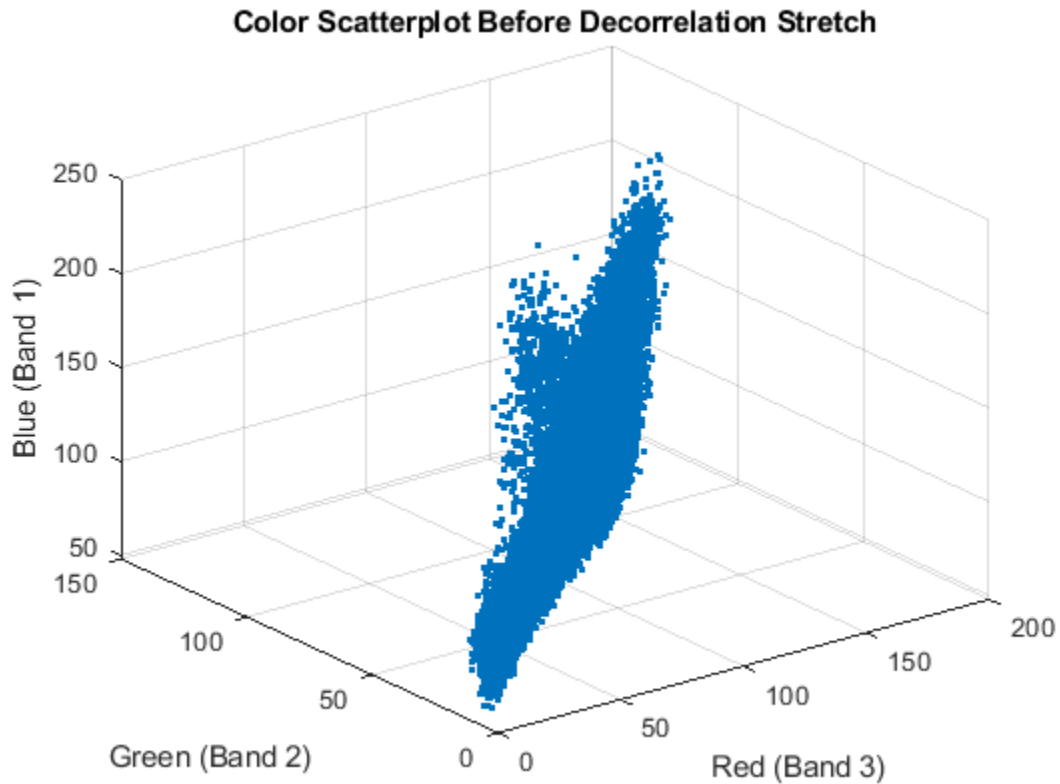
```
figure
plot3(rB(:),gB(:),bB(:),'.')
grid on
xlabel('Red (Band 3)')
ylabel('Green (Band 2)')
zlabel('Blue (Band 1)')
title('Color Scatterplot After Decorrelation Stretch')
```

Color Scatterplot After Decorrelation Stretch
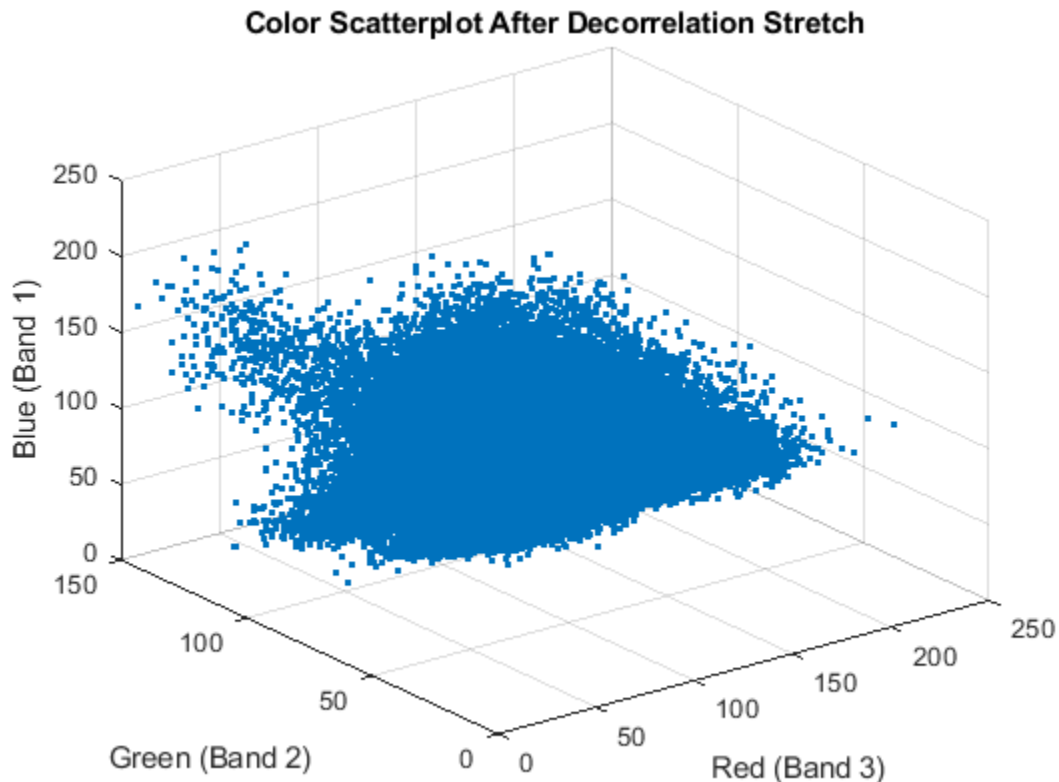
Linear Contrast Stretching

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%.

11-91

```
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
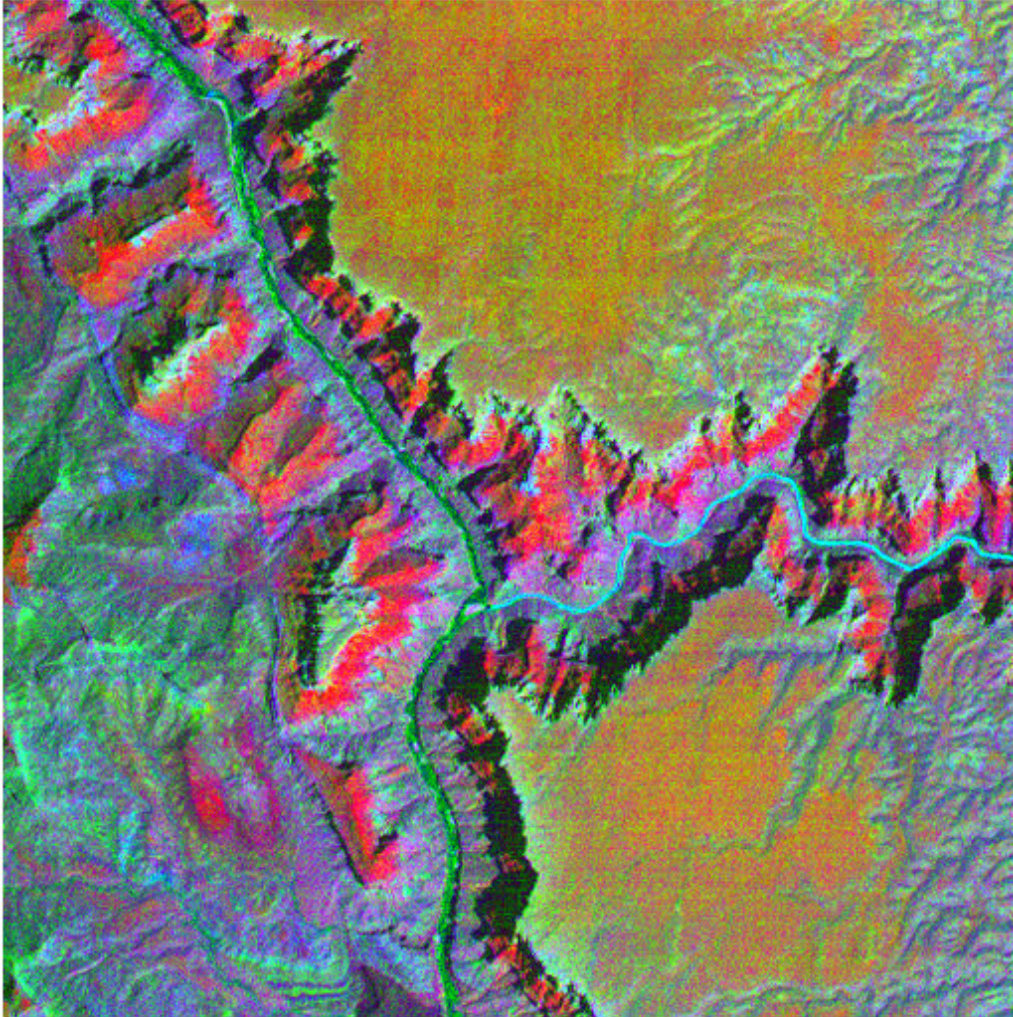
Enhance Multispectral Color Composite Images

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

Create Truecolor Composite from 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 create 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.

```
imshow(truecolor)
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)
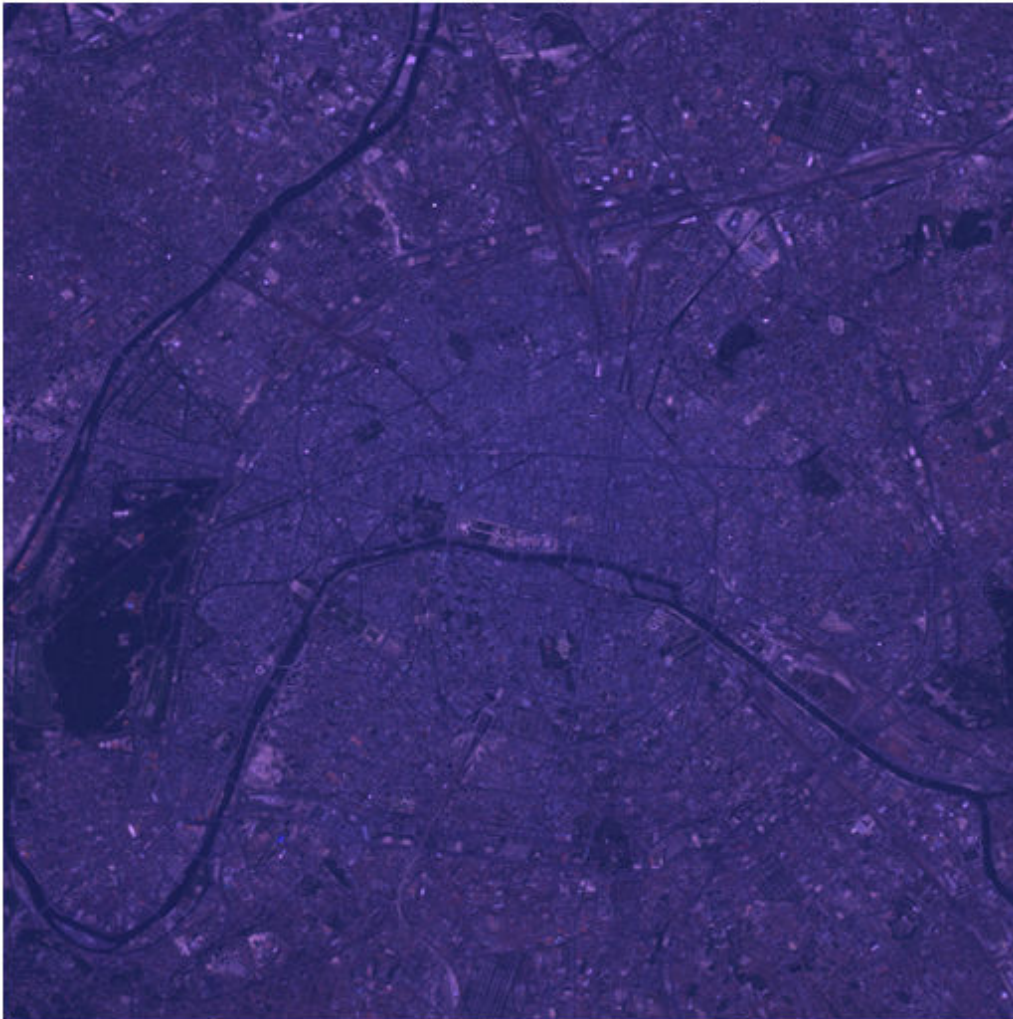
Image courtesy of Space Imaging, LLC

Explore Histogram and Scatterplot of 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.

```
imhist(truecolor(:,:,1))
title('Histogram of the Red Band (Band 3)')
```

Histogram of the Red Band (Band 3)
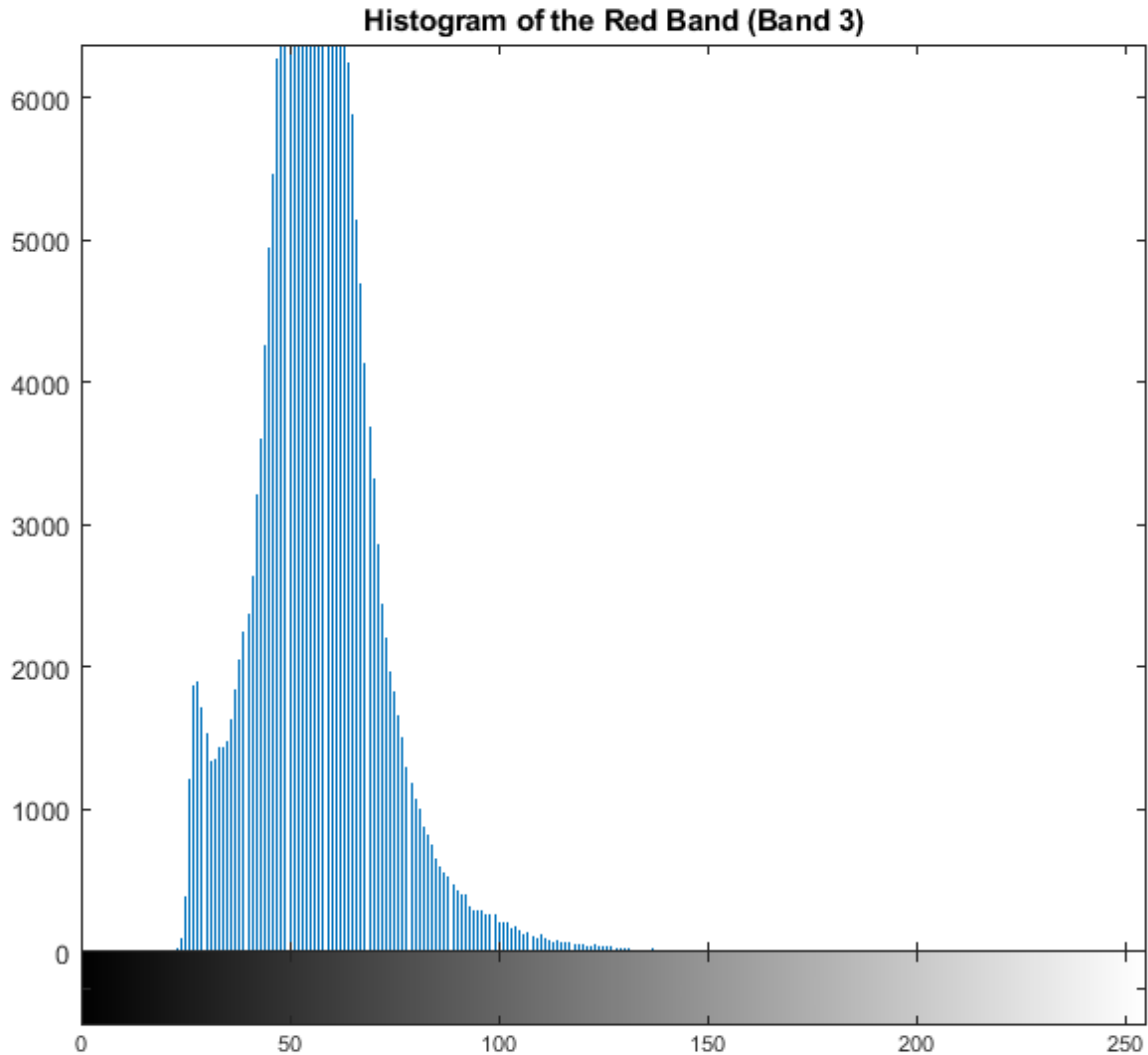
Another reason for the dull appearance of the composite is that the visible bands are highly correlated with each other. Two- and three-band scatterplots are an excellent way to gauge the degree of correlation among spectral bands. You can make them easily just by using `plot`. The linear trend of the red-green-blue scatterplot indicates that the visible bands are highly correlated. This helps explain the monochromatic look of the un-enhanced truecolor composite.

```
r = truecolor(:,:,1);
g = truecolor(:,:,2);
b = truecolor(:,:,3);

plot3(r(:),g(:),b(:),'.')
grid('on')
xlabel('Red (Band 3)')
ylabel('Green (Band 2)')
```

11-95

```
zlabel('Blue (Band 1)')
title('Scatterplot of the Visible Bands')
```


Scatterplot of the Visible Bands
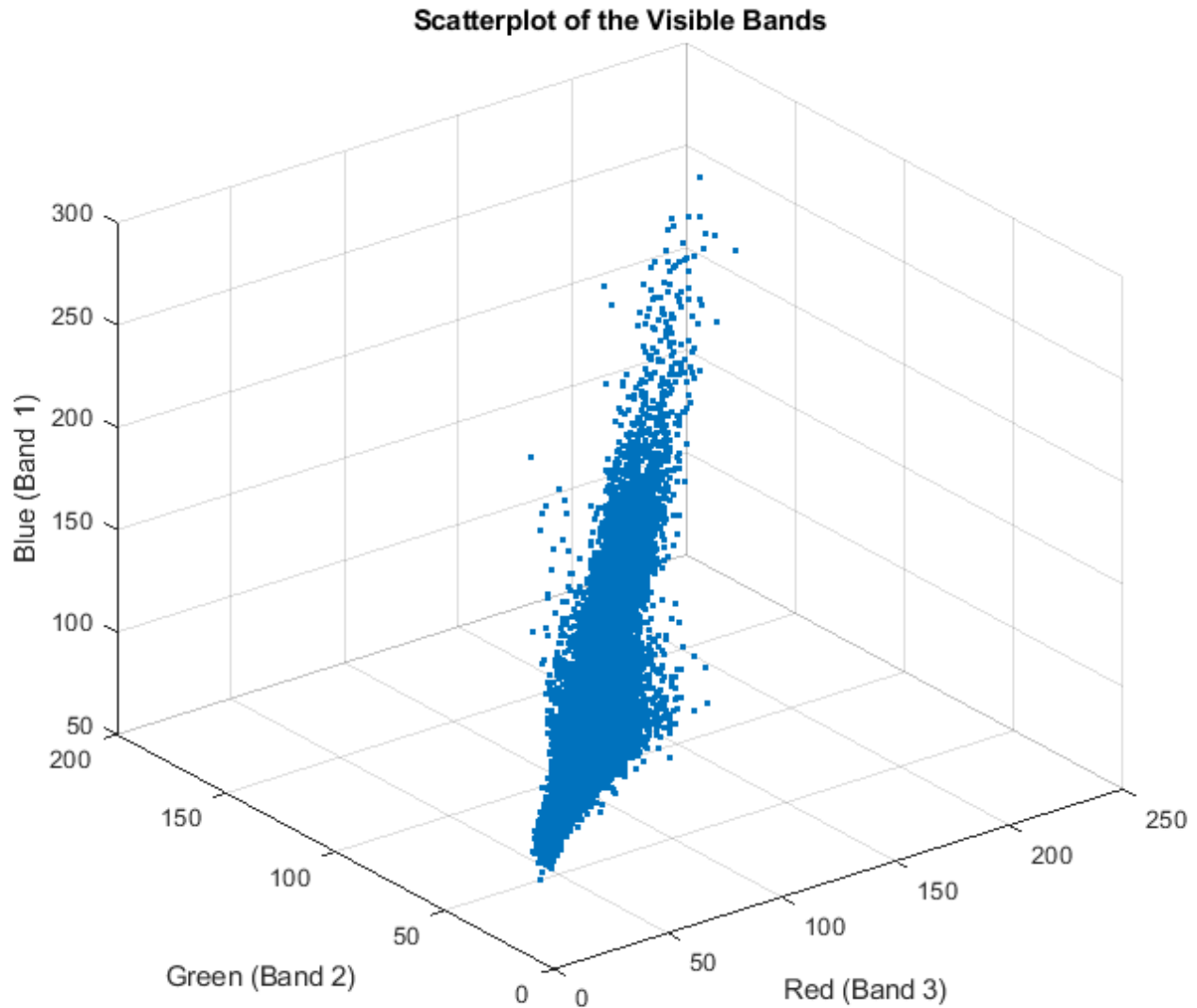
Enhance Truecolor Composite with Contrast Stretch

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));
imshow(stretched_truecolor)
title('Truecolor Composite after Contrast Stretch')
```

Truecolor Composite after Contrast Stretch

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. Create a histogram of all red pixel values in the image by using the `imhist` function.

```
imhist(stretched_truecolor(:,:,1))
title('Histogram of Red Band (Band 3) after Contrast Stretch')
```

11-97

Histogram of Red Band (Band 3) after Contrast Stretch
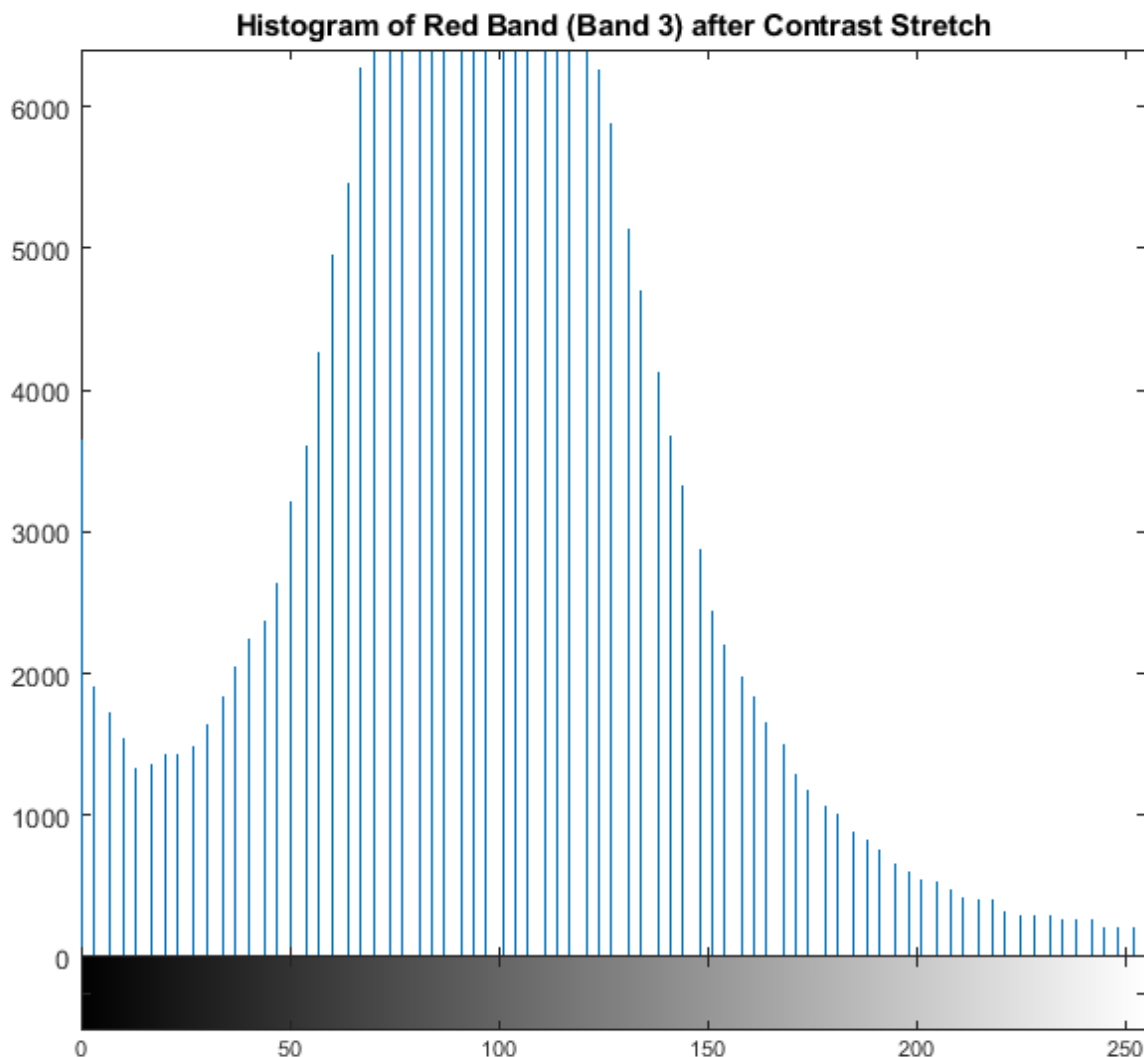
Enhance Truecolor Composite with Decorrelation Stretch

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. Specify the optional name-value pair `'Tol',0.1` to perform a linear contrast stretch after the 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.

```
decorrstretched_truecolor = decorrstretch(truecolor,'Tol',0.01);
imshow(decorrstretched_truecolor)
title('Truecolor Composite after Decorrelation Stretch')
```

Truecolor Composite after Decorrelation Stretch
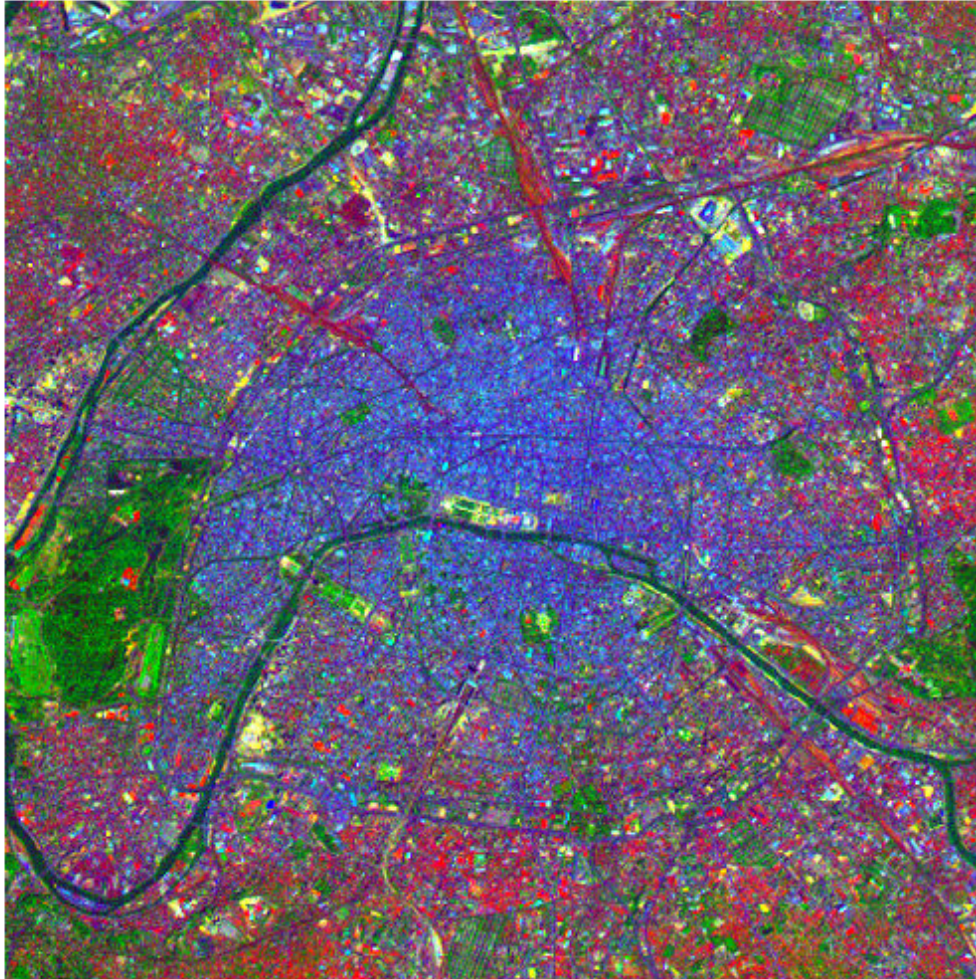
Check Correlation Following Decorrelation Stretch

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);

plot3(r(:),g(:),b(:),'.')
grid('on')
xlabel('Red (Band 3)')
ylabel('Green (Band 2)')
zlabel('Blue (Band 1)')
title('Scatterplot of the Visible Bands after Decorrelation Stretch')
```

Scatterplot of the Visible Bands after Decorrelation Stretch
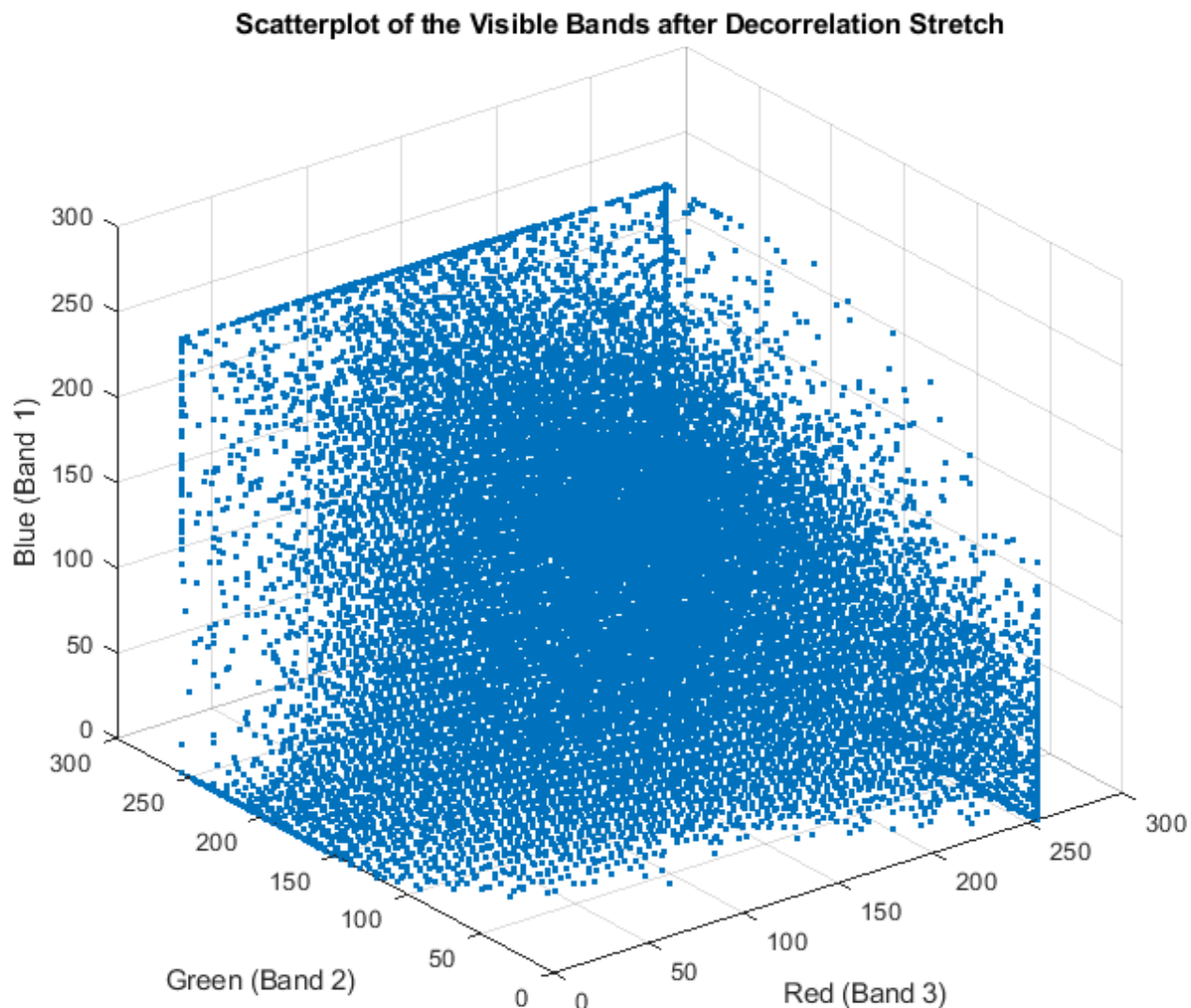
Create and Enhance CIR Composite

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. 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.

```
stretched_CIR = decorrstretch(CIR,'Tol',0.01);
imshow(stretched_CIR)
title('CIR after Decorrelation Stretch')
```


CIR after Decorrelation Stretch

See Also

decorrstretch | imadjust | imhist | multibandread | plot3 | stretchlim

Related Examples

- "Enhance Color Separation Using Decorrelation Stretching" on page 11-87

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);
```
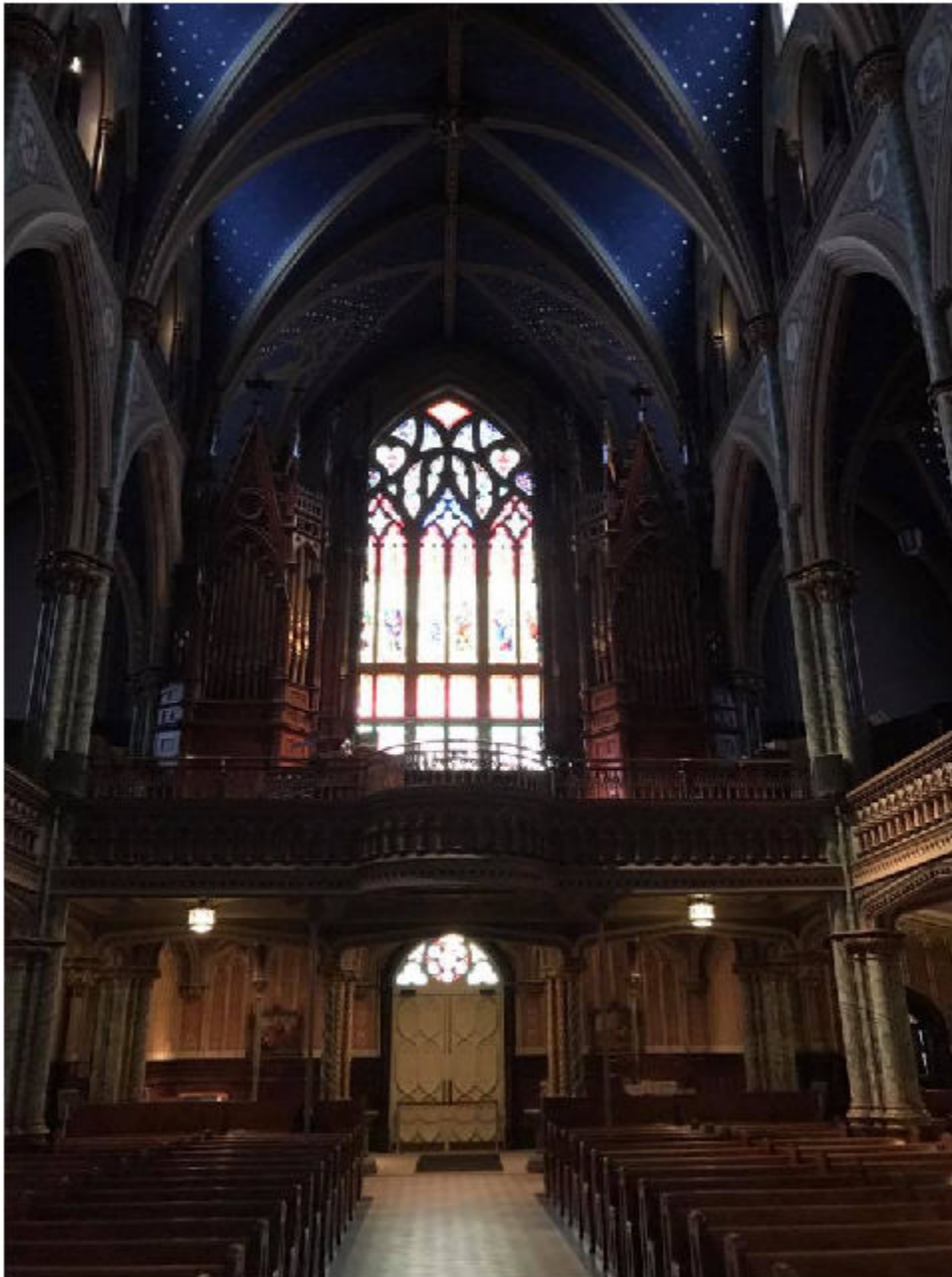
Invert the image and notice how the low-light areas in the original image appear hazy.

```
AInv = imcomplement(A);
figure, imshow(AInv);
```
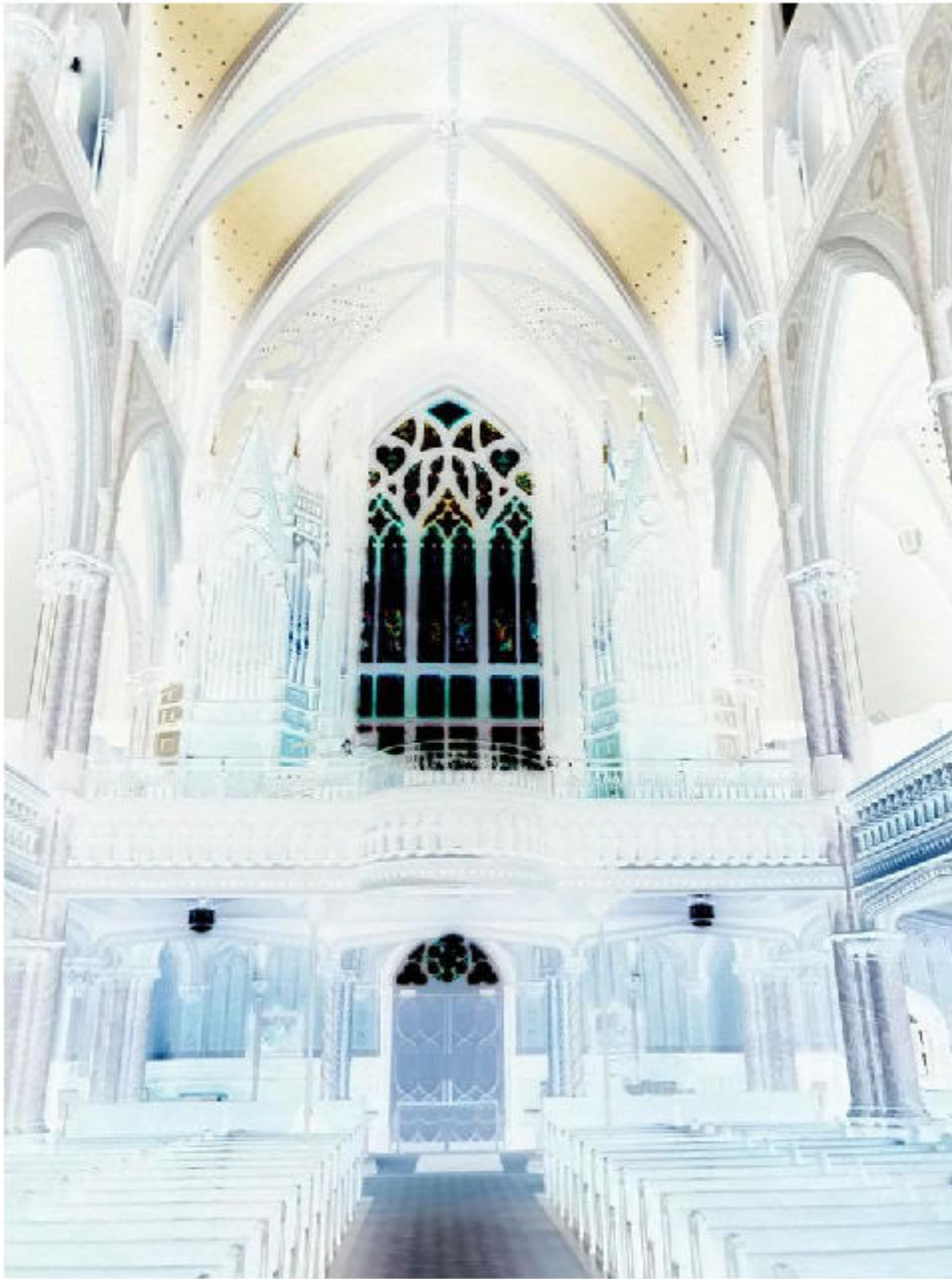
Reduce the haze using the `imreducehaze` function.

```
BInv = imreducehaze(AInv);
figure, imshow(BInv);
```
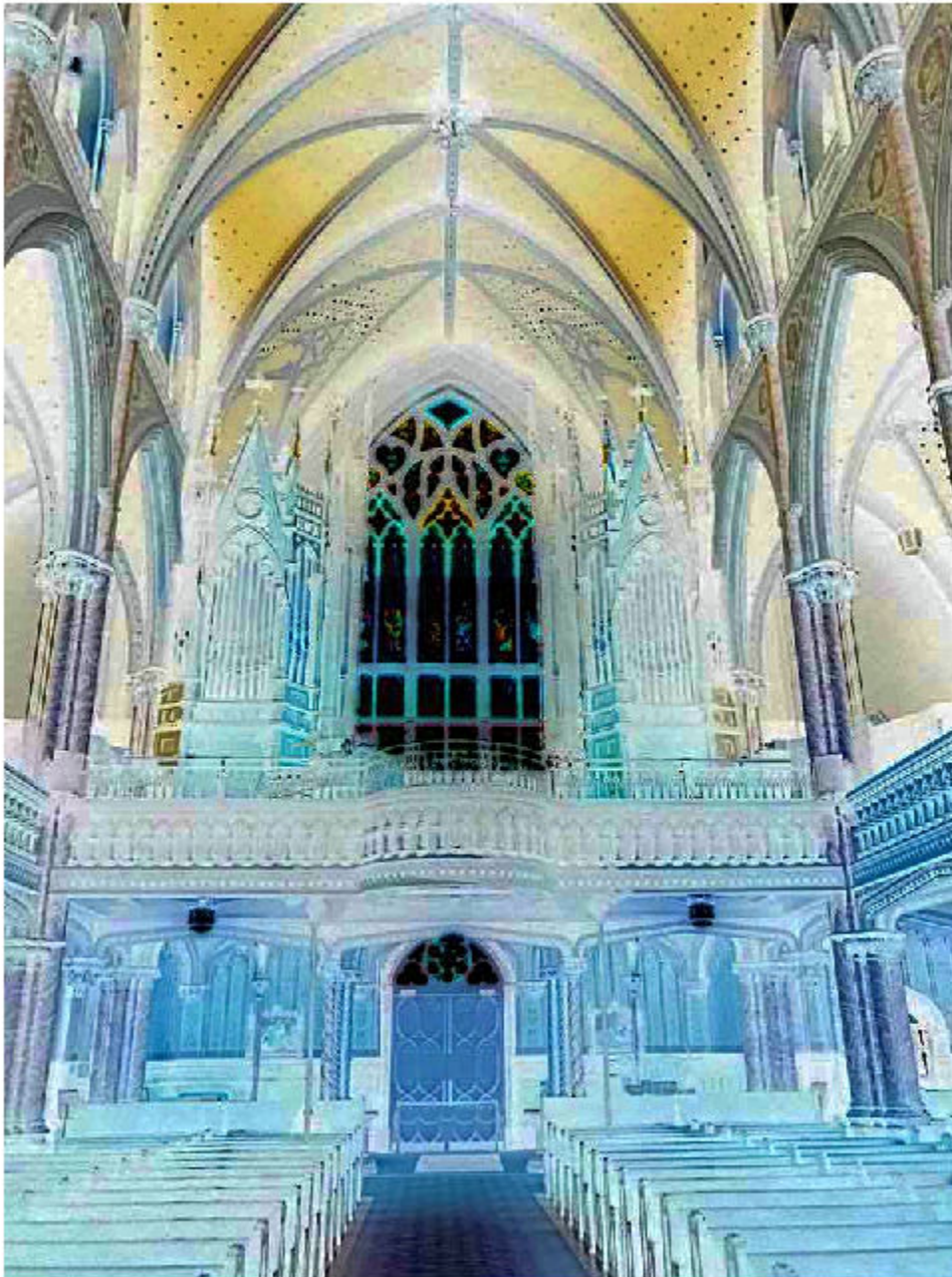
Invert the results to obtain the enhanced image.

```
B = imcomplement(BInv);
```

Display the original image and the enhanced images, side-by-side.

```
figure, montage({A, B});
```
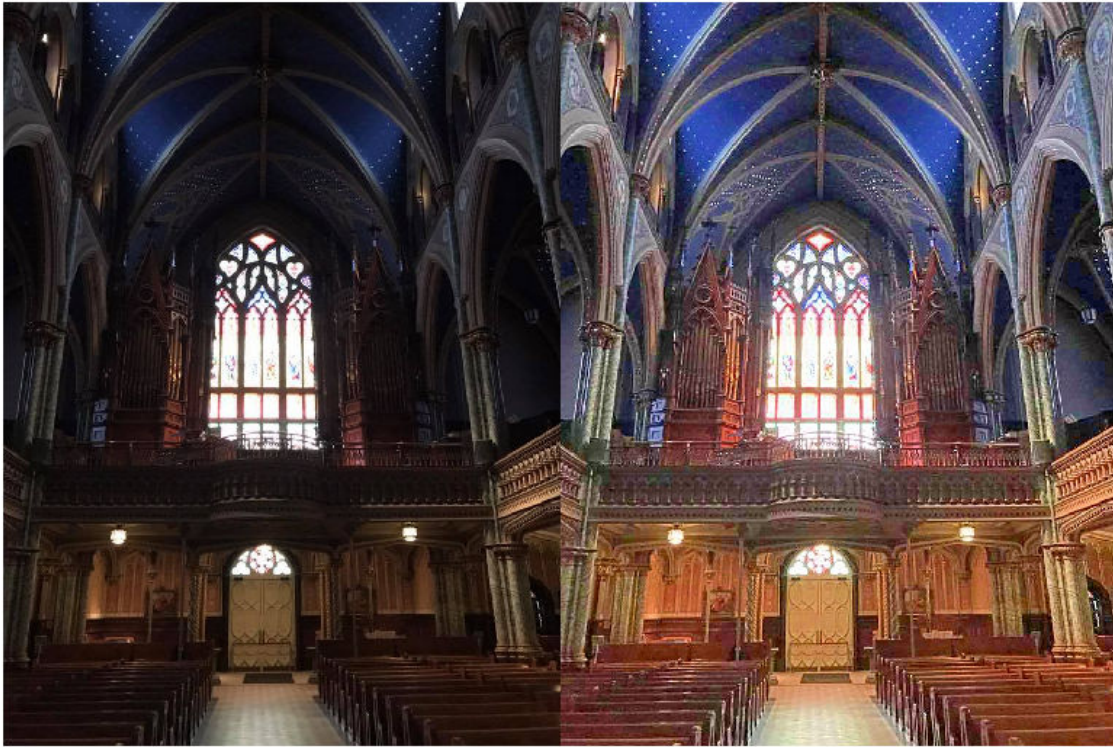

Improve Results Further Using `imreducehaze` Optional Parameters

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});
```
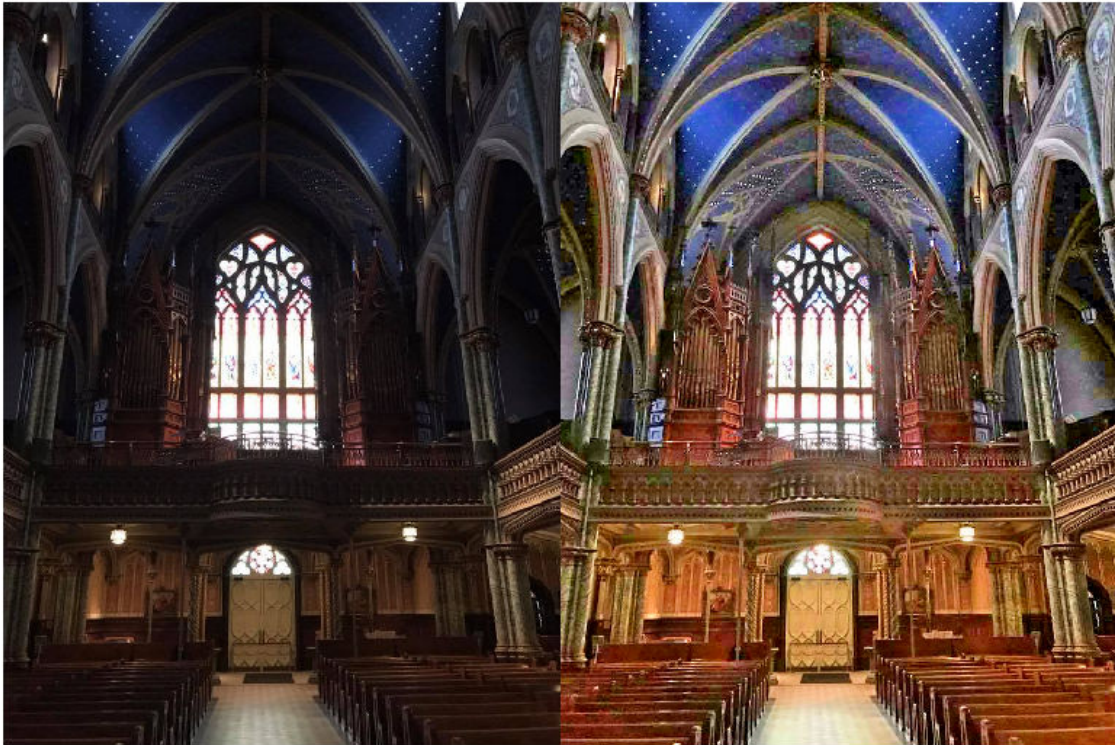
Another Example of Improving a Poorly Lit Image

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});
```
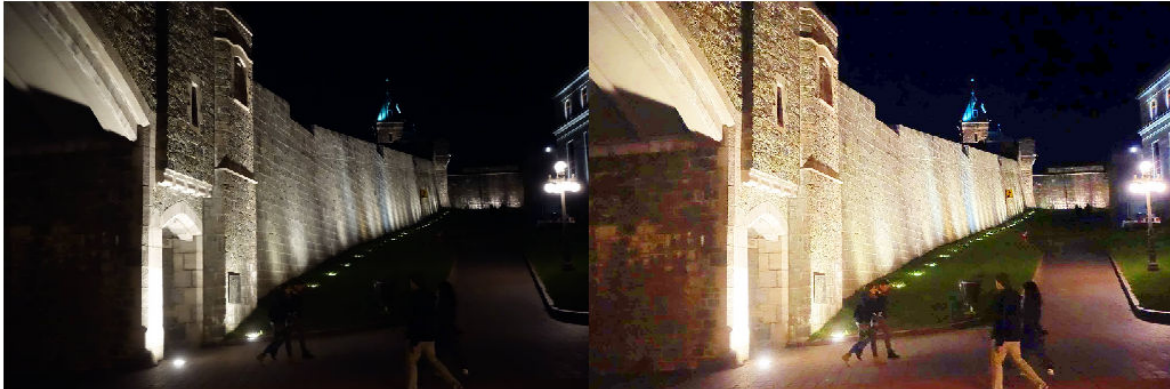
Reduce Color Distortion by using a Different Color Space

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});
```
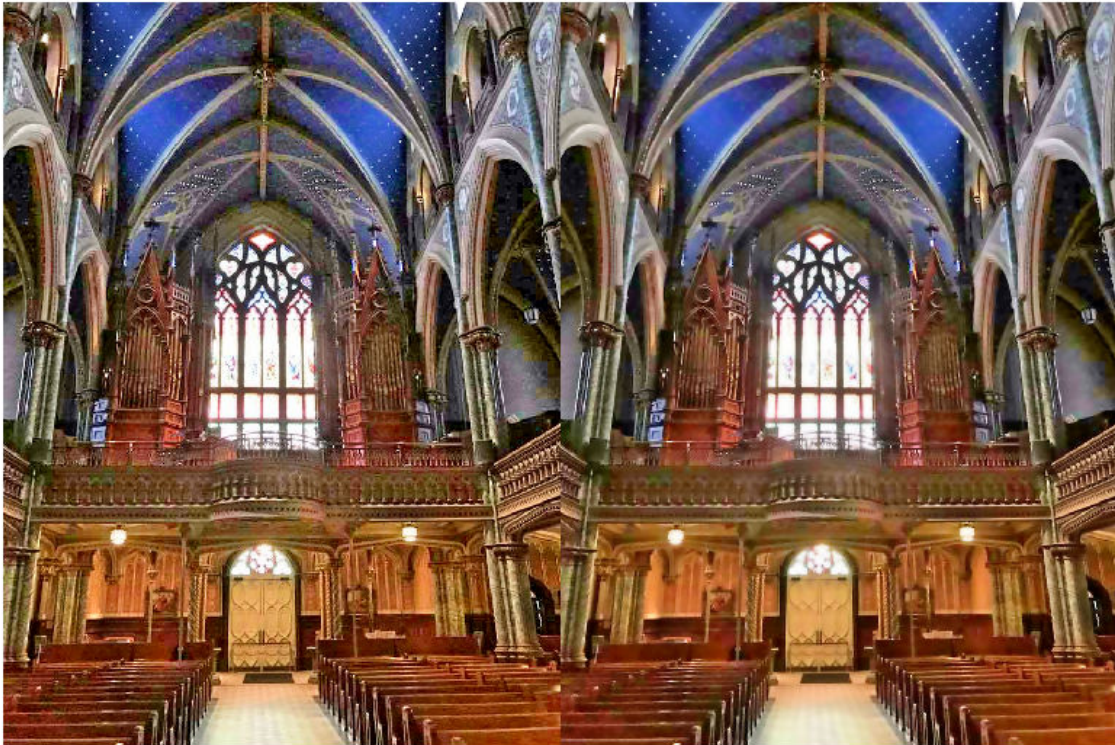
Estimate Illumination Map

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));
```

Lowlight Image · Illumination Map

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

Increase Filter Strength Radially Outward

This example shows how to create and apply filters whose strength increases radially outward.

Read and display an image.

```
I = imread('peppers.png');
I = im2double(I);
imshow(I)
```


Blur Image Using Gaussian Weighting Function

Create a blurry copy of the image using a Gaussian filter with standard deviation of 2.

```
Iblurred = imgaussfilt(I,2);
imshow(Iblurred)
```

Create a weight image as a Gaussian filter of the same size of the image. To increase the portion of the image that appears sharp, increase the value of `filterStrength`.

```
filterStrength = 50;
weights = fspecial('gaussian',[size(I,1) size(I,2)],filterStrength);
imshow(weights,[])
```
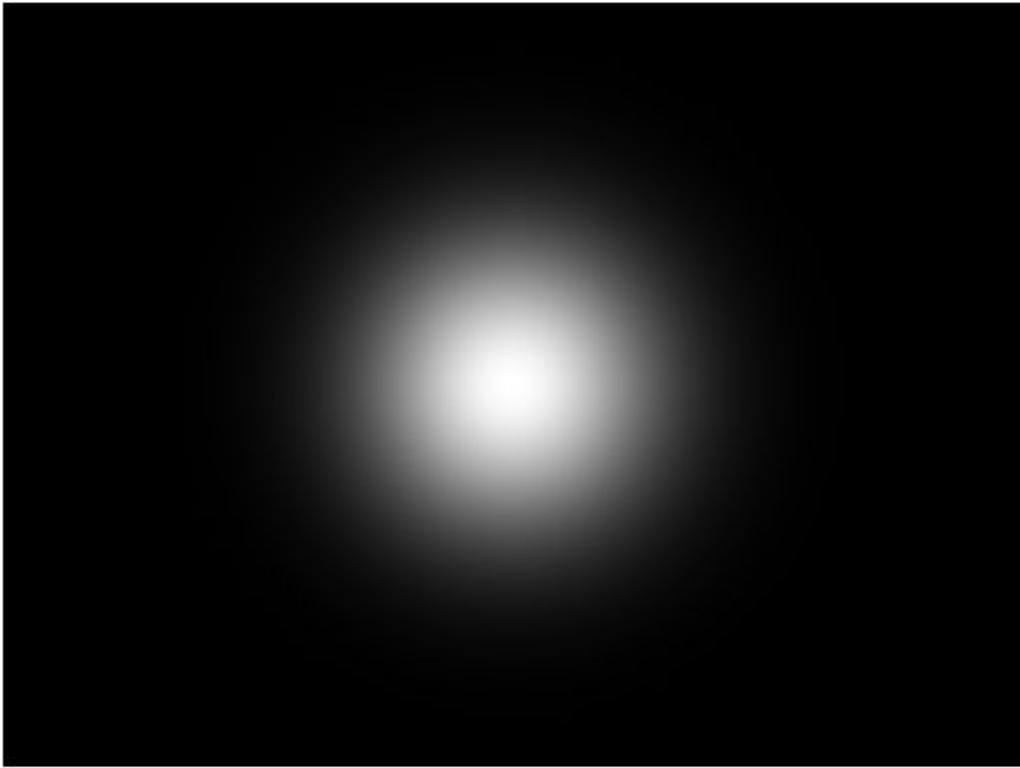
Normalize the weight image to the range [0, 1] by using the `rescale` function.

```
weights = rescale(weights);
```

Create a weighted blurred image that is a weighted sum of the original image and blurry image. MATLAB automatically replicates the weight matrix for each of the R, G, and B color channels.

```
IweightedBlurred = I.*weights + Iblurred.*(1-weights);
```

Display the result. The image is sharp in the center and becomes more blurry radially outwards. To increase the portion of the image that appears sharp, increase the value of

```
imshow(IweightedBlurred)
```

Vignette Image Using 1/R^2 Weighting Function

Get the size of the image.

```
sizex = size(I,2);
sizey = size(I,1);
```

Specify the center of the vignette.

```
xcenter = size(I,2)/2;
ycenter = size(I,1)/2;
```

Define the x- and y-coordinates of the surface.

```
[X,Y] = meshgrid((1:sizex)-xcenter,(1:sizey)-ycenter);
```

Define the radius from the center at each (x,y) coordinate.

```
R2 = X.^2 + Y.^2;
```

Define the weighting function as the inverse of R, scaled to the range [0, 1].

```
R2 = rescale(R2);
weights = (1-R2);
imshow(weights)
```
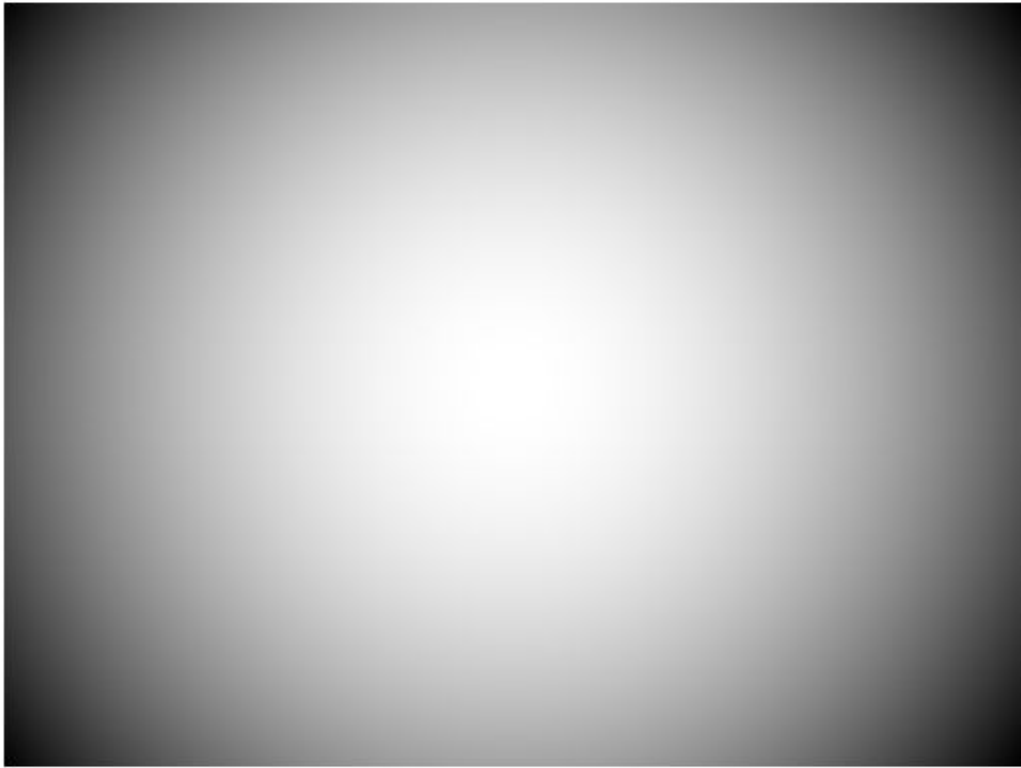
Apply the weighting function to the image and display the result.

```
I2 = I.*weights;
imshow(I2)
```

See Also
fspecial | imgaussfilt

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')
```


Original image

```
figure
imshow(Iblur1)
title('Smoothed image, \sigma = 2')
```

Smoothed image, $\sigma = 2$


```
figure
imshow(Iblur2)
title('Smoothed image, \sigma = 4')
```

Smoothed image, $\sigma = 4$


```
figure
```

```
imshow(Iblur3)
title('Smoothed image, \sigma = 8')
```

Smoothed image, $\sigma = 8$
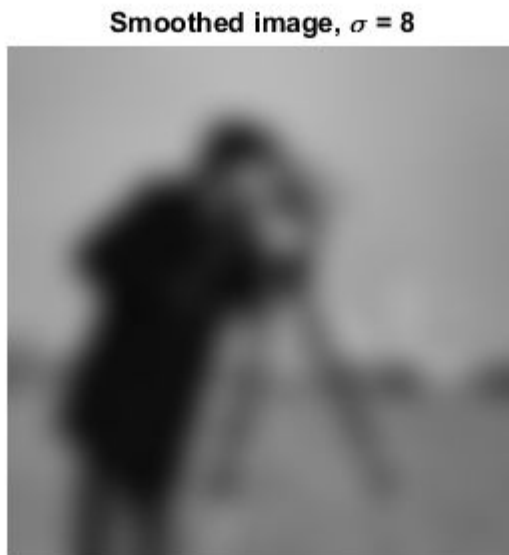
Filter the image with anisotropic Gaussian smoothing kernels. `imgaussfilt` allows the Gaussian kernel to have different standard deviations along row and column dimensions. These are called axis-aligned anisotropic Gaussian filters. Specify a 2-element vector for `sigma` when using anisotropic filters.

```
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')
```

Smoothed image, $\sigma_x = 4$, $\sigma_y = 1$


```
figure
imshow(IblurX2)
title('Smoothed image, \sigma_x = 8, \sigma_y = 1')
```

Smoothed image, $\sigma_x = 8$, $\sigma_y = 1$


```
figure
```

```
imshow(IblurY1)
title('Smoothed image, \sigma_x = 1, \sigma_y = 4')
```


Smoothed image, $\sigma_x = 1$, $\sigma_y = 4$

```
figure
imshow(IblurY2)
title('Smoothed image, \sigma_x = 1, \sigma_y = 8')
```


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')
```

Sky in original image


```
figure
imshow(IblurX1_sky), title('Sky in filtered image')
```

Sky in filtered image

Noise Removal

In this section...
"Remove Noise by Linear Filtering" on page 11-125
"Remove Noise Using an Averaging Filter and a Median Filter" on page 11-125
"Remove Noise By Adaptive Filtering" on page 11-128

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)
```
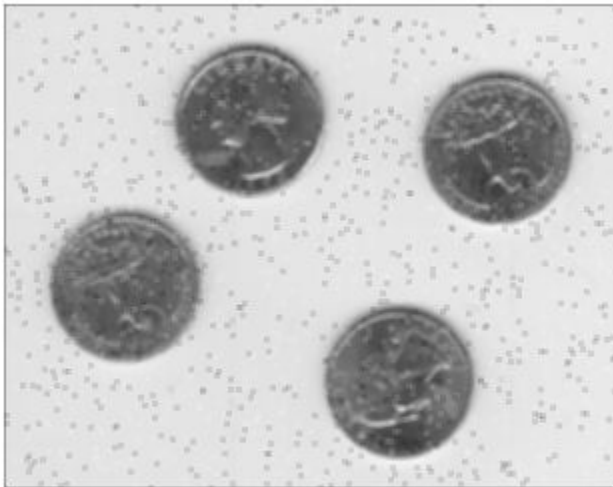

Now use a median filter to filter the noisy image, J. The example also uses a 3-by-3 neighborhood. Display the two filtered images side-by-side for comparison. Notice that medfilt2 does a better job of removing noise, with less blurring of edges of the coins.

```
Kmedian = medfilt2(J);
imshowpair(Kaverage,Kmedian,'montage')
```
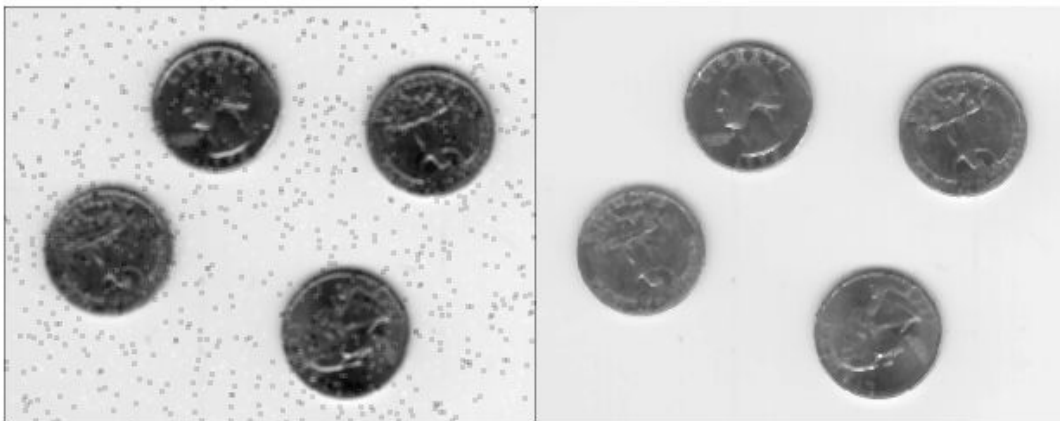
Remove Noise By Adaptive Filtering

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 = im2gray(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');
```

Portion of the Image with Added Gaussian Noise
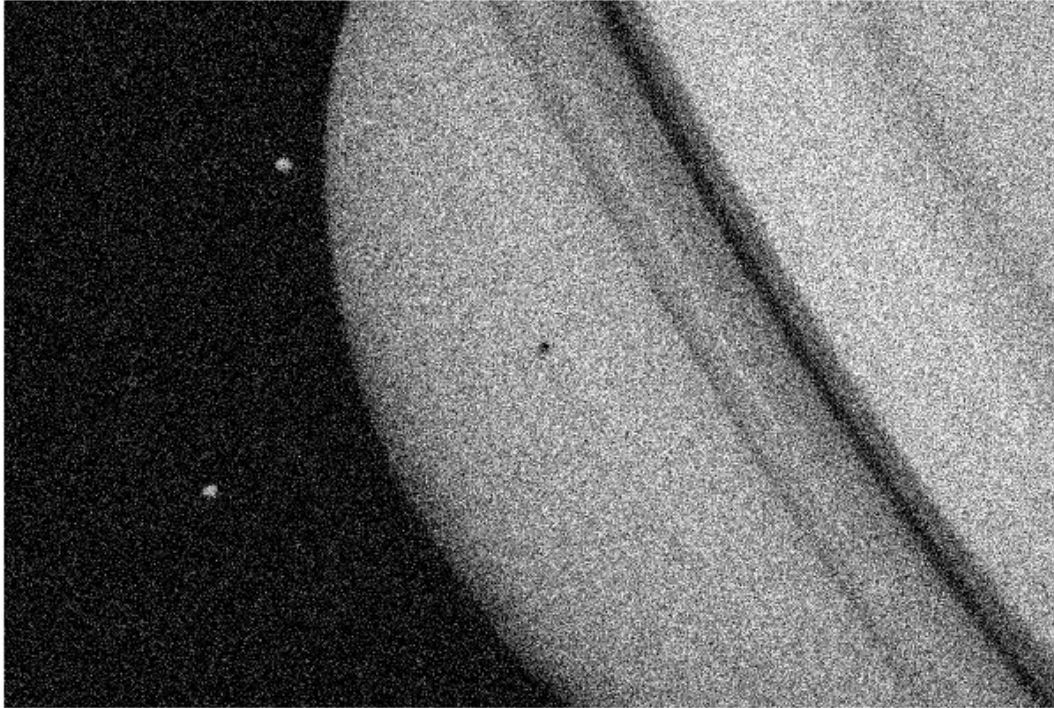
Remove the noise using the `wiener2` function.

```
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');
```

Portion of the Image with Noise Removed by Wiener Filter
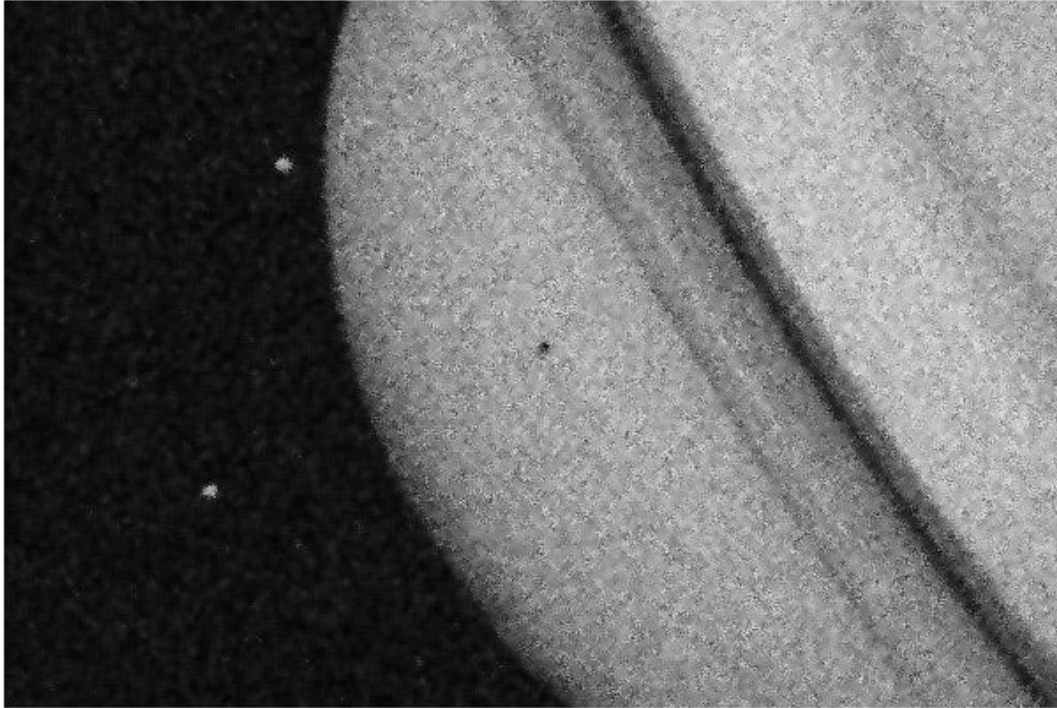
See Also

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)
```
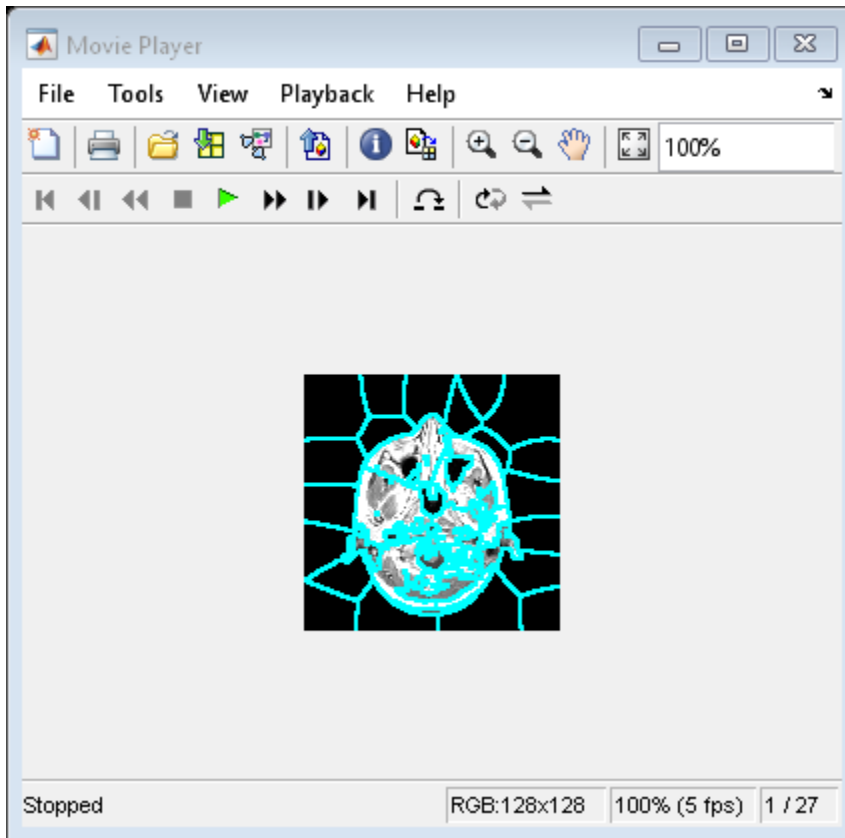
Set the color of each pixel in output image to the mean intensity of the superpixel region. Show the mean image next to the original. If you run this code, you can use `implay` to view each slice of the MRI data.

```
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);
```
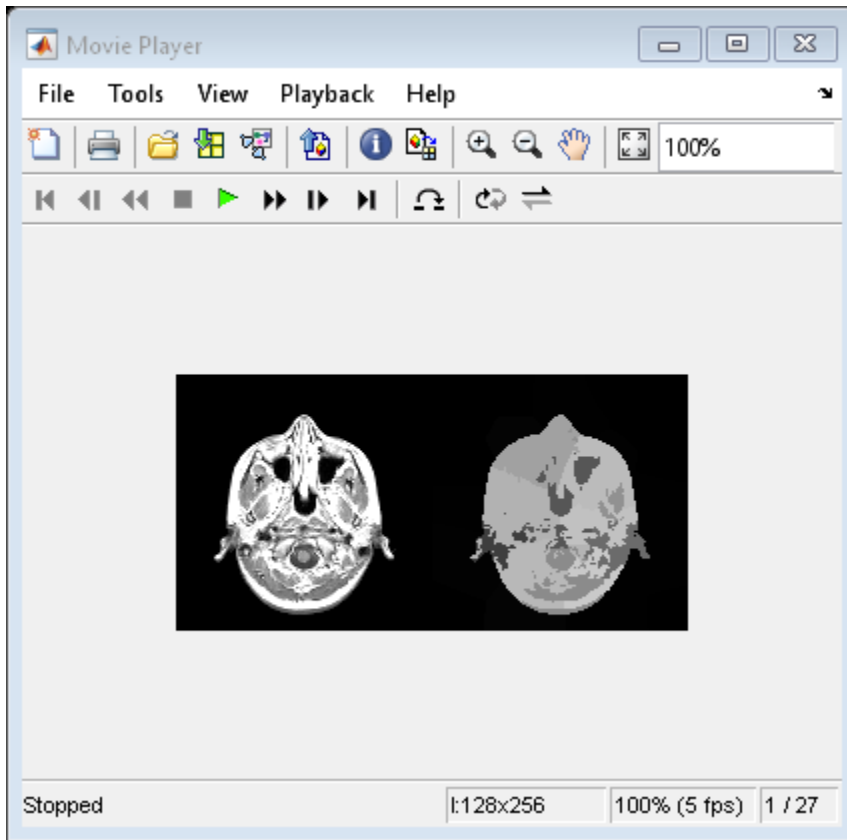
Image Quality Metrics

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.

Metric	Description
`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.
`multissim` `multissim3`	Multi-scale structural similarity (MS-SSIM) index. The MS-SSIM metric expands on the SSIM index by combining luminance information at the highest resolution level with structure and contrast information at several downsampled resolutions, or scales. The MS-accounts for variability in the perception of image details caused by factors such as viewing distance from the image, distance from the scene to the sensor, and resolution of the image acquisition sensor.

Because structural similarity is computed locally, `ssim`, `multissim`, and `multissim3` 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.

Metric	Description
`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 No-Reference Quality Assessment Model" on page 11-136
- "Obtain Local Structural Similarity Index" on page 11-146
- "Compare Image Quality at Various Compression Levels" on page 11-148

Train and Use 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.

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 11-136.

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.
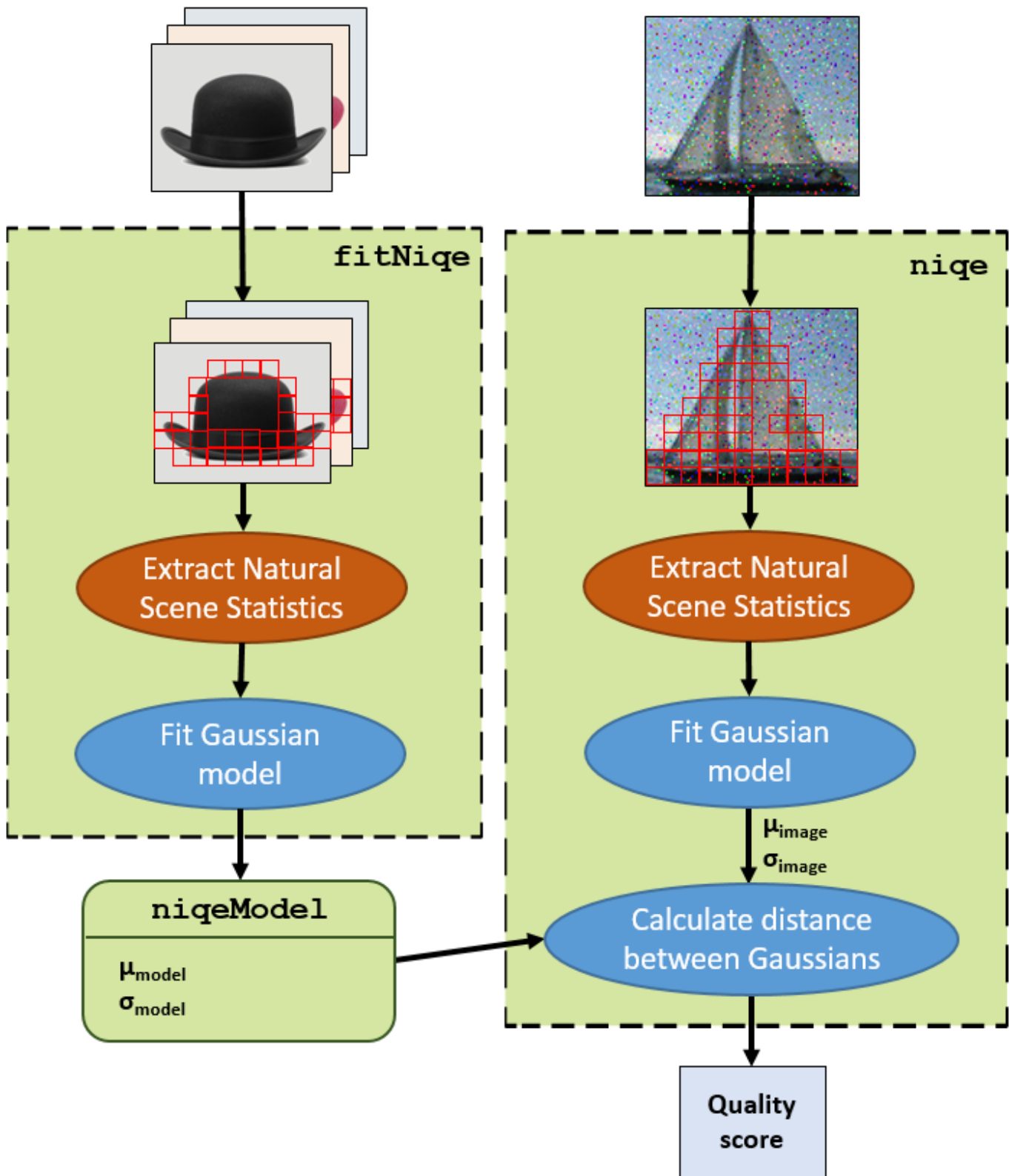
BRISQUE Workflow

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 11-138.

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.
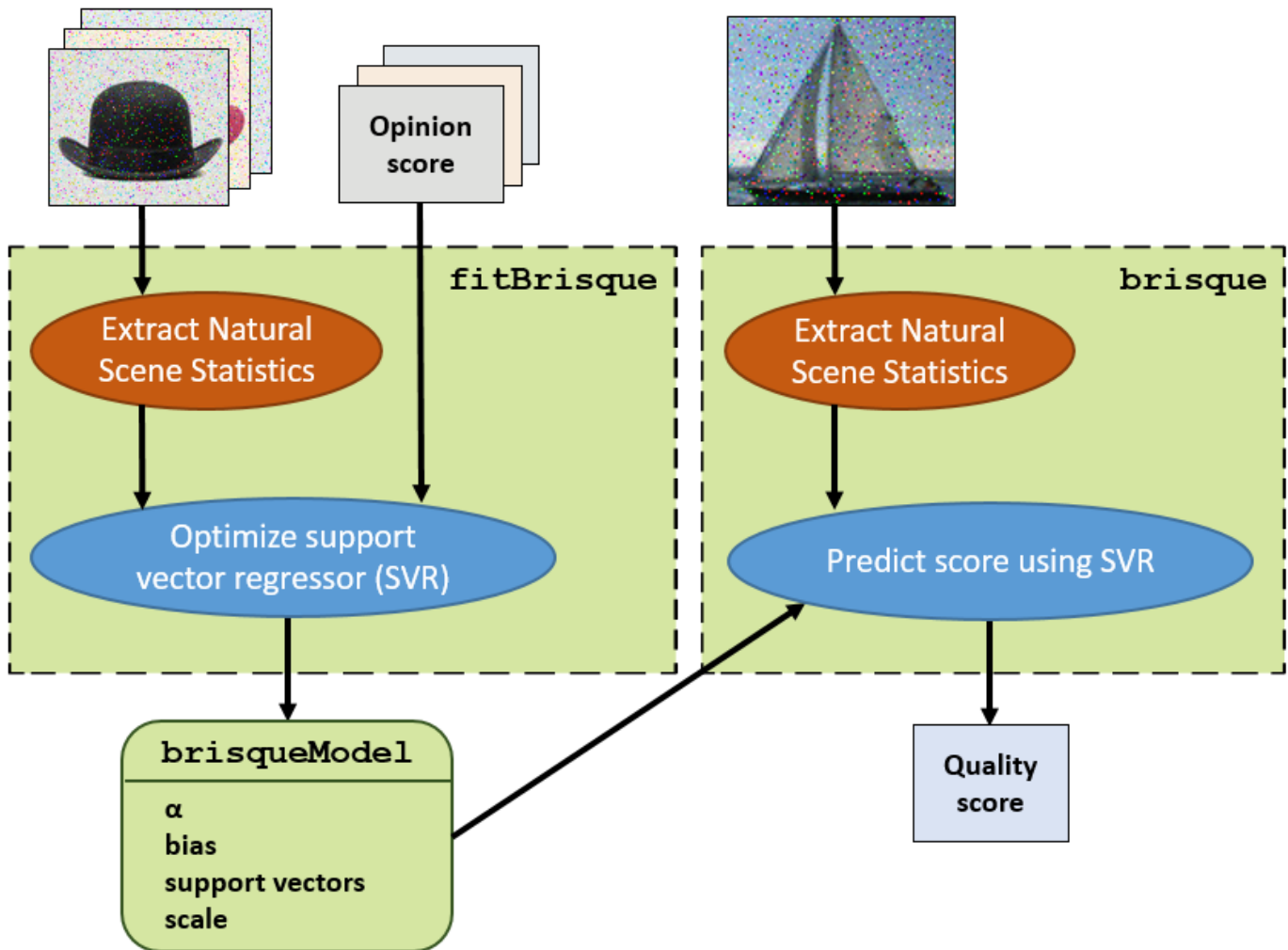
See Also

brisque | brisqueModel | fitbrisque | fitniqe | niqe | niqeModel

More About

- "Image Quality Metrics" on page 11-134
- "Compare No Reference Image Quality Metrics" on page 11-140

Compare No Reference Image Quality Metrics

This example shows how to compare the performance of various blind or no-reference image quality metrics.

Evaluating the quality of an image is an important part of image acquisition, compression, and other image enhancement workflows. It is desirable to have a fast, automated metric that closely mimics subjective measures of image quality. This example compares the performance of three no-reference quality metrics.

- BRISQUE - Blind/Referenceless image spatial quality evaluator
- NIQE - Naturalness image quality evaluator
- PIQE - Perception-based image quality evaluator

Each metric has different strengths depending on the images in the data set. To select the best metric for your data, you can compare the performance of the three metrics on sample image data. This example shows how to compare the performance in two different situations: varying levels of JPEG compression on a single image and for a video stream.

Evaluate Response to Varying Compression on Single Image

Image compression is a tradeoff between visual quality and the compression ratio, or size of the output data. The tradeoff also depends on the image content. For example, images with uniform areas can compress to smaller file sizes and exhibit fewer artifacts than images with detailed features. Image quality metrics can help analyze this tradeoff, while trying to minimize the impact of the image content on the analysis.

Read an image into the workspace.

```
im = imread('llama.jpg');
```

Write copies of the image with different JPG compression ratios. Read each compressed image back into the workspace.

```
jpegQuality = 10:10:100;
numObservations = numel(jpegQuality);
compressedFrames = cell(1,numObservations);
for ind = 1:numObservations
    q = jpegQuality(ind);
    tempFile = ['llama_compression_',num2str(q),'.jpg'];
    imwrite(im,tempFile,'Quality',q);
    compressedFrames{ind} = imread(tempFile);
end
```

Inspect the compressed images.

```
tiledlayout(1,3);

h1 = nexttile;
imshow(compressedFrames{1})
title('JPEG Quality: 10')

nexttile
imshow(compressedFrames{7})
title('JPEG Quality: 70')
```

```
nexttile
imshow(im)
title('Input Image')
linkaxes
```


Zoom in on the compressed image to see the nature of some specific artifacts. At JPEG quality 10, the blocking artifacts are obvious.

```
h1.XLim = [650 700];
h1.YLim = [490 550];
```

For each compressed JPG image, calculate the quality score using the three quality metrics.

```
pQ = zeros(1, numObservations);
nQ = zeros(1, numObservations);
bQ = zeros(1, numObservations);

for ind=1:numObservations
    bQ(ind) = brisque(compressedFrames{ind});
    nQ(ind) = niqe(compressedFrames{ind});
    pQ(ind) = piqe(compressedFrames{ind});
end
```

Visualize the score of the metrics as the JPEG quality increases. Normalize the scores so that each score has the same value for the uncompressed image. For these three metrics, lower scores correspond to higher image quality.

The BRISQUE score for a JPEG quality of 50, 60, and 70 is unrealistically lower than for uncompressed JPEG images. Therefore, for images similar to this test image, NIQE and PIQE are more reliable metrics.

```
figure
hold on
plot(jpegQuality,bQ/bQ(end),'*-');
plot(jpegQuality,nQ/nQ(end),'*-');
plot(jpegQuality,pQ/pQ(end),'*-');
legend('BRISQUE','NIQE','PIQE');
ylabel('Metric Score')
```

```
xlabel('JPEG Quality')
hold off
```
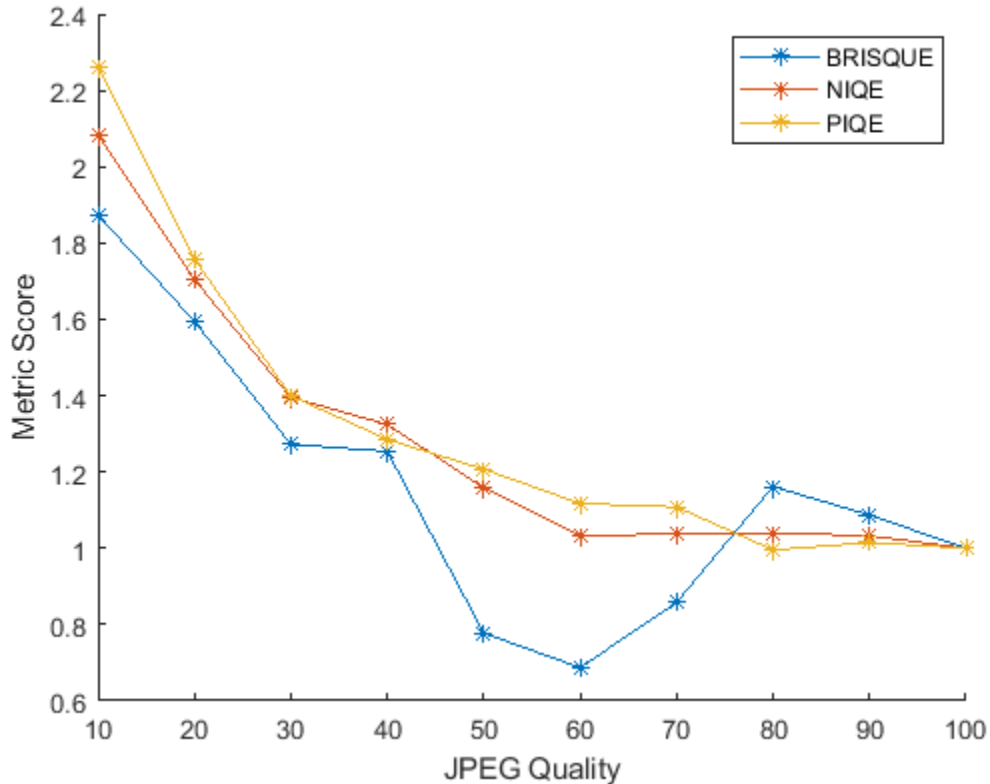

Evalute Response to Varying Compression and Content using a Video

In applications such as streaming video, there is a need to evaluate quality metrics at the receiver which may not have access to the original pristine sample. Also, the content of each frame can vary significantly. Let us simulate such a scenario to evelute the performance characteristics of these metrics.

Create a VideoReader object that reads frames from the video 'rhinos.avi'. This video has 114 frames.

```
vidObjR = VideoReader('rhinos.avi');
vidObjW = VideoWriter('varyingCompressed.avi');
open(vidObjW)
```

Create a varying compression ratio schedule to mimic real time varying bitrate transmissions

```
numFrames = vidObjR.NumFrames;
varyingQuality = sin(2*pi*(1:numFrames)*0.01);
varyingQuality = round(rescale(varyingQuality)*100);
varyingQuality = max(varyingQuality,1); % min JPEG quality is 1

figure
plot(varyingQuality);
title('JPEG Quality Schedule');
ylabel('JPEG Quality')
xlabel('Frame Index')
```
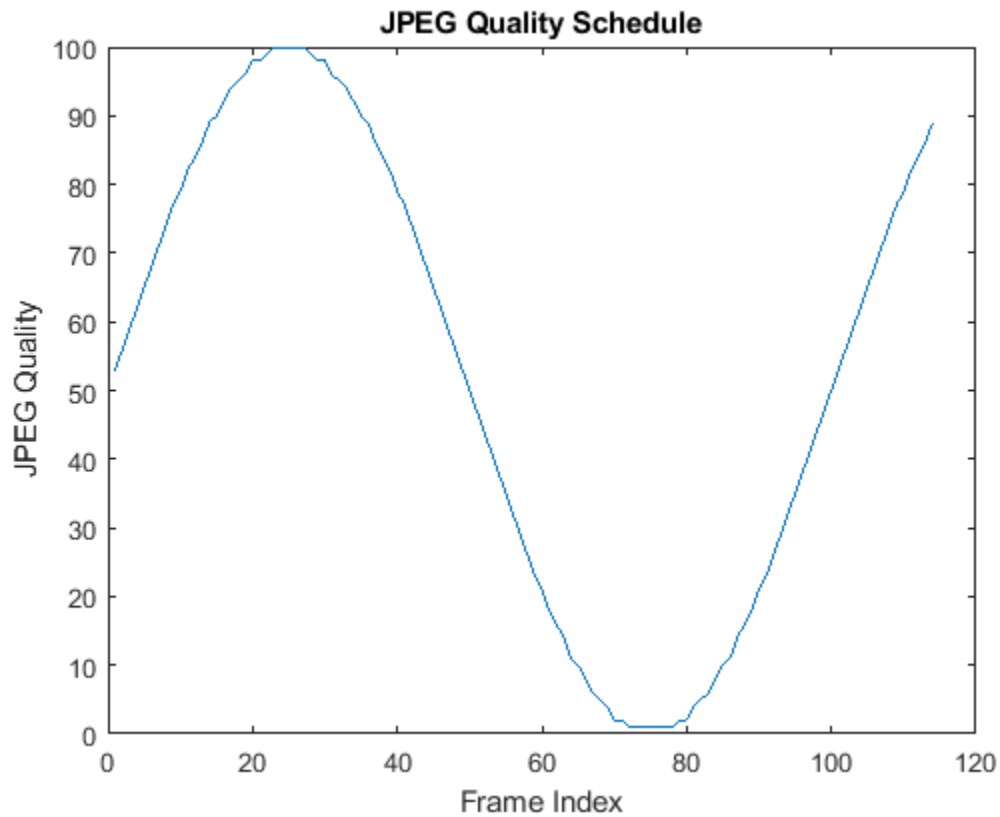
For each frame in the video, compress the frame according to the JPEG quality schedule. Compute the metrics of the compressed frame and add the compressed frame to the output video for validation.

```
pQ = zeros(1,numFrames);
nQ = zeros(1,numFrames);
bQ = zeros(1,numFrames);

ind = 1;
while hasFrame(vidObjR)
    im = readFrame(vidObjR);

    % Compress it based on the schedule
    tempFile = 'rhinos_compressed_frame.jpg';
    imwrite(im,tempFile,'Quality',varyingQuality(ind));
    frame = imread(tempFile);

    writeVideo(vidObjW,frame);

    bQ(ind) = brisque(frame);
    nQ(ind) = niqe(frame);
    pQ(ind) = piqe(frame);
    ind = ind+1;
end
close(vidObjW);
```

Visualize the trend, expect it to mimic the compression schedule. Rescale the metrics to focus on the trend, and invert the quality schedule to get the compression ratio trend. Quality metrics can still give a useful indication of perceived quality without access to the original reference frame.

```
figure
hold on
plot(rescale(bQ));
plot(rescale(nQ));
plot(rescale(pQ));
% Invert JPEG Quality to get the compression ratio
plot(1-rescale(varyingQuality),'k','LineWidth',2)
legend('BRISQUE','NIQE','PIQE','Compression Ratio');
title('Trend of Quality Metrics with Varying Compression and Content');
ylabel('Metric Score')
xlabel('Frame Index')
hold off
```
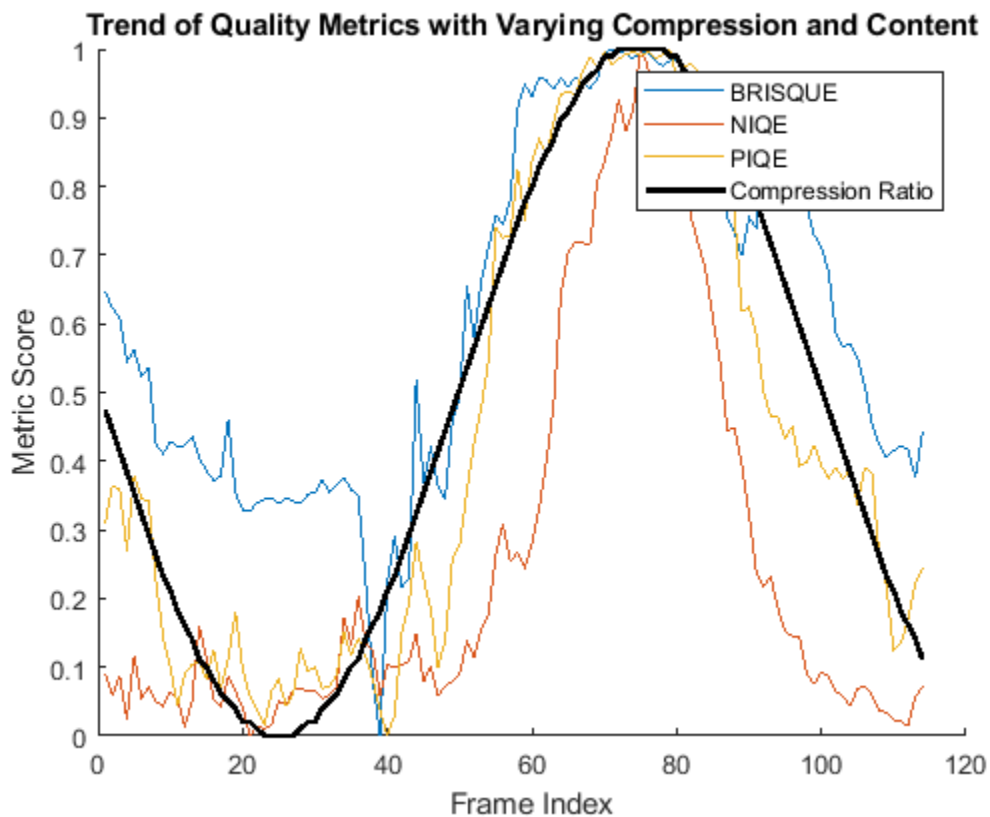

See Also

brisque | brisqueModel | fitbrisque | fitniqe | niqe | niqeModel

More About

- "Image Quality Metrics" on page 11-134
- "Train and Use No-Reference Quality Assessment Model" on page 11-136

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,[])
```
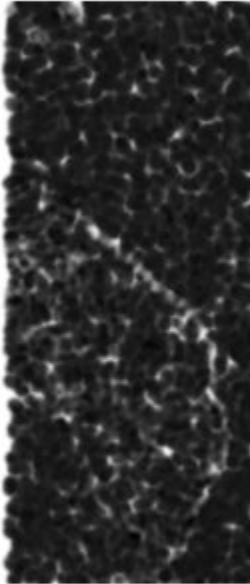

See Also

ssim

More About

*   "Image Quality Metrics" on page 11-134

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');
```
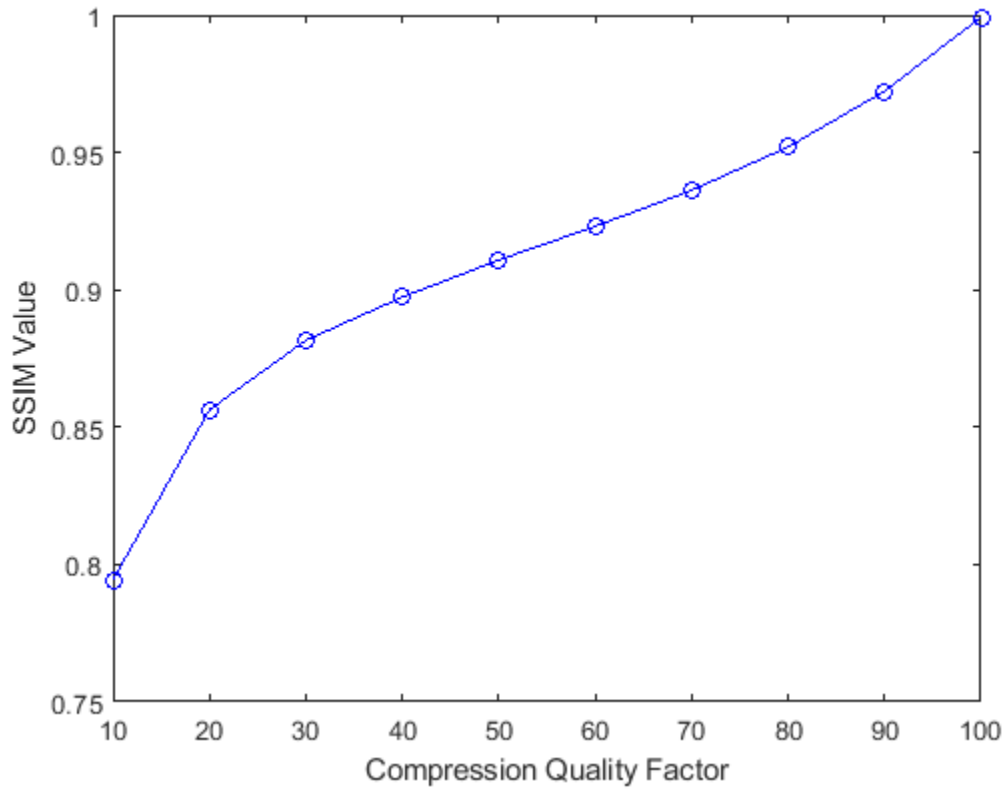
See Also

ssim

More About

*   "Image Quality Metrics" on page 11-134

Anatomy of Imatest Extended 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], [2]. 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. This version has a 16:9 aspect ratio and 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
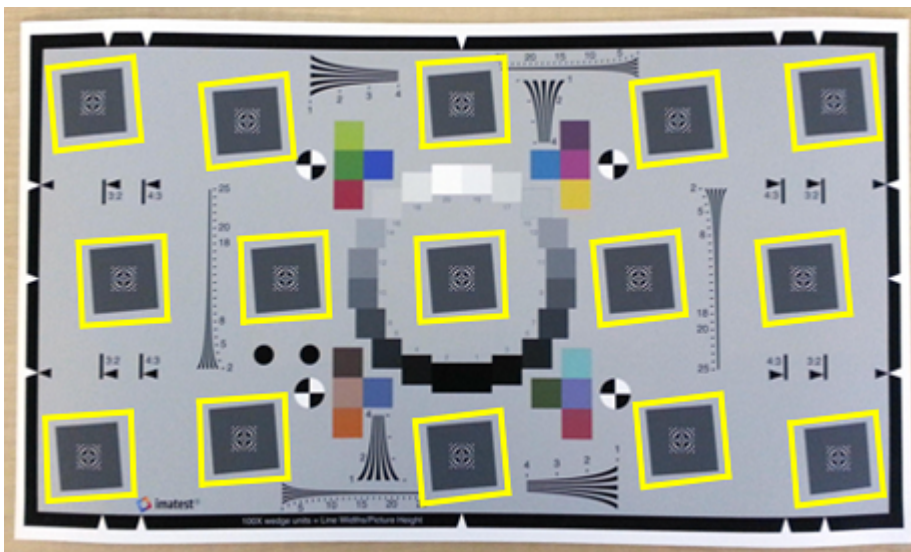
The Extended eSFR test chart has 15 gray boxes tilted 5° away from vertical. The left, top, right, and bottom edges of each box are used to measure:

- 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
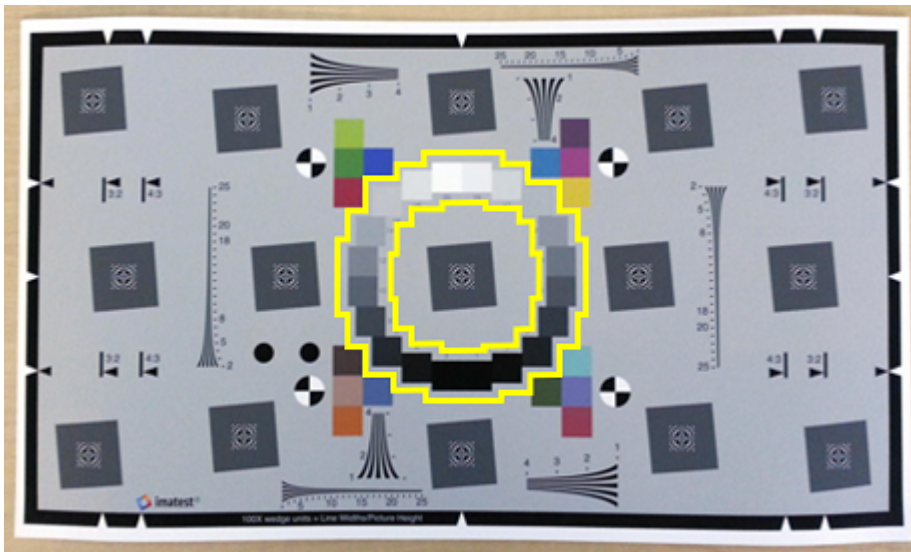
The Extended eSFR test chart has 20 gray patches of increasing brightness, arranged in a ring around the center of the image. The gray patches are used to measure:

- 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
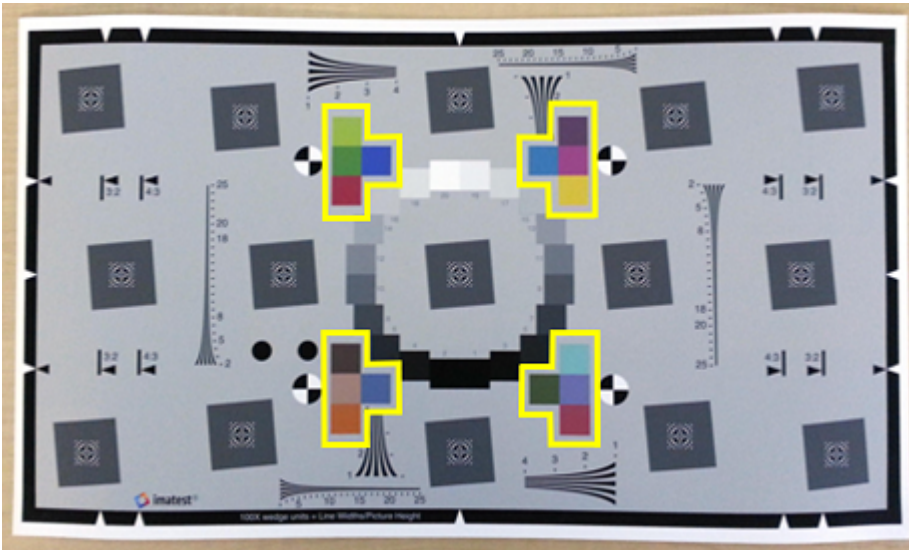
The Extended eSFR test chart has 16 color patches arranged in four groups. The color patches are used to measure:

- 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
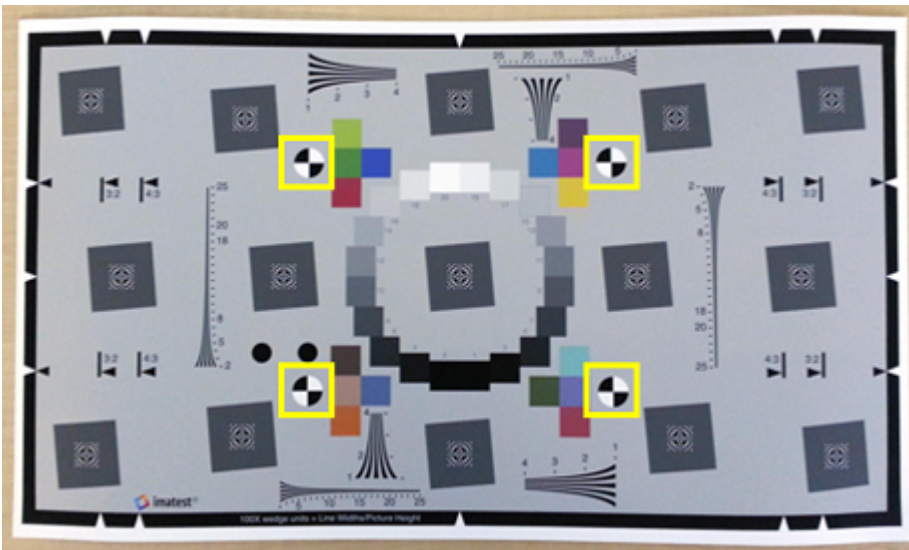
The eSFR test charts have registration markers used to orient the image properly. When you import a chart, `esfrChart` detects four black-and-white checkered circles and uses their position to define regions of interest automatically. You can optionally specify the [x, y] coordinates of the circle centers manually.

References

[1] Imatest. "Esfr". http://www.imatest.com/mathworks/esfr/.

[2] 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 11-154

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');
```


Captured Image of eSFR Chart
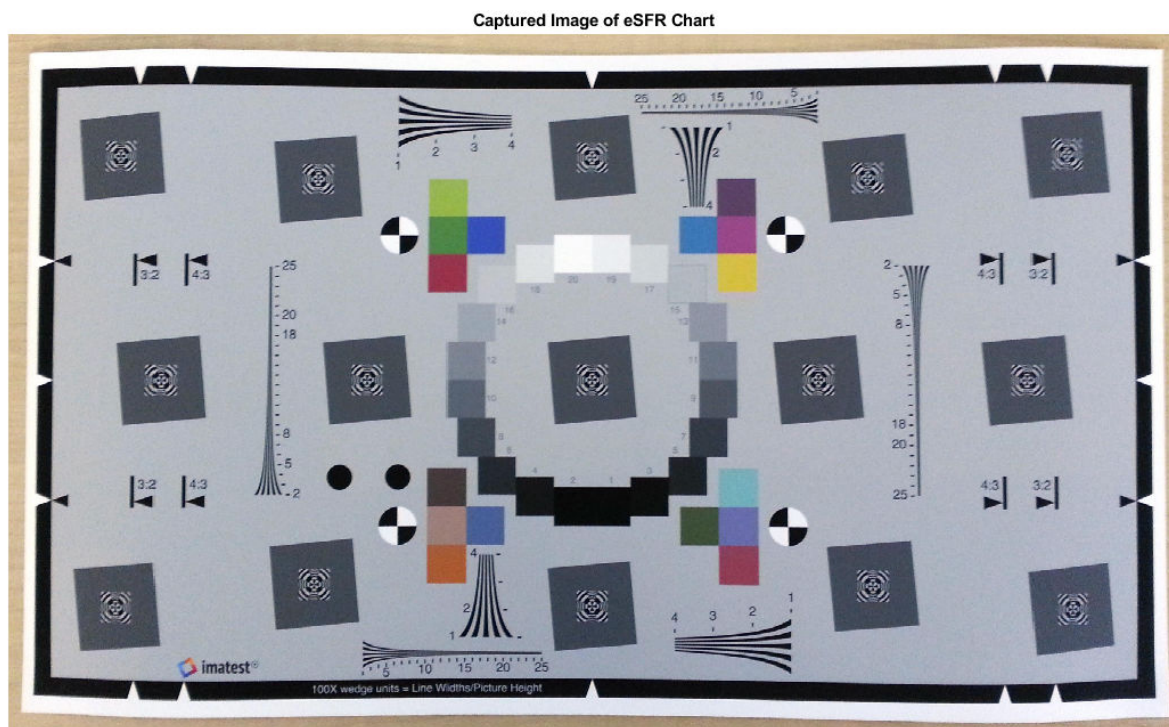
Chart courtesy of Imatest®

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)
```
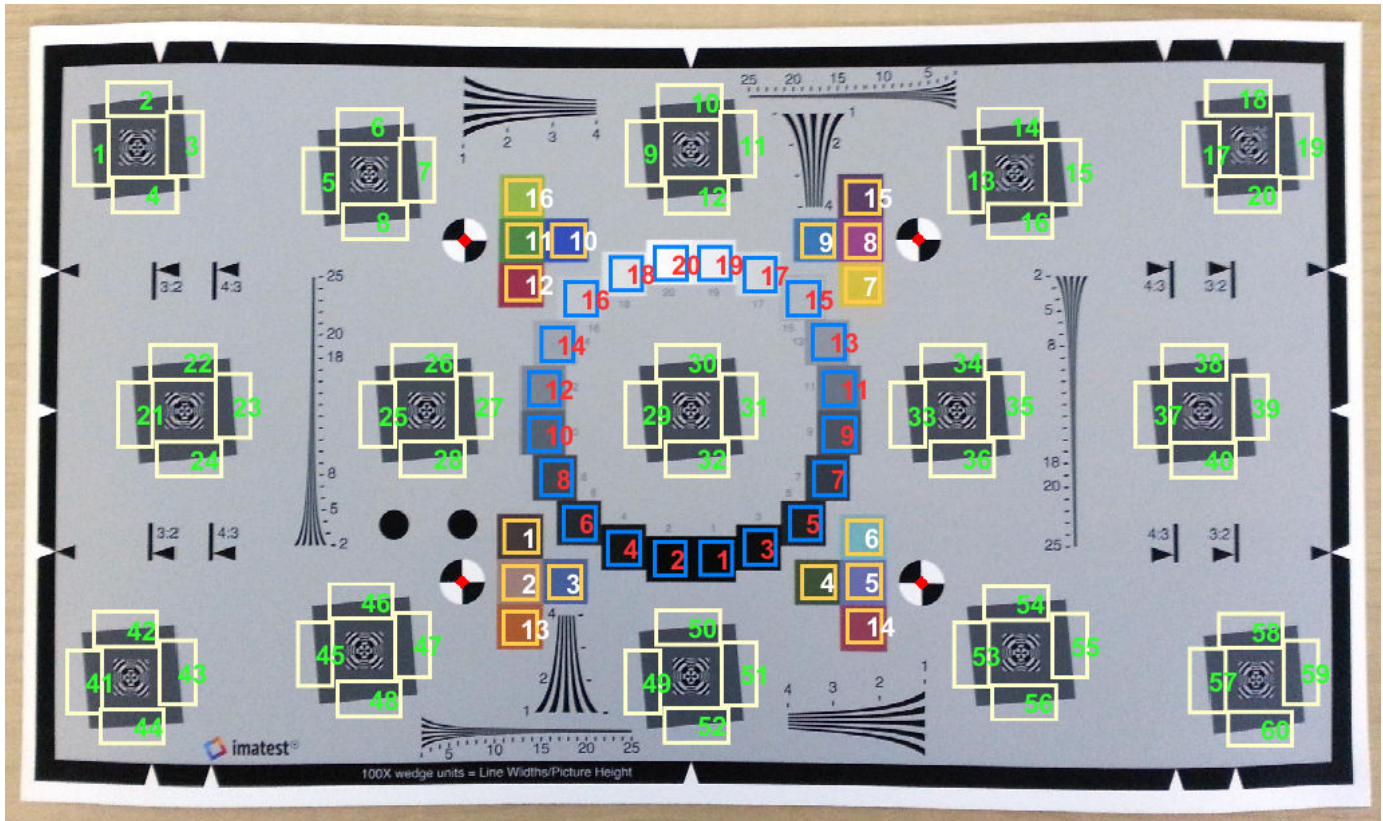
All 60 slanted edge ROIs (labeled in green) are visible and centered on appropriate edges. In addition, 20 gray patch ROIs (labeled in red) and 16 color patch ROIs (labeled in white) are visible and are contained within the boundary of each patch. The chart is correctly imported.

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 1
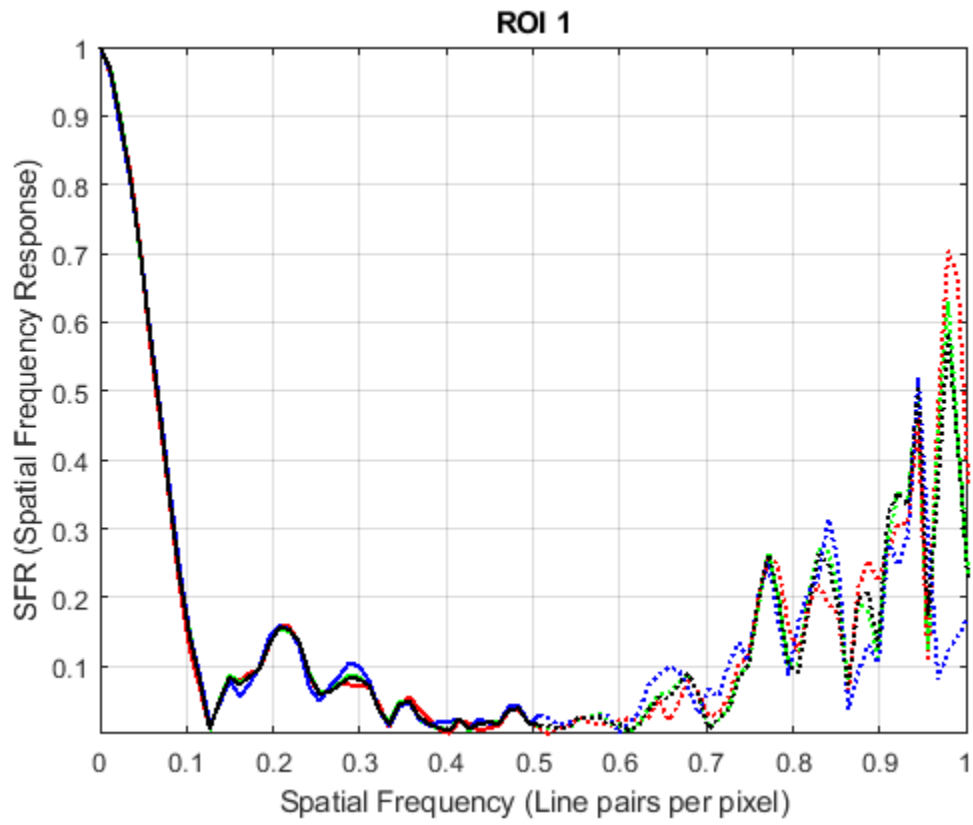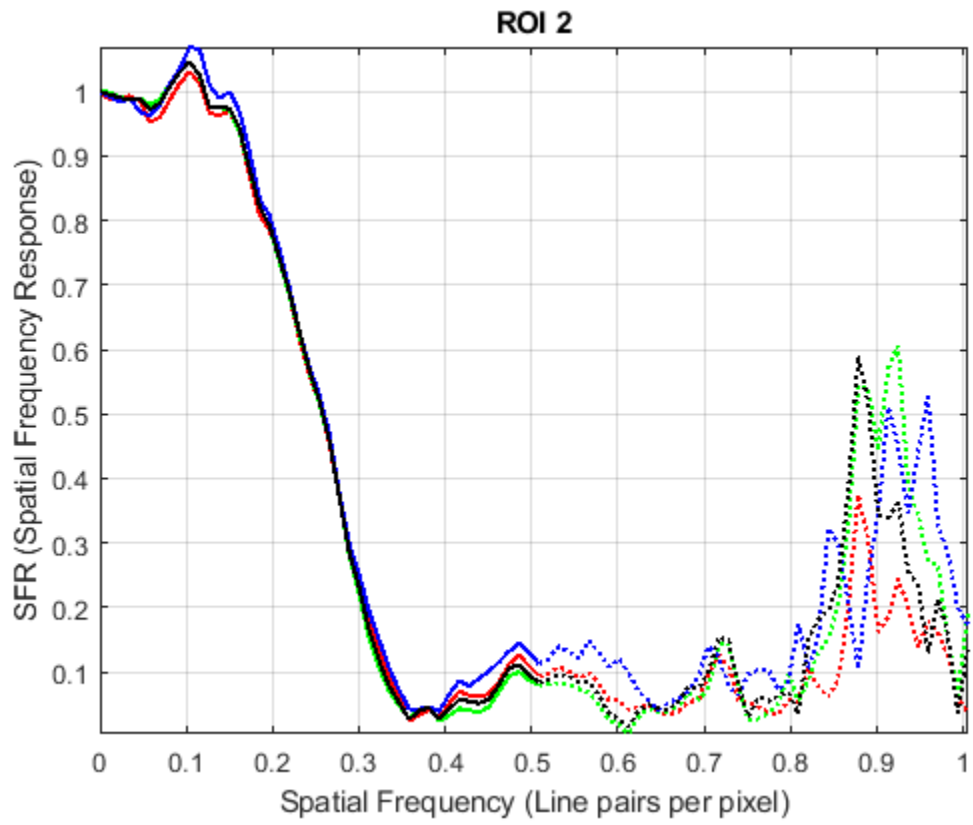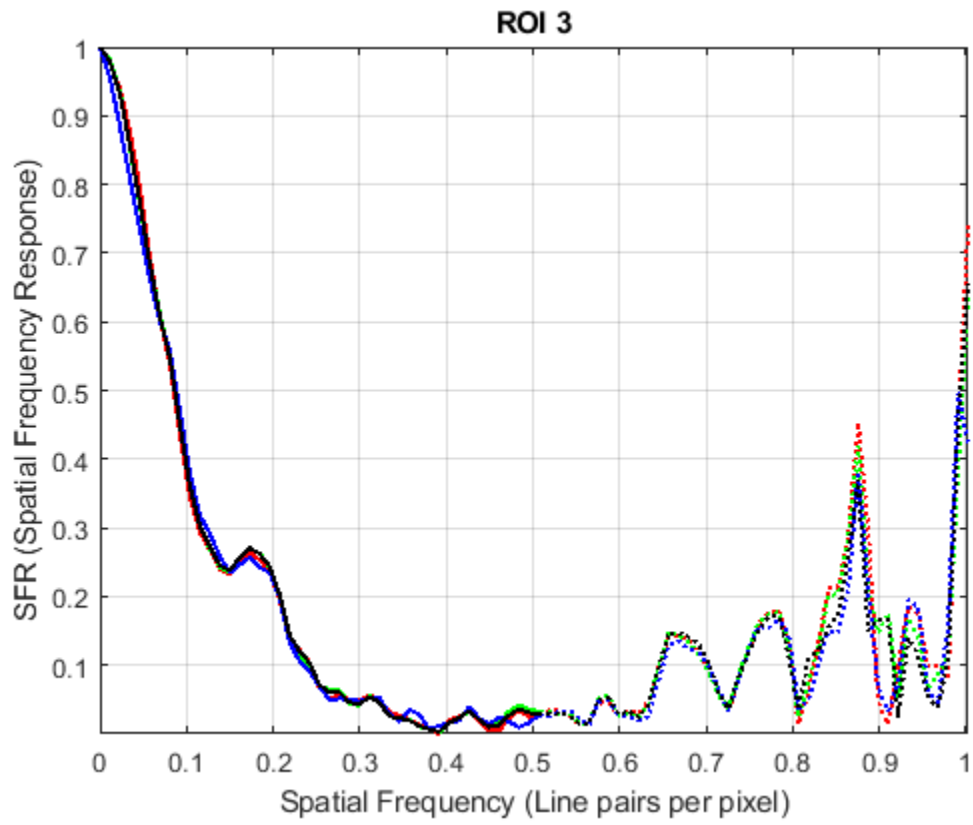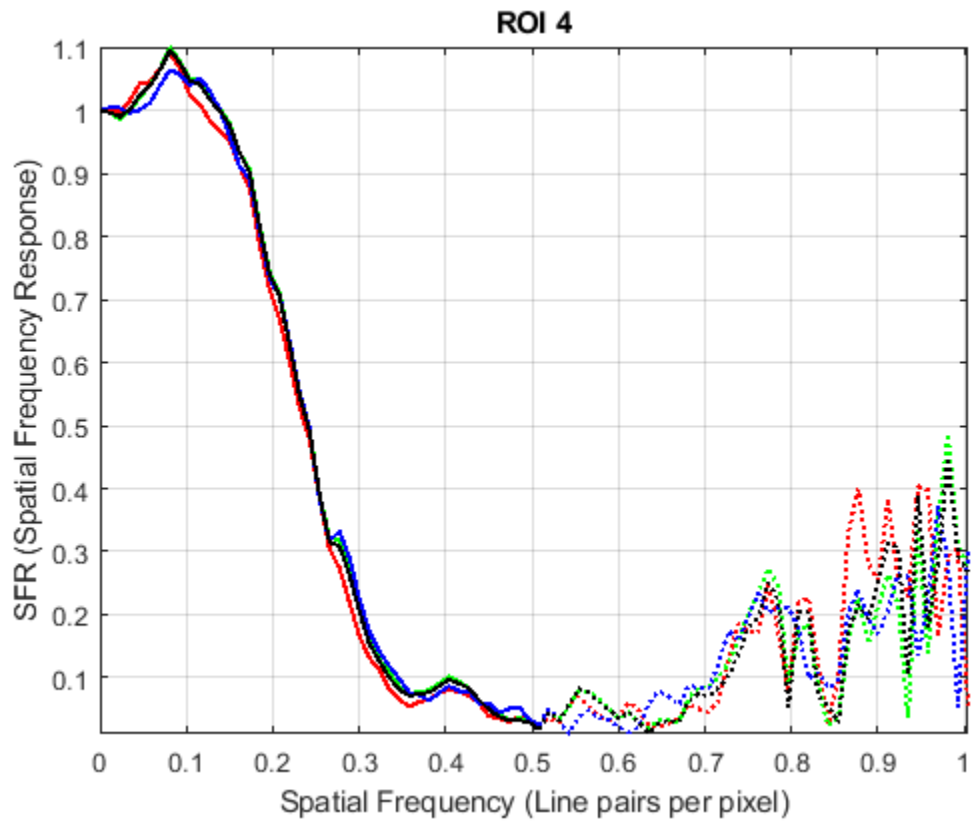
Display the average SFR of the averaged vertical and horizontal edges. The average vertical SFR drops off more rapidly than the average horizontal SFR. Therefore, the average vertical edge is less sharp than the average horizontal edge.

```
plotSFR(aggregateSharpnessTable)
```
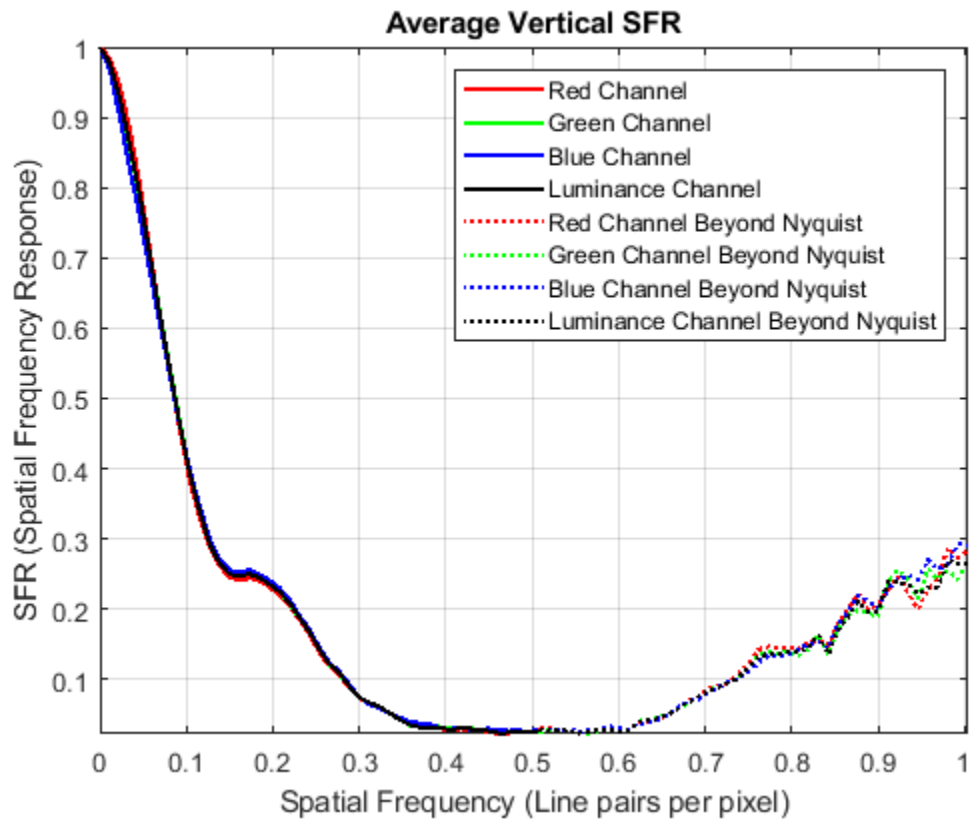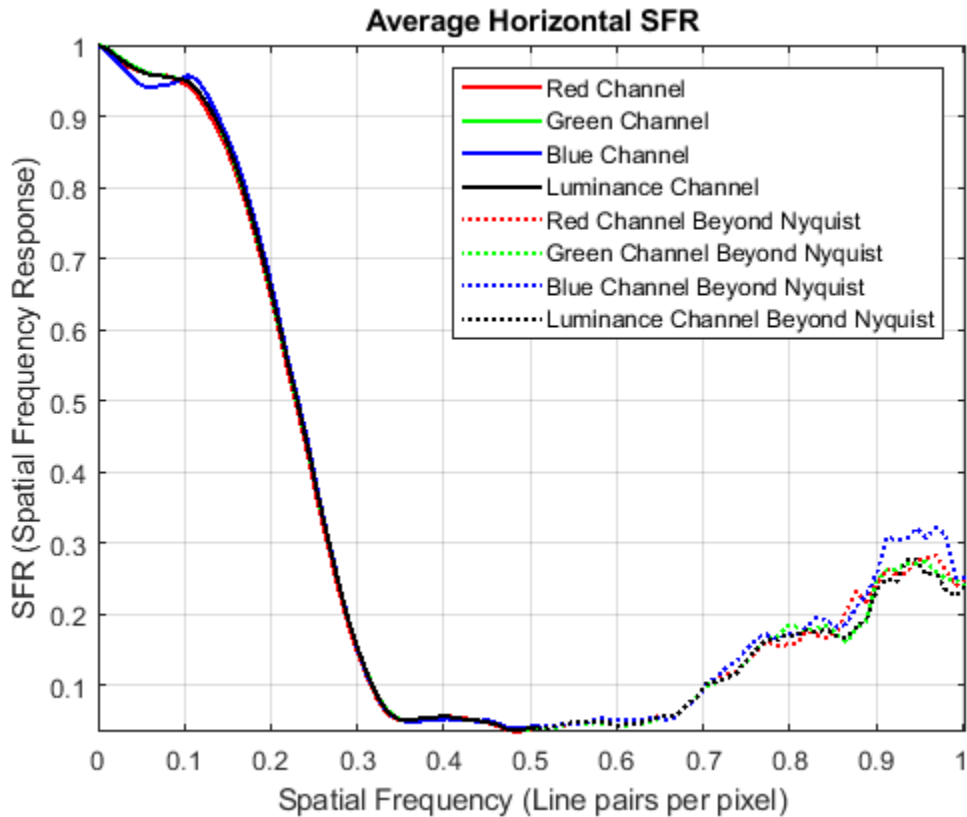
Measure Chromatic Aberration

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))])
```
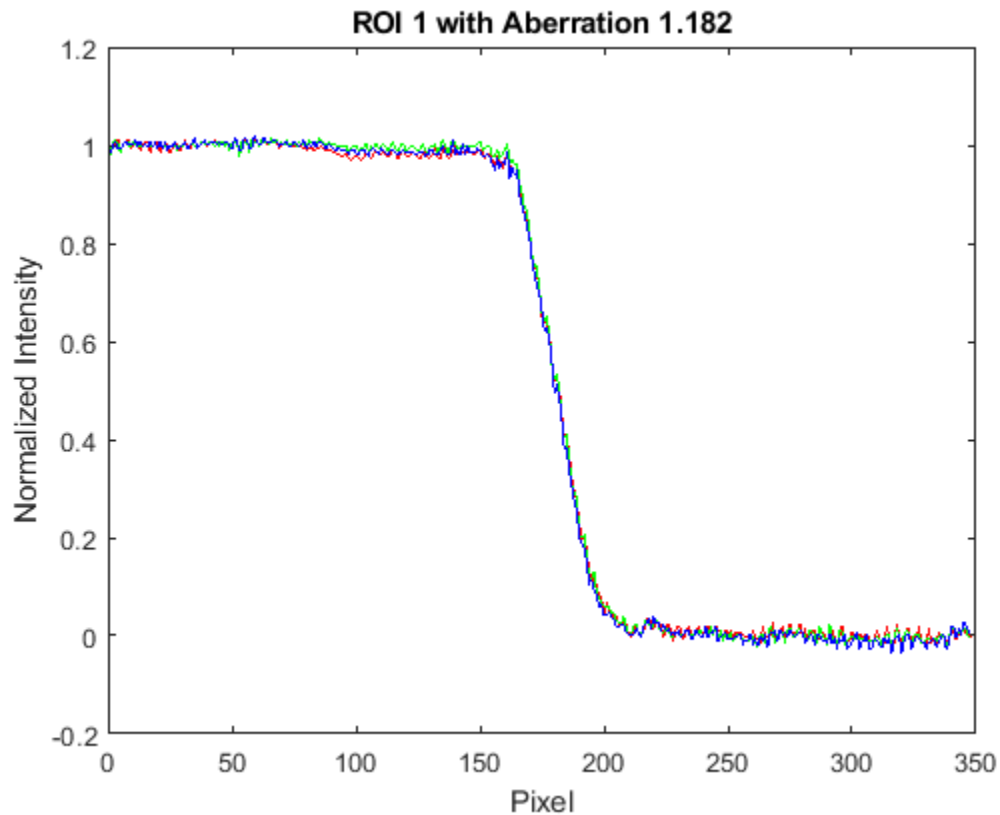
The color channels have similar normalized intensity profiles, and not much color fringing is visible along the edge.

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
```
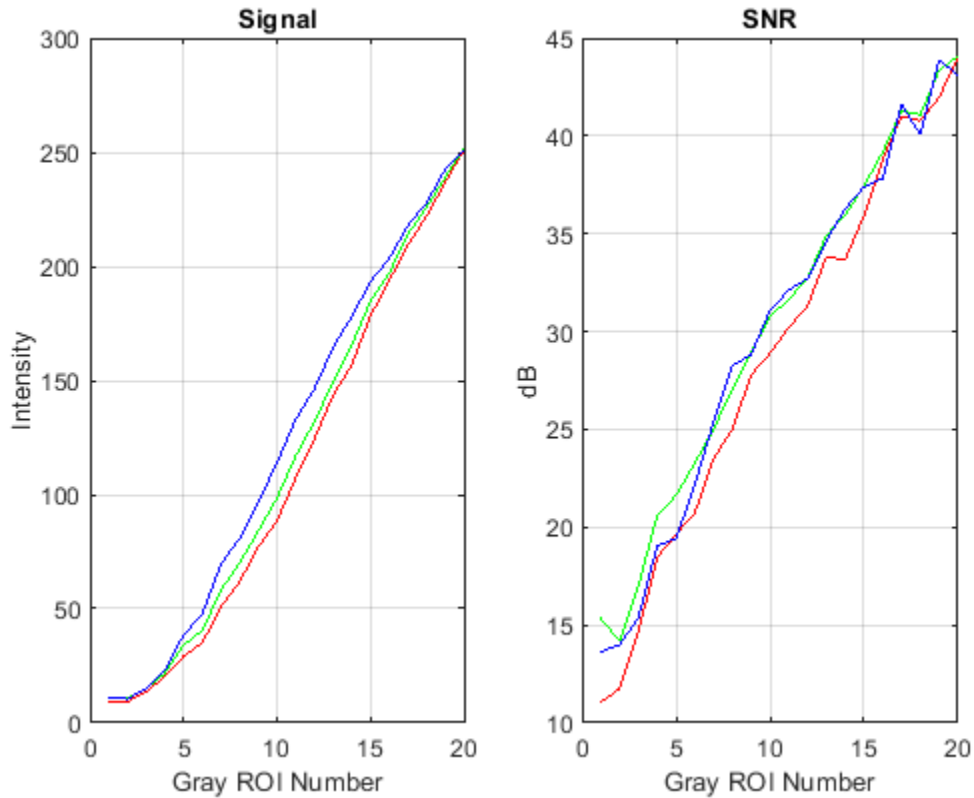

Estimate Illuminant

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)
```

Patch 1	Patch 2	Patch 3	Patch 4
$\Delta E = 15.1$	$\Delta E = 9.9$	$\Delta E = 18.5$	$\Delta E = 14.8$
Patch 5	Patch 6	Patch 7	Patch 8
$\Delta E = 13.0$	$\Delta E = 13.1$	$\Delta E = 9.2$	$\Delta E = 5.1$
Patch 9	Patch 10	Patch 11	Patch 12
$\Delta E = 22.8$	$\Delta E = 22.8$	$\Delta E = 9.5$	$\Delta E = 5.4$
Patch 13	Patch 14	Patch 15	Patch 16
$\Delta E = 16.0$	$\Delta E = 11.7$	$\Delta E = 10.9$	$\Delta E = 10.3$
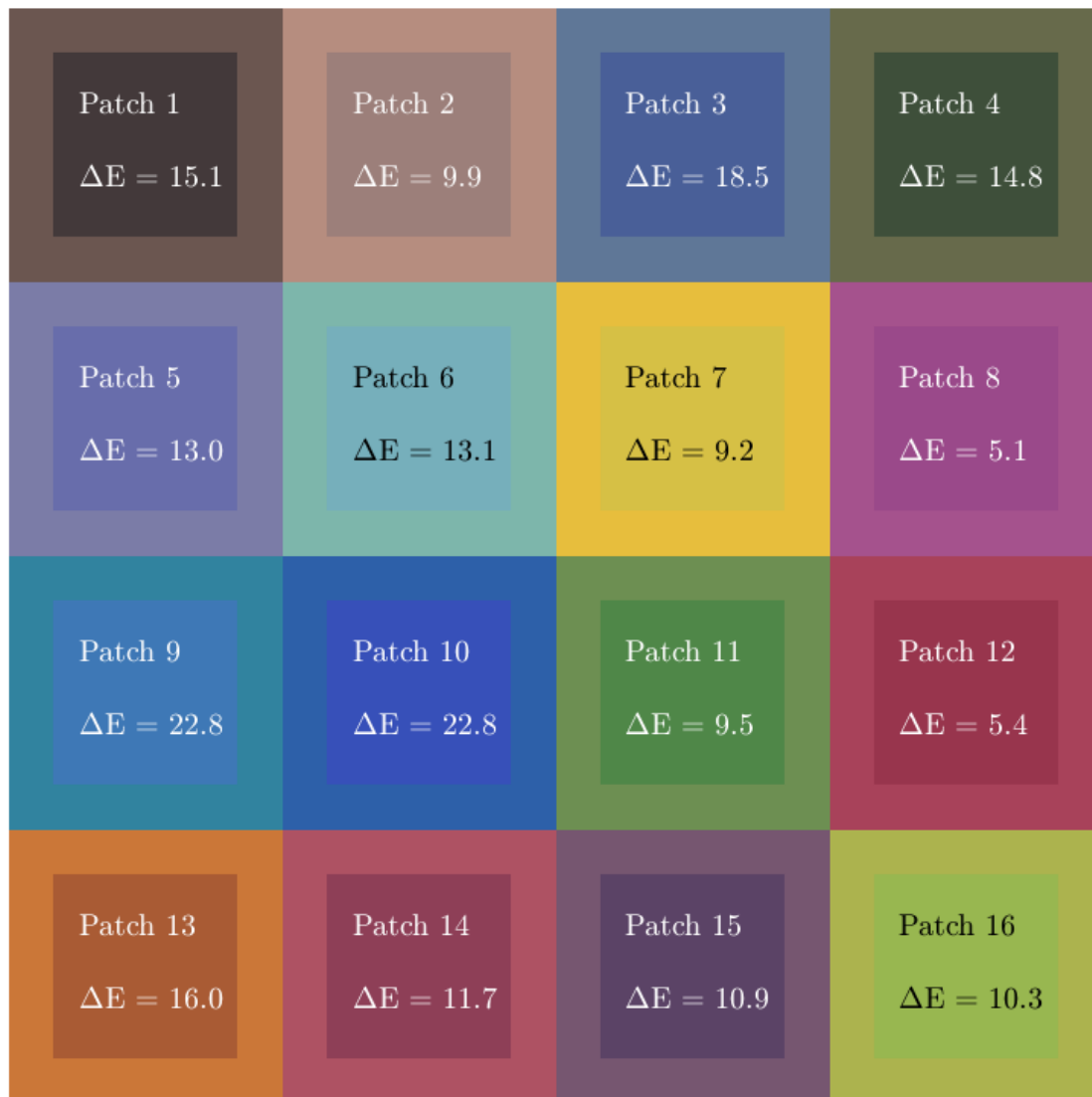
Plot the measured and reference colors in the CIE 1976 L*a*b* color space on a chromaticity diagram. Red circles indicate the reference color. Green circles indicate the measured color of each color patch.

```
figure
plotChromaticity(colorTable)
```
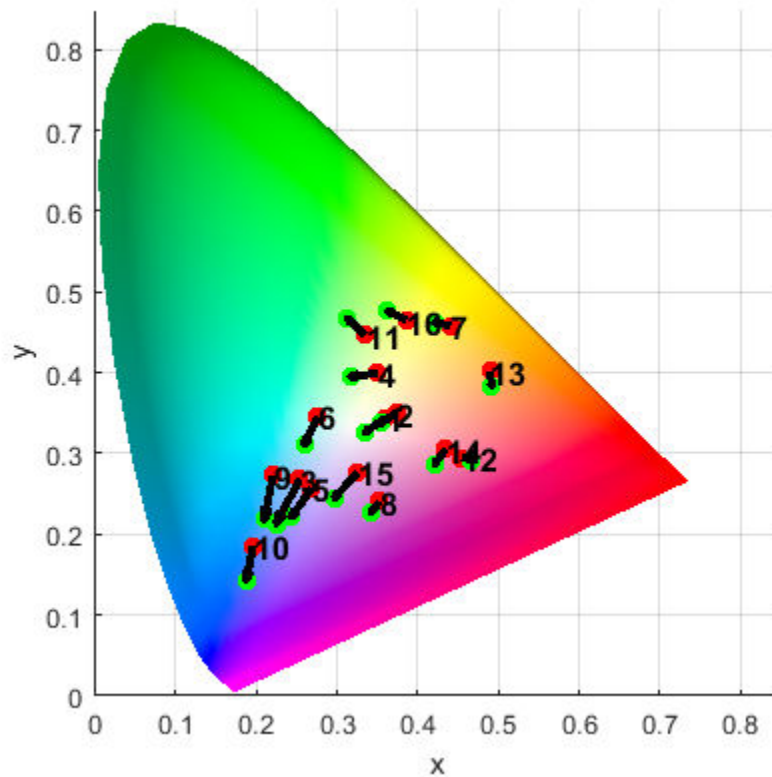
You can use the color correction matrix, `ccm`, to color-correct the test chart images. For an example, see "Correct Colors Using Color Correction Matrix" on page 11-166.

References

[1] Imatest. "Esfr". http://www.imatest.com/mathworks/esfr/.

See Also

displayChart | esfrChart | measureChromaticAberration | measureColor |
measureIlluminant | measureNoise | measureSharpness

More About

- "Anatomy of Imatest Extended eSFR Chart" on page 11-150
- "Correct Colors Using Color Correction Matrix" on page 11-166

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)
```
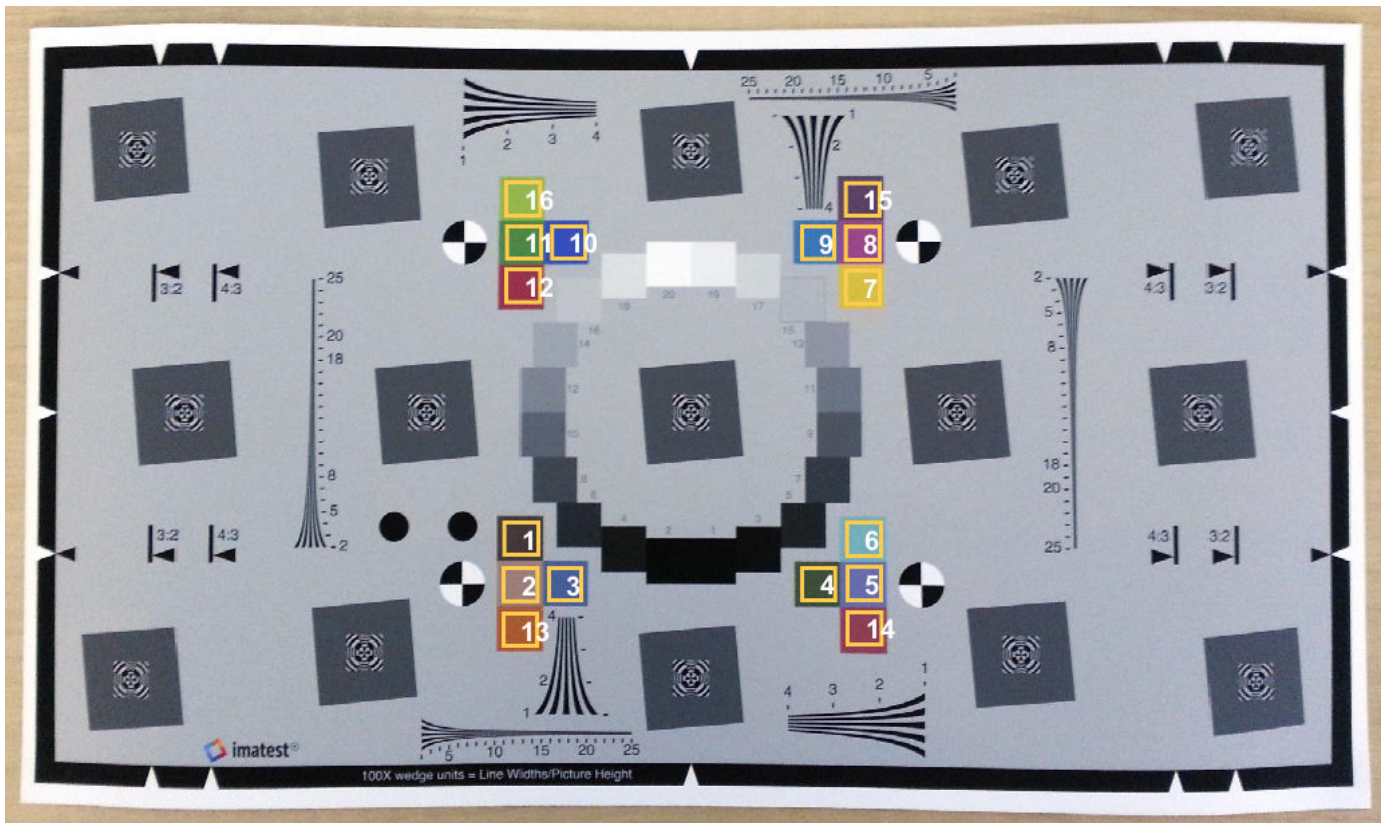

Measure the color accuracy of the 16 color patches by using the `measureColor` function. The function also returns the color correction matrix that is used to perform the color correction.

```
[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)
```
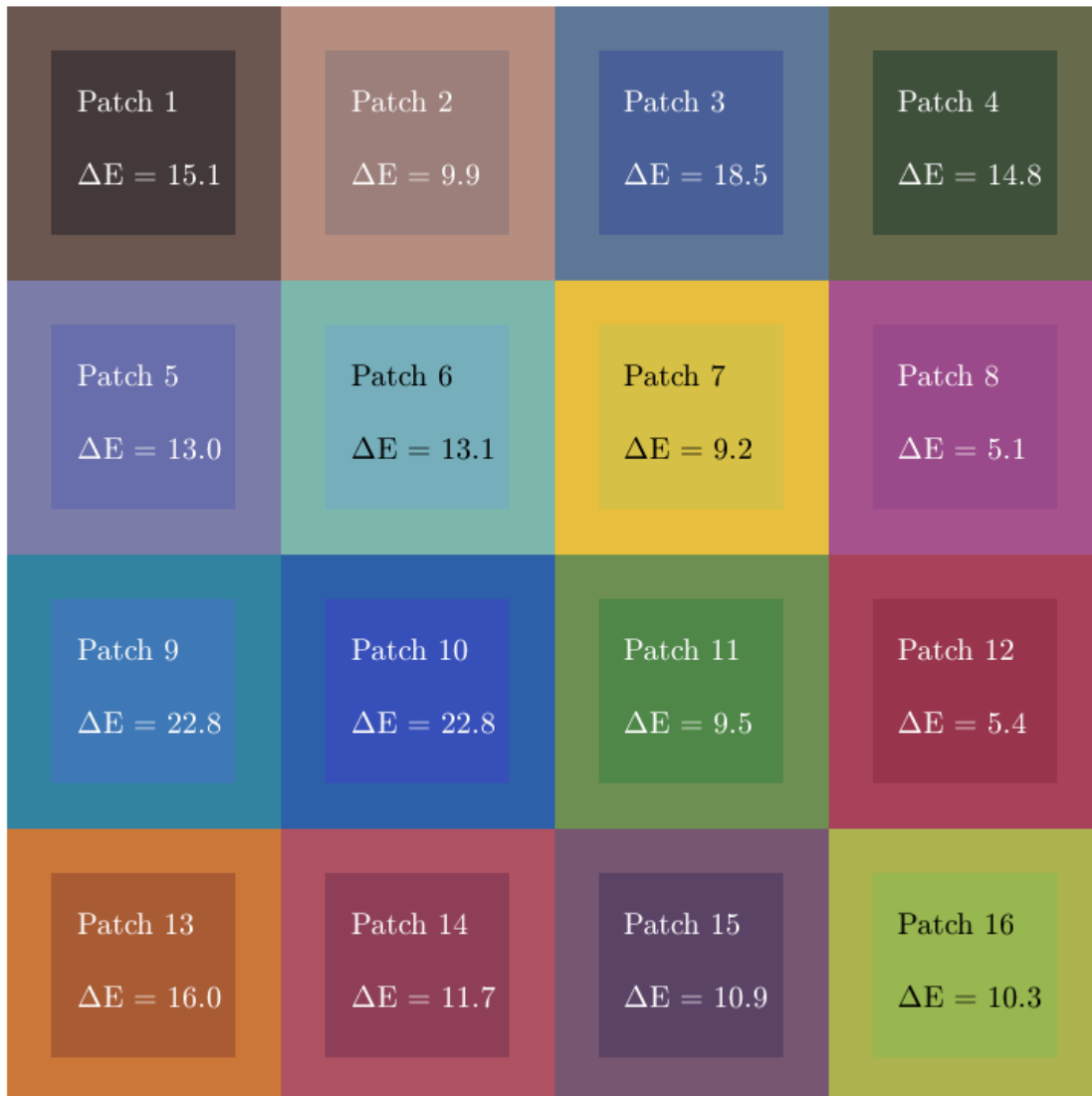
Color-Correct the Test Chart Image

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')
```

Color-Corrected Image Using Color Patches
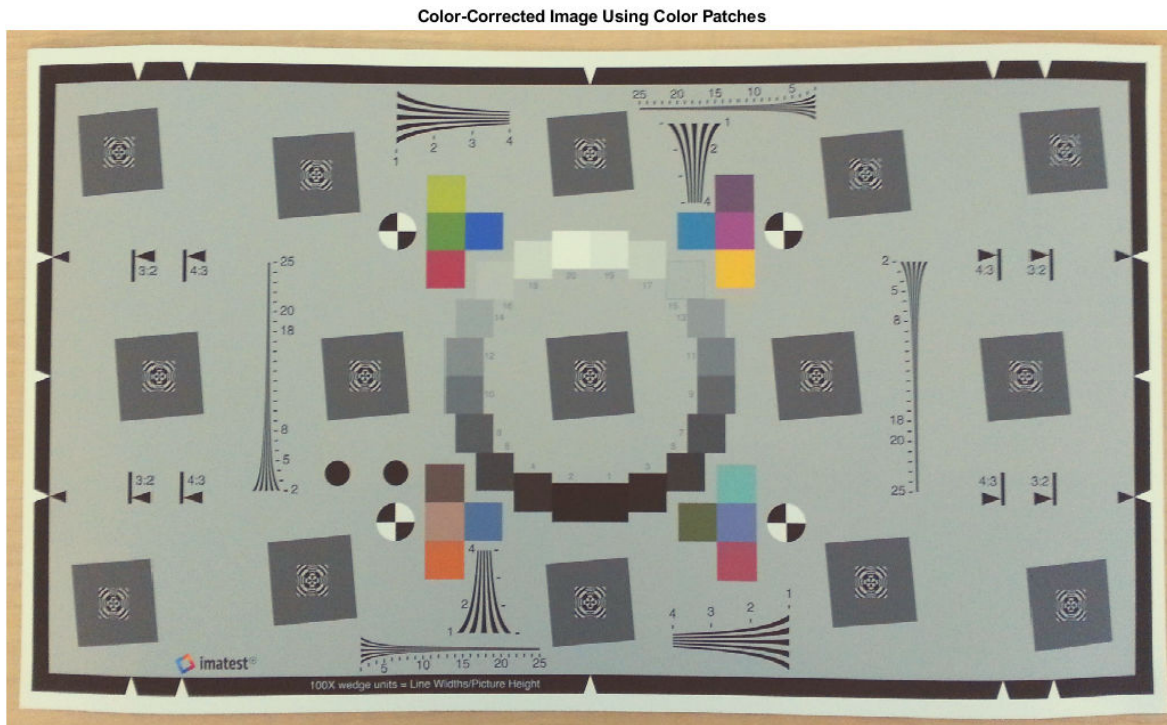
Create an `esfrChart` object that stores information about the color-corrected test chart. Measure the color accuracy of the 16 color-corrected color patches.

```
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 $\Delta E = 1.4$	Patch 2 $\Delta E = 3.8$	Patch 3 $\Delta E = 4.0$	Patch 4 $\Delta E = 2.0$
Patch 5 $\Delta E = 4.8$	Patch 6 $\Delta E = 2.2$	Patch 7 $\Delta E = 3.8$	Patch 8 $\Delta E = 5.7$
Patch 9 $\Delta E = 6.6$	Patch 10 $\Delta E = 4.1$	Patch 11 $\Delta E = 6.2$	Patch 12 $\Delta E = 6.2$
Patch 13 $\Delta E = 11.1$	Patch 14 $\Delta E = 2.7$	Patch 15 $\Delta E = 3.6$	Patch 16 $\Delta E = 4.0$

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)
```
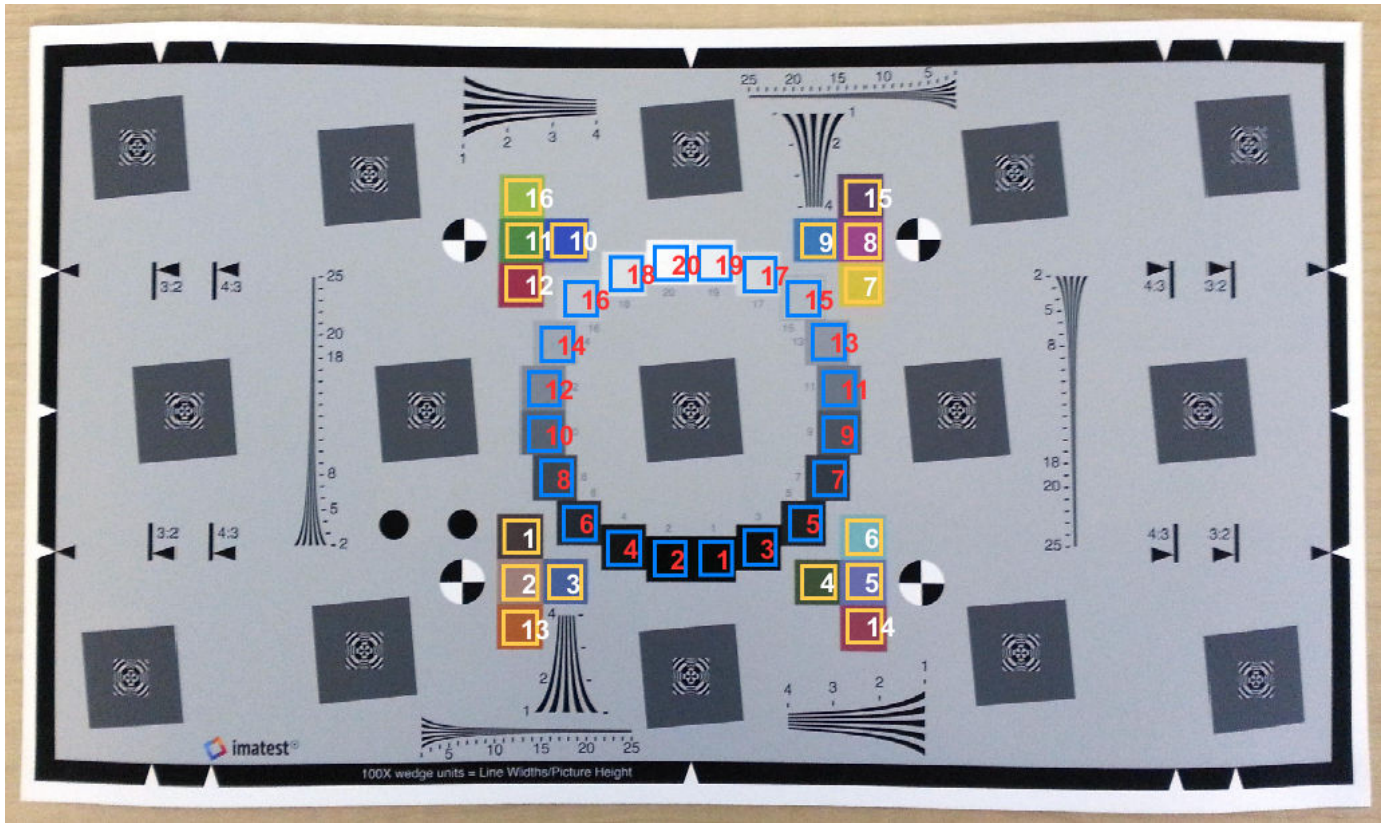
Get the reference L*a*b* values of the color and grayscale patches, which are stored in the `ReferenceColorLab` and `ReferenceGrayLab` properties of the eSFR chart object. Convert these values to the RGB color space.

```
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')
```

Color-Corrected Image Using Gray and Color Patches
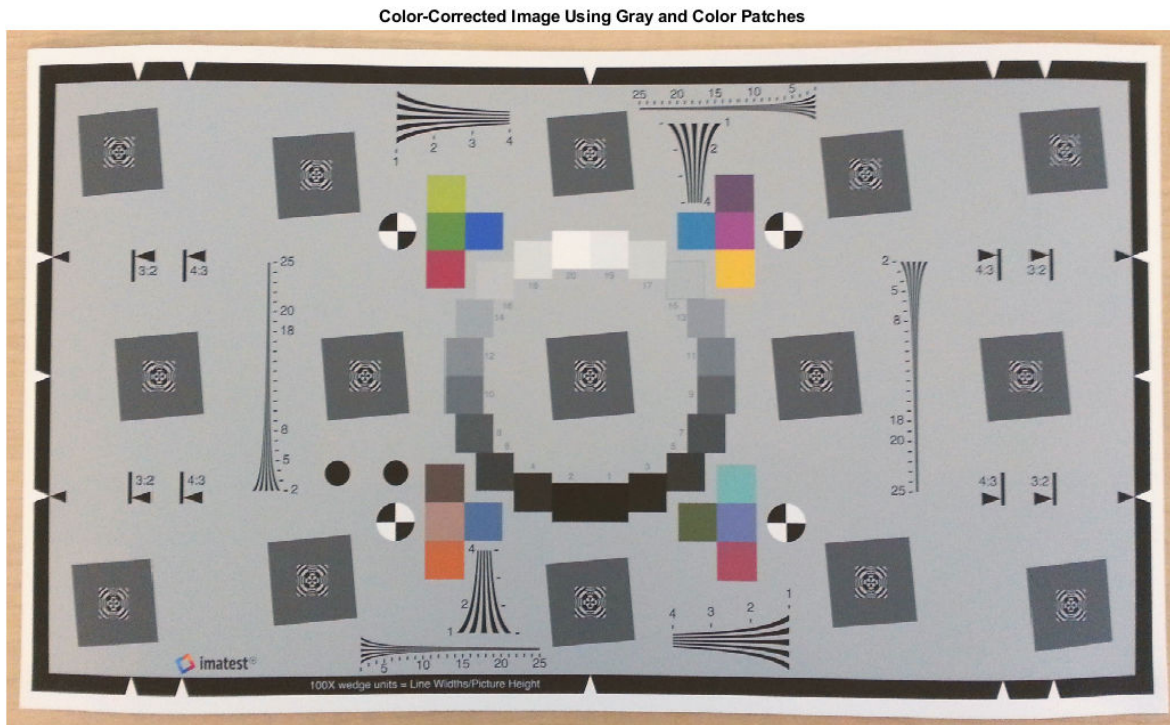
Compare the corrected and reference colors on a color patch diagram. Some of the measured color errors have decreased, while others have increased.

```
chart_cc2 = esfrChart(I_cc2);
colorTable_cc2 = measureColor(chart_cc2);
displayColorPatch(colorTable_cc2)
```

Patch 1	Patch 2	Patch 3	Patch 4
$\Delta E = 2.8$	$\Delta E = 3.1$	$\Delta E = 5.7$	$\Delta E = 3.3$
Patch 5	Patch 6	Patch 7	Patch 8
$\Delta E = 3.2$	$\Delta E = 5.2$	$\Delta E = 7.0$	$\Delta E = 3.5$
Patch 9	Patch 10	Patch 11	Patch 12
$\Delta E = 10.3$	$\Delta E = 6.4$	$\Delta E = 4.9$	$\Delta E = 4.8$
Patch 13	Patch 14	Patch 15	Patch 16
$\Delta E = 12.3$	$\Delta E = 3.2$	$\Delta E = 2.7$	$\Delta E = 2.6$

References

[1] Imatest. "Esfr". http://www.imatest.com/mathworks/esfr/.
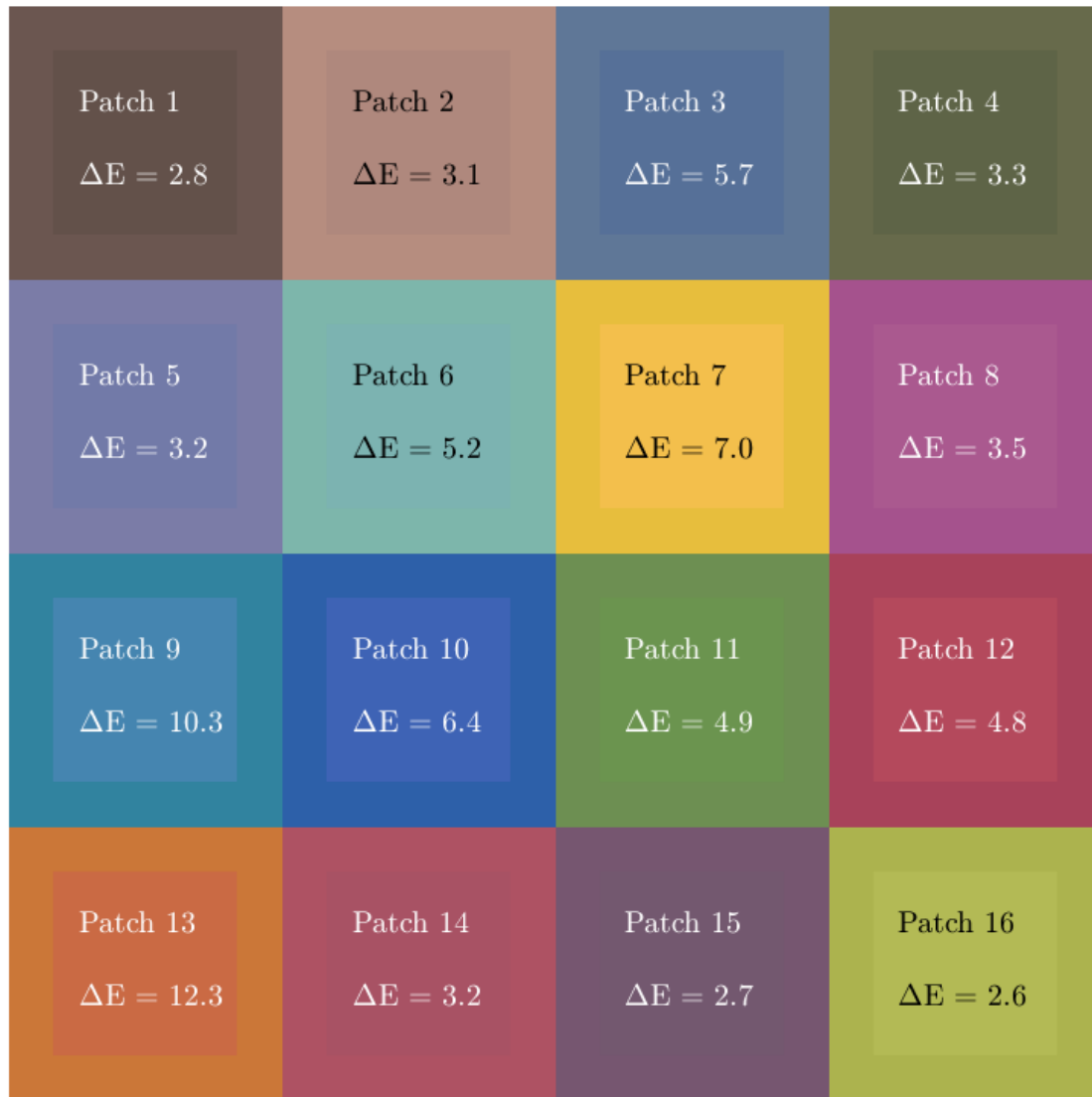
See Also

displayColorPatch | esfrChart | measureColor | plotChromaticity

More About

- "Evaluate Quality Metrics on eSFR Test Chart" on page 11-154

Install Sample Data Using Add-On Explorer

Image Processing Toolbox makes a sample 3-D MRI chest scan data set available as an optional feature. This volumetric data set is used by the example "Segment Lungs from 3-D Chest Scan" on page 13-126.
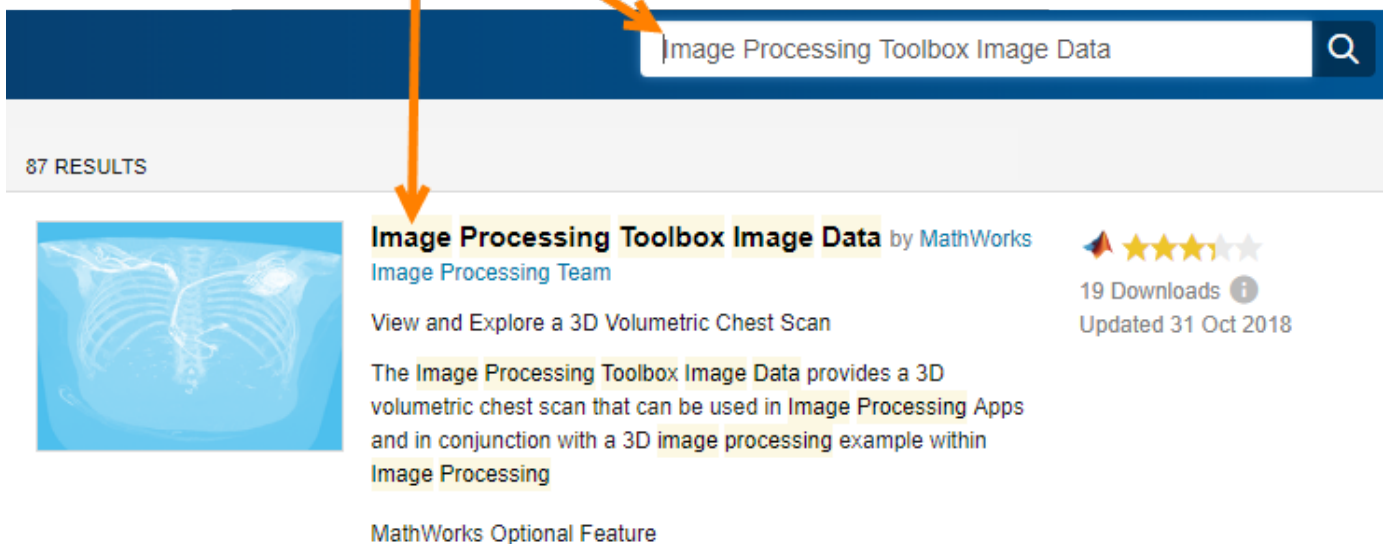
To obtain this data set, download it using the Add-On Explorer.

1   Select **Get Add-ons** from the **Add-ons** drop-down menu on the MATLAB desktop. The **Add-on Explorer** opens.
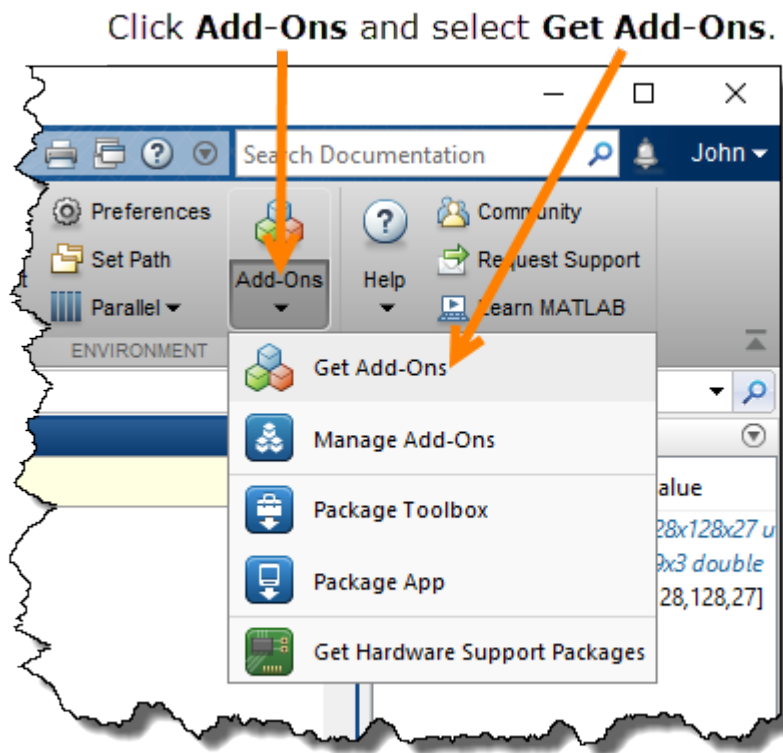
2   In the Add-On Explorer, search for the data package **Image Processing Toolbox Image Data**. The data package is a **MathWorks Optional Feature**.

Search for **Image Processing Toolbox Image Data**.

3 Click the data package in the search results. On the data package page, click **Install**. Follow the instructions presented by the installer.

ROI-Based Processing

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 Function

This example shows to create a binary mask using one of the ROI creation functions, such as drawcircle, with the mask creation function createMask.

Read an image into the workspace and display it.

```
img = imread('pout.tif');
h_im = imshow(img);
```

Create an ROI on the image using one of the ROI creation functions.

```
circ = drawcircle('Center',[113,66],'Radius',60);
```


Create a binary mask from the ROI using `createMask`. The `createMask` function returns a binary image the same size as the input image. The pixels inside the ROI are set to 1 and the pixel values everywhere else are set to 0.

```
BW = createMask(circ);
imshow(BW)
```
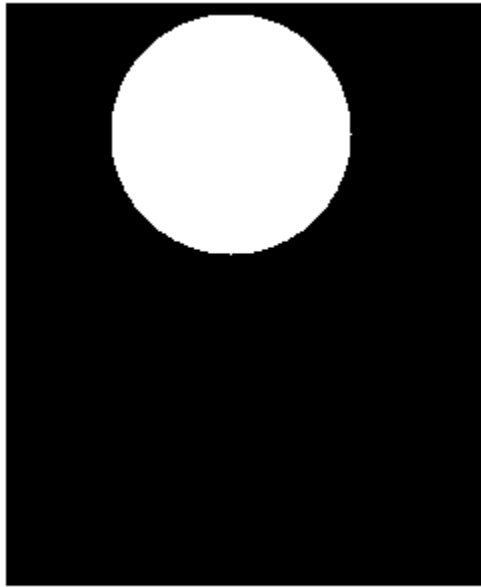
Create Binary Mask Based on Color Values

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

You can create ROI objects that represent ROIs of various shapes, including circles, ellipses, polygons, lines, polylines, rectangles, and hand-drawn shapes. You can also draw freehand shapes that get "assistance" from the underlying image, automatically following the contour of edges. In this figure, you see a polygonal ROI drawn over an image.

You can control aspects of the ROI position and appearance. You can create masks from ROIs and perform other operations. You can also specify how the ROI responds to events that occur within the ROI, such as mouse clicks and movement.
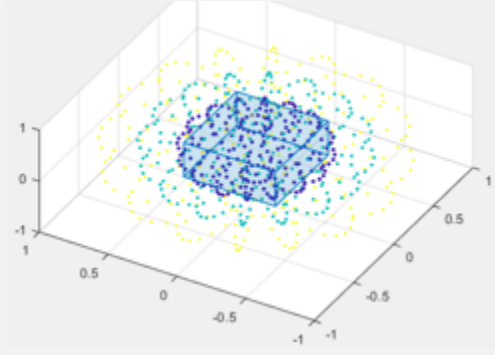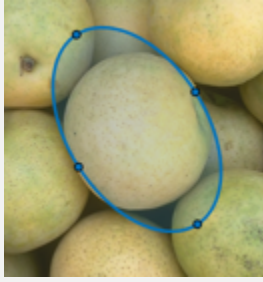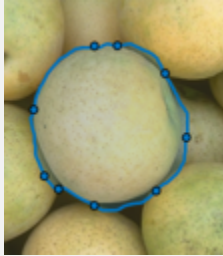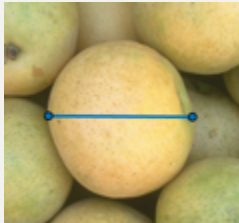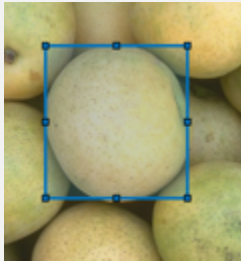
There are three ways to create ROI objects.

- Create an ROI interactively by using a creation convenience function. The creation functions enable you to draw the ROI on an image. Use this approach if you do not have prior knowledge of the size and position of the ROI and want to use the image content to assist in the placement of the ROI. For more information, see "Create ROI Using Creation Convenience Functions" on page 12-8.

- Create an ROI programmatically by specifying information about the size and shape of the ROI. Use this approach if you already know details about the size and shape of the ROI, such as the coordinates of polygon vertices or the center coordinates and radius of a circle.

- Create an ROI programmatically, then use the `draw` function to interactively draw the ROI on an image. Use this approach if you want to set the display properties and behavior of the ROI before you specify the size and position of the ROI. The `draw` function also enables you to redraw an existing ROI, preserving the appearance of the ROI. For more information, see "Create ROI Using draw Function" on page 12-10.

The table shows the supported ROIs and their respective creation convenience functions.

ROI Object	ROI Creation Convenience Function	Description
AssistedFreehand	drawassisted	Freehand ROI that snaps to edges of existing objects in the image
Circle	drawcircle	Circular ROI
Crosshair	drawcrosshair	Linear ROI that consists of two perpendicular lines

ROI Object	ROI Creation Convenience Function	Description
Cuboid	drawcuboid	3-D cuboidal ROI
Ellipse	drawellipse	Ellipsoid ROI
Freehand	drawfreehand	Freehand ROI that follows the path of the mouse
Line	drawline	Linear ROI that consists of a single line segment

ROI Object	ROI Creation Convenience Function	Description
Point	drawpoint	Point ROI
Polygon	drawpolygon	Polygonal ROI that consists of a closed set of line segments
Polyline	drawpolyline	Polyline ROI that consists of an open set of line segments
Rectangle	drawrectangle	Rectangular ROI

Create ROI Using Creation Convenience Functions

This example shows how to create an ROI object by using ROI creation convenience functions. These functions enable you to create the ROI objects interactively by drawing the ROI over an image.

Read and display an image.

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

This example creates an ellipsoid ROI. You can use a similar process to create any ROI object.

Create an `Ellipse` ROI by using the `drawellipse` function. Customize the look of the ROI by specifying the `StripeColor` name-value pair argument as yellow.

```
roi = drawellipse('StripeColor','y');
```

Inspect properties of the ROI.

```
roi
```
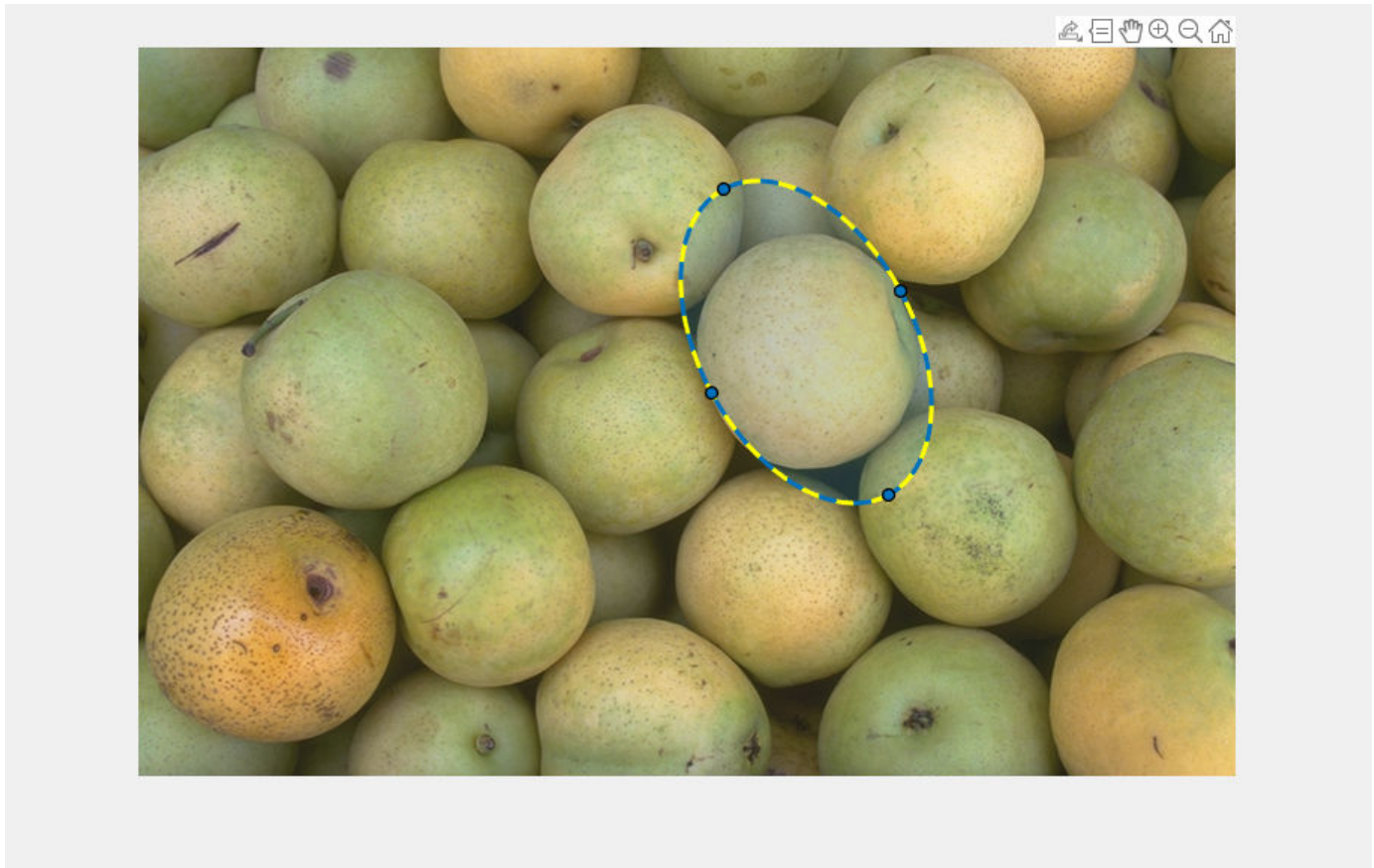
```
roi =
  Ellipse with properties:

            Center: [446.0000 197.0000]
          SemiAxes: [115.8836 71.6200]
     RotationAngle: 298.3342
       AspectRatio: 1.6180
             Label: ''
             Color: [0 0.4470 0.7410]
            Parent: [1×1 Axes]
           Visible: 'on'
          Selected: 0

  Show all properties
```

Create ROI Using draw Function

This example shows how to use the `draw` function to redraw an existing ROI interactively. This approach is useful if you want to set the display properties and behavior of the ROI before you specify

the size and position of the ROI. For example, you may want to create and customize an ROI before you have an axes in which to display the ROI.

This example creates and draws an ellipsoid ROI. You can use a similar process to create any ROI object.

Create an `Ellipse` ROI programmatically by using the `images.roi.Ellipse` function. Specify properties to customize the appearance of the ellipse. Here, the face color is cyan and border of the ROI has a red stripe. Do not specify the position of the ROI.

```
roi = images.roi.Ellipse('Color','c','StripeColor','r');
```

Inspect properties of the ROI.

```
roi

roi =
  Ellipse with properties:

          Center: []
        SemiAxes: []
   RotationAngle: 0
     AspectRatio: 1.6180
           Label: ''

  Show all properties
```

Inspect the parent axes of the ROI. The ROI is not drawn so the parent axes is empty.

```
roi.Parent

ans =
  0×0 empty GraphicsPlaceholder array.
```

Read and display an image.

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

Draw the ROI on the image by using the `draw` function. Click and drag the cursor over the image to create the elliptical shape. The displayed ROI has the face color and stripe color that you specified when you created the ROI.

```
draw(roi)
```
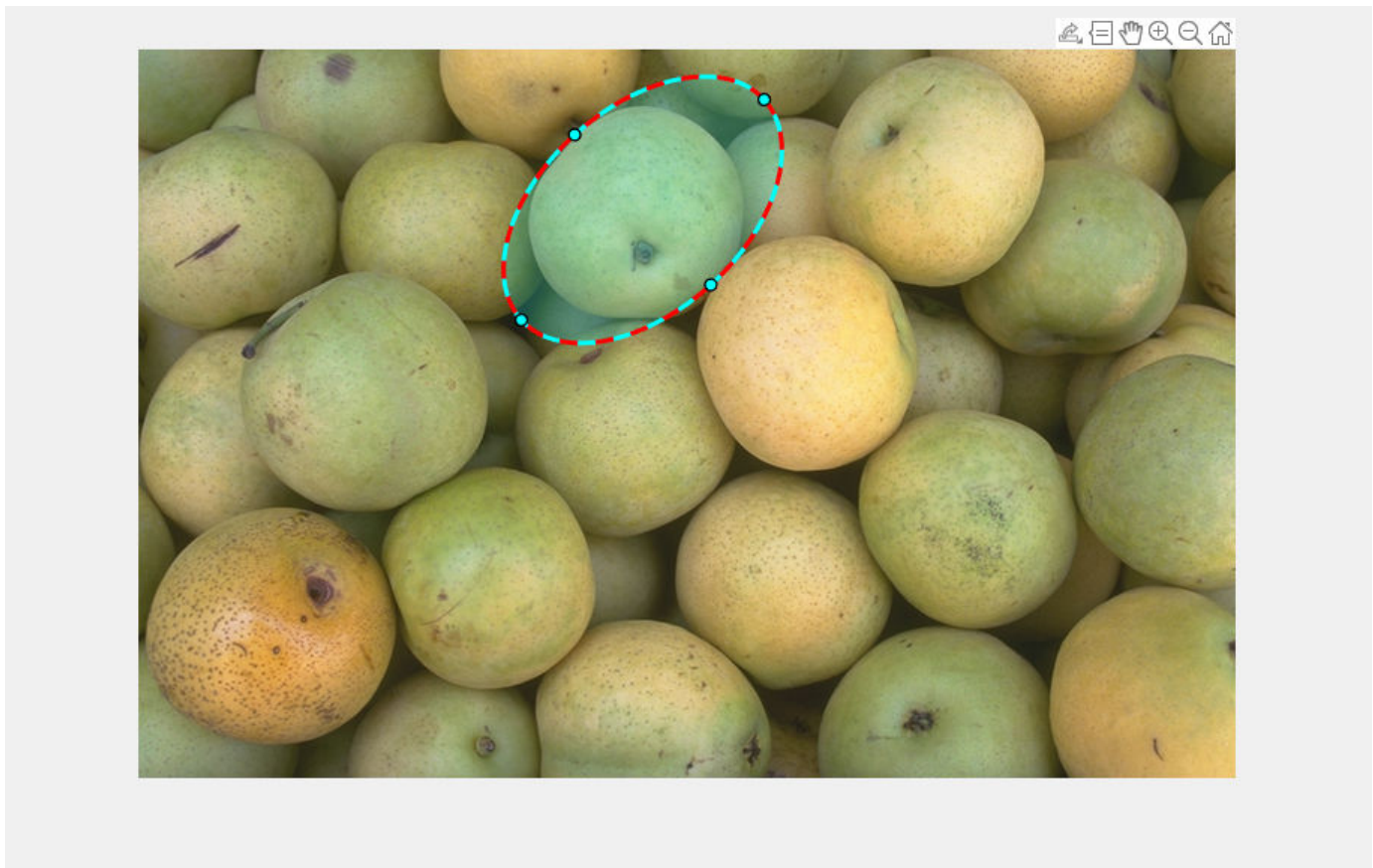
Inspect properties of the ROI. Several properties of the ROI are updated after drawing.

```
roi
```

```
roi =
  Ellipse with properties:

            Center: [337 107.5000]
          SemiAxes: [109.3766 67.5985]
     RotationAngle: 42.2208
       AspectRatio: 1.6180
             Label: ''
             Color: [0 1 1]
            Parent: [1×1 Axes]
           Visible: 'on'
          Selected: 0

  Show all properties
```

The ROI now has a parent axes. Get all graphics objects who share the same parent axes. In this example, the ROI has the same parent as the displayed image.

```
roi.Parent.Children
```

```
ans =
  2×1 graphics array:
```

```
Ellipse
Image
```

Using ROIs in Apps Created with App Designer

You can use ROIs in apps created with App Designer, parenting an ROI in a `UIAxes`. You must explicitly specify the UIAxes when calling the ROI creation function, as an input argument or using the `'Parent'` name/value pair. There are a few limitations when using ROIs in apps in this way:

- The mouse cursor does not update when you hover over the ROI. The cursor is always an arrow.
- The ROI does not change color when you hover over it.
- The ROI right-click menu (`UIContextMenu`) is not supported.

The following code, while not a typical app-creation workflow, shows how to specify an ROI in a `UIAxes` in an app (`UIFigure`).

1   Create a `UIAxes`. When you call the `uiaxes` function, it creates a `UIFigure` automatically.

```
uax = uiaxes;
```
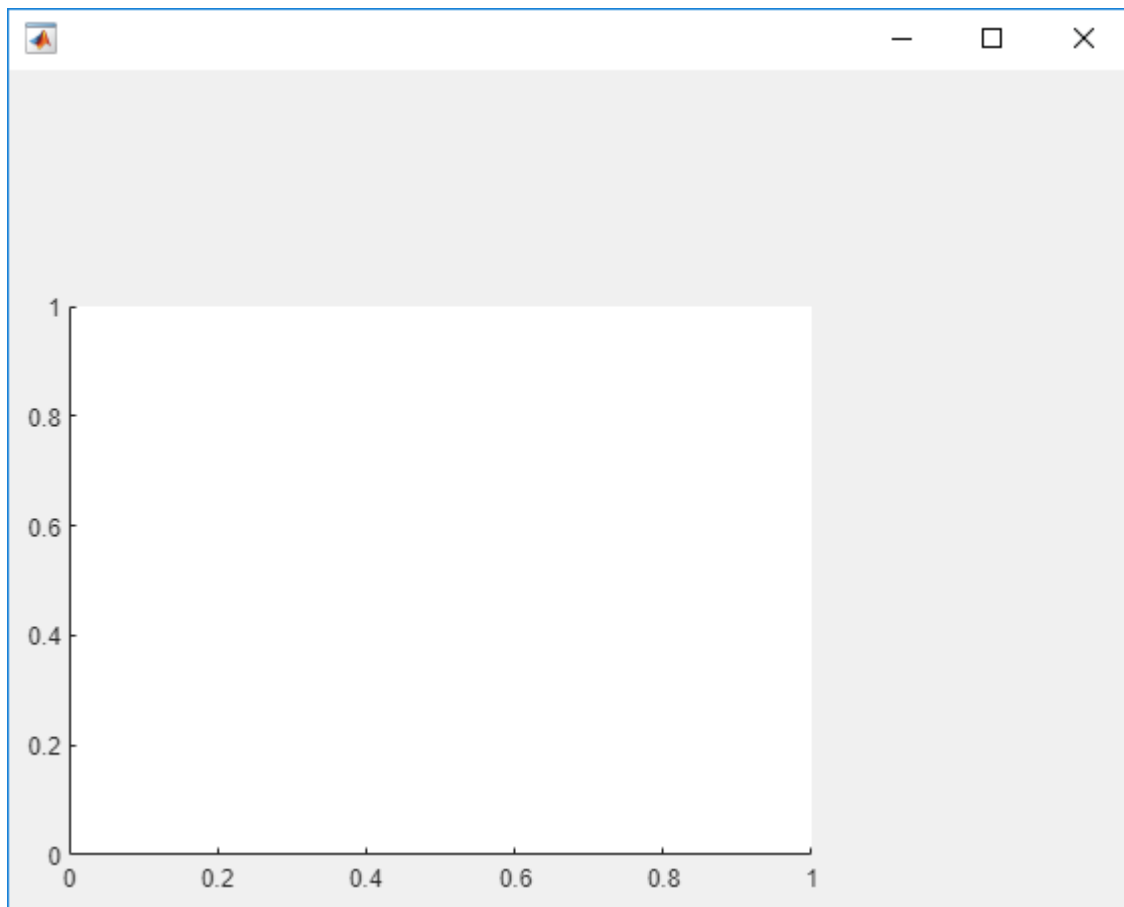

2   Create the ROI in the `UIAxes`. Call any of the ROI creation functions, such as `drawcircle`, or the ROI classes, such as `images.roi.Circle`. Specify the `UIAxes` as an argument. Move the

cursor over the axes, and click and drag the mouse to draw the ROI. The shape of the cursor does not change when used with a `UIAxes`.

```
h = drawcircle(uax);
```
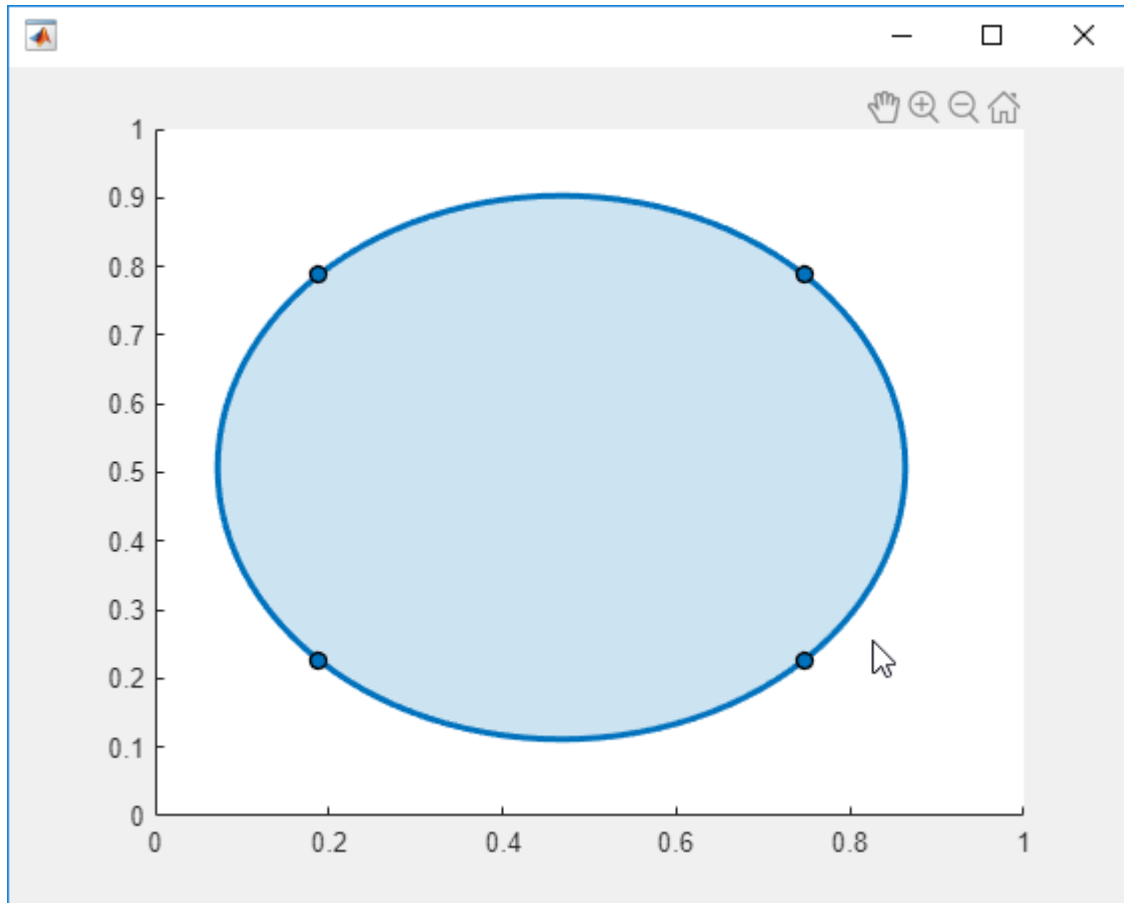

You can also create an ROI using the object creation function, such as `images.roi.Circle`. If you use the objects, you must also call the `draw` function, specifying the ROI object as an argument.

See Also

Related Examples

- "Create Binary Mask Using an ROI Function" on page 12-2

More About

- "Display Graphics in App Designer"

ROI Migration

Starting in R2018b, a new set of ROI objects replaced the previous set of ROI objects. The new objects provide better performance and more functional capabilities, such as face color transparency. With the new objects, you can also receive notification of interactions with the object, such as clicks or movement, using events. Although there are no plans to remove the old ROI objects at this time, switch to the new ROIs to take advantage of the additional capabilities and flexibility. For more information on the new ROIs, see "ROI Creation Overview" on page 12-5.

ROI Object Migration

If your code uses one of the previous ROI objects, replace this with a call to the corresponding new ROI object. Because the new ROIs offer shapes that weren't supported previously, in some cases, you have several ROIs to choose from. Two ROIs in the new system have no corresponding ROI in the previous system: `Crosshair` and `Cuboid`.

Previous ROI Object	Current ROI Object
`imellipse`	Use `Ellipse` instead. With the previous set of ROIs, you used `imellipse` to draw a circular ROI. With the new ROIs, use `Circle`.
`imfreehand`	Use `Freehand` instead. You can also use `AssistedFreehand` to create a hand-drawn ROI that "assists" your drawing by automatically following the contours of edges in the underlying image.
`imline`	Use `Line` instead.
`impoint`	Use `Point` instead.
`impoly`	Use `Polygon` instead. To create an open polygonal shape, use `Polyline`.
`imrect`	Use `Rectangle` instead.

ROI Object Function Migration

The previous set of ROIs used object functions to customize many aspects of ROI appearance and functioning. In many cases, the new ROIs replace these object functions with object properties. Instead of calling an object function, you get the value of a property or set the value of a property. For example, instead of using `getColor` to get the color of the ROI, access the value of the `Color` property of the new ROI object. For detailed information about how to migrate code to the new ROI system, see the Compatibility Considerations section of the object function reference pages associated with the previous ROI objects.

Previous ROI Object Functions	Equivalent Object Functions
`addNewPositionCallback`	Use the `addListener` object function to specify the function you want executed when the ROI moves. For more information about using events, see "ROI Events" on page 12-18.
`createMask`	Use the equivalent `createMask` object function with the new ROIs.

Previous ROI Object Functions	Equivalent Object Functions
getColor	Retrieve the value of the `Color` property of the ROI. For example, `roi_color = roi.Color;`.
getPosition	Retrieve the value of the `Position` property of the ROI. For example, `roi_pos = roi.Position;`.
getPositionConstraintFcn	Use the `DrawingArea` property to specify position constraints.
getVertices	Retrieve the value of the `Vertices` property of the ROI. For example, `roi_vert = roi.Vertices;`.
makeConstrainToRectFcn	Use the `DrawingArea` property to specify position constraints.
removeNewPositionCallback	Use the `addListener` object function to specify the function to be called with the ROI moves. To remove this callback function, delete the object returned by the `addListener` object function.
resume	Use `uiresume` instead.
setClosed	Assign a value to the ROI `Closed` property. For example, `roi.Closed = 'y'`.
setColor	Assign a value to the new ROI `Color` property. For example, `roi.Color = 'y'`.
setConstrainedPosition	Use the `DrawingArea` property to specify position constraints.
setFixedAspectRatioMode	Use `FixedAspectRatio` property of the new ROIs, setting the value to `true`.
setPosition	Assign a value to the new ROI `Position` property. The way to specify the position varies with each object. For example, `roi.Position = [50 50]`.
setPositionConstraintFcn	Use the `DrawingArea` property to specify position constraints.
setResizable	Use the `InteractionsAllowed` property, setting the value to `'translate'`.
setString	Assign a value to the new ROI `Label` property. For example, `roi.Label = 'My Label';`.
setVerticesDraggable	Use the `InteractionsAllowed` property, setting the value to `'translate'`.

Previous ROI Object Functions	Equivalent Object Functions
`wait`	Use the equivalent `wait` with the new ROI objects. Note that the new `wait` function does not support a return value containing position information.

ROI Events

With the previous ROIs, you could use the `addNewPositionCallback` object function to receive notification when the ROI moves. You specify the object and the function that you want executed when the event occurs: `id = addNewPositionCallback(h,fcn)`.

With the new ROIs, you use the `addListener` object function to receive notification when the ROI moves. You specify the object, the name of the event you want to receive notification, and the name of the function you want executed when the event occurs: `el = addlistener(roi,'ROIMoving',mycallbackfcn)`. With the new ROIs, you must specify the name of the event because you can receive notification of many other events, such as when the ROI is clicked.

To see an example, see the `Compatibility Considerations` section on the `addNewPositionCallback` reference page.

See Also

More About

- "ROI Creation Overview" on page 12-5

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`, specify:

- The input grayscale image to be filtered
- A binary mask image that defines the ROI
- A 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 (that is, [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
Circle | createMask | drawcircle | 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);
```
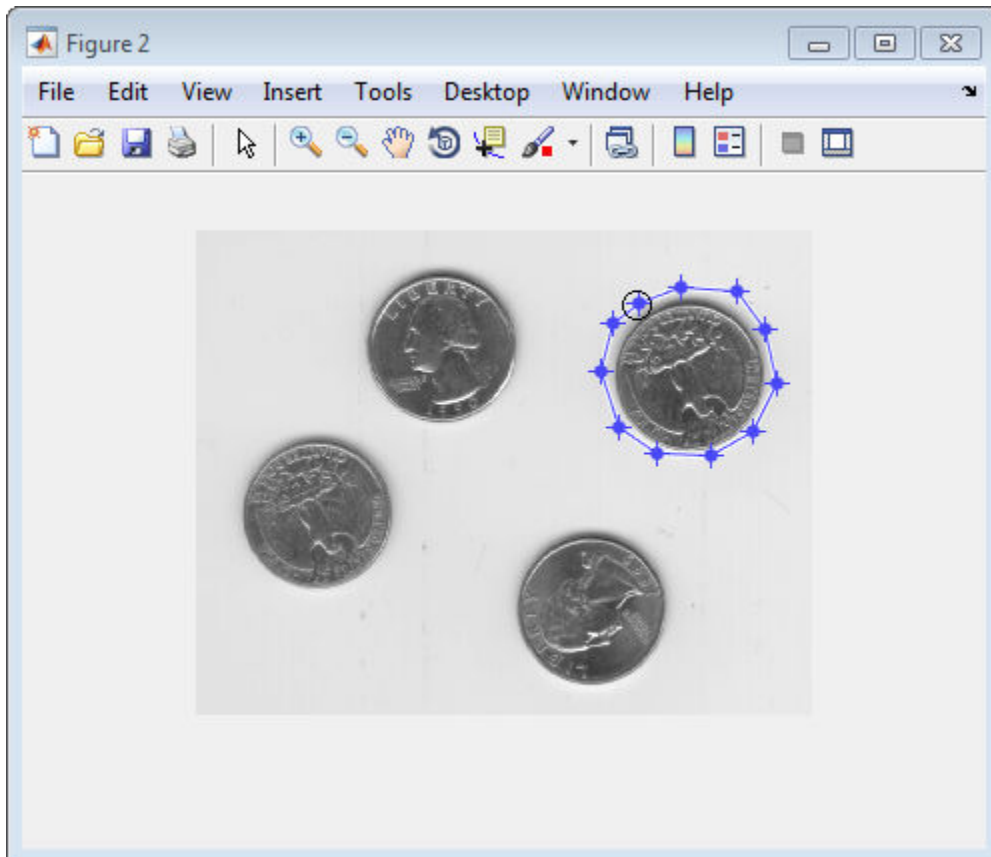

Double-click to finish defining the region. `roipoly` creates a binary image with the region filled with 1-valued pixels.

Display the mask image.

```
figure
imshow(mask)
```
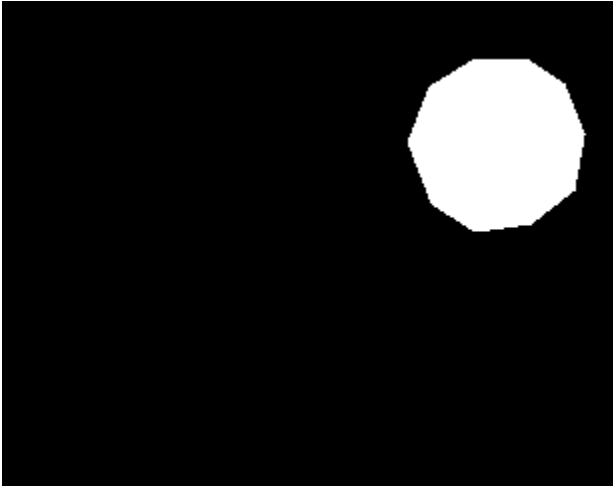

Fill the region, using `regionfill`, specifying the image to be filled and the mask image as inputs. Display the result. Note the image contains one less coin.

```
J = regionfill(I,mask);
figure
imshow(J)
```


See Also
createMask | drawpolygon | regionfill | roipoly

Calculate Properties of Image Regions Using Image Region Analyzer

This example shows how to calculate the properties of regions in binary images by using the **Image Region Analyzer** app. This example finds the largest region, measure by area, 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 section, click **Image Region Analyzer** .

On the app toolstrip, click **Load Image**, and then select **Load Image from Workspace** to load the image from the workspace into the app. In the Import From Workspace dialog box, select the image you read into the workspace, and then click **OK**.

You can also open the app from the command line by using the `imageRegionAnalyzer` function, specifying the image you want to analyze: `imageRegionAnalyzer(BW);`.

The **Image Region Analyzer** app displays the image you selected and a table where each row is a region identified in the image and each column is a property of that region, such as the area, perimeter, and orientation. (The **Image Region Analyzer** app uses the `regionprops` function to identify regions in the image and calculate properties of these regions.)

To explore the image, move the cursor over the image to access the pan and zoom controls.

The app calculates more properties than it initially includes in the table. To view additional properties in the table, click **Choose Properties** and select the properties you want to view. The app updates the table automatically, adding a new column to the table for each property.

To explore properties of the image, sort the information in the table. Initially, the app lists the properties in the order it finds them, starting in the upper-left corner of the image. To change the sorting order, in the Properties section of the app toolstrip, click **Sort Table** and select the property

on which you want to sort. For example, to find the largest region in the image, sort on the `Area` property. The **Image Region Analyzer** app sorts the table by size.

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.

See Also
Image Region Analyzer | bwareafilt | bwpropfilt | regionprops

Filter Images on Properties Using Image Region Analyzer App

This example shows how to create a new binary image, such as a mask image, by filtering an existing binary image based on 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** section, click Image Region Analyzer .
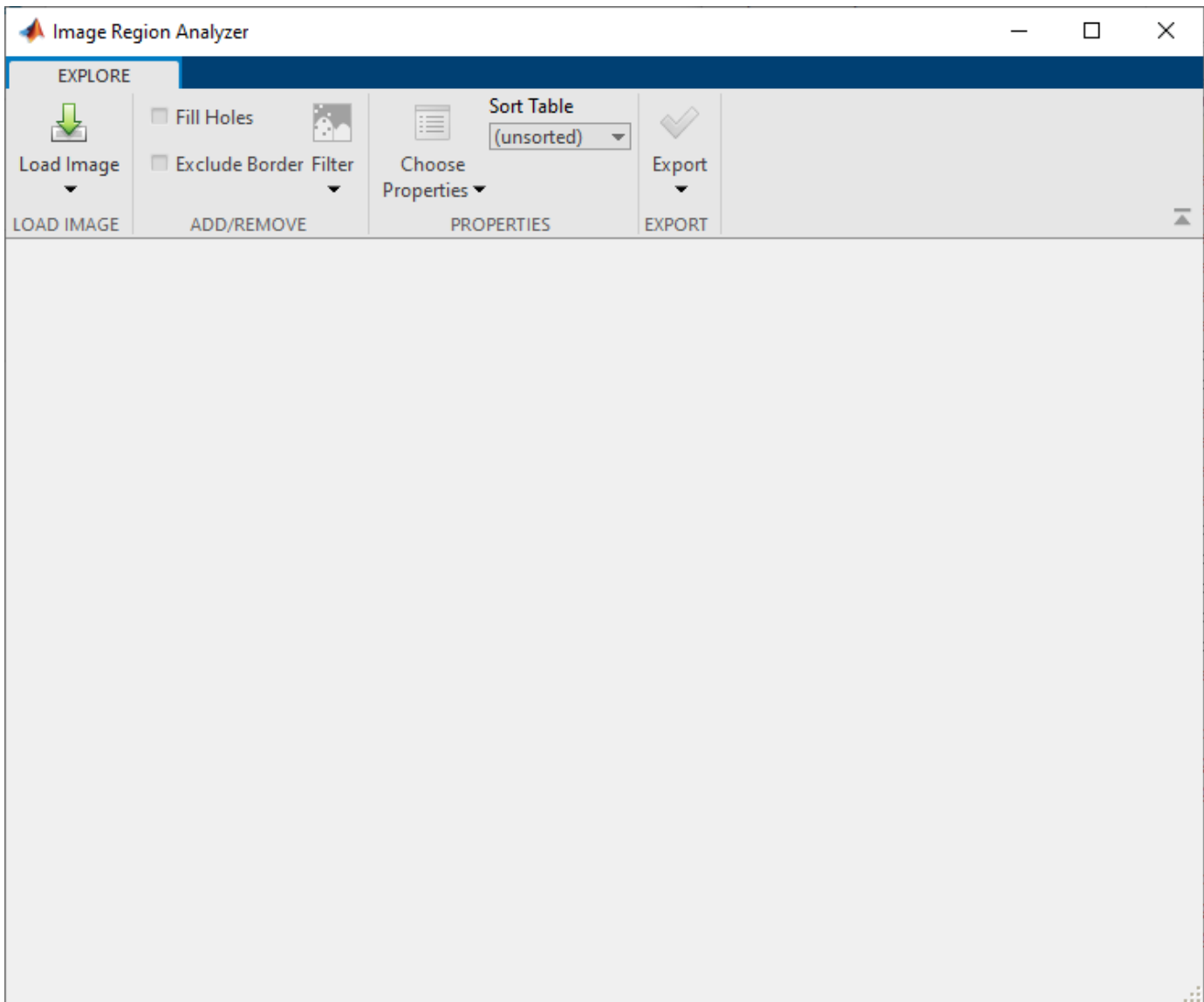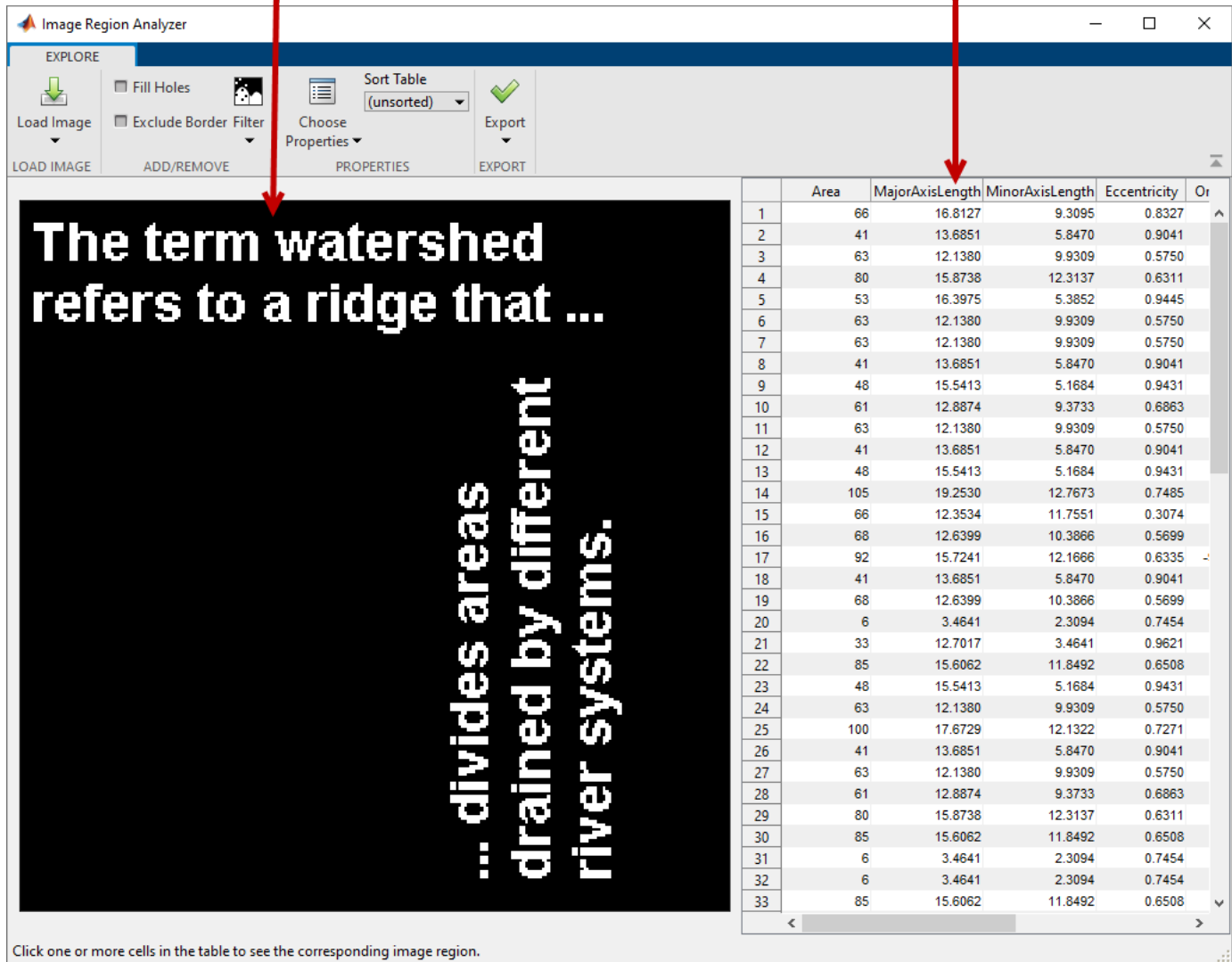
On the app toolstrip, click **Load Image**, and then select **Load Image from Workspace** to load the image from the workspace into the app. In the Import from Workspace dialog box, select the image you read into the workspace, and then click **OK**.

You can also open the app from the command line using the `imageRegionAnalyzer` function, specifying the image you want to analyze: `imageRegionAnalyzer(BW);`.

The **Image Region Analyzer** app displays the image you selected and a table where each row is a region identified in the image and each column is a property of that region, such as the area, perimeter, and orientation. (The **Image Region Analyzer** app uses the `regionprops` command to identify regions in the image and calculate properties of those regions.)

To filter on the value of a region property, on the app toolstrip, click **Filter.** Then, select the property on which you want to filter.
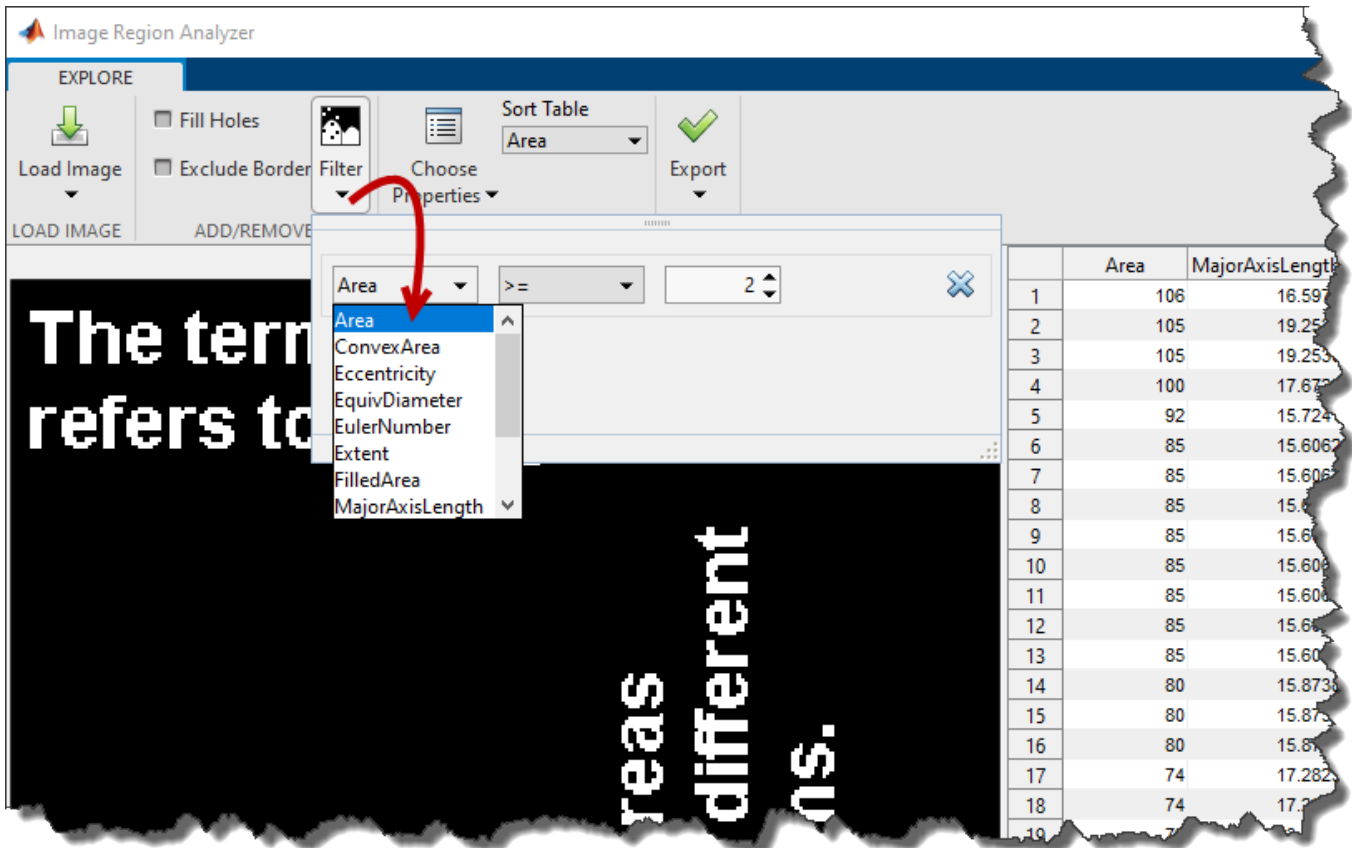
Next, specify the filter criteria. For example, to create an image that removes all but the largest regions, choose the greater than or equal to symbol (>=), and then specify the minimum value. To identify the minimum value for the desired property, you can sort the values in the table by that property. The app uses the `bwpropfilt` and `bwareafilt` functions to filter binary images.
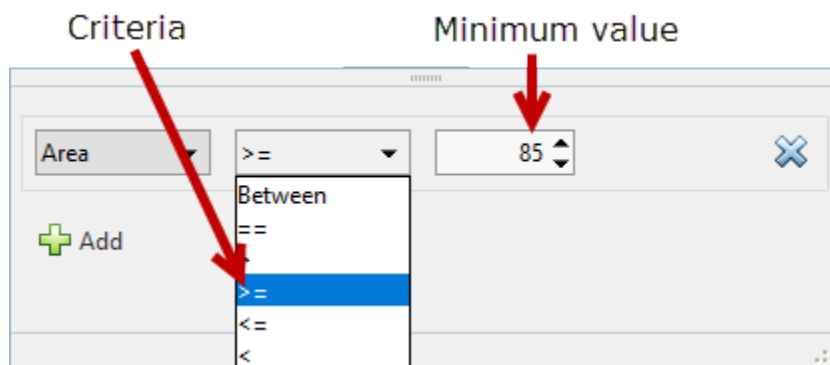
To filter on another property, click **Add**. The app displays another row in which you can select a property and specify filter criteria. The result is the intersection (logical AND) of the two filtering operations.

If you are creating a mask image, you can optionally perform cleanup operations on the mask, such as clearing all foreground pixels that touch the border and filling holes in objects. Filling holes can

12-35

change the area of regions in the image, and areas that had been filtered because they were below the threshold can become visible.
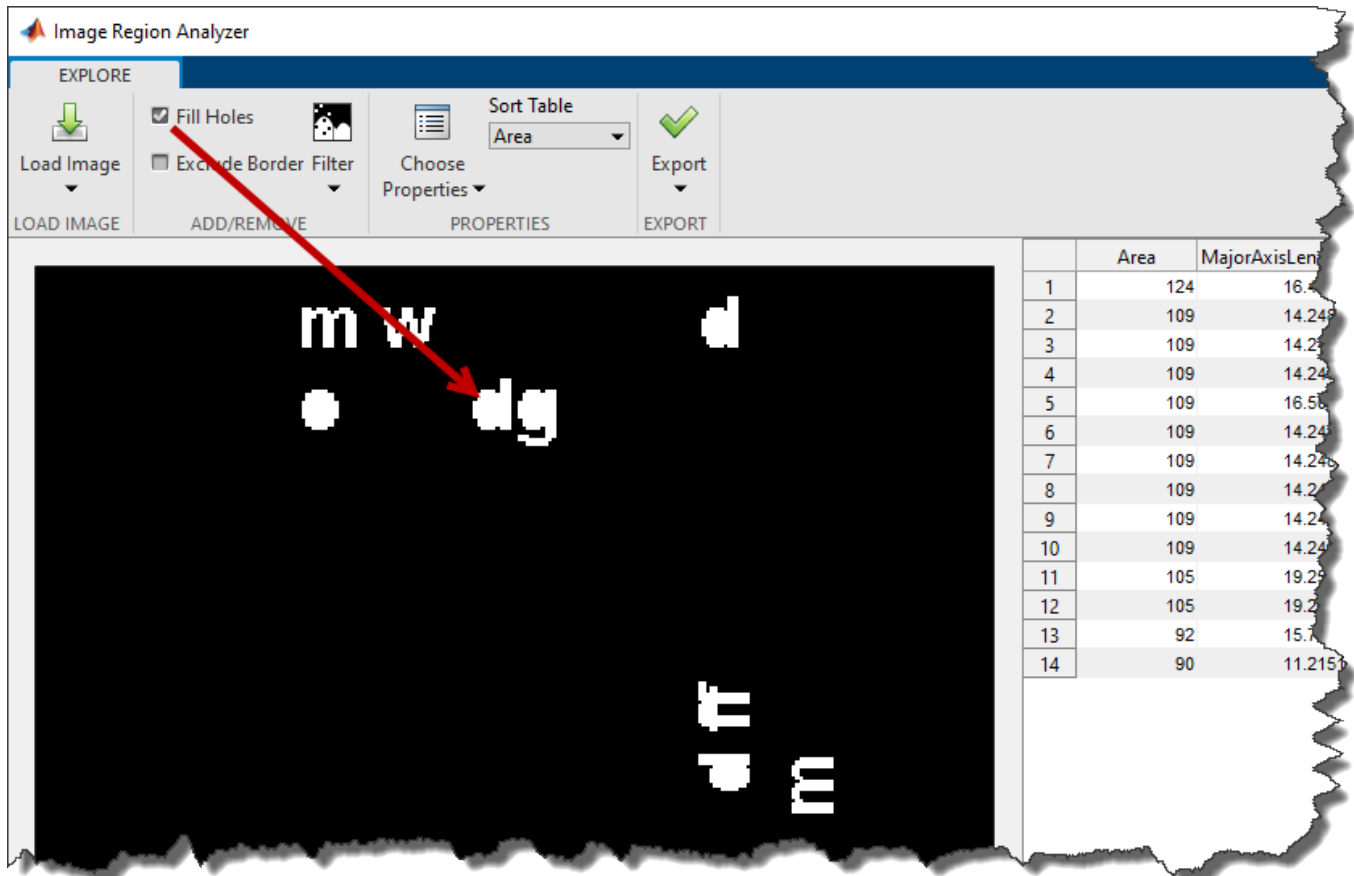
When you are done filtering the image, you can save it. Click **Export** and select **Export Image**. In the Export to Workspace dialog box, accept the default name for the mask image, or specify another name. Then, click **OK**.
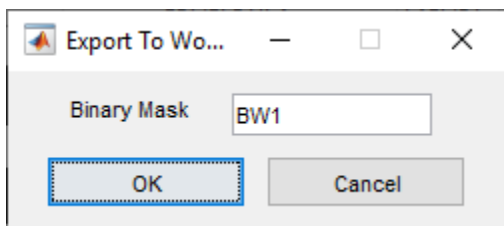
See Also
Image Region Analyzer | bwareafilt | bwpropfilt | regionprops

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);
```
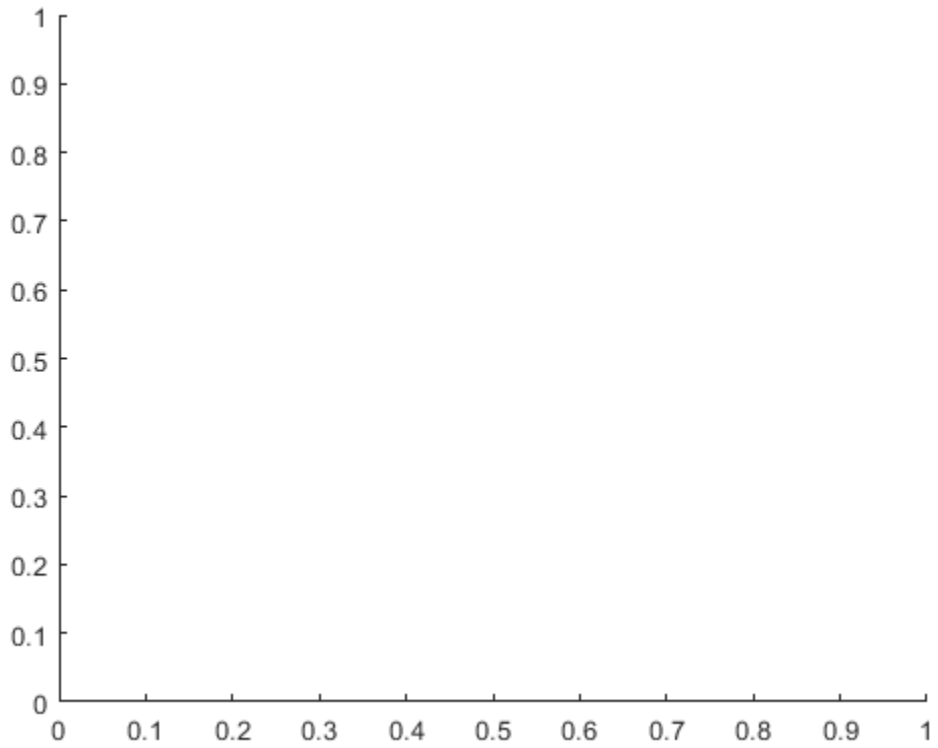
Stack both images on the same axes.

```
hImage1 = imshow(im, 'Parent', hAxes);
drawnow; % Ensure the image gets drawn.
hold on;
hImage2 = imshow(imgray, 'Parent', hAxes);
hold off;
```

Create a circular ROI on the axes.

```
centerLocation = [220, 100];
radius = 60;
hC = images.roi.Circle(...
    'Parent', hAxes,...
    'FaceAlpha',0,...
    'Center',centerLocation,...
    'Radius', radius);
```
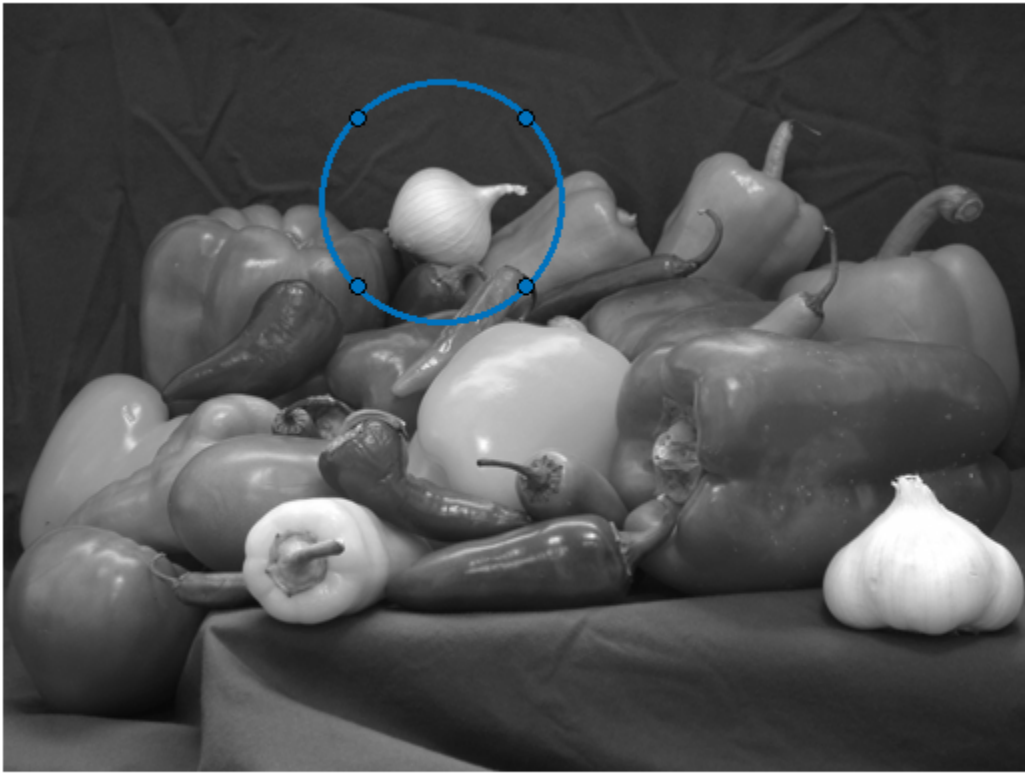
Create a listener that listens to changes in the position of the ROI (the circle). The updateAlpha function is defined at the end of this example.

```
addlistener(hC,'MovingROI', @updateAlpha);
```

Execute the callback manually the first time

```
updateAlpha(hC)
```
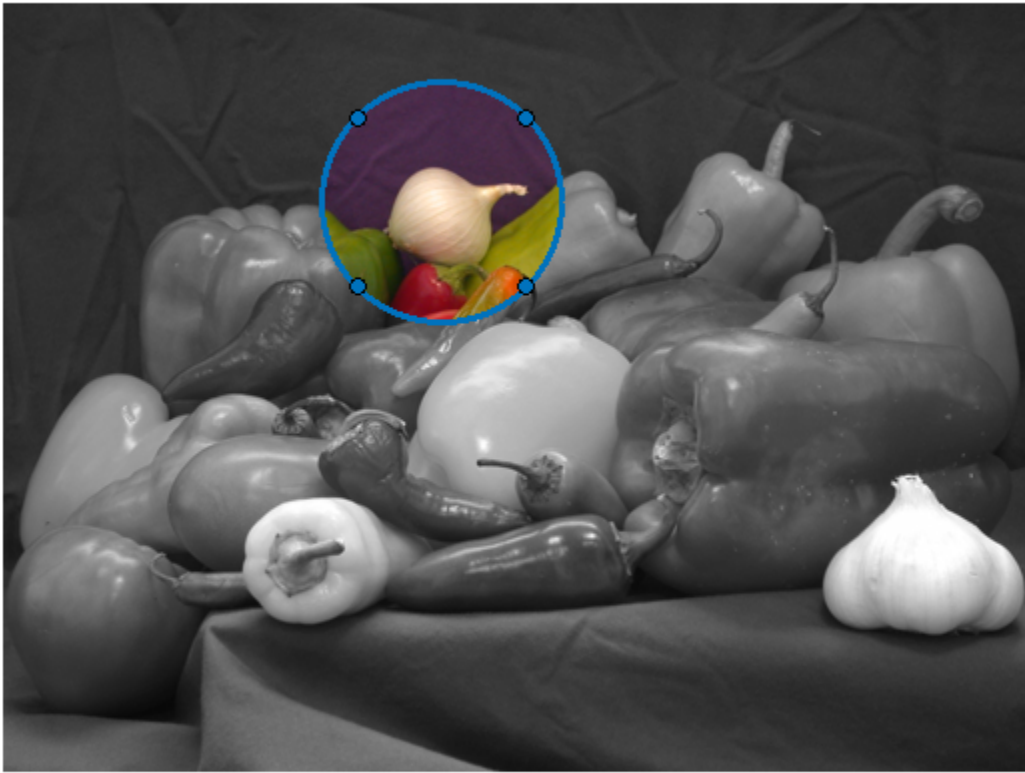
Simulate zooming in to a region.

```
hC.Parent.XLim = [75 370];
hC.Parent.YLim = [0 205];
```
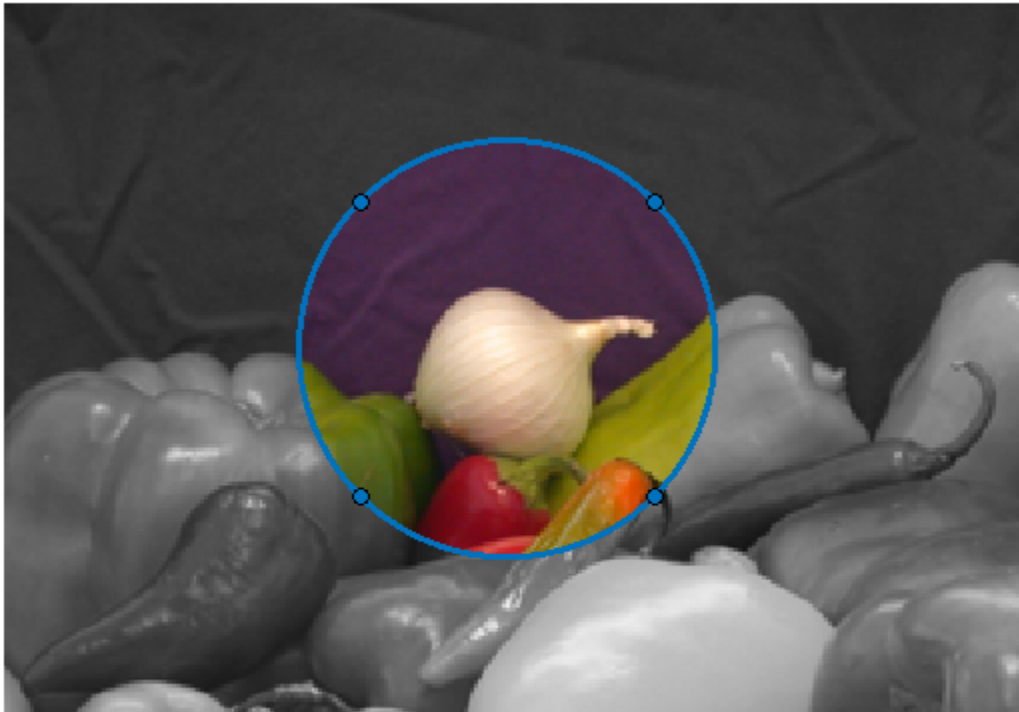
This file contains the source code for a function that implements this image comparison tool. This code listens for two additional events. When a user to enter the 't/T' key to switch which image is on top. The code also listens for the mouse scroll wheel to increase or decrease the radius of the ROI.

```
edit helperImageComparer
```

An Animation of the Tool in Use
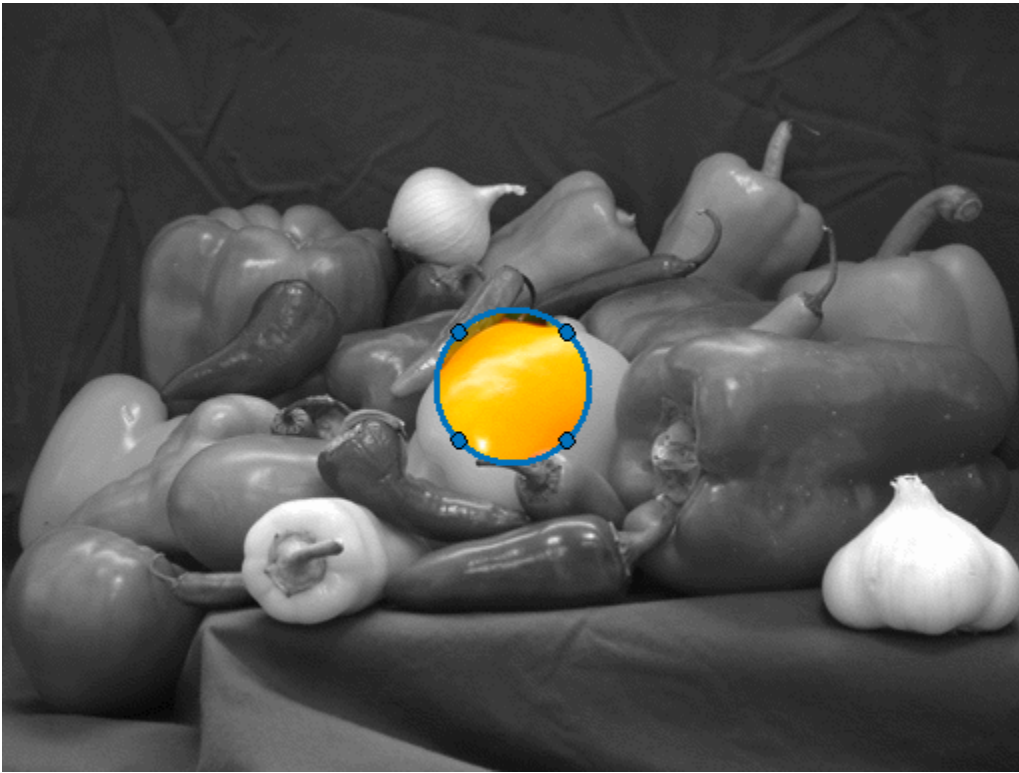
Callback Function to Update Alpha Layer

Callback function to update the alpha layer as the ROI object is moved.

```
function updateAlpha(hC, ~)
hImages = findobj(hC.Parent,'Type','image');
% Create a BW mask from the Circle ROI
mask = hC.createMask(hImages(1).CData);
% Set the alpha data so that the underlying image shows through
% only inside the circle
set(hImages(1),'AlphaData', ~mask);
end
```

See Also
Circle | addlistener | createMask

More About
- "Callbacks — Programmed Response to User Action"
- "Overview Events and Listeners"

Use Freehand ROIs to Refine Segmentation Masks

This example shows how to refine a logical segmentation mask by converting it into a Freehand ROI object. In this method, you take advantage of the interactive reshaping capbility of the Freehand ROI object to make a better fit of the edge of the segmentation mask to the shape of the image region that you want to segment.

Introduction - Image Segmentation

Segmentation algorithms are used to segment interesting parts of an image. To illustrate, this example uses K-means clustering to segment bone and tissue in an MRI image.

```
im = dicomread('knee1.dcm');
segmentedLabels = imsegkmeans(im,3);
boneMask = segmentedLabels==2;
imshowpair(im, boneMask);
```
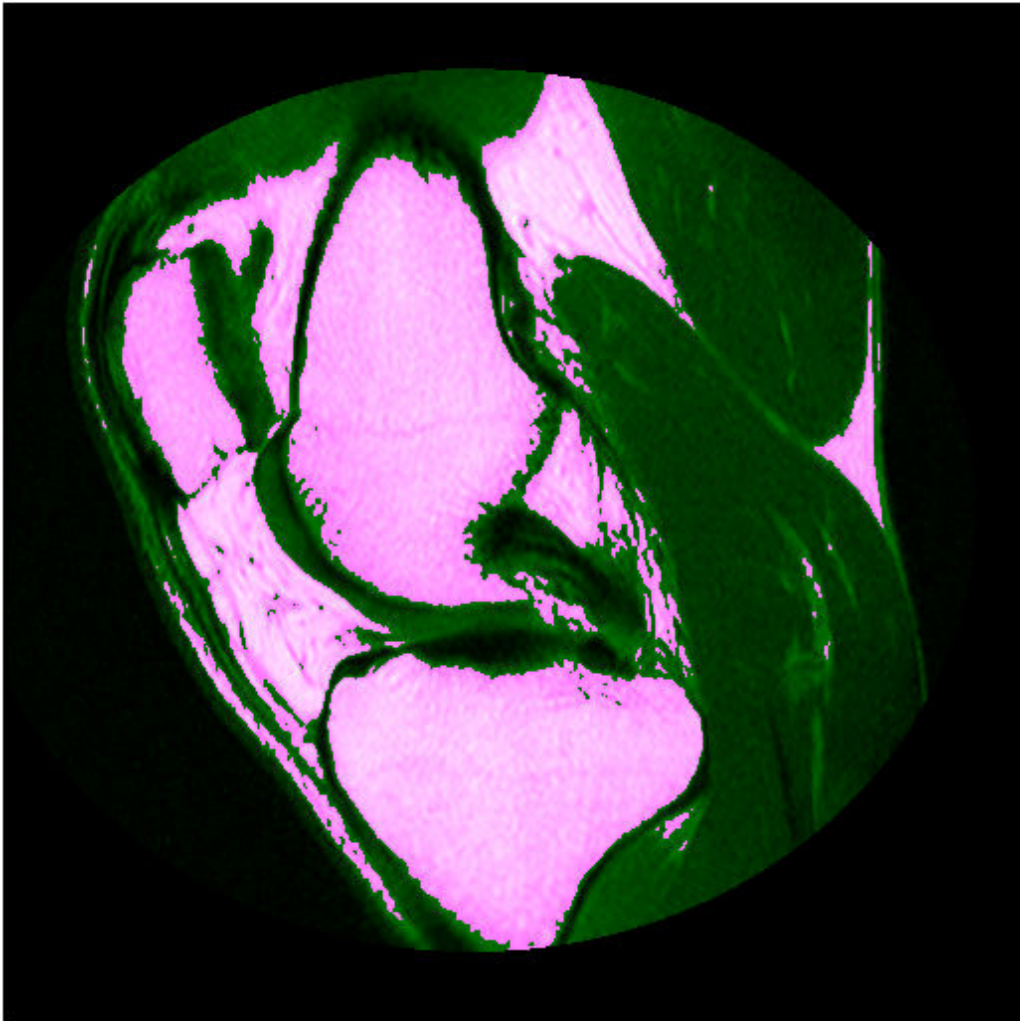
Post-Process the Segmentation Masks

Often, the results of automated segmentation algorithms need additional post-processing to clean up the masks. As a first step, select the two largest bones from the mask, the femur and the tibia.

```
boneMask = bwareafilt(boneMask, 2);
imshowpair(im, boneMask);
```
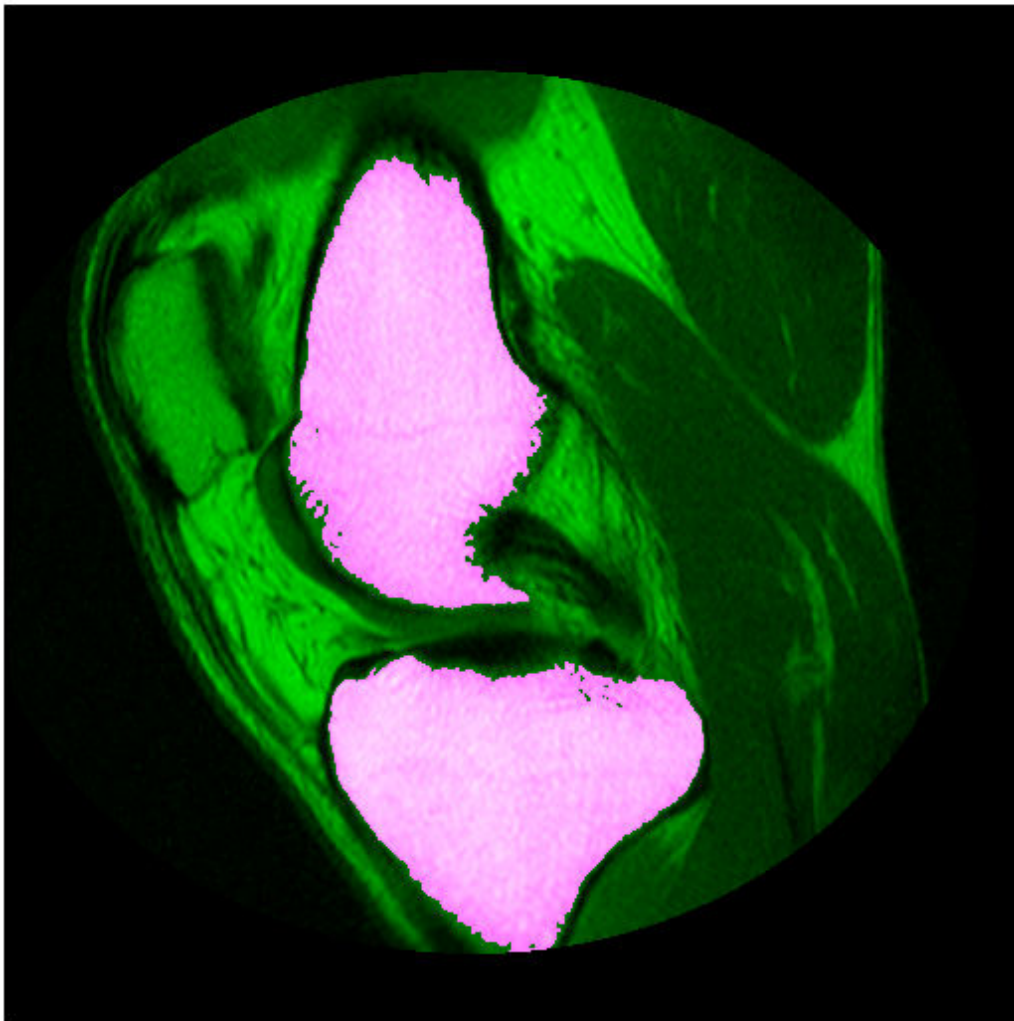
Convert Mask to Freehand ROI object

To refine the edges of the automatic k-means segmentation, convert the two masks into interactive freehand ROI objects. First, retrieve the locations of boundary pixels that delineate these two segmented regions. Note that these ROI objects are densely sampled--their `Position` property has the same resolution as the image pixels.

```
blocations = bwboundaries(boneMask,'noholes');
figure
imshow(im, []);
for ind = 1:numel(blocations)
    % Convert to x,y order.
    pos = blocations{ind};
    pos = fliplr(pos);
    % Create a freehand ROI.
```

```
    drawfreehand('Position', pos);
end
```
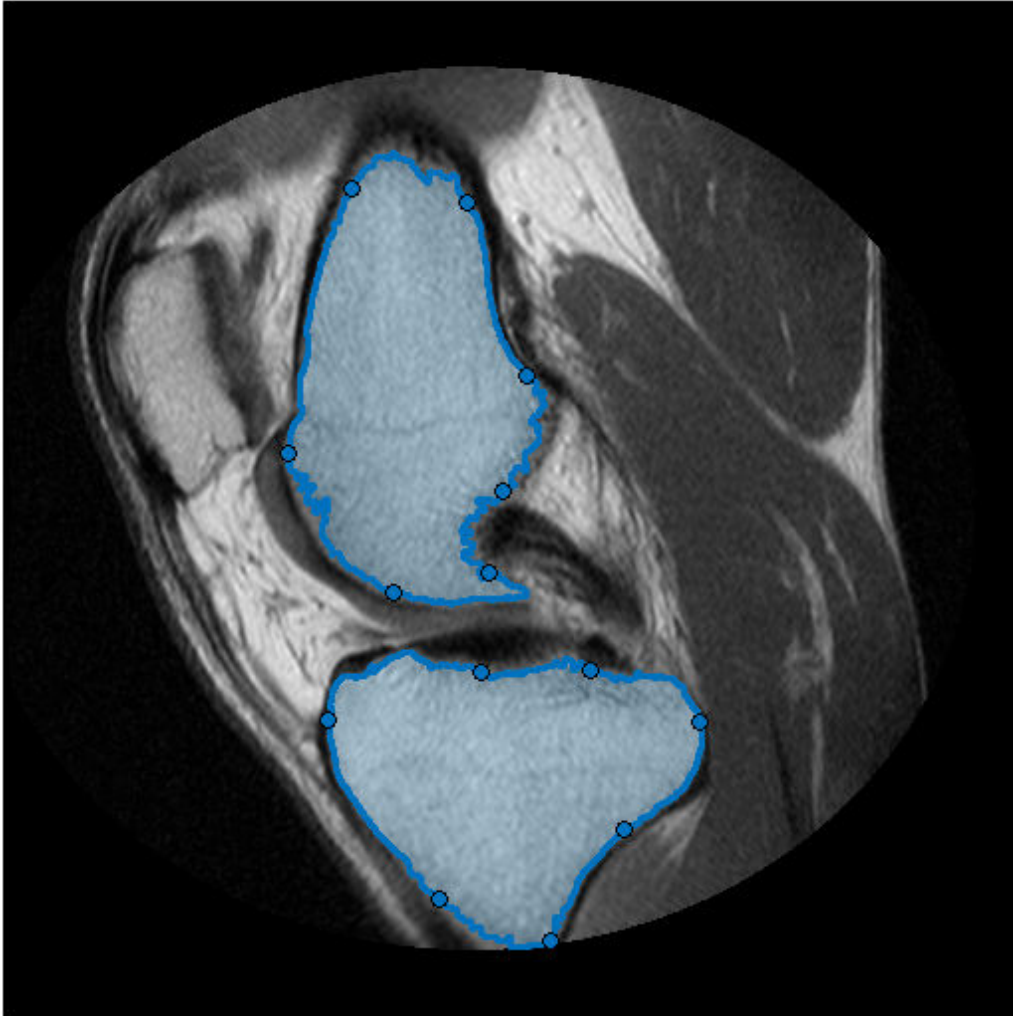

Edit the ROIs

The Freehand ROI object allows simple 'rubber-band' interactive edits. To edit the ROI, click and drag any of the waypoints along the ROI boundary. You can add additional waypoints anywhere on the boundary by double-clicking the ROI edge or by using the context menu accessible by right-clicking the edge.

Convert the Freehand ROIs Back to Masks

After editing the ROIs, convert these ROI objects back to binary masks using the ROI object's `createMask` method. Note the additional step required to include the boundary pixels in the final mask.

```matlab
% Convert edited ROI back to masks.
hfhs = findobj(gca, 'Type', 'images.roi.Freehand');
editedMask = false(size(im));

for ind = 1:numel(hfhs)
    % Accumulate the mask from each ROI
    editedMask = editedMask | hfhs(ind).createMask();

    % Include the boundary of the ROI in the final mask.
    % Ref: https://blogs.mathworks.com/steve/2014/03/27/comparing-the-geometries-of-bwboundaries
    % Here, we have a dense boundary, so we can take the slightly more
    % performant approach of just including the boundary pixels directly in
    % the mask.
    boundaryLocation = hfhs(ind).Position;
    bInds = sub2ind(size(im), boundaryLocation(:,2), boundaryLocation(:,1));
    editedMask(bInds) = true;
end
```

See Also

Freehand | bwareafilt | bwboundaries | createMask | dicomread | drawfreehand | imsegkmeans

Rotate Image Interactively Using Rectangle ROI

This example shows how to rotate an image by using a Rectangle ROI with a callback function that calls `imrotate` when you move the ROI.

Image rotation is a common preprocessing step. In this example, an image needs to be rotated by an unknown amount to align the horizon with the x-axis. You can use the `imrotate` function to rotate the image, but you need prior knowledge of the rotation angle. By using an interactive rotatable ROI, you can rotate the image in real time to match the rotation of the ROI.

Create the Rotatable Rectangle ROI

Display an image in an Axes.

```
im = imread('baby.jpg');
hIm = imshow(im);
```

Get the size of the image.

```
sz = size(im);
```

Determine the position and size of the Rectangle ROI as a 4-element vector of the form [x y w h]. The ROI will be drawn at the center of the image and have half of the image width and height.

```
pos = [(sz(2)/4) + 0.5, (sz(1)/4) + 0.5, sz(2)/2, sz(1)/2];
```

Create a rotatable Rectangle ROI at the specified position and set the Rotatable property to `true`. You can then rotate the rectangle by clicking and dragging near the corners. As the ROI moves, it broadcasts an event MovingROI. By adding a listener for that event and a callback function that executes when the event occurs, you can rotate the image in response to movements of the ROI.

```
h = drawrectangle('Rotatable',true,...
    'DrawingArea','unlimited',...
    'Position',pos,...
    'FaceAlpha',0);
```

Place a prompt in the label.

```
h.Label = 'Rotate rectangle to rotate image';
```
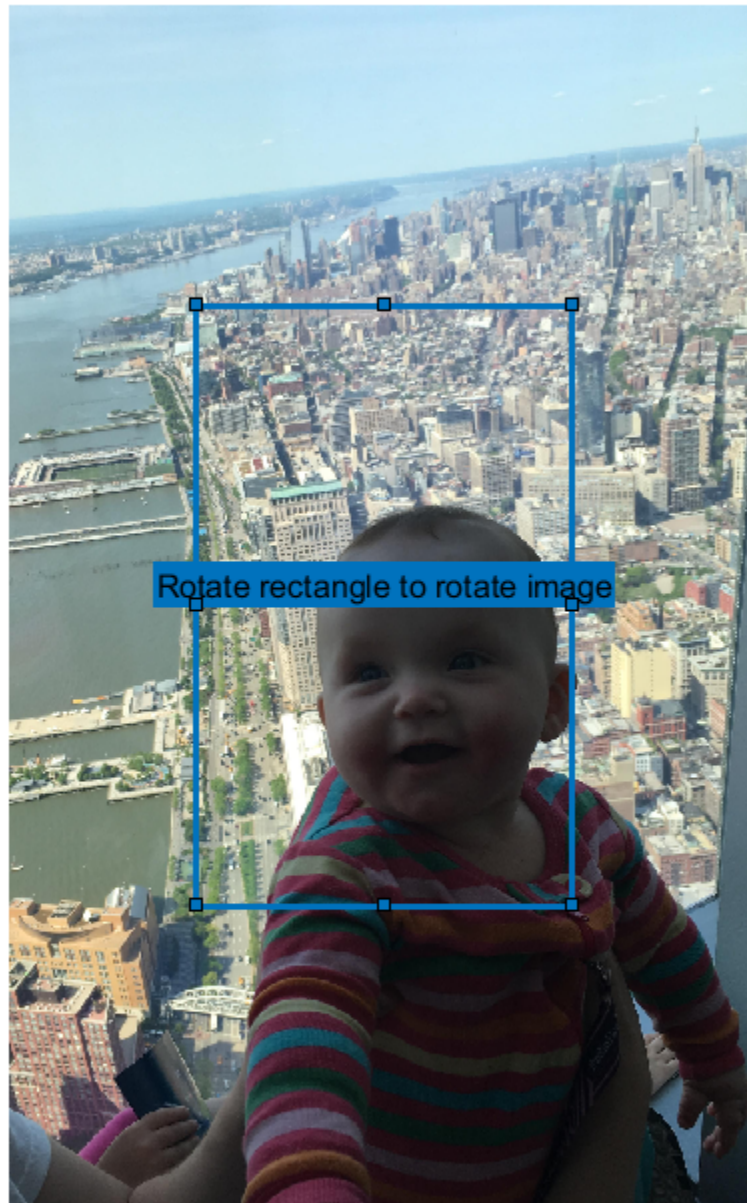
Add a listener that listens for any movement of the ROI.

```
addlistener(h,'MovingROI',@(src,evt) rotateImage(src,evt,hIm,im));
```

Call imrotate in Callback Function

Define a callback function that executes as the Rectangle ROI moves. This function retrieves the current rotation angle of the ROI, calls imrotate on the image with that rotation angle, and updates the display. The function also updates the label to display the current rotation angle.

```
function rotateImage(src,evt,hIm,im)

% Only rotate the image when the ROI is rotated. Determine if the
% RotationAngle has changed
if evt.PreviousRotationAngle ~= evt.CurrentRotationAngle

    % Update the label to display current rotation
    src.Label = [num2str(evt.CurrentRotationAngle,'%30.1f') ' degrees'];

    % Rotate the image and update the display
    im = imrotate(im,evt.CurrentRotationAngle,'nearest','crop');
    hIm.CData = im;

end

end
```
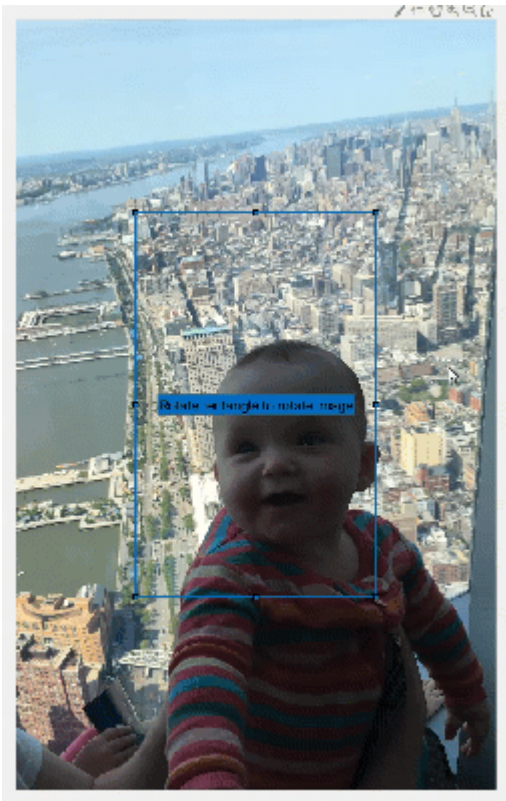

See Also

Rectangle | addlistener | drawrectangle | imrotate

More About

- "Callbacks — Programmed Response to User Action"
- "Overview Events and Listeners"

Subsample or Simplify a Freehand ROI

This example shows how to subsample or reduce the number of points in a `Freehand` ROI object.

Introduction

The `drawfreehand` function creates a smooth-looking, freehand, region-of-interest (ROI). However, the edge of the ROI is actually made of discrete points distributed all along the boundary. Two factors contribute to how smooth a freehand ROI looks: 1) the density of points and 2) the `Smoothing` property of the freehand ROI object.

When drawing interactively, mouse motion determines the density of points. For large complex ROIs, the number of points used can be quite large.

The `Smoothing` property controls how the boundary looks. By default, the `Freehand` object uses a Gaussian smoothing kernel with a sigma value of 1 and a filter size of 5. Changing this value only changes how the boundary looks, it does not change the underlying `Position` property of the object.

Default Density of Points

Reducing the density of points can help reduce the space required to store the ROI data and may also speed up any computation that depends on the number of these points. One way to reduce the density of points is to subsample the points, for example, pick every other point.

Create a sample freehand ROI by converting a mask to an ROI. The ROI is very dense since every boundary pixel will correspond to a point in the ROI.

```
im = imread('football.jpg');
bw = im(:,:,1)>200;
bw = bwareafilt(bw, 1);
bloc = bwboundaries(bw,'noholes');
roipos = fliplr(bloc{1});
imshow(im);
hfh = drawfreehand('Position', roipos);
```
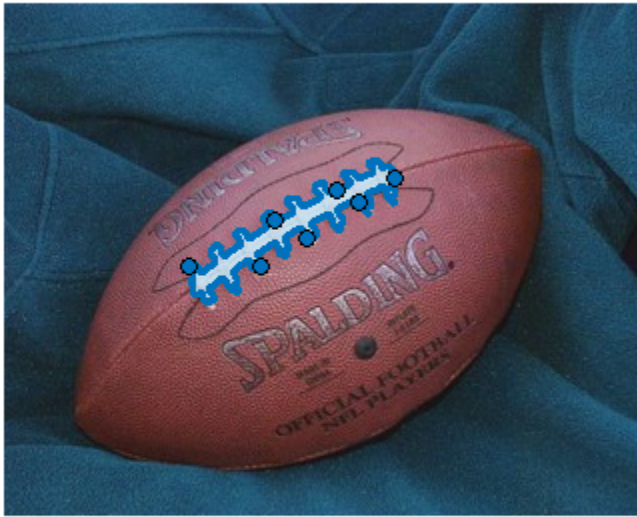
To visualize the density of the points, turn every point in the ROI into a waypoint.

```
hfh.Waypoints(:) = true;

title('Original density');
snapnow

% Zoom in.
xlim([80 200]);
ylim([70 160]);
snapnow
```

Original density

Original density

Subsampling the Position Points

Subsample the points that make up the `Position` property of the freehand ROI. Since the freehand ROI is very dense. Subsampling can substantially reduce the size without loosing fidelity. Query the initial, full, fine-grained position.

```
fpos = hfh.Position;
```

Subsample, picking every other point.

```
cpos = fpos(1:2:end,:);
```

Update the Position property of the ROI.

```
hfh.Position = cpos;
```

Original density

To see the density, turn all points into waypoints.

```
hfh.Waypoints(:) = true;
title('Simple Subsample');
snapnow
```

Simple Subsample

Subsampling - Using Rate of Change

A better approach to subsample the points would be to selectively start removing points which have low *curvature*. It makes more sense to remove a point that is along a relatively straight portion of the ROI rather than one near a curve. One simple approach to define a curvature value is to measure the rate of the change in position locations.

Measure the rate of change. The neighbor of the first point is the last point.

```
dfpos = diff([fpos(end,:); fpos]);
```

Define an ad-hoc measure of curvature based on a simple low-pass filter.

```
cm = sum(abs(conv2(dfpos, ones(3,2),'same')),2);
```

Sort by curvature.

```
[~, cmInds] = sort(cm);
```

Pick 3/4 of the points with lower curvature values to remove from the ROI.

```
numPointsToCull = round(0.25*size(fpos,1));
```

Remove those positions.

```
cpos = fpos;
cpos(cmInds(1:numPointsToCull),:) = [];
```

Update the ROI, turning on all Waypoints to see the impact.

```
hfh.Position = cpos;
hfh.Waypoints(:) = true;
```

```
title('Curvature Based Subsample (factor of 4)');
snapnow
```

Curvature Based Subsample (factor of 4)
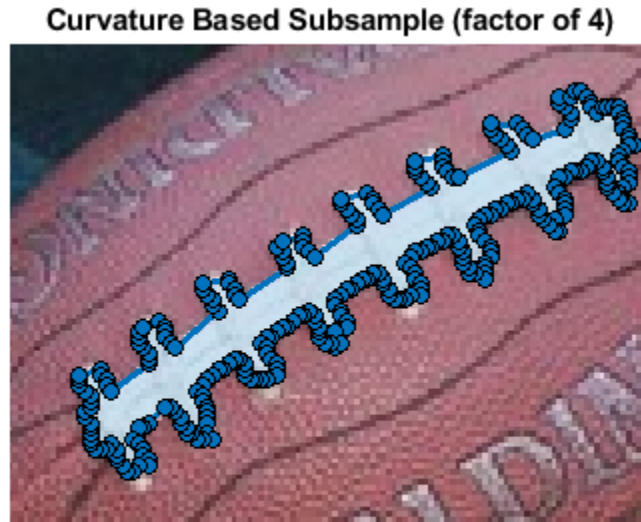
Subsampling - Using `reduce` method on freehand ROI objects

An even better approach to subsample the points would be to use the `reduce` method on the ROI object. The `reduce` method operates directly on the `Position` property of the ROI object. You can affect the number of points removed by specifying a tolerance value between [0 1.0] as an optional input argument. The default value of tolerance is 0.01.

Reset the `Position` property and call `reduce` on the ROI object.

```
hfh.Position = fpos;
reduce(hfh);

% View the updated ROI, turning all the points into waypoints to see the
% impact.
hfh.Waypoints(:) = true;
title('Subsampling using reduce method');
snapnow
```

Subsampling using reduce method

Interactive Subsampling

Another way to subsample is to use events to make this process easier. First create a listener to interactively change the number of points that the freehand ROI uses. Use the `UserData` property of the Freehand object to cache the full resolution `Position` data, along with the current value of tolerance. Then add a custom context menu to the ROI object by creating a new `uimenu` and parenting it to the `UIContextMenu` of the Freehand object. This menu option allows you to *finalize* the ROI, which deletes the temporary cache.

Restore the original ROI, and cache the original position along with its curvature measure in UserData.

```
hfh.Waypoints(:) = true;
hfh.UserData.fpos = fpos;
hfh.UserData.tol = 0;
```

Subsampling using reduce method

Respond to mouse scroll.

```
h = gcf;
h.WindowScrollWheelFcn = @(h, evt) changeSampleDensity(hfh, evt);
```

Add a context menu to finalize the ROI and perform any clean up needed.

```
uimenu(hfh.UIContextMenu, 'Text','Finalize',...
    'MenuSelectedFcn', @(varargin)finalize(hfh));

title('Scroll to change density interactively');
```

Scroll to change density interactively

Animation of the Interactive Subsampling
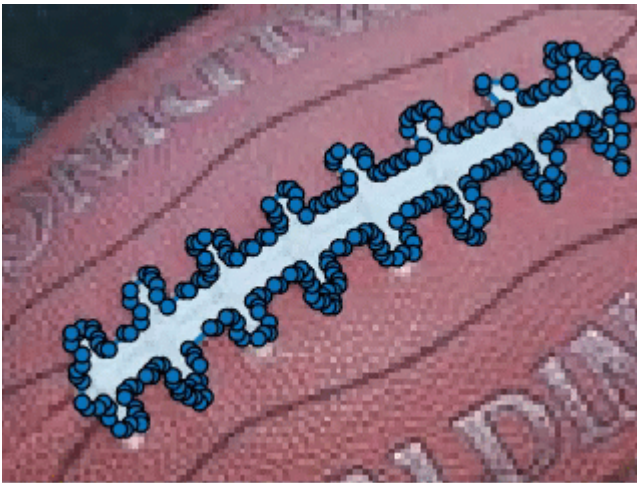
Callback Function - Change the Sample Density Based on Mouse Scroll

This function gets called on scroll action. Scrolling up increases the density, and scrolling down decreases it. This allows you to interactively select the number of points to retain.

```
function changeSampleDensity(hfh, evt)
% Restore Position property of ROI.
hfh.Position = hfh.UserData.fpos;
% Change tolerance by a fixed amount based on the direction of the scroll.
% This code changes the tolerance by 0.01 for every scroll increment.
tol = hfh.UserData.tol + 0.01 * (evt.VerticalScrollCount);
% Restrict the range of tolerance values to be from 0 to 0.15, which is the
```

```
% useful range.
tol = max(min(tol, 0.15), 0);
% Call |reduce| with the specified tolerance.
reduce(hfh,tol);
hfh.UserData.tol = tol;
% Update the ROI and turn all the points into waypoints to show the
% density.
hfh.Waypoints(:) = true;
end
```

Callback Function - Finalize the Freehand ROI

Delete and create a new Freehand ROI with the subsampled points to save on space.

```
function finalize(hfh)
h = ancestor(hfh, 'figure');
% Reset the mouse scroll wheel callback.
h.WindowScrollWheelFcn = [];
% Save finalized set of points.
pos = hfh.Position;
% Delete and create a new Freehand ROI with the new |Position| value.
delete(hfh);
drawfreehand(gca, 'Position', pos);
end
```

See Also
Freehand | bwareafilt | bwboundaries | drawfreehand

More About
- "Callbacks — Programmed Response to User Action"
- "Overview Events and Listeners"

Measure Distances in an Image

This example shows how to use line ROIs to measure distances in an image. You can also calibrate the measurements to real world values and specify the units. The example illustrates how you can seamlessly add, edit, and remove ROIs without needing to enter into any specific drawing mode.

Read Image into the Workspace and Display Image

Read image into the workspace.

```
im = imread('concordorthophoto.png');
```

Gather data about the image, such as its size, and store the data in a structure that you can pass to callback functions.

```
sz = size(im);
myData.Units = 'pixels';
myData.MaxValue = hypot(sz(1),sz(2));
myData.Colormap = hot;
myData.ScaleFactor = 1;
```

Display the image in an axes.

```
hIm = imshow(im);
```

Specify a callback function for the `ButtonDownFcn` callback on the image. Pass the `myData` structure to the callback function. This callback function creates the line objects and starts drawing the ROIs.

```
hIm.ButtonDownFcn = @(~,~) startDrawing(hIm.Parent,myData);
```

Create Callback Function to Start Drawing ROIs

Create the function used with the `ButtonDownFcn` callback to create line ROIs. This function:

1. Instantiates a line ROI object.

2. Sets up listeners to react to clicks and movement of the ROI.

3. Adds a custom context menu to the ROIs that includes a 'Delete All' option.

4. Begins drawing the ROI, using the point clicked in the image as the starting point.

```
function startDrawing(hAx,myData)

% Create a line ROI object. Specify the initial color of the line and
% store the |myData| structure in the |UserData| property of the ROI.
h = images.roi.Line('Color',[0, 0, 0.5625],'UserData',myData);

% Set up a listener for movement of the line ROI. When the line ROI moves,
% the |updateLabel| callback updates the text in the line ROI label and
% changes the color of the line, based on its length.
addlistener(h,'MovingROI',@updateLabel);
```

```
% Set up a listener for clicks on the line ROI. When you click on the line
% ROI, the |updateUnits| callback opens a GUI that lets you specify the
% known distance in real-world units, such as, meters or feet.
addlistener(h,'ROIClicked',@updateUnits);

% Get the current mouse location from the |CurrentPoint| property of the
% axes and extract the _x_ and _y_ coordinates.
cp = hAx.CurrentPoint;
cp = [cp(1,1) cp(1,2)];

% Begin drawing the ROI from the current mouse location. Using the
% |beginDrawingFromPoint| method, you can draw multiple ROIs.
h.beginDrawingFromPoint(cp);

% Add a custom option to the line ROI context menu to delete all existing
% line ROIs.
c = h.UIContextMenu;
uimenu(c,'Label','Delete All','Callback',@deleteAll);

end
```


Create Callback Function to Update ROI Label and Color

Create the function that is called whenever the line ROI is moving, that is, when the `'MovingROI'` event occurs. This function updates the ROI label with the length of the line and changes the color of the line based on its length.

This function is called repeatedly when the ROI moves. If you want to update the ROI only when the movement has finished, listen for the `'ROIMoved'` event instead.

```
function updateLabel(src,evt)

% Get the current line position.
pos = evt.Source.Position;
```

```
% Determine the length of the line.
diffPos = diff(pos);
mag = hypot(diffPos(1),diffPos(2));

% Choose a color from the color map based on the length of the line. The
% line changes color as it gets longer or shorter.
color = src.UserData.Colormap(ceil(64*(mag/src.UserData.MaxValue)),:);

% Apply the scale factor to line length to calibrate the measurements.
mag = mag*src.UserData.ScaleFactor;

% Update the label.
set(src,'Label',[num2str(mag,'%30.1f') ' ' src.UserData.Units],'Color',color);

end
```

Create Callback Function to Update Measurement Units

Create the function that is called whenever you double-click the ROI label. This function opens a popup dialog box in which you can enter information about the real-world distance and units.

This function listens for the `'ROIClicked'` event, using event data to check the type of click and the part of the ROI that was clicked.

The popup dialog box prompts you to enter the known distance and units for this measurement. With this information, you can calibrate all the ROI measurements to real world units.

```
function updateUnits(src,evt)

% When you double-click the ROI label, the example opens a popup dialog box
% to get information about the actual distance. Use this information to
% scale all line ROI measurements.
if strcmp(evt.SelectionType,'double') && strcmp(evt.SelectedPart,'label')

    % Display the popup dialog box.
    answer = inputdlg({'Known distance','Distance units'},...
        'Specify known distance',[1 20],{'10','meters'});

    % Determine the scale factor based on the inputs.
    num = str2double(answer{1});

    % Get the length of the current line ROI.
    pos = src.Position;
    diffPos = diff(pos);
    mag = hypot(diffPos(1),diffPos(2));

    % Calculate the scale factor by dividing the known length value by the
    % current length, measured in pixels.
    scale = num/mag;

    % Store the scale factor and the units information in the |myData|
    % structure.
    myData.Units = answer{2};
    myData.MaxValue = src.UserData.MaxValue;
    myData.Colormap = src.UserData.Colormap;
    myData.ScaleFactor = scale;
```

```matlab
    % Reset the data stored in the |UserData| property of all existing line
    % ROI objects. Use |findobj| to find all line ROI objects in the axes.
    hAx = src.Parent;
    hROIs = findobj(hAx,'Type','images.roi.Line');
    set(hROIs,'UserData',myData);

    % Update the label in each line ROI object, based on the information
    % collected in the input dialog.
    for i = 1:numel(hROIs)

        pos = hROIs(i).Position;
        diffPos = diff(pos);
        mag = hypot(diffPos(1),diffPos(2));

        set(hROIs(i),'Label',[num2str(mag*scale,'%30.1f') ' ' answer{2}]);

    end

    % Reset the |ButtonDownFcn| callback function with the current |myData|
    % value.
    hIm = findobj(hAx,'Type','image');
    hIm.ButtonDownFcn = @(~,~) startDrawing(hAx,myData);

end

end
```
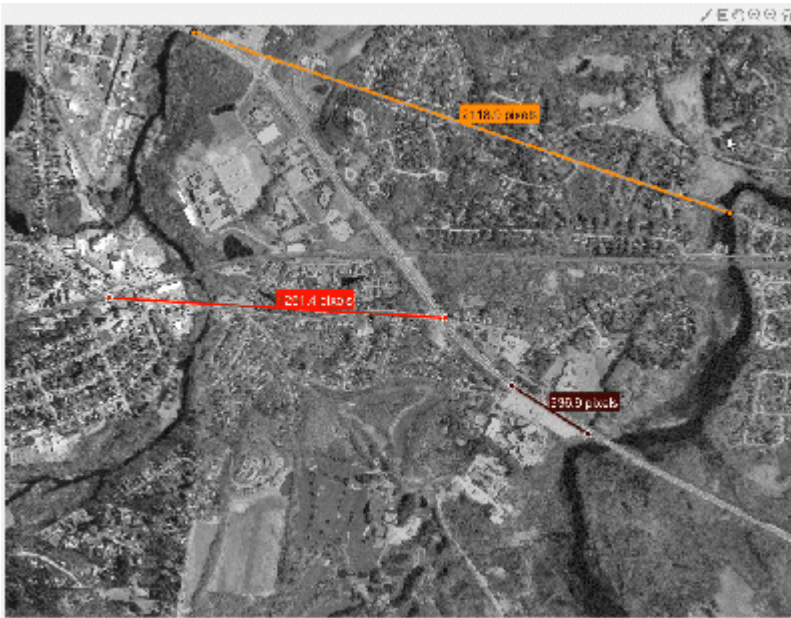

Create Callback Function to Delete All ROIs

Create the function to delete all ROIs. You added a custom context menu item to each line ROI in the startDrawing callback function. This is the callback associated with that custom context menu. This callback uses the findobj function to search for the ROI Type and deletes any found ROIs.

```matlab
function deleteAll(src,~)
```

```
hFig = ancestor(src,'figure');
hROIs = findobj(hFig,'Type','images.roi.Line');
delete(hROIs)

end
```


See Also

Line | addlistener | beginDrawingFromPoint | drawline

More About

- "Callbacks — Programmed Response to User Action"
- "Overview Events and Listeners"

Use Polyline to Create An Angle Measurement Tool

This example shows the steps need to create a three-point measurement tool using a Polyline ROI

Display Image and Polyline ROI

Display an image and a polyline at a right angle. We want to prompt the user to manipulate the polyline to measure angles of interest.

```matlab
im = imread('gantrycrane.png');

% Display image in the axes
imshow(im)

% Get size of image
m = size(im,1);
n = size(im,2);

% Get center point of image for initial positioning
midy = ceil(m/2);
midx = ceil(n/2);

% Position first point vertically above the middle
firstx = midx;
firsty = midy - ceil(m/4);
lastx = midx + ceil(n/4);
lasty = midy;

% Add empty context menu to replace default menu
c = uicontextmenu;

% Create a two-segment right-angle polyline centered in the image
h = drawpolyline('Parent',gca,...
    'Position',[firstx,firsty;midx,midy;lastx,lasty],...
    'Label','Modify angle to begin...',...
    'Color',[0.8,0.2,0.2],...
    'UIContextMenu',c);
```
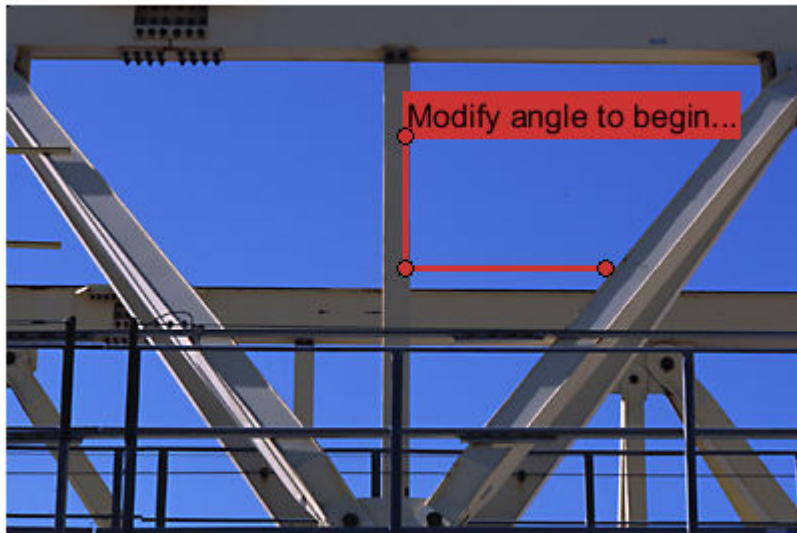
```matlab
% Listen to event that fires as the ROI is being moved
addlistener(h,'MovingROI',@(src,evt) updateAngle(src,evt));

% Intercept attempts to add or remove vertices
addlistener(h,'AddingVertex',@(src,evt) storePositionInUserData(src,evt));
addlistener(h,'VertexAdded',@(src,evt) recallPositionInUserData(src,evt));
addlistener(h,'DeletingVertex',@(src,evt) storePositionInUserData(src,evt));
addlistener(h,'VertexDeleted',@(src,evt) recallPositionInUserData(src,evt));
```

Update Label in Callback

As the user reshapes the polyline, we want to recompute the angle and update the label displayed.

```matlab
function updateAngle(src,evt)

% Get the current position
p = evt.CurrentPosition;

% Find the angle
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)];
theta = acos(dot(v1,v2)/(norm(v1)*norm(v2)));

% Convert it to degrees
angleDegrees = (theta * (180/pi));

% Update the Label to display the angle
src.Label = sprintf('(%1.0f) degrees',angleDegrees);

end
```

12-73

Prevent Users From Adding or Removing Vertices

The polyline allows users to add and remove vertices; however, an angle measurement tool assumes that there are only three vertices at any time. We want to add listeners to intercept when a user is attempting to add or remove a vertex. In the callback for the 'AddingVertex' and DeletingVertex' events, the vertex of interest has not yet been added or deleted. We will cached the current polyline position in the UserData property and restore it in the callback for the 'VertexAdded' and 'VertexDeleted' events. This will prevent the user from interactively adding or removing vertices for the polyline.

```matlab
function storePositionInUserData(src,~)

% Before a vertex is added/removed, store the Position in the UserData
% property
src.UserData = src.Position;

end

function recallPositionInUserData(src,~)

% Restore the previous Position to prevent users from adding/removing
% vertices
src.Position = src.UserData;

end
```

See Also
Polyline | addlistener | drawpolyline

More About
*   "Callbacks — Programmed Response to User Action"
*   "Overview Events and Listeners"

Create Freehand ROI Editing Tool

This example shows how to create a simple tool to edit the shape of a freehand ROI using another ROI object. By default, `Freehand` ROI objects include waypoints that you can click and drag to adjust the shape of the ROI. You can also add waypoints interactively to any part of the boundary.

Another way to edit the shape of freehand ROIs, offered by many popular image manipulation programs, is an 'eraser' or 'brush' tool. This example implements one of these tools, using another ROI object to edit the freehand ROI.

Create Freehand ROI

Create a Freehand ROI that follows the shape of a segmentation mask. For more details on this process, see "Use Freehand ROIs to Refine Segmentation Masks" on page 12-44.

Read MRI data into the workspace.

```
im = dicomread('knee1.dcm');
```

Segment the MRI image and select the two largest regions of the mask.

```
segmentedLabels = imsegkmeans(im,3);
boneMask = segmentedLabels==2;
boneMask = bwareafilt(boneMask, 1);
```

Get the coordinates of the boundaries of the two segmented regions.

```
blocations = bwboundaries(boneMask,'noholes');
```

Convert the locations returned by `bwboundaries` to x,y order.

```
pos = blocations{1};
pos = fliplr(pos);
```

Display the image.

```
figure
hImage = imshow(im,[]);
```

Create a freehand ROI inside the segmented mask.

```
hf = drawfreehand('Position', pos);
```
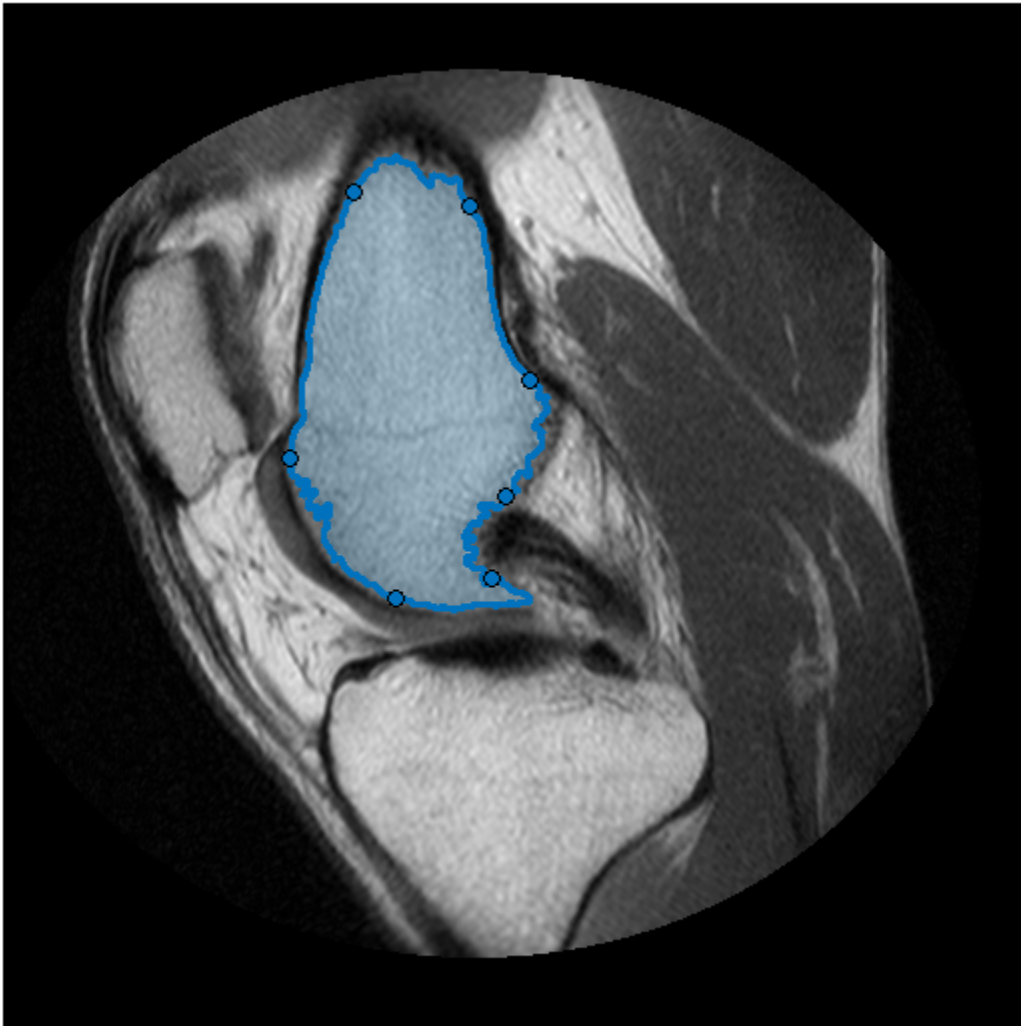
Create the Freehand ROI Editing Tool

Create a Circle ROI that will be used as the eraser or brush ROI editing tool. (You can use any of the `images.roi.*` classes by making a small change, mentioned below).

```
he = images.roi.Circle(...
    'Center', [50 50],...
    'Radius', 10,...
    'Parent', gca,...
    'Color','r');
```
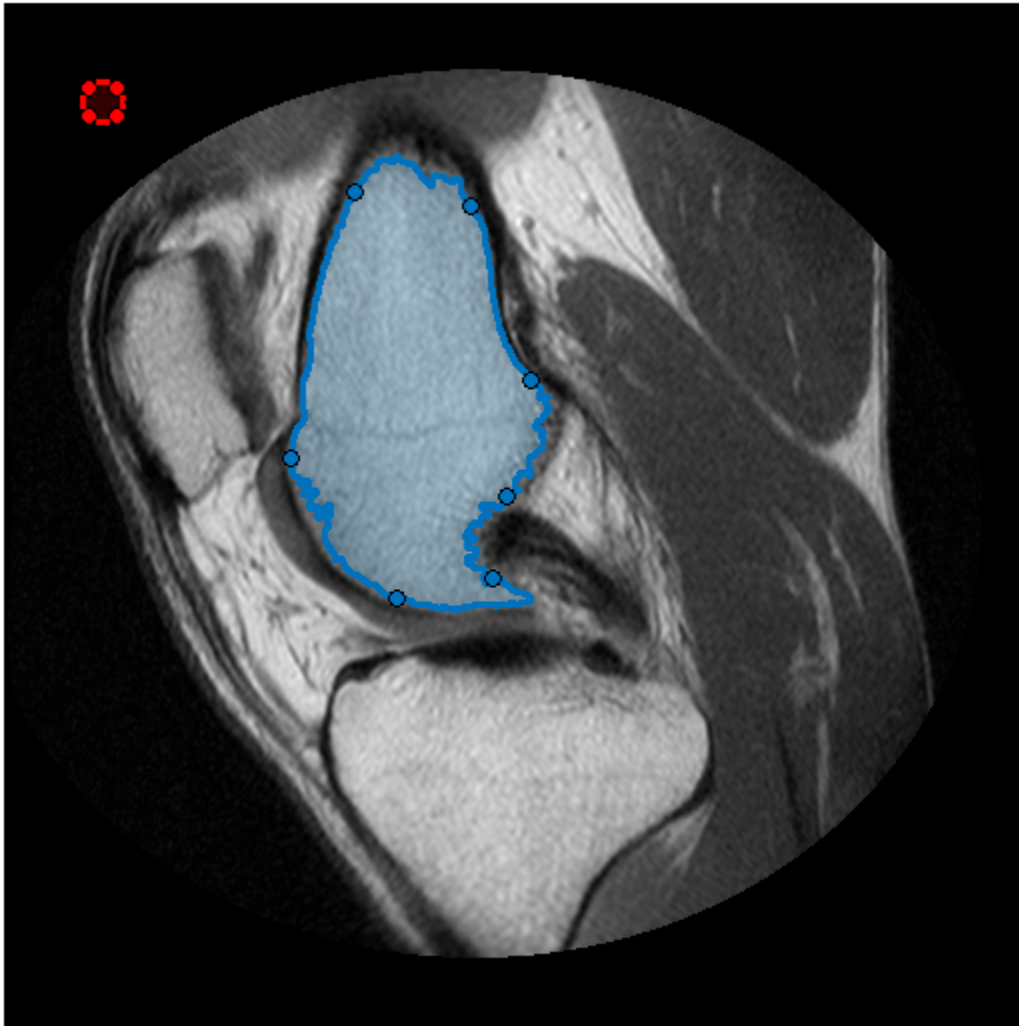
Associate two event listeners with the Circle ROI. One listens for ROI movement and the other listens for when movement stops. The ROI moving callback function , the example makes sure to have its position snap to pixel locations and also to change color (Red/Green) to indicate if the edit operation will remove or add to the target freehand ROI. Once the editor ROI stops moving, we will create corresponding binary masks for the editor ROI and the target freehand ROI and make the required edit. Finally, we'll transform the updated mask back to a freehand ROI object.Wire up a listener to react whenever this editor ROI is moved

```
addlistener(he,'MovingROI', @(varargin)editorROIMoving(he, hf));
addlistener(he,'ROIMoved', @(varargin)editFreehand(hf, he));
```

Interactively Edit the Freehand ROI

This animation shows the add and remove edit operation.
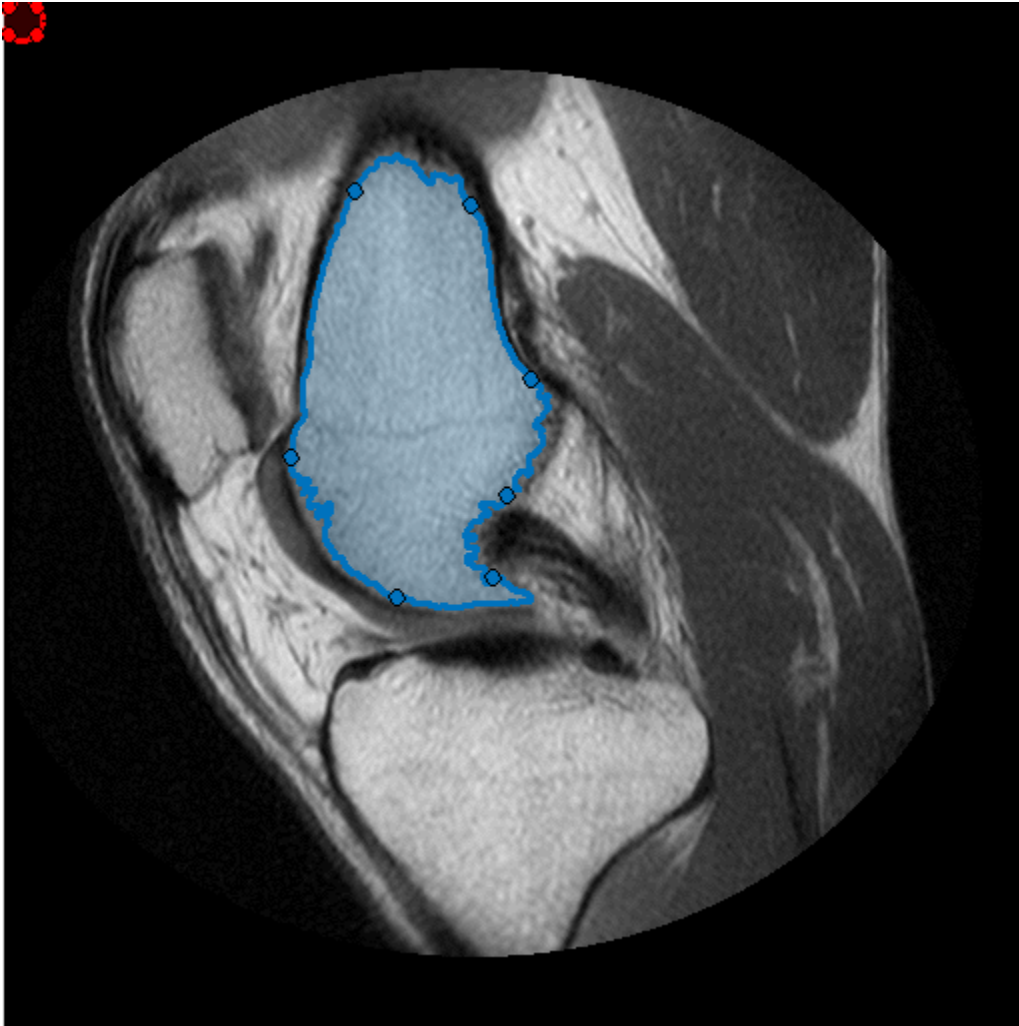
This is the ROI moving callback function. This function ensure that the editor ROI snaps to the pixel grid, and changes the color of the editor ROI to indicate if it will add to the freehand ROI or a remove a region from the freehand ROI. If the center of the editor ROI is outside the target freehand ROI, removes operation, otherwise it will 'add'.

```matlab
function editorROIMoving(he, hf)
% Snap editor ROI to grid
he.Position = round(he.Position);

% Check if the circle ROI's center is inside or outside the freehand ROI.
center = he.Center;
isAdd = hf.inROI(center(1), center(2));
if isAdd
    % Green if inside (since we will add to the freehand).
    he.Color = 'g';
else
    % Red otherwise.
    he.Color = 'r';
end
end
```

This is the edit freehand ROI callback that adds or removes the region of the editor ROI that intersects the target freehand ROI.

```matlab
function editFreehand(hf, he)

% Create a mask for the target freehand.
tmask = hf.createMask();
[m, n,~] = size(tmask);
% Include the boundary pixel locations
boundaryInd = sub2ind([m,n], hf.Position(:,2), hf.Position(:,1));
tmask(boundaryInd) = true;

% Create a mask from the editor ROI
emask = he.createMask();
boundaryInd = sub2ind([m,n], he.Position(:,2), he.Position(:,1));
emask(boundaryInd) = true;

% Check if center of the editor ROI is inside the target freehand. If you
% use a different editor ROI, ensure to update center computation.
center = he.Center; %
isAdd = hf.inROI(center(1), center(2));
if isAdd
    % Add the editor mask to the freehand mask
    newMask = tmask|emask;
else
    % Delete out the part of the freehand which intersects the editor
    newMask = tmask&~emask;
end

% Update the freehand ROI
perimPos = bwboundaries(newMask, 'noholes');
hf.Position = [perimPos{1}(:,2), perimPos{1}(:,1)];

end
```

See Also

Circle | Freehand | addlistener | bwareafilt | bwboundaries | createMask | dicomread | drawfreehand | imsegkmeans | inROI

More About

- "Callbacks — Programmed Response to User Action"
- "Overview Events and Listeners"

Use Wait Function After Drawing ROI

This example shows how to define a custom wait function that blocks the MATLAB® command line until you finish positioning a rectangle.

Display an image.

```
imshow('pears.png')
```

Draw a rectangle ROI in the top left corner of the image.

```
h = drawrectangle('Position',[1 1 100 100]);
```


Use a custom wait function to block the MATLAB command line while you interact with the rectangle. This example specifies a function called `customWait` (defined at the end of the example).

While the command line is blocked, resize and reposition the rectangle so that it encompasses one pear. Double-click on the rectangle to resume execution of the `customWait` function. The function returns the final position of the rectangle.

```
pos = customWait(h)
```
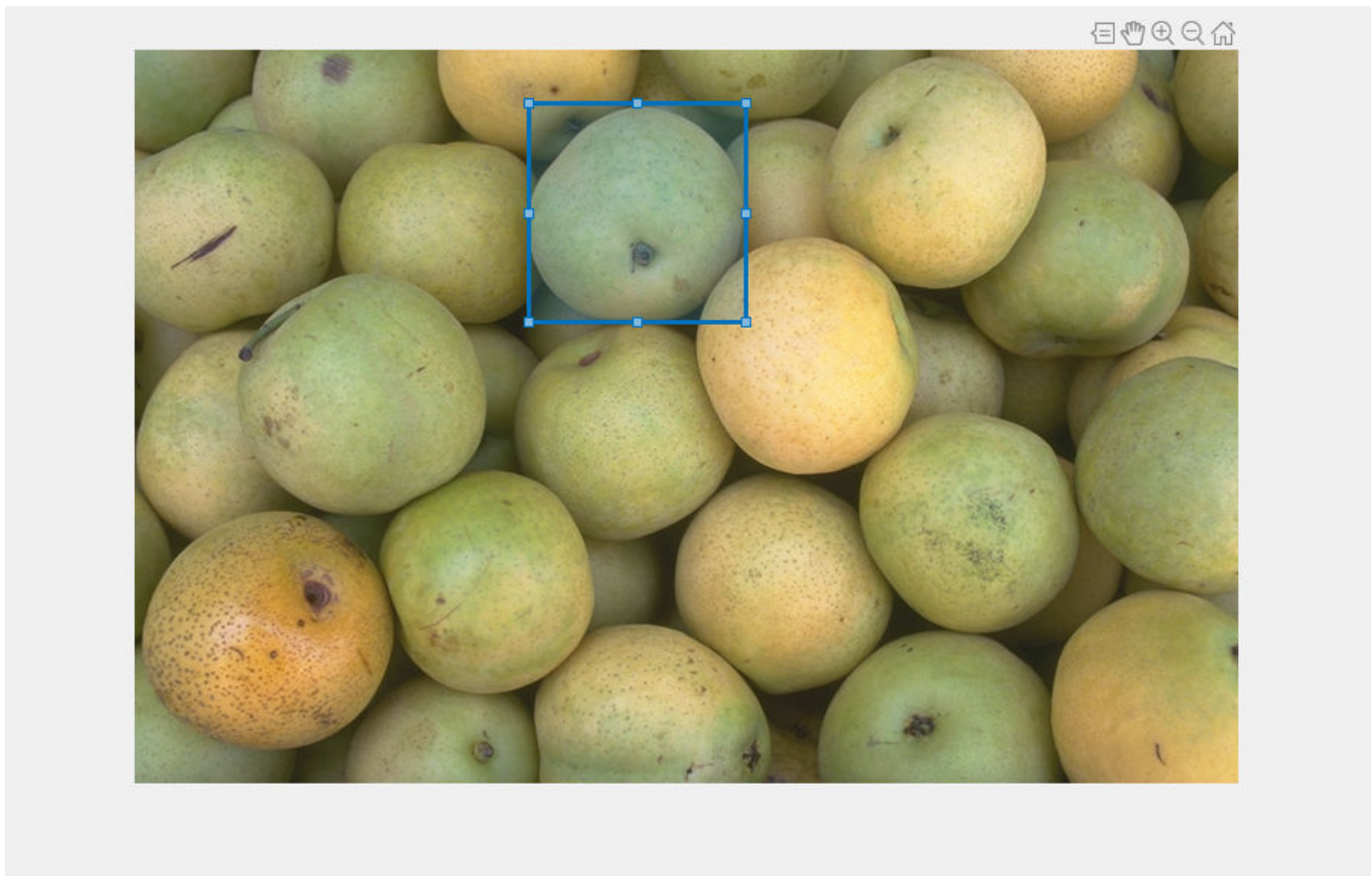
```
pos = 1×4

  262.0000   36.0000   144.0000   145.0000
```

This is the custom wait function that blocks the program execution when you click an ROI. When you have finished interacting with the ROI, the function returns the position of the ROI.

```matlab
function pos = customWait(hROI)

% Listen for mouse clicks on the ROI
l = addlistener(hROI,'ROIClicked',@clickCallback);

% Block program execution
uiwait;

% Remove listener
delete(l);

% Return the current position
pos = hROI.Position;

end
```

This click callback function resumes program execution when you double-click the ROI. Note that event data is passed to the callback function as an `images.roi.ROIClickedEventData` object,

which enables you to define callback functions that respond to different types of actions. For example, you could define a callback function to resume program execution when you click the ROI while pressing the Shift key or when you click a specific part of the ROI such as the label.

```matlab
function clickCallback(~,evt)

if strcmp(evt.SelectionType,'double')
    uiresume;
end

end
```

See Also
Rectangle | addlistener | drawrectangle | uiresume | uiwait

More About
*   "Callbacks — Programmed Response to User Action"
*   "Overview Events and Listeners"

Interactive Image Inpainting Using Exemplar Matching

This example shows how to interactively select image regions and the, inpaint the selected region by using exemplar based-matching method. Interactive inpainting allows you to select a region multiple times and perform inpainting iteratively to achieve the desired results.

In this example, you perform region filling and object removal by:

- Interactively selecting the inpainting region.
- Dynamically updating the parameter values.
- Visualizing the results dynamically.

Read Image into the Workspace

Read an image to inpaint into the workspace. The image has missing image regions to be filled through inpainting.

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

Create an Interactive Figure Window

Create an interactive figure window to display the image to be inpainted. In the window, you can select the region of interest (ROI), and dynamically update the parameter values.

```
h = figure('Name','Interactive Image Inpainting','Position',[0,0,700,400]);

% Create a panel in the current figure to interactively set the parameter
% values.
dataPanel = uipanel(h,'Position',[0.01 0.5 0.25 0.5],'Title','Set parameter values','FontSize',10

% Add an user control interface for specifying the patch size.
% Set the default patch size value to 9.
uicontrol(dataPanel,'Style','text','String','Enter Patch Size','FontSize',10,'Position',[1 150 12
data.patchSize = uicontrol(dataPanel,'Style','edit','String',num2str(9),'Position',[7 130 60 20]

% Add an user control interface for selecting the fill order.
% Set the default fill order to gradient.
uicontrol(dataPanel,'Style','text','String','Select Filling Order','FontSize',10,'Position',[5 10
data.fillOrder = uicontrol(dataPanel,'Style','popupmenu','String',{'gradient','tensor'},'Position

% Create a panel in the current figure to display the image.
viewPanel = uipanel(h,'Position',[0.25 0 0.8 1],'Title','Interactive Inpainting','FontSize',10);
ax = axes(viewPanel);
```

Display the image in the interactive figure window.

```
hImage = imshow(I,'Parent',ax);
```

Select and Inpaint Image Regions Interactively

Select ROIs interactively and dynamically inpaint the selected ROIs by using the callback function `clickCallback`. Assign a function handle that references the `clickCallback` function to the `ButtonDownFcn` property of the image object.

```
hImage.ButtonDownFcn = @(hImage,eventdata)clickCallback(hImage,eventdata,data);
```
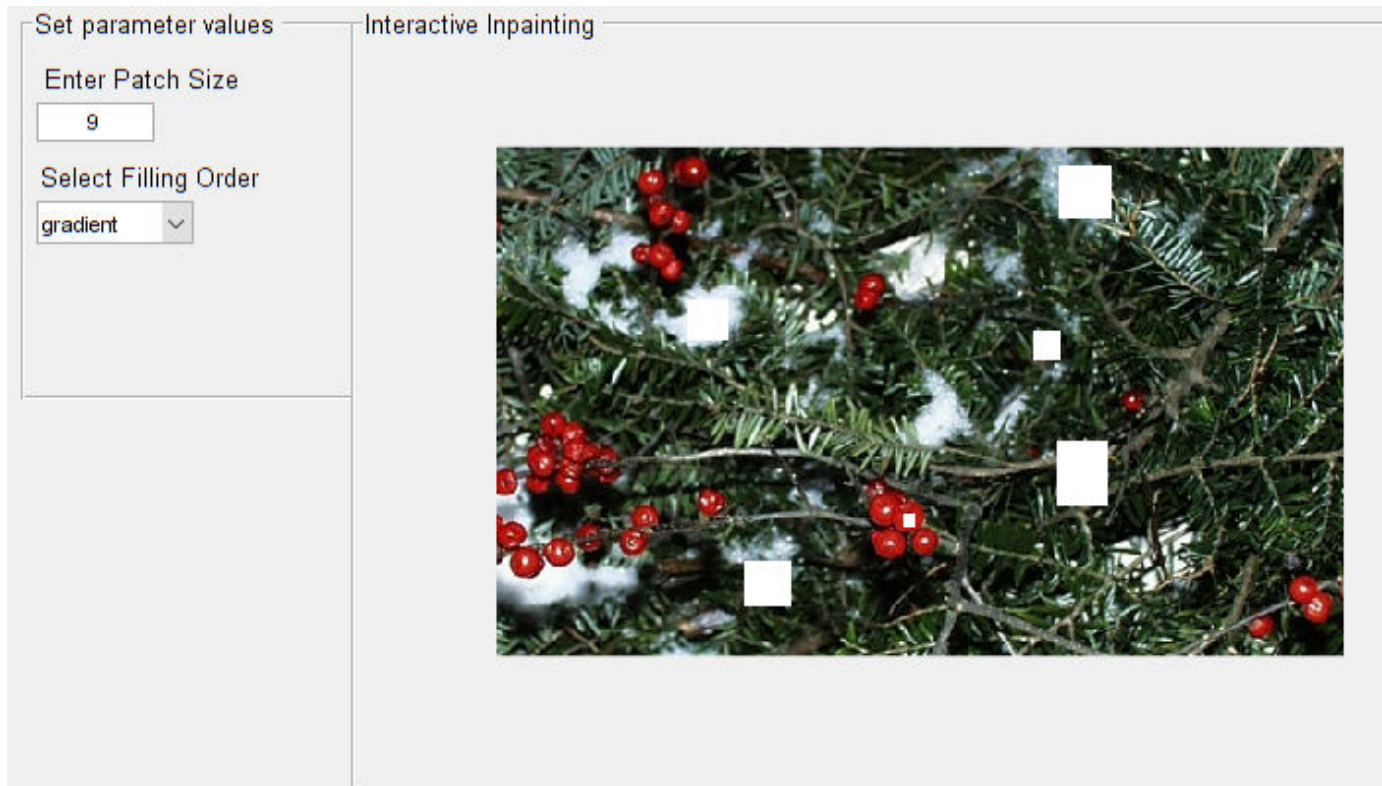
Interactively inpaint the image by following these steps.

Step 1: Choose the patch size and the fill order for inpainting. To inpaint with local parameter values, modify the patch size and the fill order to desired values by using user controls in the interactive figure window.

The choice of patch size and the fill order influences the quality of inpainting and their best value depend on the characteristics of the image region to be inpainted.

The default patch size value is set to 9.

- To inpaint regions with regular textures, choose a larger patch size and achieve seamless inpainting.
- To inpaint regions that are locally uniform with respect to a small neighborhood, choose smaller patch size.

The default fill order is set to `'gradient'`. You can choose `'gradient'` or `'tensor'` based fill order for inpainting image regions. However, `'tensor'` based fill order is more suitable for inpainting image regions with linear structures and regular textures.

Step 2: Create a freehand ROI interactively by using your mouse. Position the pointer on the axes and click and drag to draw the ROI shape. Release the pointer to close the shape.

The function dynamically updates the parameter values specified by using the user control interface and inpaints the selected ROI. Repeat steps 1 and 2, in order to inpaint all the desired regions in the image.

12-85
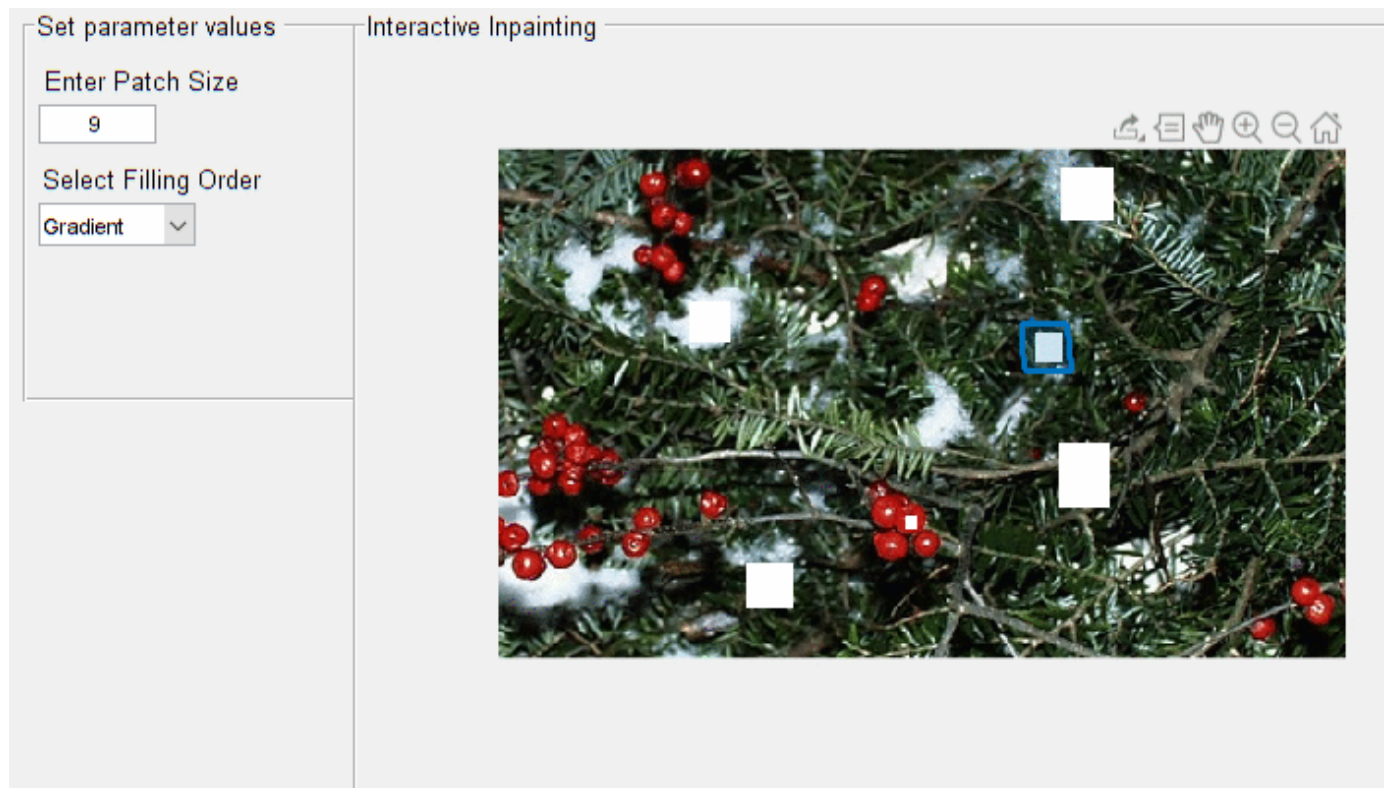
Create Callback Function to Select and Inpaint ROIs

Create the `clickCallback` to be used with the `ButtonDownFcn` to interactively select and inpaint ROIs.

```
function clickCallback(src,~,data)
% Get the parameter values for inpainting.
fillOrder = data.fillOrder.String{data.fillOrder.Value};
pSize = data.patchSize.String;
patchSize = str2double(pSize);
% Select and draw freehand ROI.
h = drawfreehand('Parent',src.Parent);
% Create a binary mask of the selected ROI.
mask = h.createMask(src.CData);
% Run exemplar-based inpainting algorithm with user given parameters.
newImage = inpaintExemplar(src.CData,mask,'PatchSize',patchSize,'FillOrder',fillOrder);
% Update input image with output.
src.CData = newImage;
% Delete ROI handle.
delete(h);
end
```

See Also
Freehand | createMask | drawfreehand | inpaintExemplar

More About

- "Callbacks — Programmed Response to User Action"
- "Overview Events and Listeners"

Image Segmentation

This topic describes a range of techniques and apps that are used to segment images.

Texture Segmentation Using Gabor Filters

This example shows how to use texture segmentation to identify regions based on their texture. The goal is to segment the dog from the bathroom floor. The segmentation is visually obvious because of the difference in texture between the regular, periodic pattern of the bathroom floor, and the regular, smooth texture of the dog's fur.

From experimentation, it is known that Gabor filters are a reasonable model of simple cells in the Mammalian vision system. Because of this, Gabor filters are thought to be a good model of how humans distinguish texture, and are therefore a useful model to use when designing algorithms to recognize texture. This example uses the basic approach described in (A. K. Jain and F. Farrokhnia, "Unsupervised Texture Segmentation Using Gabor Filters",1991) to perform texture segmentation.

Read and display input image

Read and display the input image. This example shrinks the image to make the example run more quickly.

```
A = imread('kobi.png');
A = imresize(A,0.25);
Agray = rgb2gray(A);
figure
imshow(A)
```


Design Array of Gabor Filters

Design an array of Gabor Filters which are tuned to different frequencies and orientations. The set of frequencies and orientations is designed to localize different, roughly orthogonal, subsets of

frequency and orientation information in the input image. Regularly sample orientations between [0,150] degrees in steps of 30 degrees. Sample wavelength in increasing powers of two starting from 4/sqrt(2) up to the hypotenuse length of the input image. These combinations of frequency and orientation are taken from [Jain,1991] cited in the introduction.

```
imageSize = size(A);
numRows = imageSize(1);
numCols = imageSize(2);

wavelengthMin = 4/sqrt(2);
wavelengthMax = hypot(numRows,numCols);
n = floor(log2(wavelengthMax/wavelengthMin));
wavelength = 2.^(0:(n-2)) * wavelengthMin;

deltaTheta = 45;
orientation = 0:deltaTheta:(180-deltaTheta);

g = gabor(wavelength,orientation);
```

Extract Gabor magnitude features from source image. When working with Gabor filters, it is common to work with the magnitude response of each filter. Gabor magnitude response is also sometimes referred to as "Gabor Energy". Each MxN Gabor magnitude output image in `gabormag(:,:,ind)` is the output of the corresponding Gabor filter `g(ind)`.

```
gabormag = imgaborfilt(Agray,g);
```

Post-process the Gabor Magnitude Images into Gabor Features.

To use Gabor magnitude responses as features for use in classification, some post-processing is required. This post processing includes Gaussian smoothing, adding additional spatial information to the feature set, reshaping our feature set to the form expected by the `pca` and `kmeans` functions, and normalizing the feature information to a common variance and mean.

Each Gabor magnitude image contains some local variations, even within well segmented regions of constant texture. These local variations will throw off the segmentation. We can compensate for these variations using simple Gaussian low-pass filtering to smooth the Gabor magnitude information. We choose a sigma that is matched to the Gabor filter that extracted each feature. We introduce a smoothing term K that controls how much smoothing is applied to the Gabor magnitude responses.

```
for i = 1:length(g)
    sigma = 0.5*g(i).Wavelength;
    K = 3;
    gabormag(:,:,i) = imgaussfilt(gabormag(:,:,i),K*sigma);
end
```

When constructing Gabor feature sets for classification, it is useful to add a map of spatial location information in both X and Y. This additional information allows the classifier to prefer groupings which are close together spatially.

```
X = 1:numCols;
Y = 1:numRows;
[X,Y] = meshgrid(X,Y);
featureSet = cat(3,gabormag,X);
featureSet = cat(3,featureSet,Y);
```

Reshape data into a matrix X of the form expected by the `kmeans` function. Each pixel in the image grid is a separate datapoint, and each plane in the variable `featureSet` is a separate feature. In this

example, there is a separate feature for each filter in the Gabor filter bank, plus two additional features from the spatial information that was added in the previous step. In total, there are 24 Gabor features and 2 spatial features for each pixel in the input image.

```
numPoints = numRows*numCols;
X = reshape(featureSet,numRows*numCols,[]);
```

Normalize the features to be zero mean, unit variance.

```
X = bsxfun(@minus, X, mean(X));
X = bsxfun(@rdivide,X,std(X));
```

Visualize the feature set. To get a sense of what the Gabor magnitude features look like, Principal Component Analysis can be used to move from a 26-D representation of each pixel in the input image into a 1-D intensity value for each pixel.

```
coeff = pca(X);
feature2DImage = reshape(X*coeff(:,1),numRows,numCols);
figure
imshow(feature2DImage,[])
```
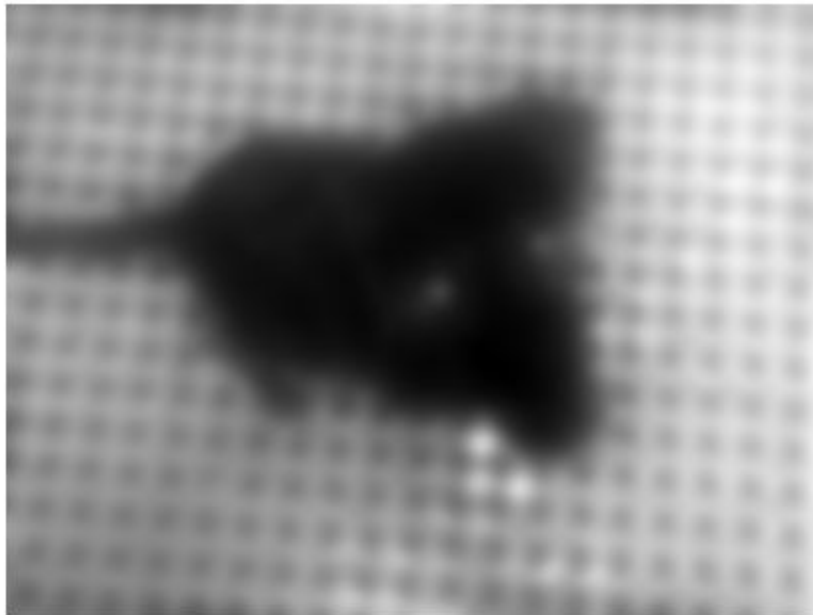

It is apparent in this visualization that there is sufficient variance in the Gabor feature information to obtain a good segmentation for this image. The dog is very dark compared to the floor because of the texture differences between the dog and the floor.

Classify Gabor Texture Features using kmeans

Repeat k-means clustering five times to avoid local minima when searching for means that minimize objective function. The only prior information assumed in this example is how many distinct regions

of texture are present in the image being segmented. There are two distinct regions in this case. This part of the example requires the Statistics and Machine Learning Toolbox™.

```
L = kmeans(X,2,'Replicates',5);
```

Visualize segmentation using `label2rgb`.

```
L = reshape(L,[numRows numCols]);
figure
imshow(label2rgb(L))
```
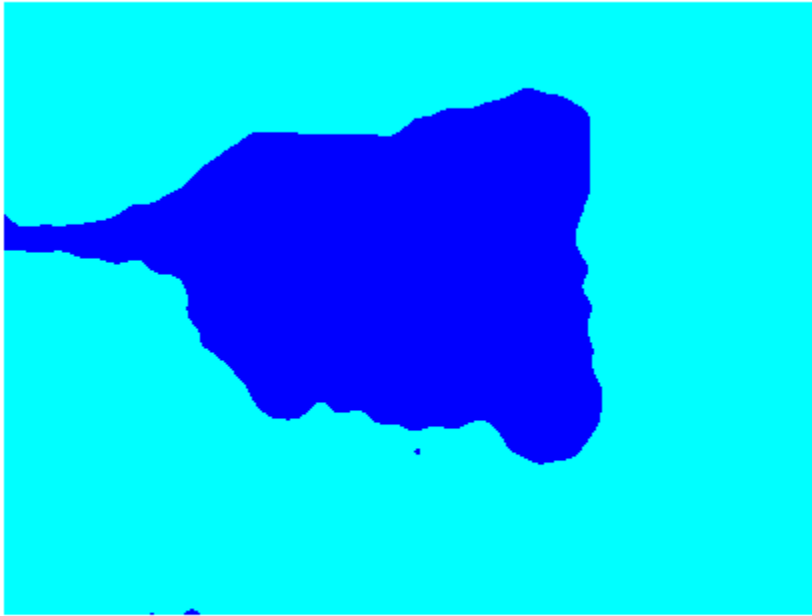

Visualize the segmented image using `imshowpair`. Examine the foreground and background images that result from the mask BW that is associated with the label matrix L.

```
Aseg1 = zeros(size(A),'like',A);
Aseg2 = zeros(size(A),'like',A);
BW = L == 2;
BW = repmat(BW,[1 1 3]);
Aseg1(BW) = A(BW);
Aseg2(~BW) = A(~BW);
figure
imshowpair(Aseg1,Aseg2,'montage');
```
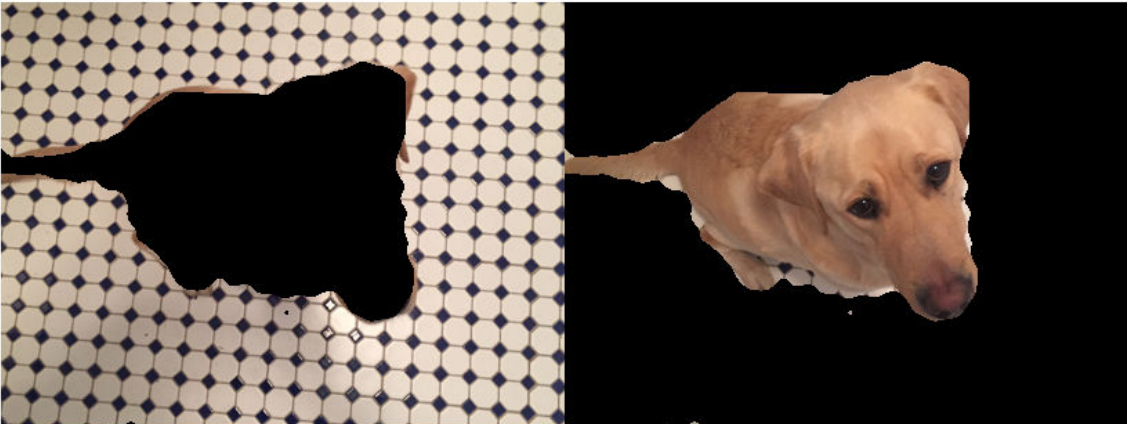
Texture Segmentation Using Texture Filters

This example shows how to identify and segment regions based on their texture.

Read Image

Read and display a grayscale image of textured patterns on a bag.

```
I = imread('bag.png');
imshow(I)
title('Original Image')
```

Original Image
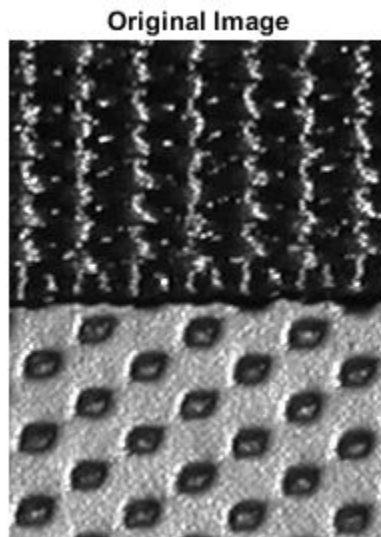
Create Texture Image

Use `entropyfilt` to create a texture image. The function `entropyfilt` returns an array where each output pixel contains the entropy value of the 9-by-9 neighborhood around the corresponding pixel in the input image `I`. Entropy is a statistical measure of randomness.

You can also use `stdfilt` and `rangefilt` to achieve similar segmentation results. For comparison to the texture image of local entropy, create texture images `S` and `R` showing the local standard deviation and local range, respectively.

```
E = entropyfilt(I);
S = stdfilt(I,ones(9));
R = rangefilt(I,ones(9));
```

Use `rescale` to rescale the texture images `E` and `S` so that pixel values are in the range [0, 1] as expected of images of data type `double`.

```
Eim = rescale(E);
Sim = rescale(S);
```

Display the three texture images in a montage.

```
montage({Eim,Sim,R},'Size',[1 3],'BackgroundColor','w',"BorderSize",20)
title('Texture Images Showing Local Entropy, Local Standard Deviation, and Local Range')
```

Texture Images Showing Local Entropy, Local Standard Deviation, and Local Range
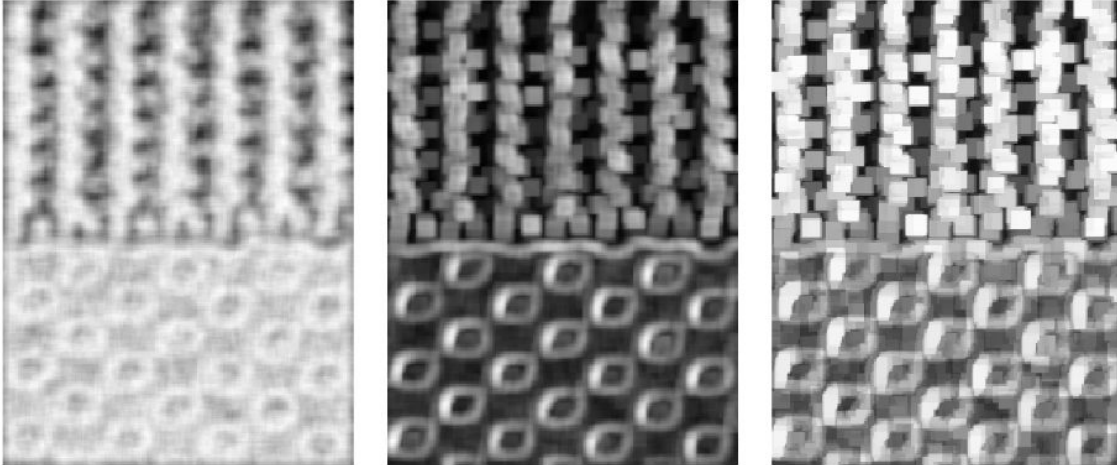
Create Mask for Bottom Texture

This example continues by processing the entropy texture image `Eim`. You can repeat a similar process for the other two types of texture images with other morphological functions to achieve similar segmentation results.

Threshold the rescaled image `Eim` to segment the textures. A threshold value of 0.8 is selected because it is roughly the intensity value of pixels along the boundary between the textures.

```
BW1 = imbinarize(Eim,0.8);
imshow(BW1)
title('Thresholded Texture Image')
```

Thresholded Texture Image

The segmented objects in the binary image BW1 are white. If you compare BW1 to I, you notice the top texture is overly segmented (multiple white objects) and the bottom texture is segmented almost in its entirety. Remove the objects in the top texture by using bwareaopen.

```
BWao = bwareaopen(BW1,2000);
imshow(BWao)
title('Area-Opened Texture Image')
```

Area-Opened Texture Image

Use `imclose` to smooth the edges and to close any open holes in the object in `BWao`. Specify the same 9-by-9 neighborhood that was used by `entropyfilt`.

```
nhood = ones(9);
closeBWao = imclose(BWao,nhood);
imshow(closeBWao)
title('Closed Texture Image')
```

Closed Texture Image

Use `imfill` to fill holes in the object in `closeBWao`. The mask for the bottom texture is not perfect because the mask does not extend to the bottom of the image. However, you can use the mask to segment the textures.

```
mask = imfill(closeBWao,'holes');
imshow(mask);
title('Mask of Bottom Texture')
```

Mask of Bottom Texture

Use Mask to Segment Textures

Separate the textures into two different images.

```
textureTop = I;
textureTop(mask) = 0;
textureBottom = I;
textureBottom(~mask) = 0;
montage({textureTop,textureBottom},'Size',[1 2],'BackgroundColor','w',"BorderSize",20)
title('Segmented Top Texture (Left) and Segmented Bottom Texture (Right)')
```

Segmented Top Texture (Left) and Segmented Bottom Texture (Right)
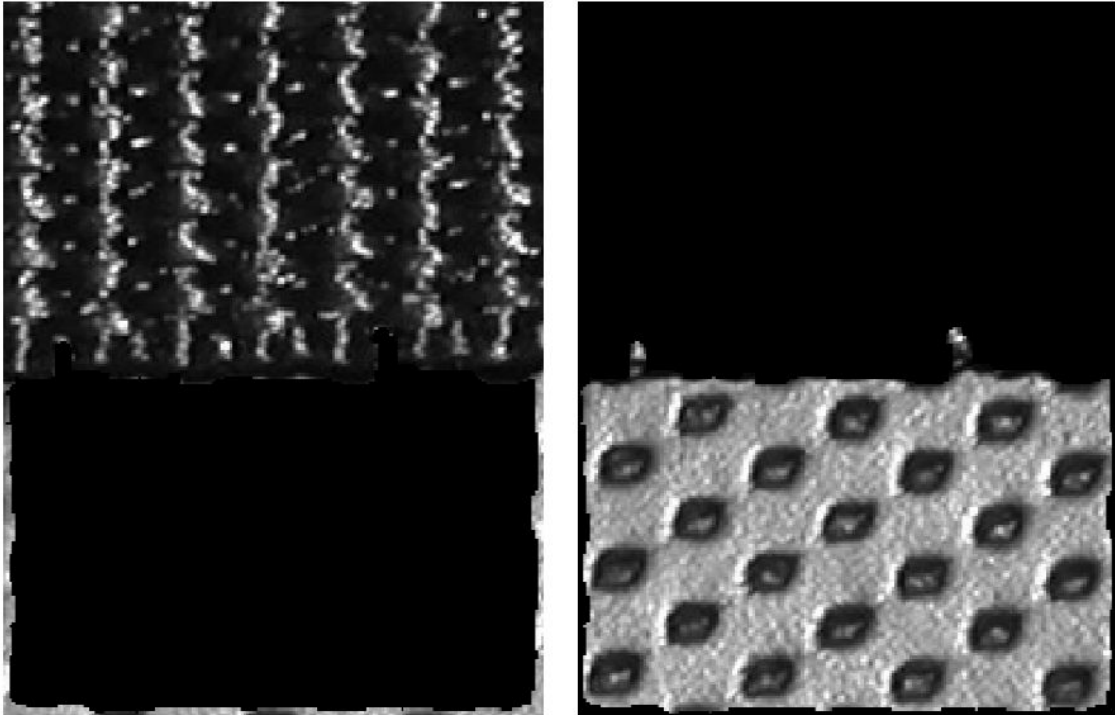
Display Segmentation Results

Create a label matrix that has the label 1 where the mask is `false` and the label 2 where the mask is `true`. Overlay label matrix on the original image.

```
L = mask+1;
imshow(labeloverlay(I,L))
title('Labeled Segmentation Regions')
```

Labeled Segmentation Regions
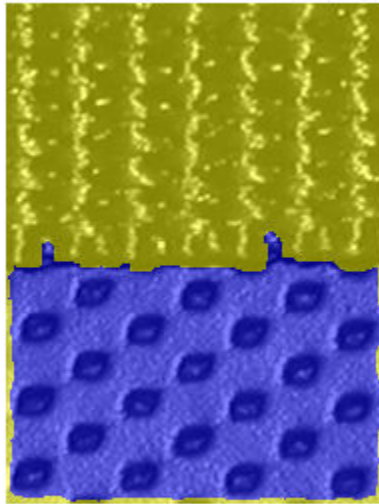
Outline the boundary between the two textures in cyan.

```
boundary = bwperim(mask);
imshow(labeloverlay(I,boundary,"Colormap",[0 1 1]))
title('Boundary Between Textures')
```

Boundary Between Textures
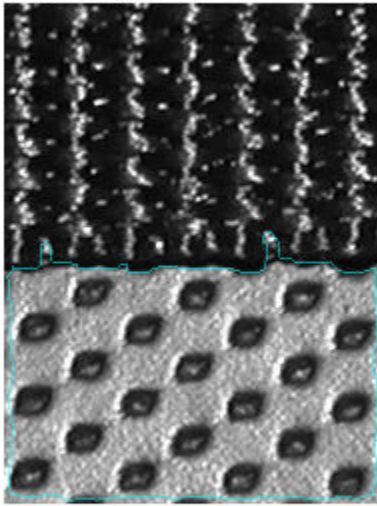
See Also
bwareaopen | bwperim | entropyfilt | imbinarize | imclose | imfill | rangefilt

Color-Based Segmentation Using the L*a*b* Color Space

This example shows how to identify different colors in fabric by analyzing the L*a*b* colorspace. The fabric image was acquired using the Image Acquisition Toolbox™.

Step 1: Acquire Image

Read in the `fabric.png` image, which is an image of colorful fabric. Instead of using `fabric.png`, you can acquire an image using the following functions in the Image Acquisition Toolbox.

```
% Access a Matrox(R) frame grabber attached to a Pulnix TMC-9700 camera, and
% acquire data using an NTSC format.
% vidobj = videoinput('matrox',1,'M_NTSC_RGB');

% Open a live preview window.  Point camera onto a piece of colorful fabric.
% preview(vidobj);

% Capture one frame of data.
% fabric = getsnapshot(vidobj);
% imwrite(fabric,'fabric.png','png');

% Delete and clear associated variables.
% delete(vidobj)
% clear vidobj;

fabric = imread('fabric.png');
imshow(fabric)
title('Fabric')
```

Fabric

Step 2: Calculate Sample Colors in L*a*b* Color Space for Each Region

You can see six major colors in the image: the background color, red, green, purple, yellow, and magenta. Notice how easily you can visually distinguish these colors from one another. The L*a*b* colorspace (also known as CIELAB or CIE L*a*b*) enables you to quantify these visual differences.

The L*a*b* color space is derived from the CIE XYZ tristimulus values. The L*a*b* space consists of a luminosity 'L*' or brightness layer, chromaticity layer 'a*' indicating where color falls along the red-green axis, and chromaticity layer 'b*' indicating where the color falls along the blue-yellow axis.

Your approach is to choose a small sample region for each color and to calculate each sample region's average color in 'a*b*' space. You will use these color markers to classify each pixel.

To simplify this example, load the region coordinates that are stored in a MAT-file.

```
load regioncoordinates;

nColors = 6;
sample_regions = false([size(fabric,1) size(fabric,2) nColors]);

for count = 1:nColors
  sample_regions(:,:,count) = roipoly(fabric,region_coordinates(:,1,count), ...
                                      region_coordinates(:,2,count));
```

```
end
```

```
imshow(sample_regions(:,:,2))
title('Sample Region for Red')
```

Sample Region for Red
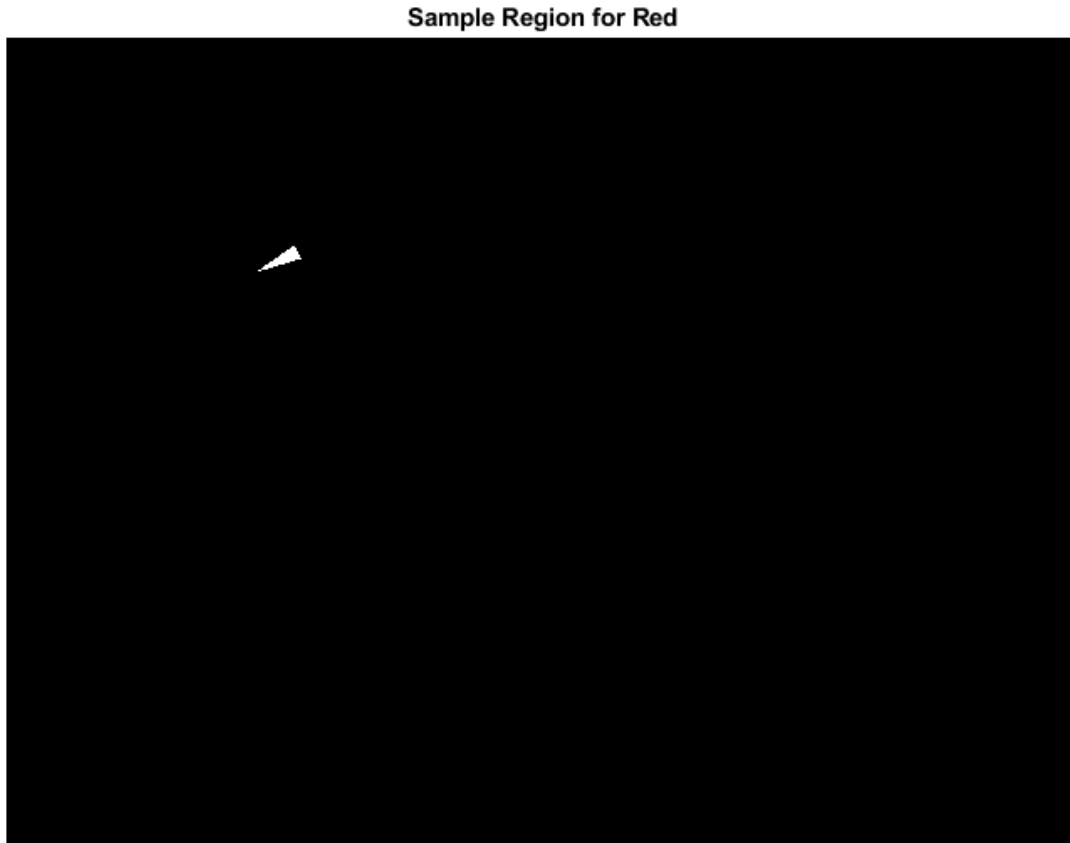
Convert your fabric RGB image into an L*a*b* image using `rgb2lab` .

```
lab_fabric = rgb2lab(fabric);
```

Calculate the mean 'a*' and 'b*' value for each area that you extracted with `roipoly`. These values serve as your color markers in 'a*b*' space.

```
a = lab_fabric(:,:,2);
b = lab_fabric(:,:,3);
color_markers = zeros([nColors, 2]);

for count = 1:nColors
  color_markers(count,1) = mean2(a(sample_regions(:,:,count)));
  color_markers(count,2) = mean2(b(sample_regions(:,:,count)));
end
```

For example, the average color of the red sample region in 'a*b*' space is

```
fprintf('[%0.3f,%0.3f] \n',color_markers(2,1),color_markers(2,2));
```

13-17

```
[69.828,20.106]
```

Step 3: Classify Each Pixel Using the Nearest Neighbor Rule

Each color marker now has an 'a*' and a 'b*' value. You can classify each pixel in the `lab_fabric` image by calculating the Euclidean distance between that pixel and each color marker. The smallest distance will tell you that the pixel most closely matches that color marker. For example, if the distance between a pixel and the red color marker is the smallest, then the pixel would be labeled as a red pixel.

Create an array that contains your color labels, i.e., 0 = background, 1 = red, 2 = green, 3 = purple, 4 = magenta, and 5 = yellow.

```
color_labels = 0:nColors-1;
```

Initialize matrices to be used in the nearest neighbor classification.

```
a = double(a);
b = double(b);
distance = zeros([size(a), nColors]);
```

Perform classification

```
for count = 1:nColors
  distance(:,:,count) = ( (a - color_markers(count,1)).^2 + ...
                      (b - color_markers(count,2)).^2 ).^0.5;
end

[~,label] = min(distance,[],3);
label = color_labels(label);
clear distance;
```

Step 4: Display Results of Nearest Neighbor Classification

The label matrix contains a color label for each pixel in the fabric image. Use the label matrix to separate objects in the original fabric image by color.

```
rgb_label = repmat(label,[1 1 3]);
segmented_images = zeros([size(fabric), nColors],'uint8');

for count = 1:nColors
  color = fabric;
  color(rgb_label ~= color_labels(count)) = 0;
  segmented_images(:,:,:,count) = color;
end
```

Display the five segmented colors as a montage. Also display the background pixels in the image that are not classified as a color.

```
montage({segmented_images(:,:,:,2),segmented_images(:,:,:,3) ...
    segmented_images(:,:,:,4),segmented_images(:,:,:,5) ...
    segmented_images(:,:,:,6),segmented_images(:,:,:,1)});
title("Montage of Red, Green, Purple, Magenta, and Yellow Objects, and Background")
```

Montage of Red, Green, Purple, Magenta, and Yellow Objects, and Background
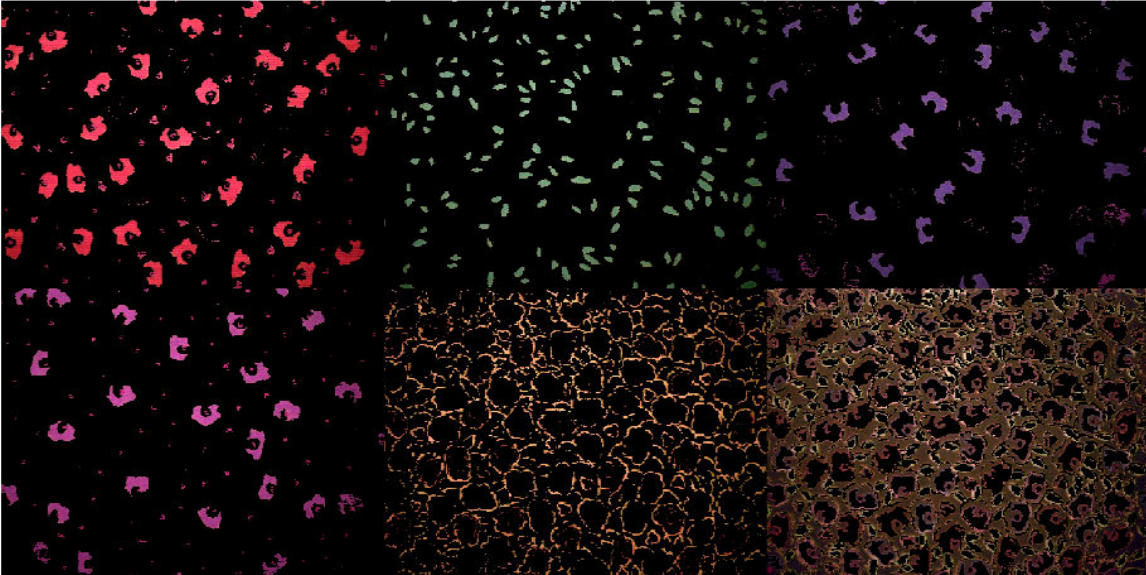
Step 5: Display 'a*' and 'b*' Values of the Labeled Colors

You can see how well the nearest neighbor classification separated the different color populations by plotting the 'a*' and 'b*' values of pixels that were classified into separate colors. For display purposes, label each point with its color label.

```
purple = [119/255 73/255 152/255];
plot_labels = {'k', 'r', 'g', purple, 'm', 'y'};

figure
for count = 1:nColors
  plot(a(label==count-1),b(label==count-1),'.','MarkerEdgeColor', ...
      plot_labels{count}, 'MarkerFaceColor', plot_labels{count});
  hold on;
end

title('Scatterplot of the segmented pixels in ''a*b*'' space');
xlabel('''a*'' values');
ylabel('''b*'' values');
```

Scatterplot of the segmented pixels in 'a*b*' space
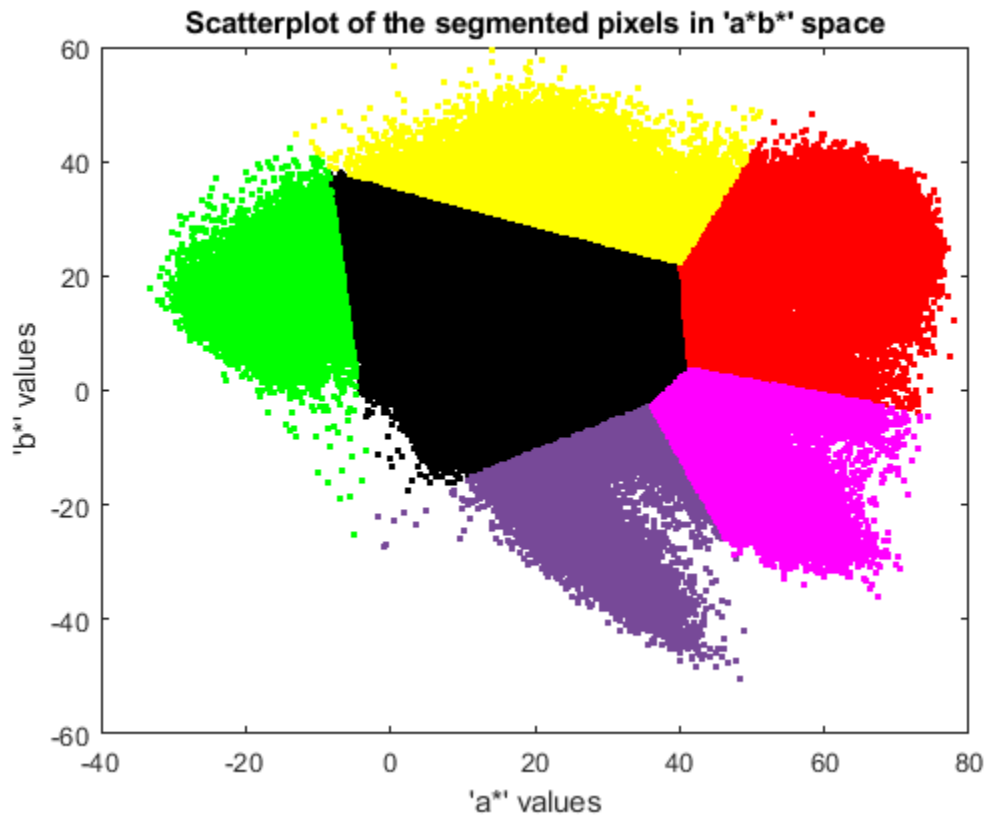
Color-Based Segmentation Using K-Means Clustering

This example shows how to segment colors in an automated fashion using the L*a*b* color space and K-means clustering.

Step 1: Read Image

Read in `hestain.png`, which is an image of tissue stained with hemotoxylin and eosin (H&E). This staining method helps pathologists distinguish different tissue types.

```
he = imread('hestain.png');
imshow(he), title('H&E image');
text(size(he,2),size(he,1)+15,...
    'Image courtesy of Alan Partin, Johns Hopkins University', ...
    'FontSize',7,'HorizontalAlignment','right');
```


H&E image
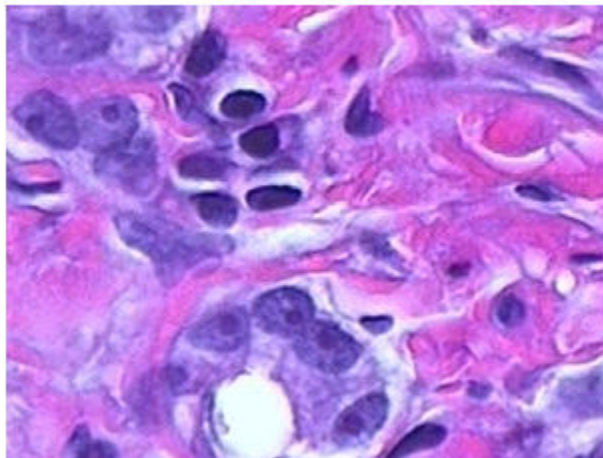
Image courtesy of Alan Partin, Johns Hopkins University

Step 2: Convert Image from RGB Color Space to L*a*b* Color Space

How many colors do you see in the image if you ignore variations in brightness? There are three colors: white, blue, and pink. Notice how easily you can visually distinguish these colors from one another. The L*a*b* color space (also known as CIELAB or CIE L*a*b*) enables you to quantify these visual differences.

The L*a*b* color space is derived from the CIE XYZ tristimulus values. The L*a*b* space consists of a luminosity layer 'L*', chromaticity-layer 'a*' indicating where color falls along the red-green axis, and chromaticity-layer 'b*' indicating where the color falls along the blue-yellow axis. All of the color information is in the 'a*' and 'b*' layers. You can measure the difference between two colors using the Euclidean distance metric.

Convert the image to L*a*b* color space using `rgb2lab`.

```
lab_he = rgb2lab(he);
```

13-21

Step 3: Classify the Colors in 'a*b*' Space Using K-Means Clustering

Clustering is a way to separate groups of objects. K-means clustering treats each object as having a location in space. It finds partitions such that objects within each cluster are as close to each other as possible, and as far from objects in other clusters as possible. K-means clustering requires that you specify the number of clusters to be partitioned and a distance metric to quantify how close two objects are to each other.

Since the color information exists in the 'a*b*' color space, your objects are pixels with 'a*' and 'b*' values. Convert the data to data type `single` for use with `imsegkmeans`. Use `imsegkmeans` to cluster the objects into three clusters.

```
ab = lab_he(:,:,2:3);
ab = im2single(ab);
nColors = 3;
% repeat the clustering 3 times to avoid local minima
pixel_labels = imsegkmeans(ab,nColors,'NumAttempts',3);
```

For every object in your input, `imsegkmeans` returns an index, or a label, corresponding to a cluster. Label every pixel in the image with its pixel label.

```
imshow(pixel_labels,[])
title('Image Labeled by Cluster Index');
```


Image Labeled by Cluster Index
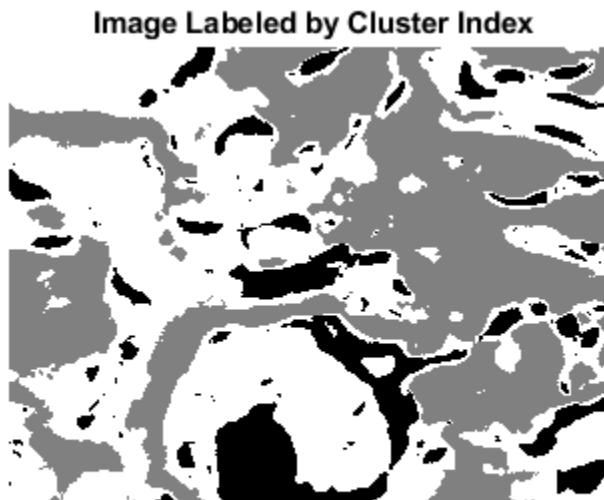
Step 4: Create Images that Segment the H&E Image by Color

Using `pixel_labels`, you can separate objects in `hestain.png` by color, which will result in three images.

```
mask1 = pixel_labels==1;
cluster1 = he .* uint8(mask1);
imshow(cluster1)
title('Objects in Cluster 1');
```

Objects in Cluster 1
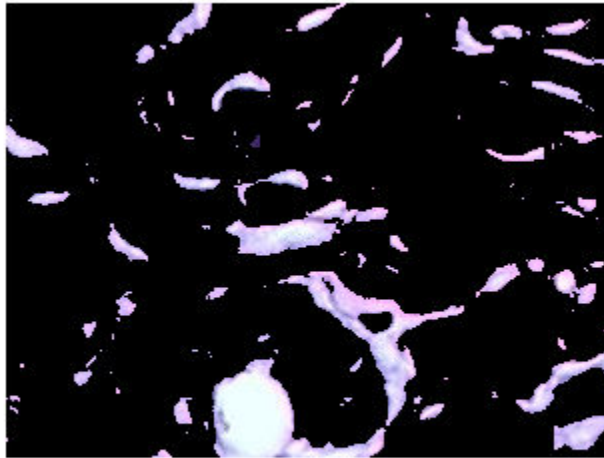
```
mask2 = pixel_labels==2;
cluster2 = he .* uint8(mask2);
imshow(cluster2)
title('Objects in Cluster 2');
```


Objects in Cluster 2
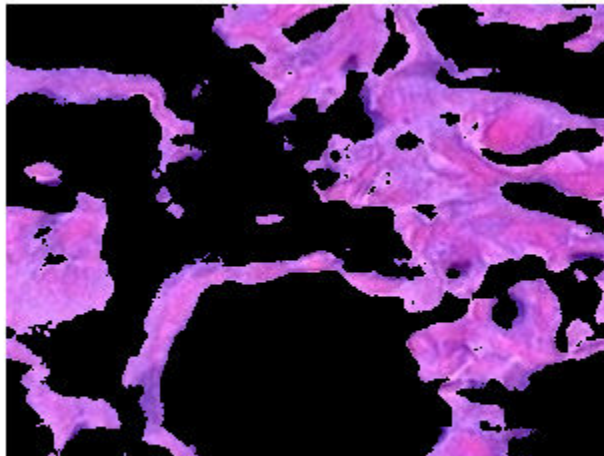
```
mask3 = pixel_labels==3;
cluster3 = he .* uint8(mask3);
imshow(cluster3)
title('Objects in Cluster 3');
```

Objects in Cluster 3
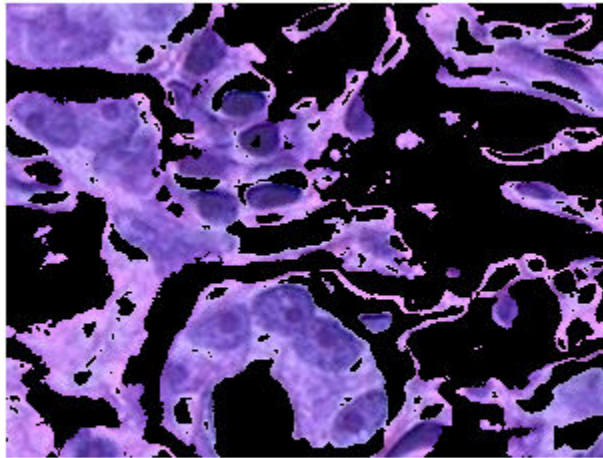
Step 5: Segment the Nuclei

Cluster 3 contains the blue objects. Notice that there are dark and light blue objects. You can separate dark blue from light blue using the 'L*' layer in the L*a*b* color space. The cell nuclei are dark blue.

Recall that the 'L*' layer contains the brightness values of each color. Extract the brightness values of the pixels in this cluster and threshold them with a global threshold using `imbinarize`. The mask `is_light_blue` gives the indices of light blue pixels.

```
L = lab_he(:,:,1);
L_blue = L .* double(mask3);
L_blue = rescale(L_blue);
idx_light_blue = imbinarize(nonzeros(L_blue));
```

Copy the mask of blue objects, `mask3`, then remove the light blue pixels from the mask. Apply the new mask to the original image and display the result. Only dark blue cell nuclei are visible.

```
blue_idx = find(mask3);
mask_dark_blue = mask3;
mask_dark_blue(blue_idx(idx_light_blue)) = 0;

blue_nuclei = he .* uint8(mask_dark_blue);
imshow(blue_nuclei)
title('Blue Nuclei');
```

Blue Nuclei
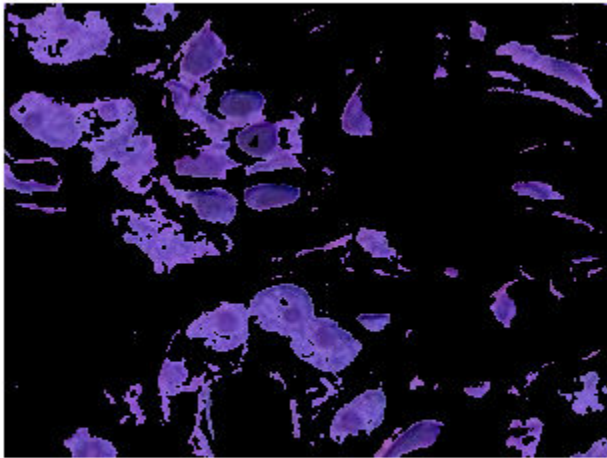
Marker-Controlled Watershed Segmentation

This example shows how to use watershed segmentation to separate touching objects in an image. The watershed transform finds "catchment basins" and "watershed ridge lines" in an image by treating it as a surface where light pixels are high and dark pixels are low.

Segmentation using the watershed transform works better if you can identify, or "mark," foreground objects and background locations. Marker-controlled watershed segmentation follows this basic procedure:

1   Compute a segmentation function. This is an image whose dark regions are the objects you are trying to segment.

2   Compute foreground markers. These are connected blobs of pixels within each of the objects.

3   Compute background markers. These are pixels that are not part of any object.

4   Modify the segmentation function so that it only has minima at the foreground and background marker locations.

5   Compute the watershed transform of the modified segmentation function.

Step 1: Read in the Color Image and Convert it to Grayscale

```
rgb = imread('pears.png');
I = rgb2gray(rgb);
imshow(I)

text(732,501,'Image courtesy of Corel(R)',...
     'FontSize',7,'HorizontalAlignment','right')
```

Image courtesy of Corel(R)

Step 2: Use the Gradient Magnitude as the Segmentation Function

Compute the gradient magnitude. The gradient is high at the borders of the objects and low (mostly) inside the objects.

```
gmag = imgradient(I);
imshow(gmag,[])
title('Gradient Magnitude')
```

Gradient Magnitude
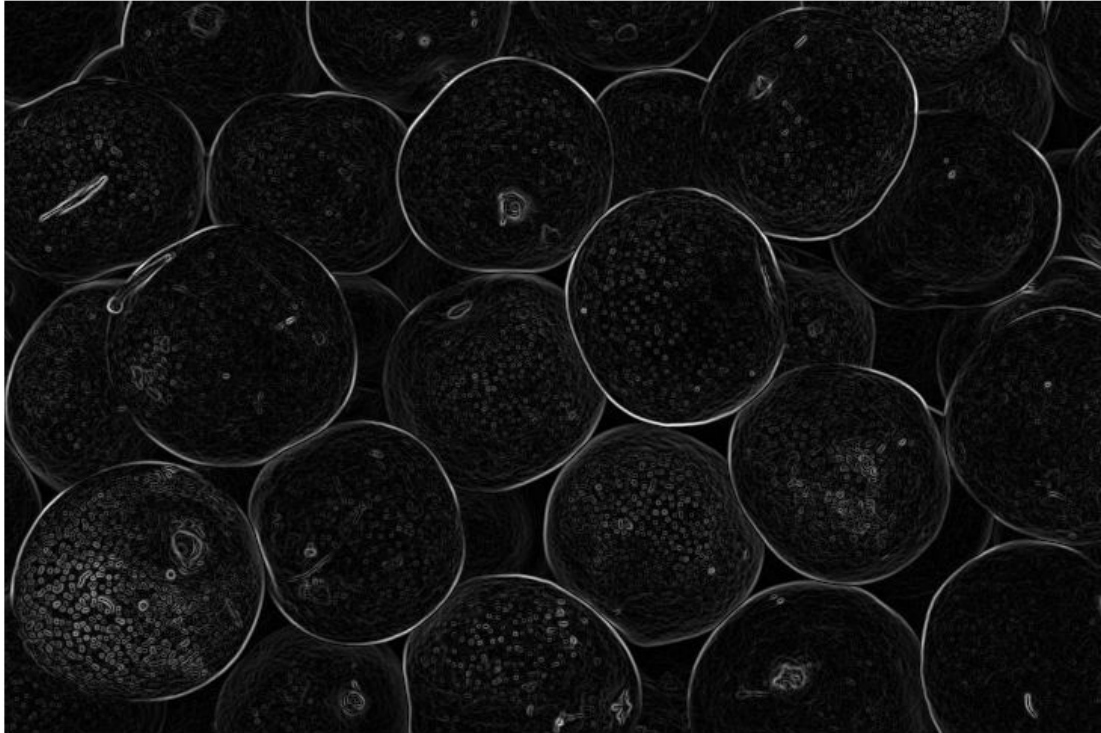
Can you segment the image by using the watershed transform directly on the gradient magnitude?

```
L = watershed(gmag);
Lrgb = label2rgb(L);
imshow(Lrgb)
title('Watershed Transform of Gradient Magnitude')
```

Watershed Transform of Gradient Magnitude
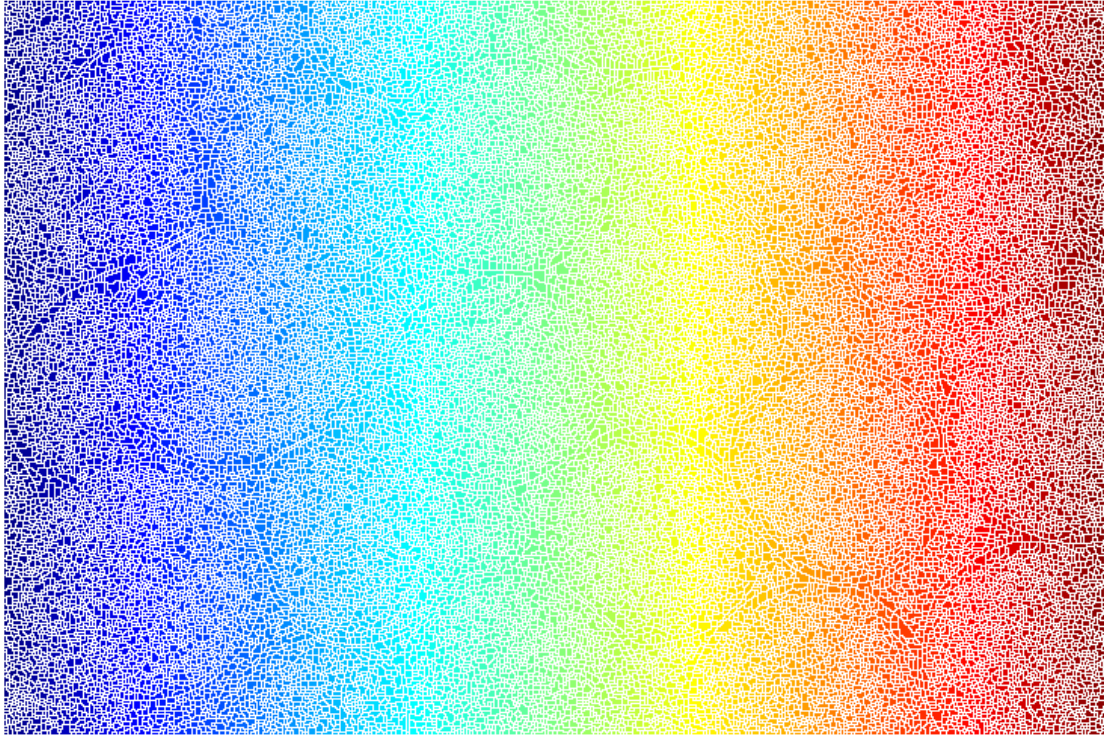
No. Without additional preprocessing such as the marker computations below, using the watershed transform directly often results in "oversegmentation."

Step 3: Mark the Foreground Objects

A variety of procedures could be applied here to find the foreground markers, which must be connected blobs of pixels inside each of the foreground objects. In this example you'll use morphological techniques called "opening-by-reconstruction" and "closing-by-reconstruction" to "clean" up the image. These operations will create flat maxima inside each object that can be located using `imregionalmax`.

Opening is an erosion followed by a dilation, while opening-by-reconstruction is an erosion followed by a morphological reconstruction. Let's compare the two. First, compute the opening using `imopen`.

```
se = strel('disk',20);
Io = imopen(I,se);
imshow(Io)
title('Opening')
```

Next compute the opening-by-reconstruction using `imerode` and `imreconstruct`.

```
Ie = imerode(I,se);
Iobr = imreconstruct(Ie,I);
imshow(Iobr)
title('Opening-by-Reconstruction')
```

Opening-by-Reconstruction
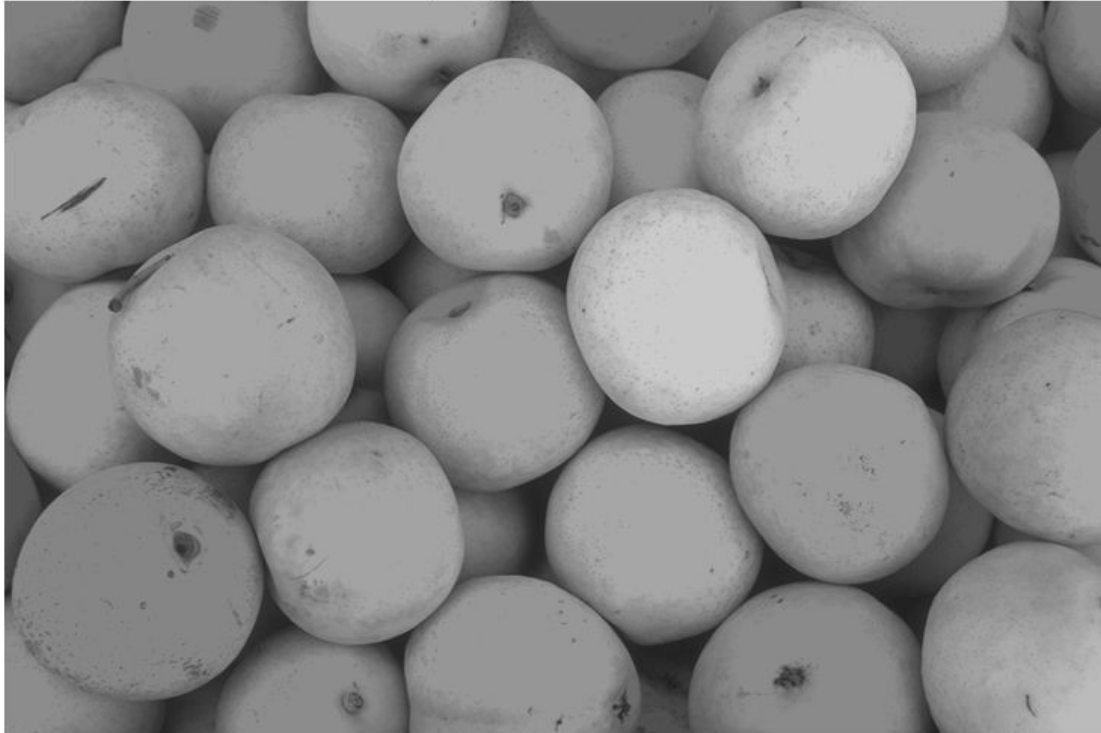
Following the opening with a closing can remove the dark spots and stem marks. Compare a regular morphological closing with a closing-by-reconstruction. First try `imclose`:

```
Ioc = imclose(Io,se);
imshow(Ioc)
title('Opening-Closing')
```

Opening-Closing
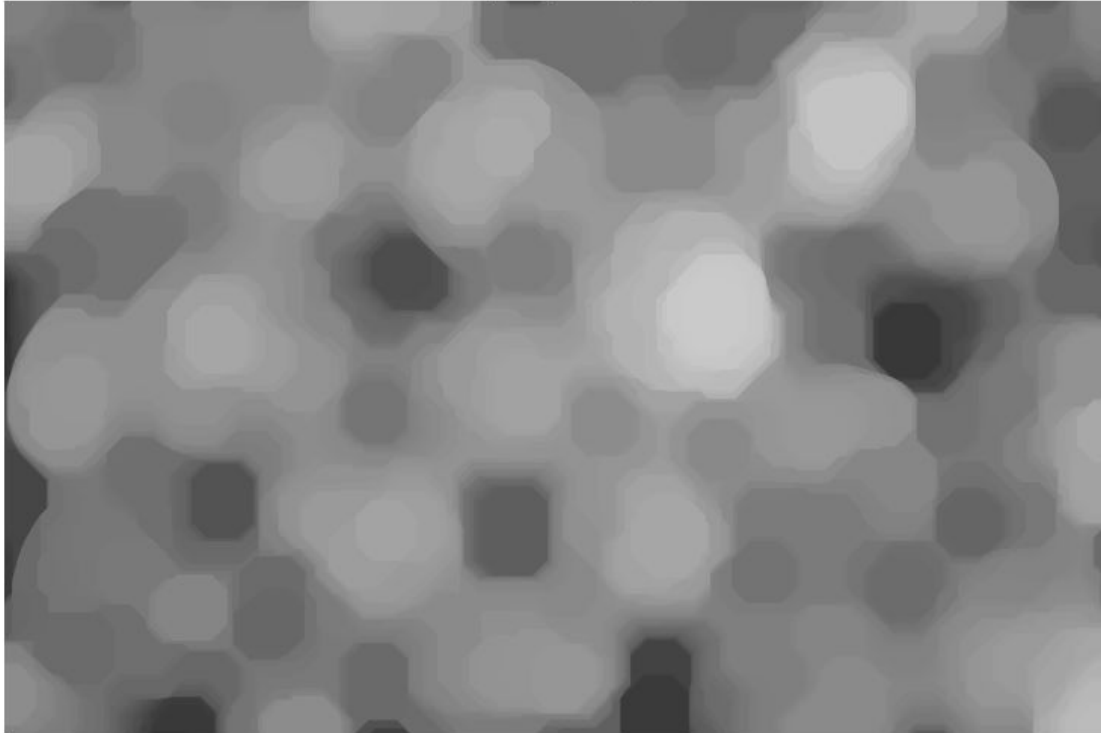
Now use `imdilate` followed by `imreconstruct`. Notice you must complement the image inputs and output of `imreconstruct`.

```
Iobrd = imdilate(Iobr,se);
Iobrcbr = imreconstruct(imcomplement(Iobrd),imcomplement(Iobr));
Iobrcbr = imcomplement(Iobrcbr);
imshow(Iobrcbr)
title('Opening-Closing by Reconstruction')
```

Opening-Closing by Reconstruction
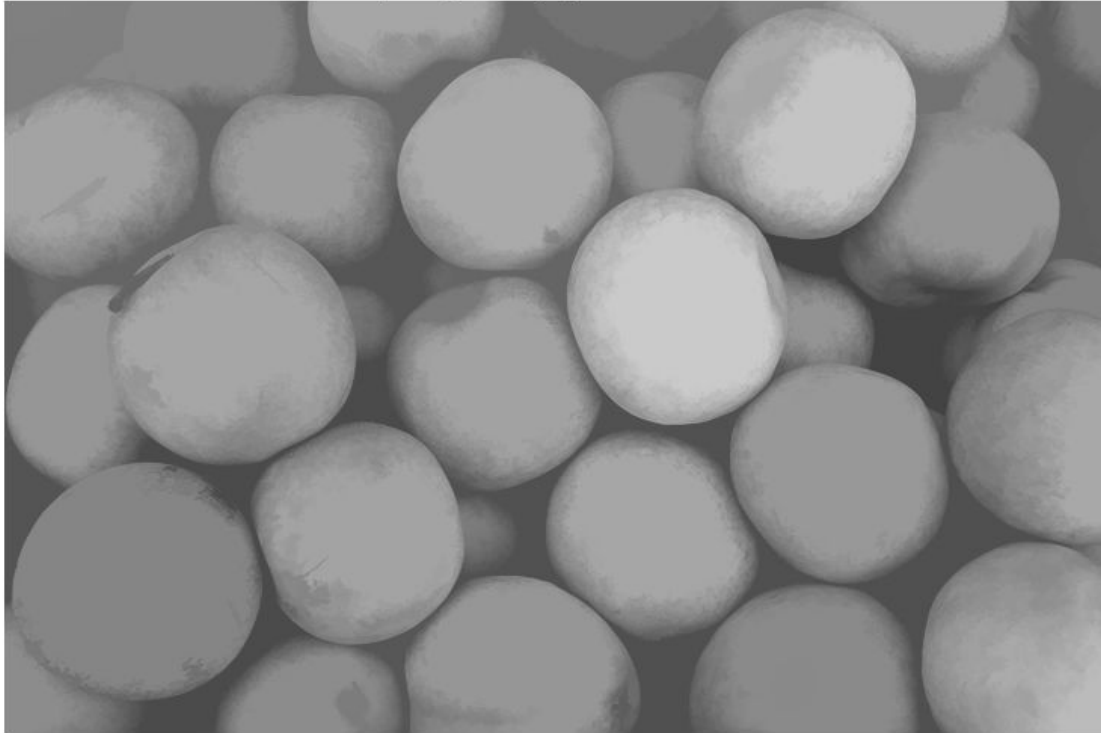
As you can see by comparing `Iobrcbr` with `Ioc`, reconstruction-based opening and closing are more effective than standard opening and closing at removing small blemishes without affecting the overall shapes of the objects. Calculate the regional maxima of `Iobrcbr` to obtain good foreground markers.

```
fgm = imregionalmax(Iobrcbr);
imshow(fgm)
title('Regional Maxima of Opening-Closing by Reconstruction')
```

Regional Maxima of Opening-Closing by Reconstruction
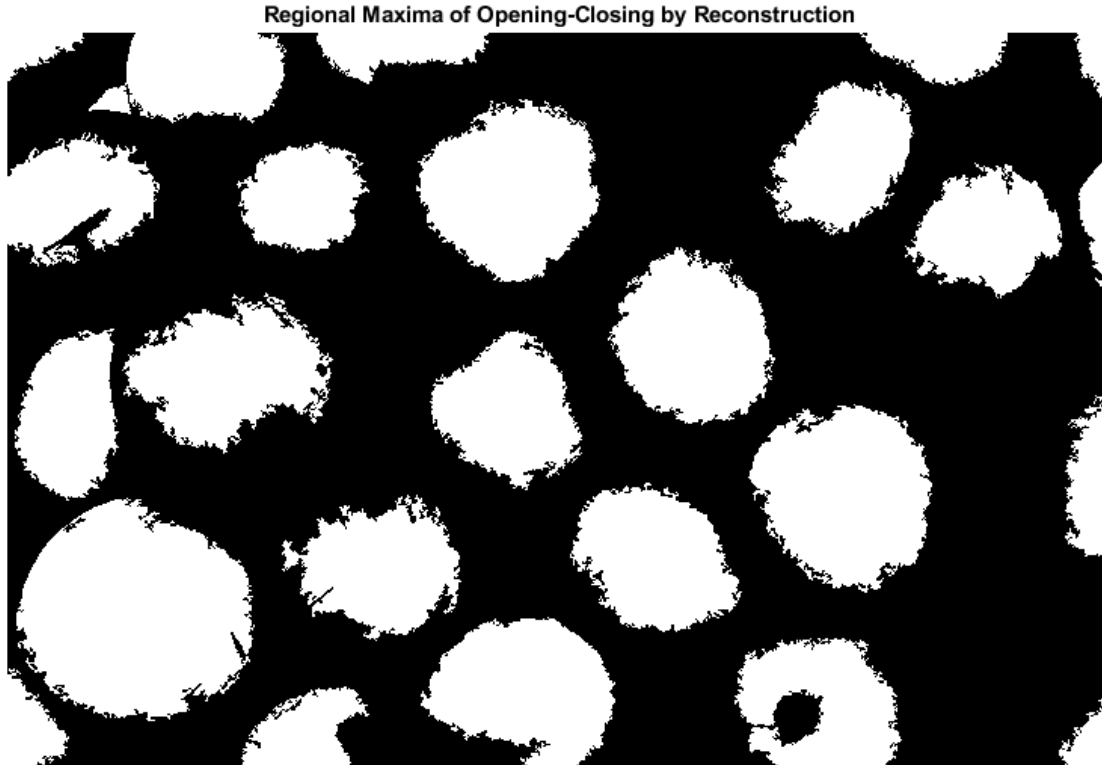
To help interpret the result, superimpose the foreground marker image on the original image.

```
I2 = labeloverlay(I,fgm);
imshow(I2)
title('Regional Maxima Superimposed on Original Image')
```

Regional Maxima Superimposed on Original Image
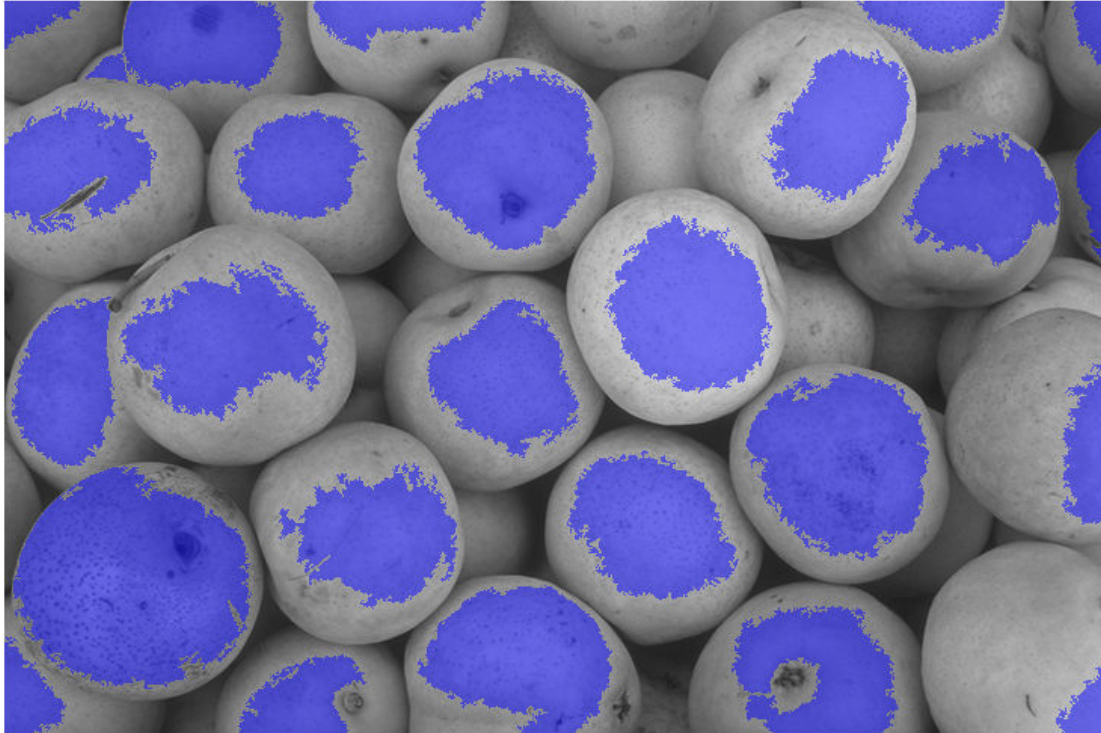
Notice that some of the mostly-occluded and shadowed objects are not marked, which means that these objects will not be segmented properly in the end result. Also, the foreground markers in some objects go right up to the objects' edge. That means you should clean the edges of the marker blobs and then shrink them a bit. You can do this by a closing followed by an erosion.

```
se2 = strel(ones(5,5));
fgm2 = imclose(fgm,se2);
fgm3 = imerode(fgm2,se2);
```

This procedure tends to leave some stray isolated pixels that must be removed. You can do this using bwareaopen, which removes all blobs that have fewer than a certain number of pixels.

```
fgm4 = bwareaopen(fgm3,20);
I3 = labeloverlay(I,fgm4);
imshow(I3)
title('Modified Regional Maxima Superimposed on Original Image')
```

Modified Regional Maxima Superimposed on Original Image
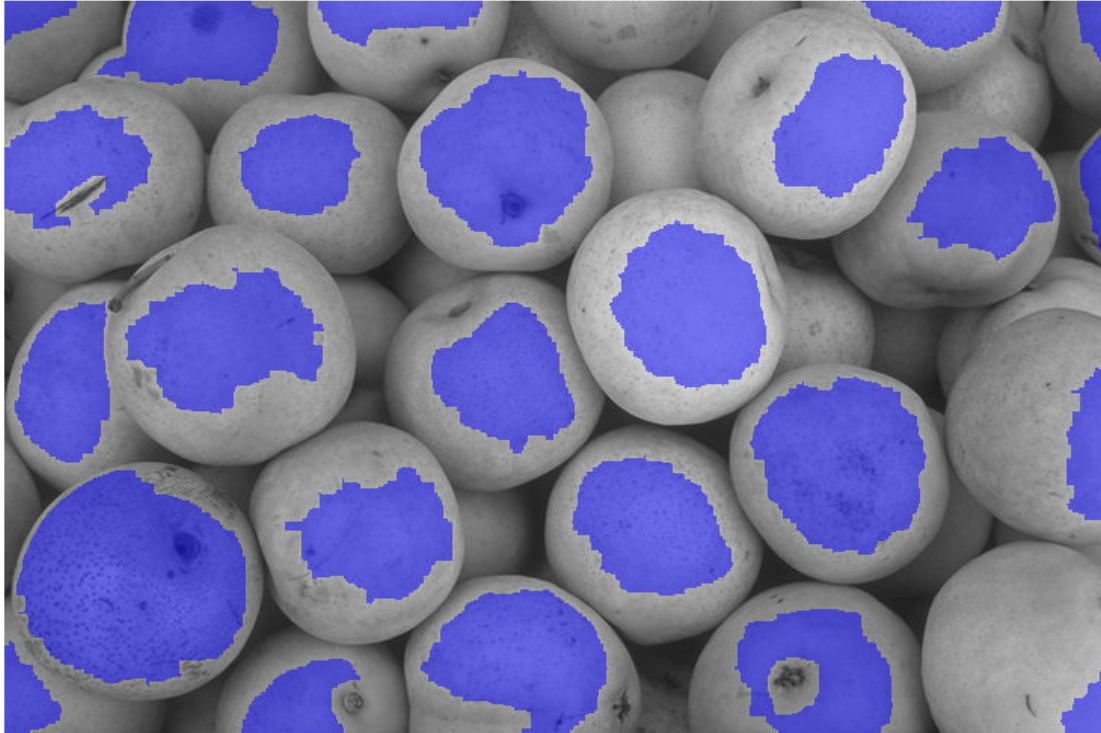
Step 4: Compute Background Markers

Now you need to mark the background. In the cleaned-up image, `Iobrcbr`, the dark pixels belong to the background, so you could start with a thresholding operation.

```
bw = imbinarize(Iobrcbr);
imshow(bw)
title('Thresholded Opening-Closing by Reconstruction')
```

Thresholded Opening-Closing by Reconstruction
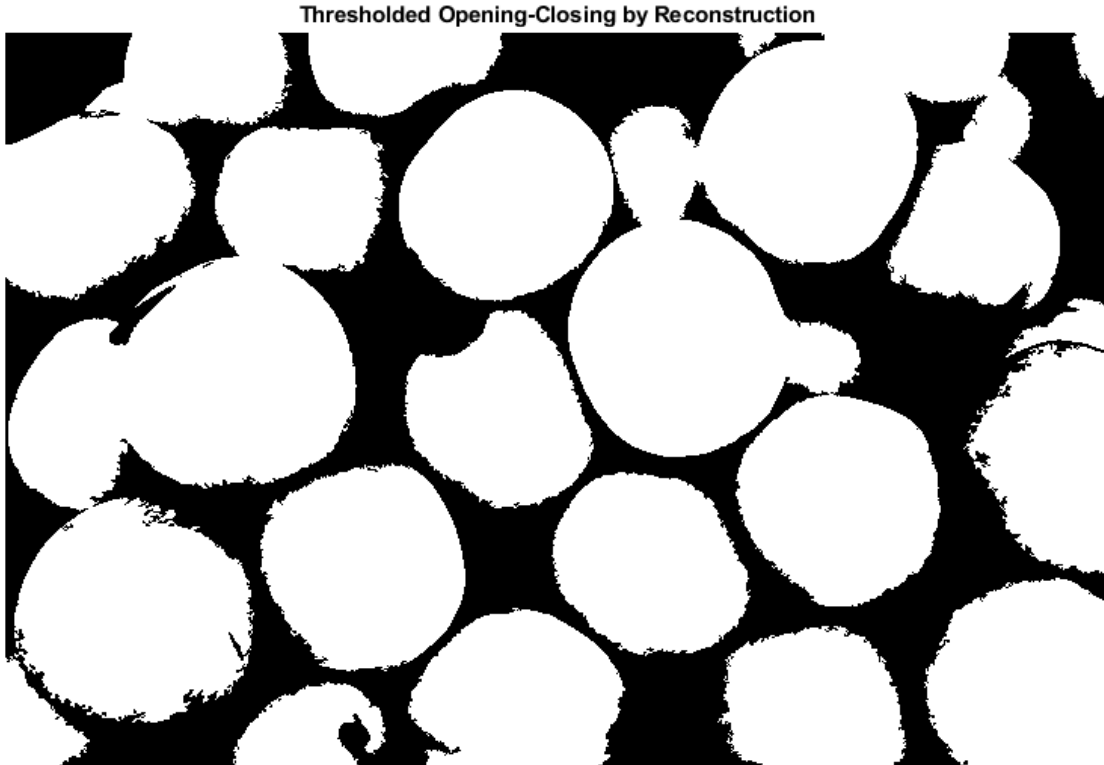
The background pixels are in black, but ideally we don't want the background markers to be too close to the edges of the objects we are trying to segment. We'll "thin" the background by computing the "skeleton by influence zones", or SKIZ, of the foreground of bw. This can be done by computing the watershed transform of the distance transform of bw, and then looking for the watershed ridge lines (DL == 0) of the result.

```
D = bwdist(bw);
DL = watershed(D);
bgm = DL == 0;
imshow(bgm)
title('Watershed Ridge Lines')
```

Watershed Ridge Lines
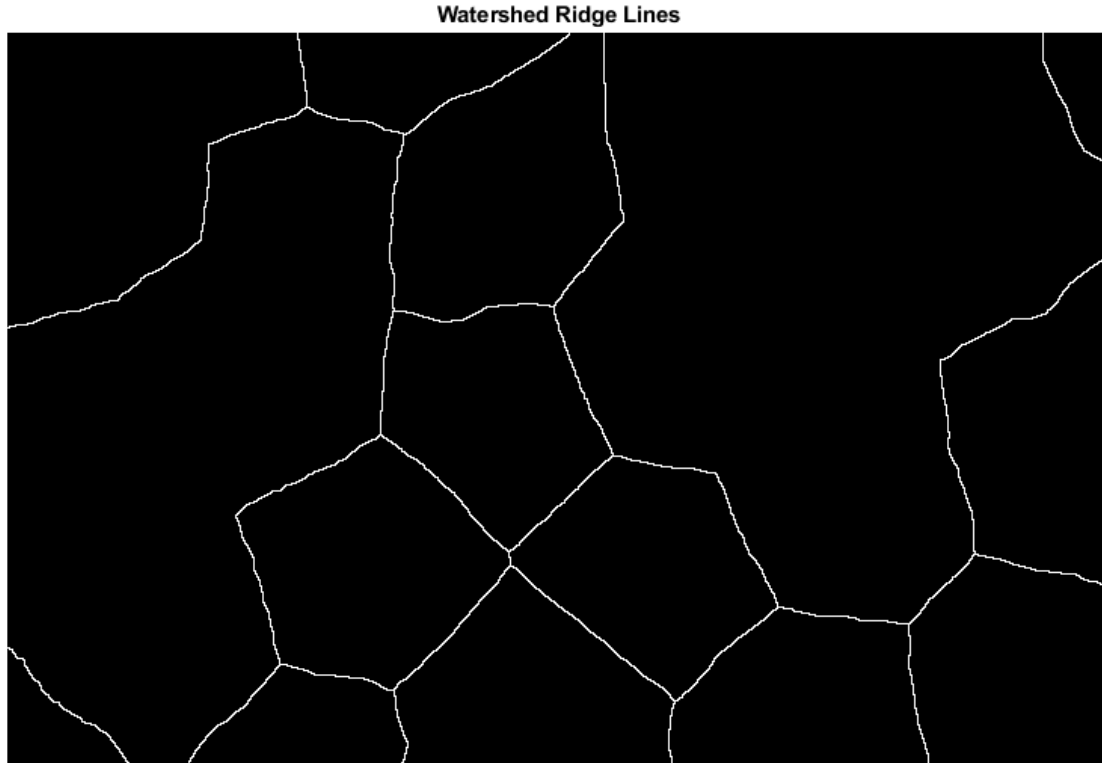
Step 5: Compute the Watershed Transform of the Segmentation Function.

The function `imimposemin` can be used to modify an image so that it has regional minima only in certain desired locations. Here you can use `imimposemin` to modify the gradient magnitude image so that its only regional minima occur at foreground and background marker pixels.

```
gmag2 = imimposemin(gmag, bgm | fgm4);
```

Finally, compute the watershed-based segmentation.

```
L = watershed(gmag2);
```

Step 6: Visualize the Result

One visualization technique is to superimpose the foreground markers, background markers, and segmented object boundaries on the original image. You can use dilation as needed to make certain aspects, such as the object boundaries, more visible. Object boundaries are located where `L == 0`. The binary foreground and background markers are scaled to different integer values so that they are assigned different labels.

```
labels = imdilate(L==0,ones(3,3)) + 2*bgm + 3*fgm4;
I4 = labeloverlay(I,labels);
imshow(I4)
title('Markers and Object Boundaries Superimposed on Original Image')
```

Markers and Object Boundaries Superimposed on Original Image
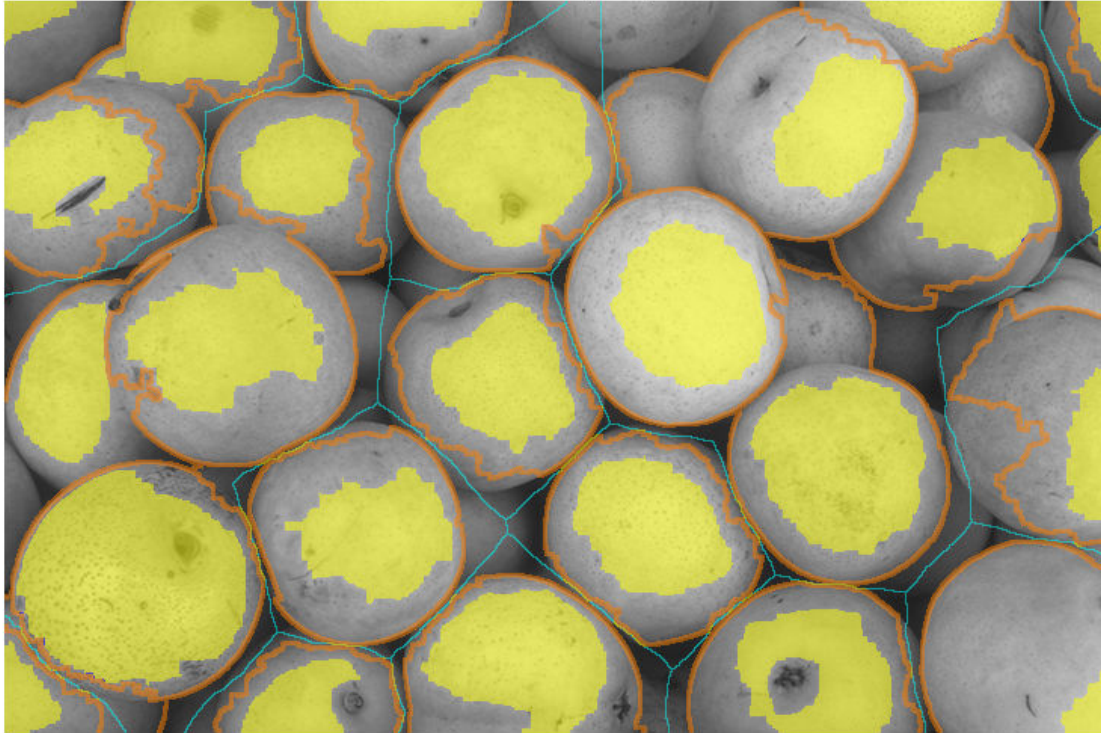
This visualization illustrates how the locations of the foreground and background markers affect the result. In a couple of locations, partially occluded darker objects were merged with their brighter neighbor objects because the occluded objects did not have foreground markers.

Another useful visualization technique is to display the label matrix as a color image. Label matrices, such as those produced by `watershed` and `bwlabel`, can be converted to truecolor images for visualization purposes by using `label2rgb`.

```
Lrgb = label2rgb(L,'jet','w','shuffle');
imshow(Lrgb)
title('Colored Watershed Label Matrix')
```

Colored Watershed Label Matrix
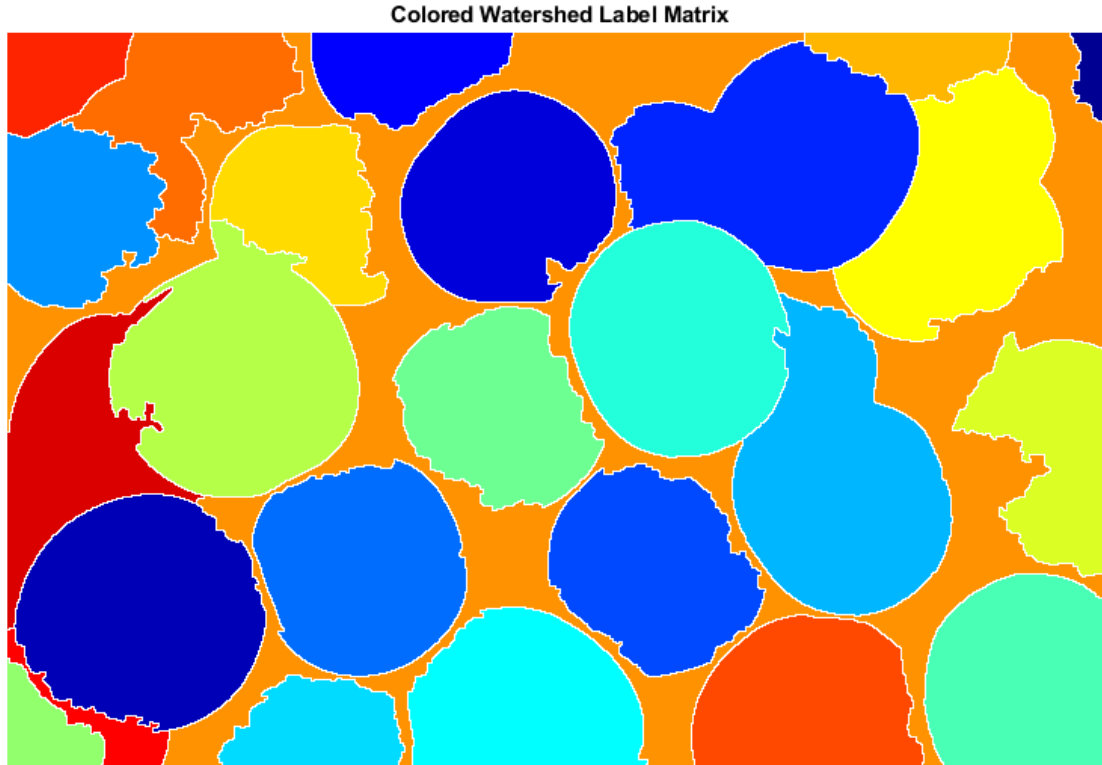
You can use transparency to superimpose this pseudo-color label matrix on top of the original intensity image.

```
figure
imshow(I)
hold on
himage = imshow(Lrgb);
himage.AlphaData = 0.3;
title('Colored Labels Superimposed Transparently on Original Image')
```

Colored Labels Superimposed Transparently on Original Image
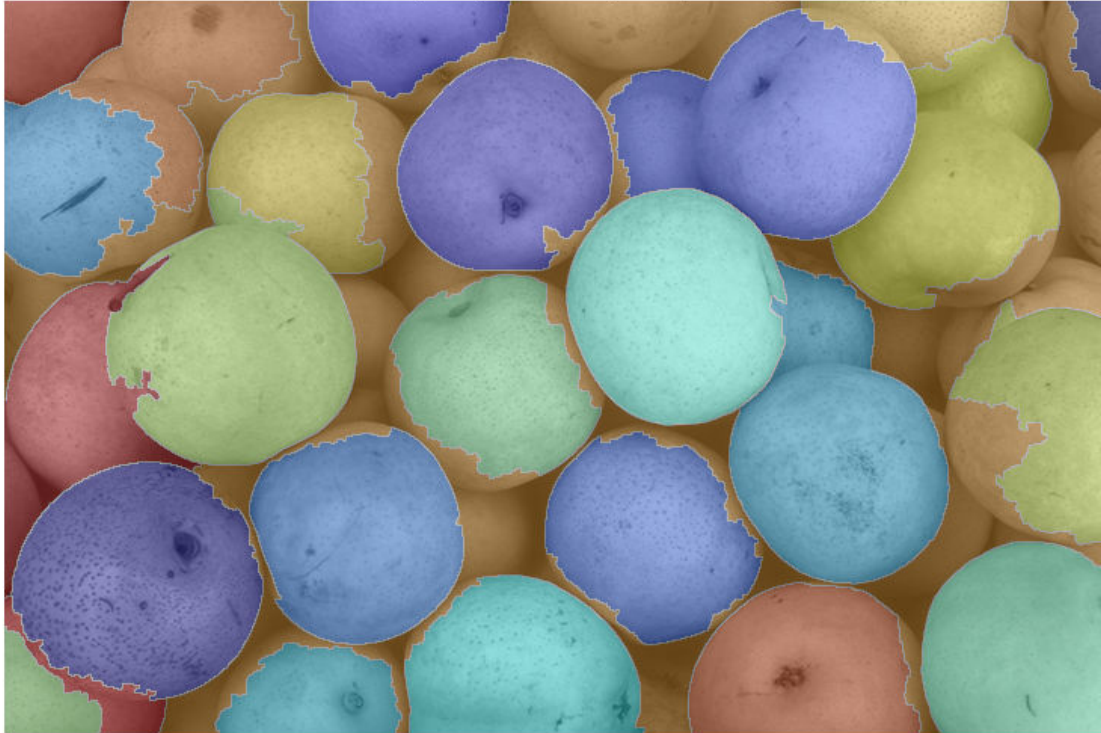
See Also

bwareaopen | bwdist | imclose | imcomplement | imdilate | imerode | imgradient | imopen | imreconstruct | imregionalmax | label2rgb | labeloverlay | watershed

Segment Image and Create Mask Using Color Thresholder App

This example shows how to segment an image and create a binary mask image using the **Color Thresholder** app. The example segments the foreground (the peppers) from the background (the purple cloth) based on color values.

In the **Color Thresholder** app, image segmentation can be an iterative process. For example, try segmenting the image in several of the color spaces supported by the app because one color space might isolate a particular color better than another. In any of the supported color spaces, you can initially perform an automatic segmentation by selecting a region in the foreground or background. Then, you can refine the segmentation by using color component controls provided by the app.

The last part of this example shows how to save the results of your work, create a mask image, and get the MATLAB® code the app used to perform the segmentation.

Open Image in Color Thresholder

Read a color image into the workspace.

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

Open the **Color Thresholder** app from the MATLAB toolstrip. On the Apps tab, in the **Image Processing and Computer Vision** section, click **Color Thresholder** .
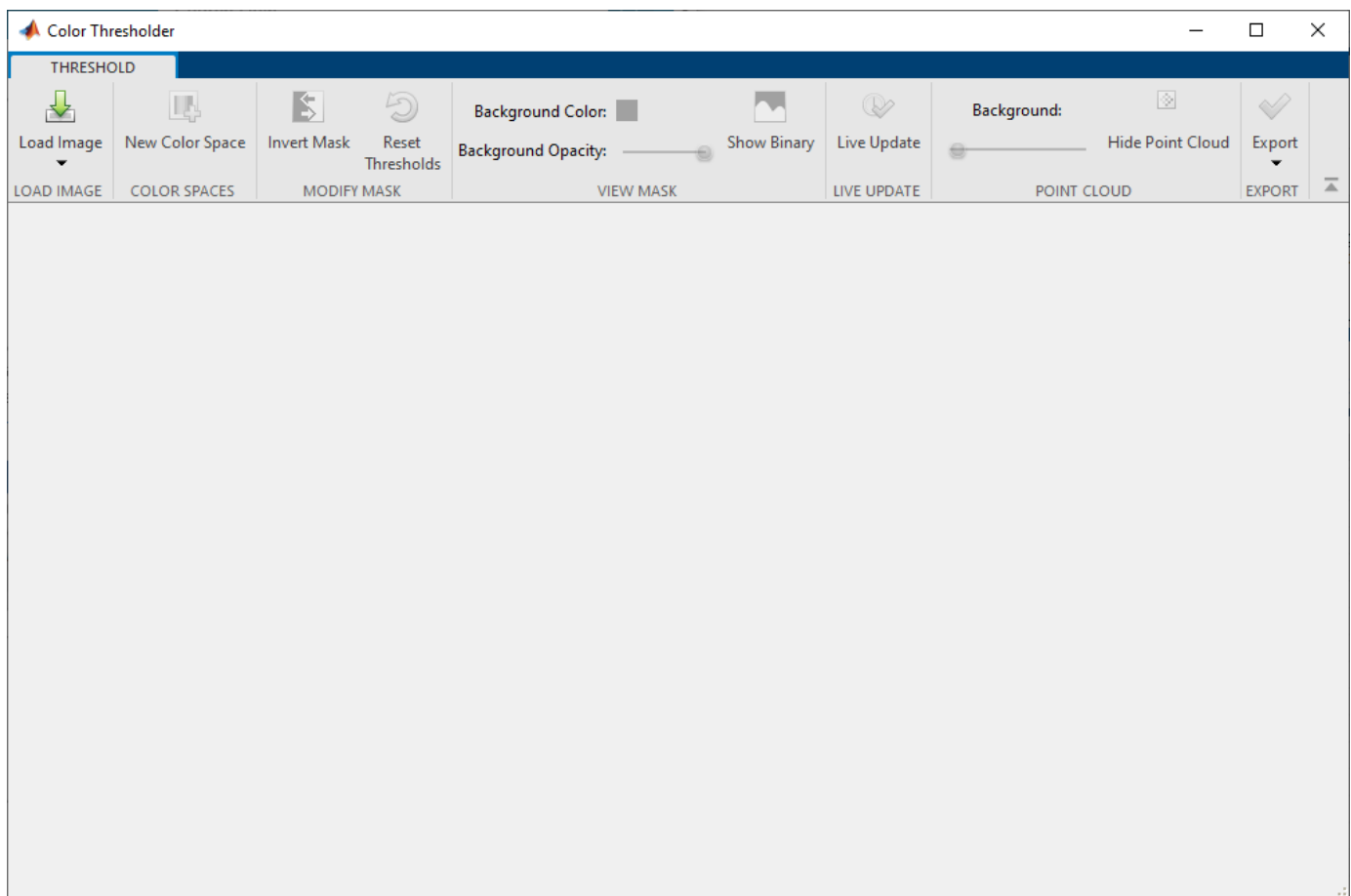
Load the image into the **Color Thresholder** app. Click **Load Image**, and then select **Load Image from Workspace.** In the Import From Workspace dialog box, select the image from the workspace, and then click **OK**.
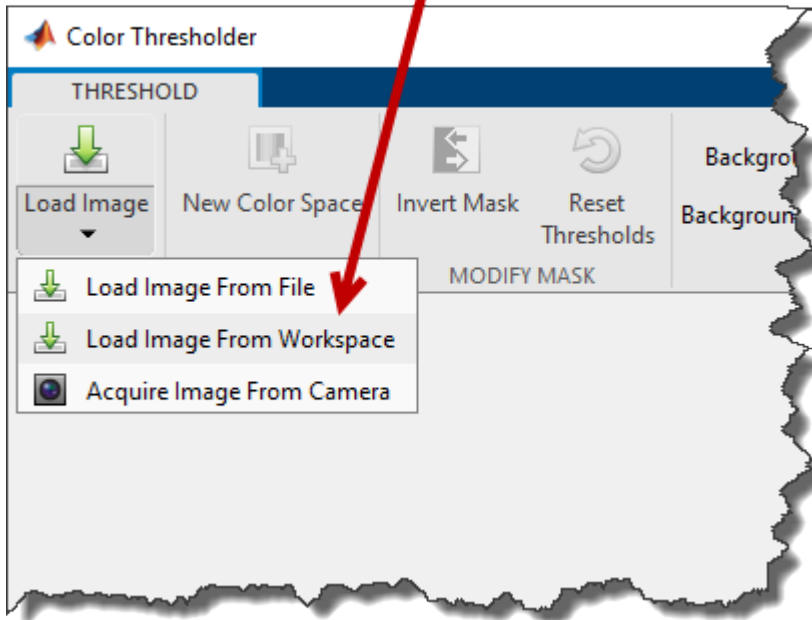
You can also open the app from the command line by using the `colorThresholder` function, specifying the name of the image: `colorThresholder(rgb);`. You can also acquire an image "Acquire Live Images in the Color Thresholder App" on page 13-55.

The **Color Thresholder** app displays the image in the **Choose a Color Space** tab, with point clouds representing the image in these color spaces: RGB, HSV, YCbCr, and *L*a*b**. For color-based segmentation, select the color space that provides the best color separation. Using the mouse, rotate the point cloud representations to see how they isolate individual colors. Segmentation using the **Color Thresholder** app can be an iterative process—try several different color spaces before you achieve a segmentation that meets your needs. For this example, start the process by selecting the YCbCr color space.
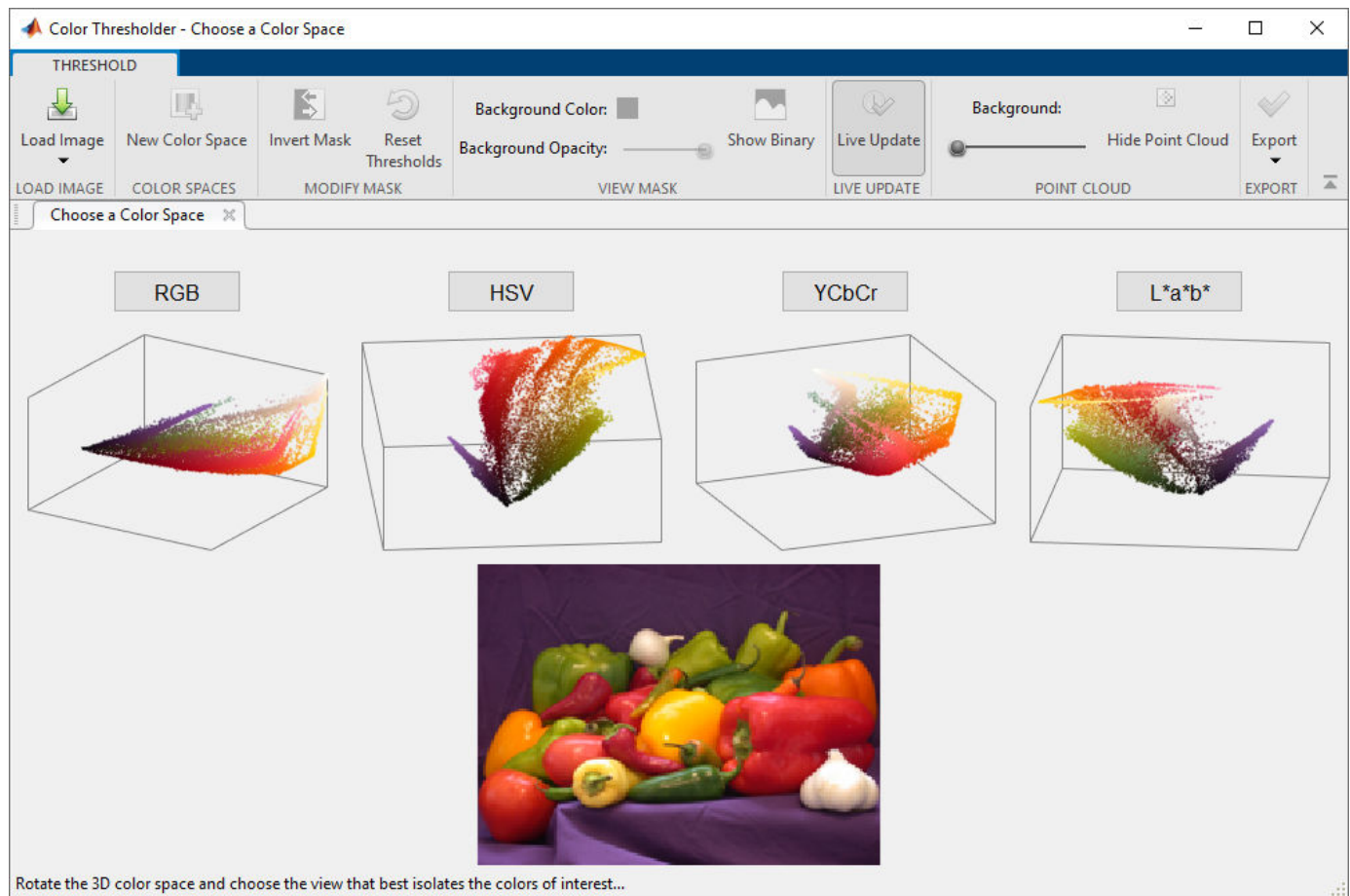
Segment YCbCr Image in Color Thresholder App

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 and the point cloud representation. The color controls vary depending on the color space. For the YCbCr color space, the **Color Thresholder** app displays three histograms representing the three color components: the **Y** component represents brightness, the **Cb** component represents the blue-yellow spectrum, and the **Cr** component represents the red-green spectrum.
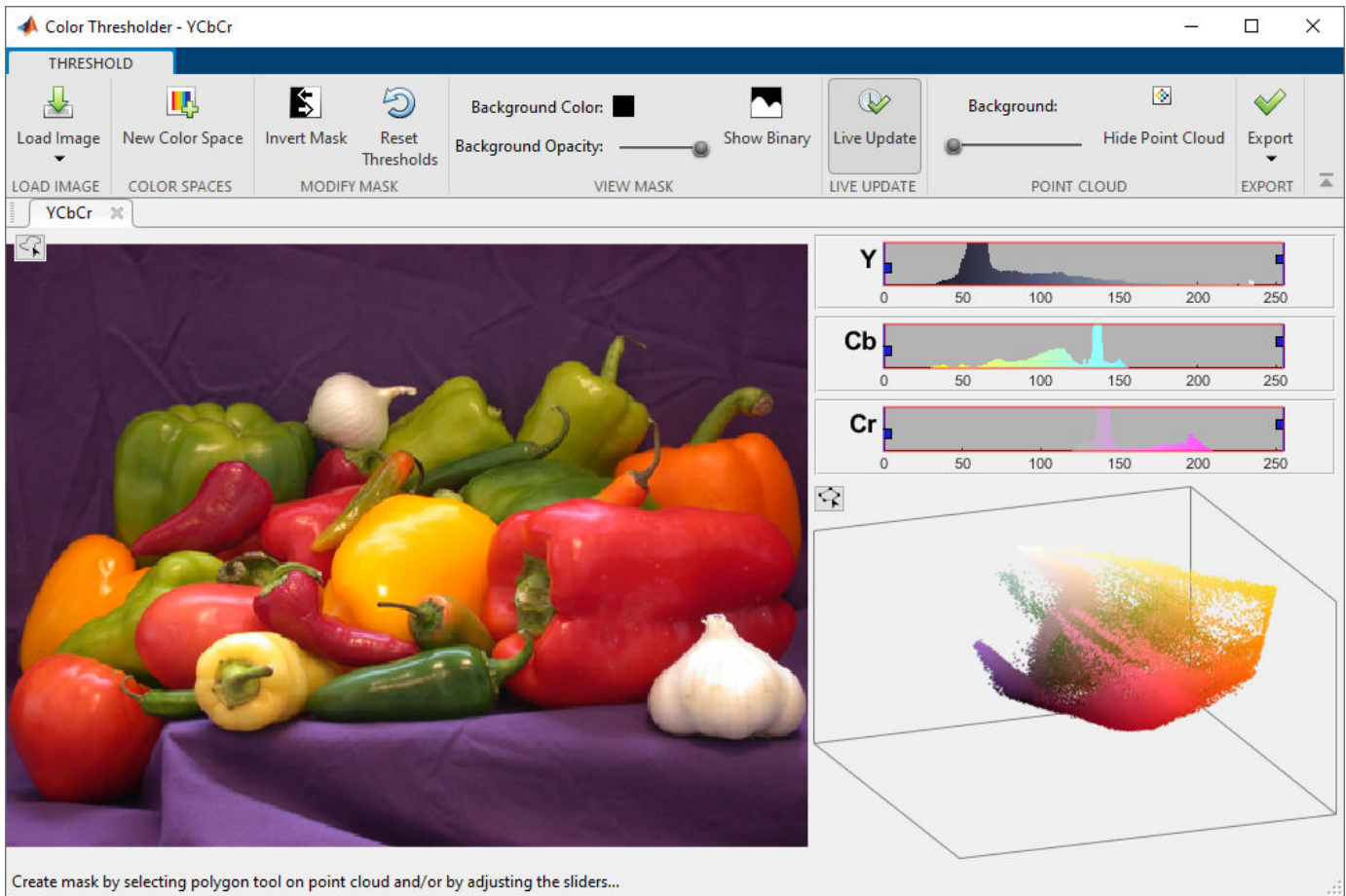
To explore the image, move the cursor over the image to access the pan and zoom controls.
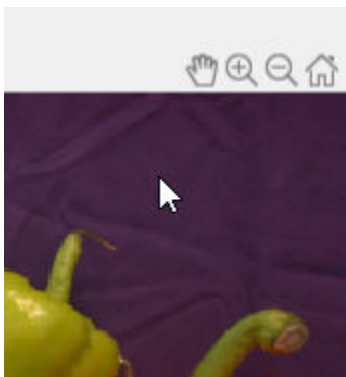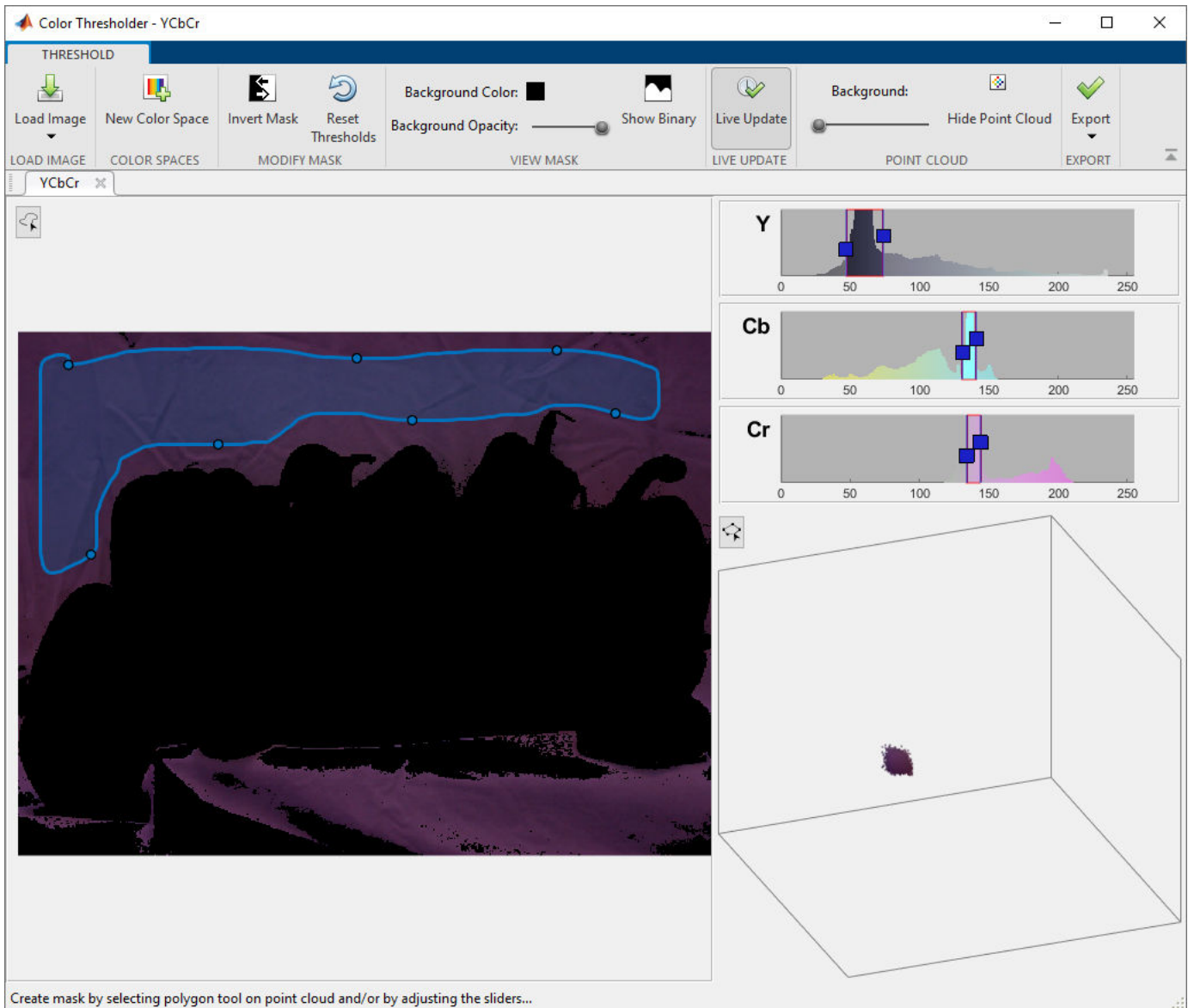
Automatic Thresholding

First, segment the image using automatic thresholding. Because the background color (purple cloth) is close to a uniform color, segment it rather than the foreground objects (the peppers). You can always invert the mask later using the **Invert Mask** option.

Define a region using the freehand ROI tool. Click the button [icon] in the upper-left corner of the image and draw an ROI on the background. You can draw multiple regions.

After drawing the region, the **Color Thresholder** app automatically thresholds the image based on the colors you selected in the region you drew. The **Y**, **Cb**, and **Cr** color controls change to reflect the segmentation. This automatic thresholding does not create a clean segmentation of the background and foreground, especially at the lower border between the foreground and background. For this example, the background color is lighter near the bottom of the image. If you want to delete a region you drew and start over, right-click anywhere in the region and select **Delete Freehand**.

Refine the Automatic Thresholding with the Color Controls

To fine tune the automatic thresholding, use the color controls. For each **Y**, **Cb**, and **Cr** color control, you can set the range of values by dragging the lower and upper bounds in that histogram. Using these color controls you can significantly improve the segmentation of the foreground.
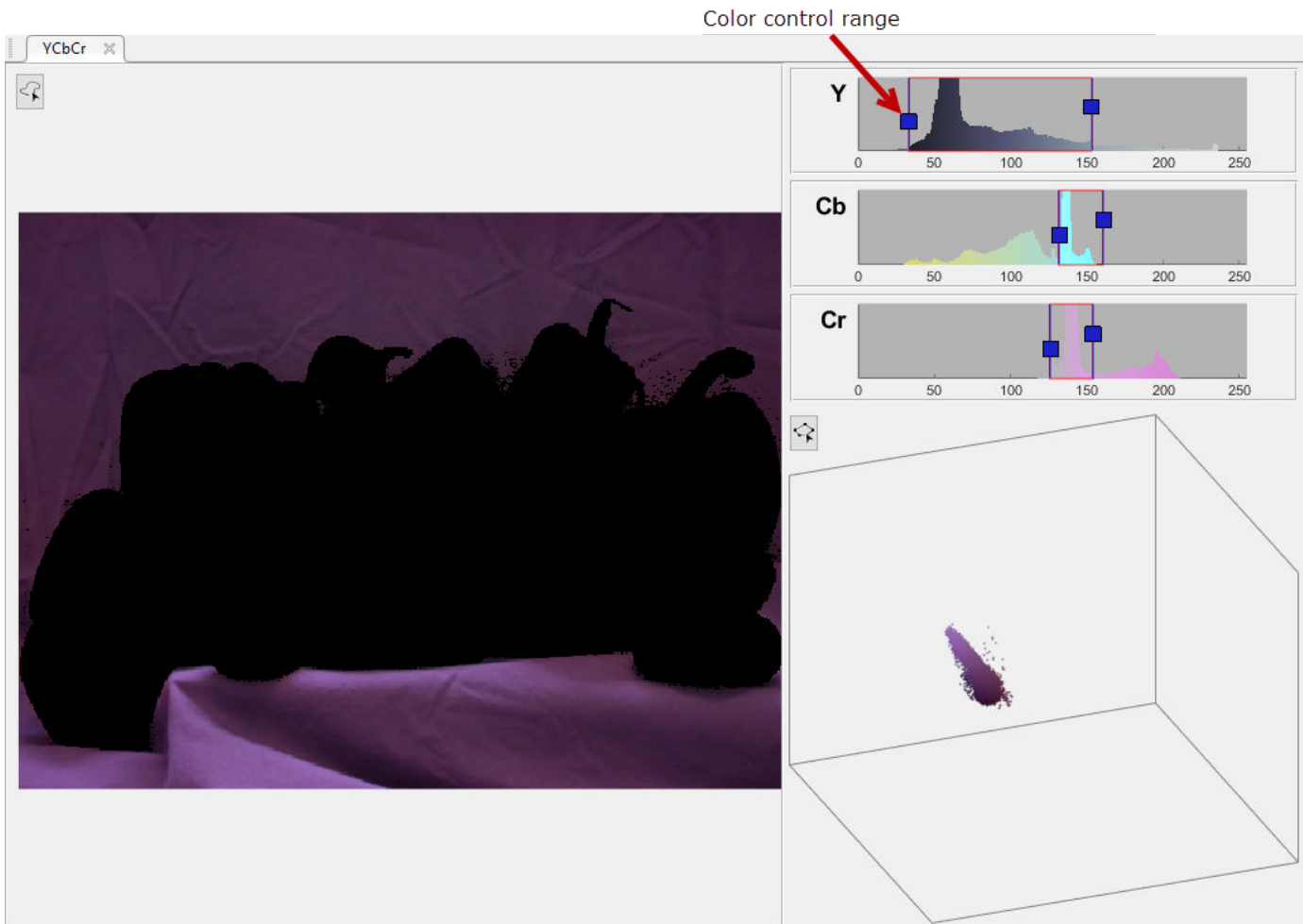
Threshold Image Color Values Using Point Cloud

Another approach to segmenting the image in the YCbCr color space is to draw an ROI on the point cloud to select a range of colors.

On the app toolstrip, click **Reset Thresholds** to revert back to the original image. In the bottom-right pane of the app, click and drag the point cloud to rotate until you isolate the view of the color you are

interested in thresholding. Click the ⬦ button in the upper left corner of the point cloud. The **Color Thresholder** app converts the 3-D point cloud into a 2-D representation and activates the polygon ROI tool. Draw an ROI around the color you want to segment (purple). This method can create a better segmentation than the initial automatic thresholding approach.

13-47
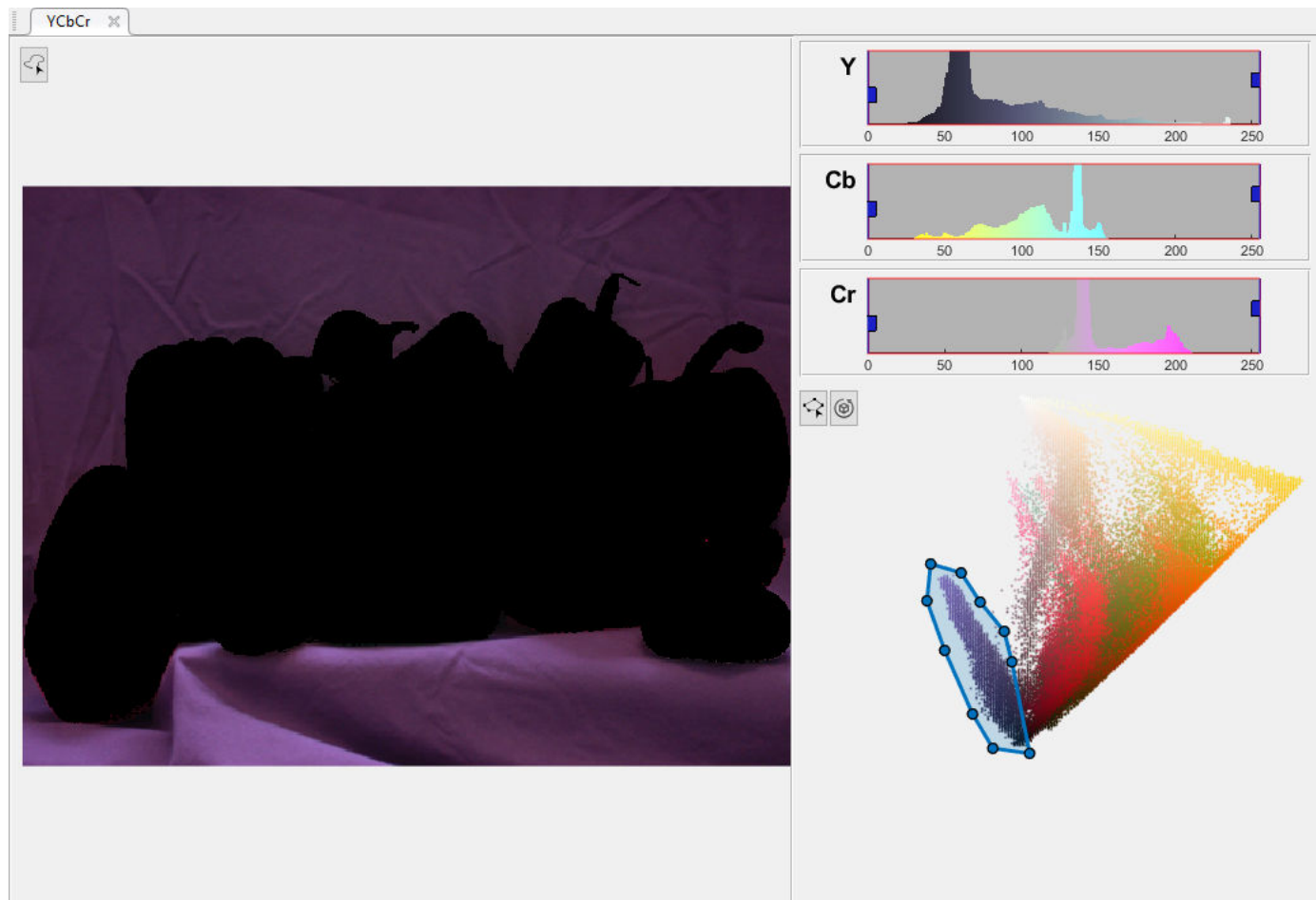
Segment Image in Another Color Space

To segment the image in another color space, click **New Color Space** in the app toolstrip. In the **Choose a Color Space** tab, choose the HSV color space.

The **Color Thresholder** app creates a new tab displaying the image and the color component controls for the HSV color space. In this color space, **H** represents hue, **S** represents saturation, and **V** represents value. The HSV color space uses a dual-direction knob for the **H** component and two histogram sliders for the **S** and **V** components. The tab also contains the point cloud representation of the colors in the image.
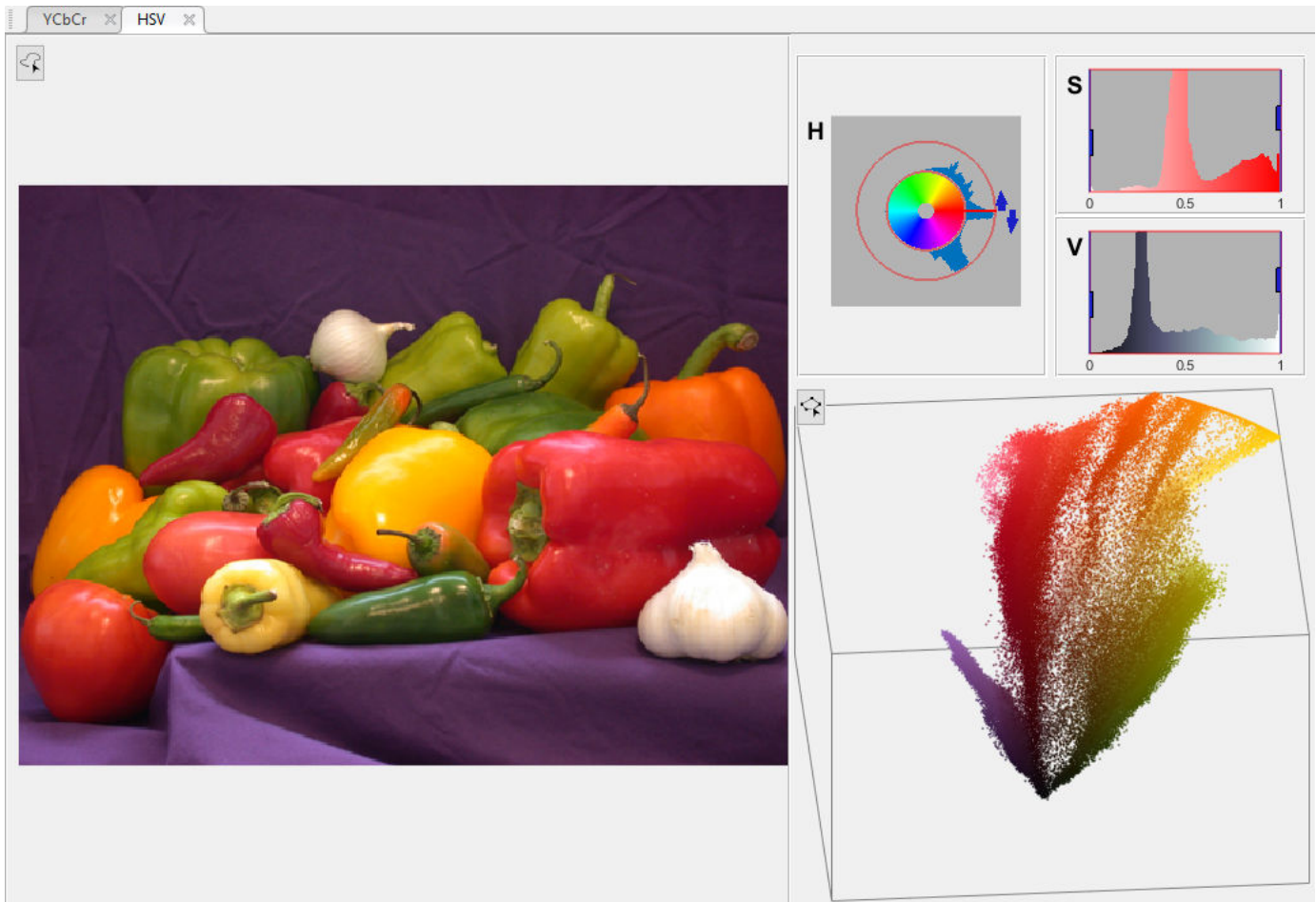
As in the previous iteration, you can use all of the same techniques: automatic thresholding and interactive use of the color component controls, including the point cloud. When you use the color controls, you can see the segmentation in progress. In the pane with the **H** control, change the range of the hue by clicking and dragging one arrow at a time. Experiment with the controls until you have a clean separation of the background from the foreground. You can clean up small imperfections after you create the mask image using toolbox functions, such as morphological operators.
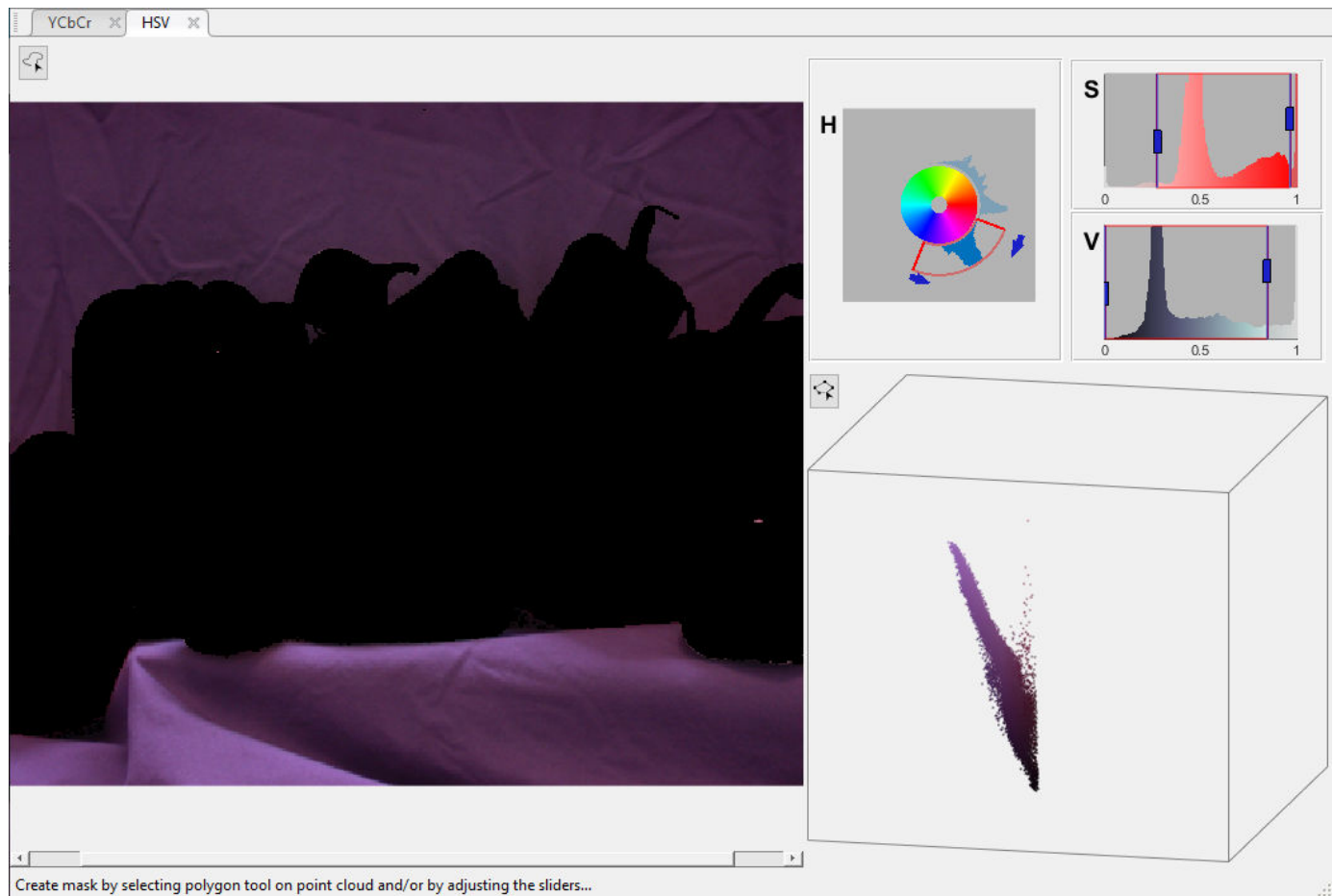
Create Mask Image Using Color Thresholder App

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.

Because the example segmented the background (the purple cloth) rather than the foreground objects (the peppers), swap the foreground and background by clicking **Invert Mask**.
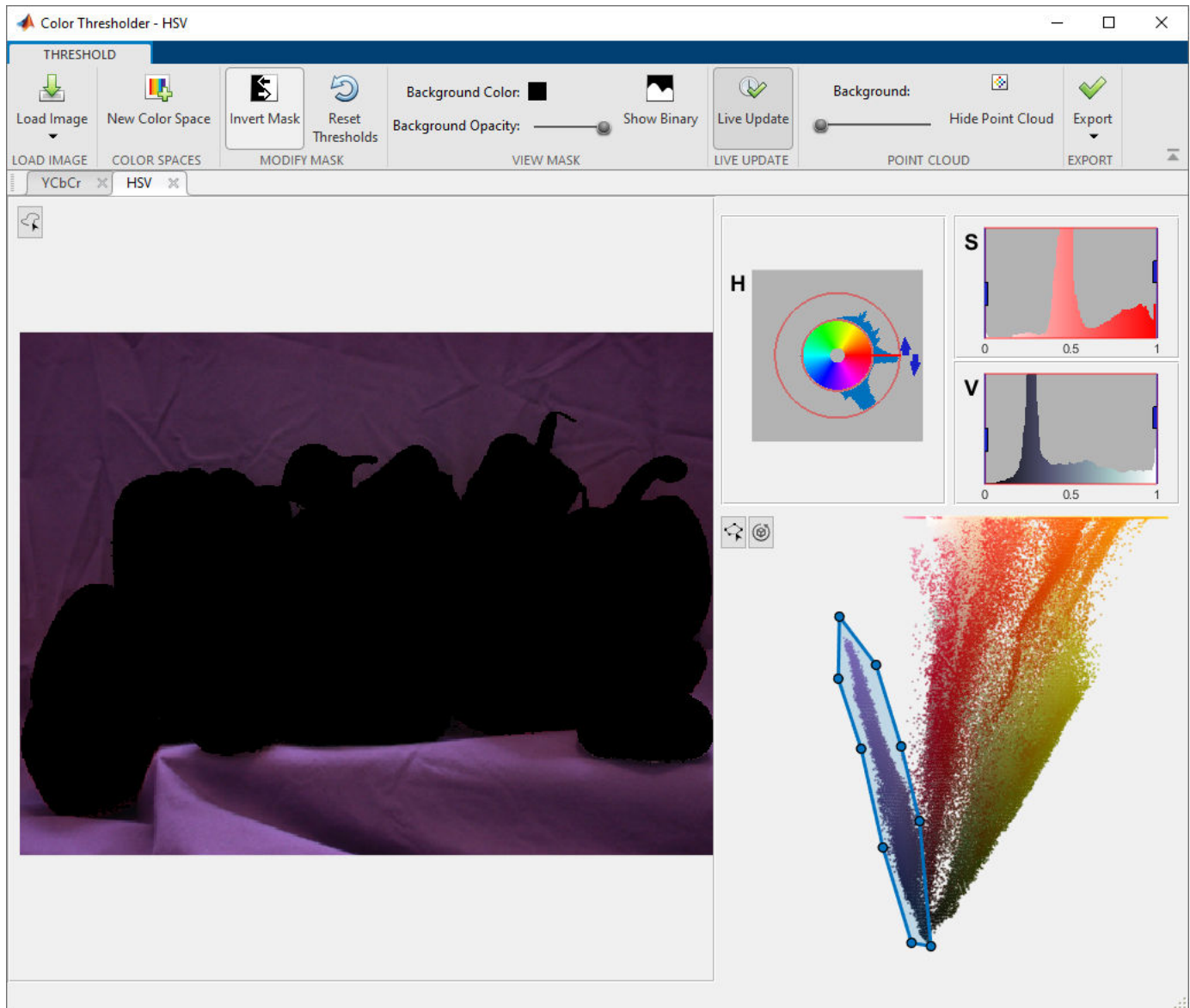
View the binary mask image that you created by clicking **Show Binary** on the app toolstrip.
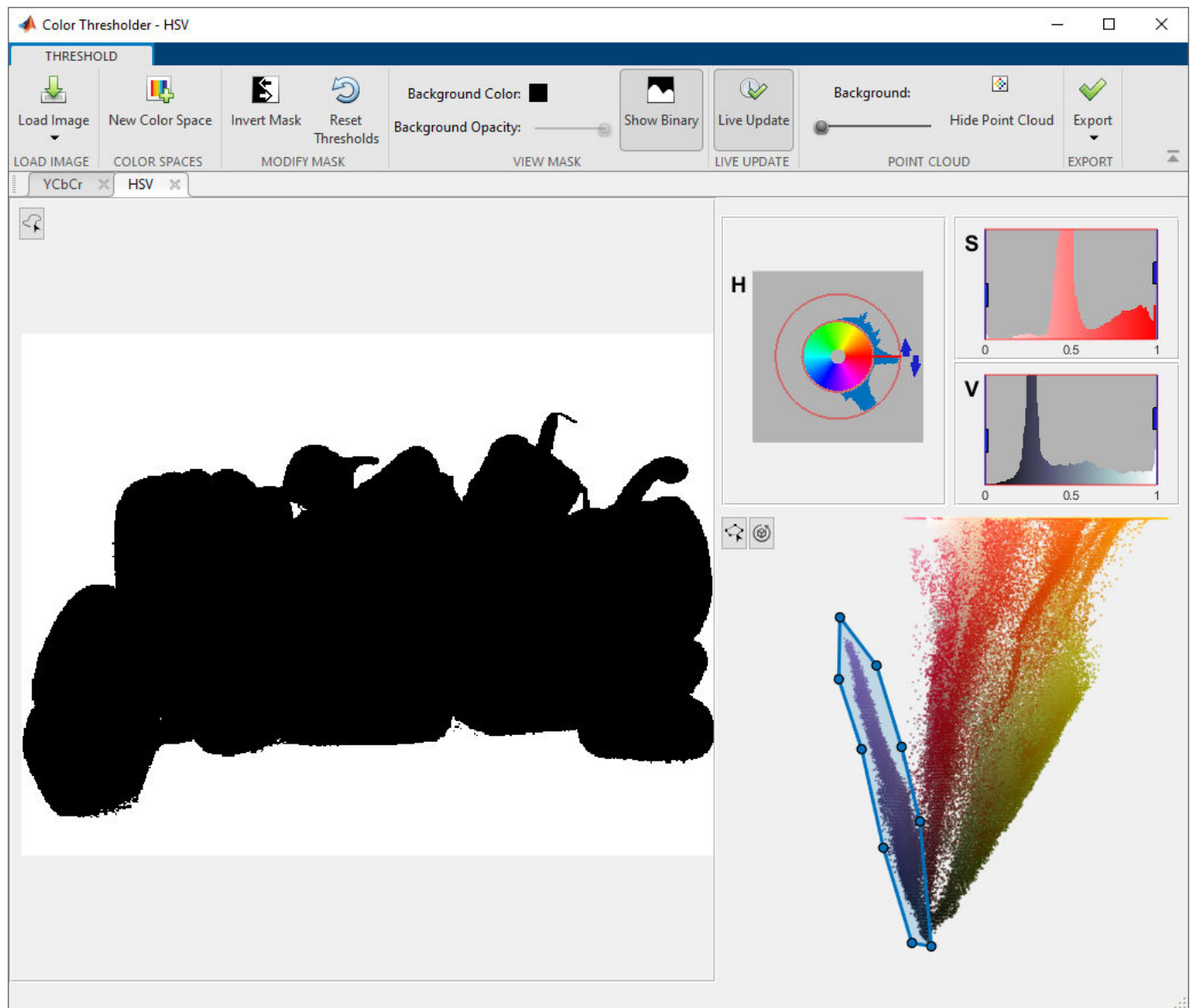
Save the mask image in the workspace. On the mask toolstrip, click **Export** and select **Export Images**.

In the Export To Workspace dialog box, specify variable names for the binary mask image. You can also save the original input RGB image and the segmented version of the original image.
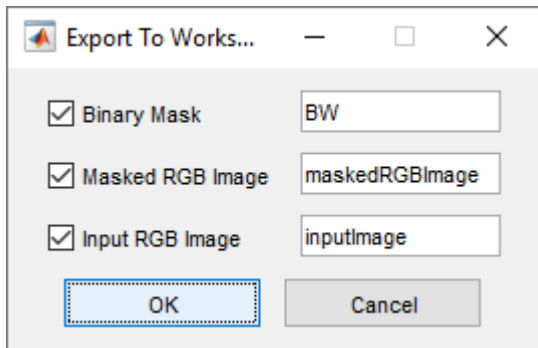
To save the MATLAB code required to recreate the segmentation, click **Export** and select **Export Function**. The Color Thresholder app opens the MATLAB Editor with the code that creates the segmentation. To save the code, click **Save** on the MATLAB Editor toolstrip. You can run this code, passing it an RGB image, to create the same mask image programmatically.

```matlab
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
```

See Also
Color Thresholder

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.
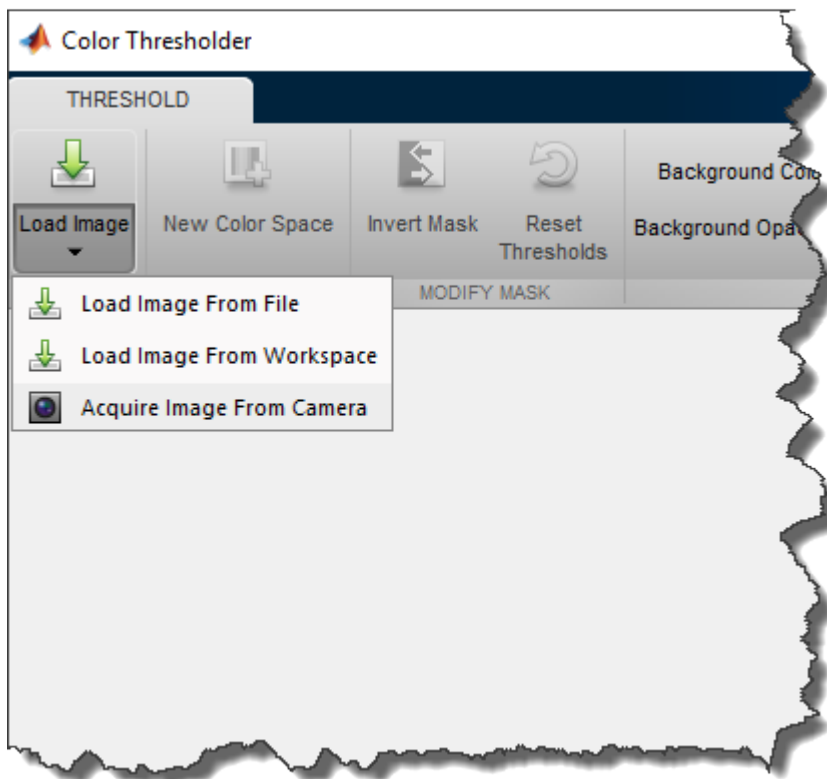
3 On the **Image Capture** tab, if you have only one webcam connected to your system, it is selected by default, and a live preview window opens. If you have multiple cameras connected and want to use a different one, select the camera in the **Camera** list.

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.
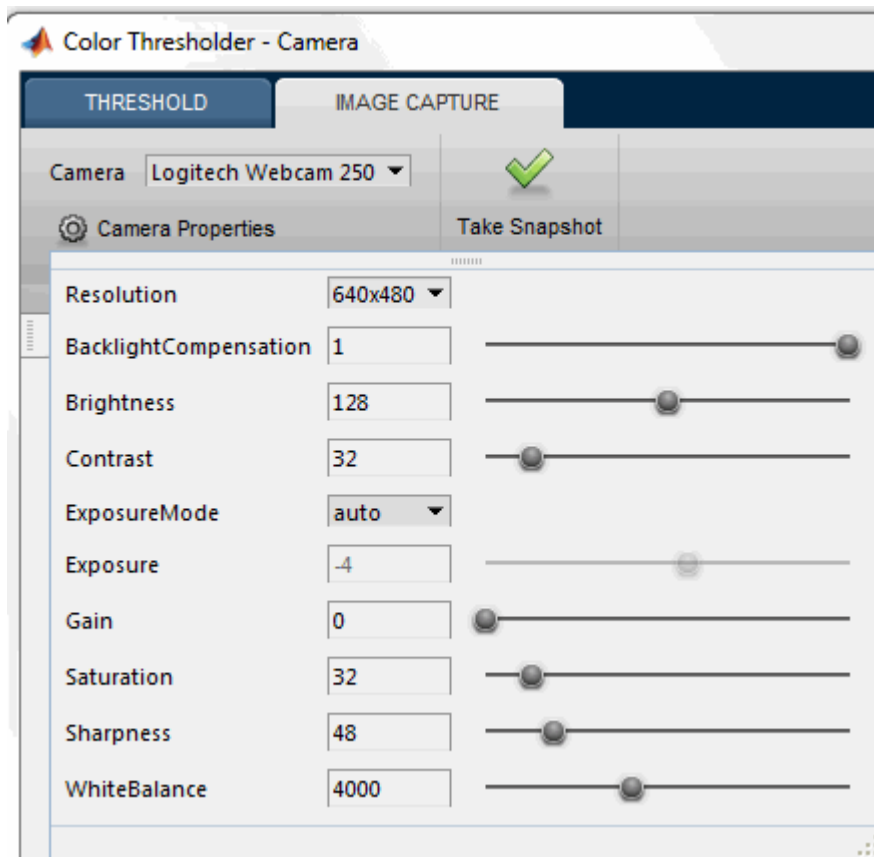
Use the sliders or drop-downs to change property settings. The **Preview** window updates dynamically when you change a setting. When you are done setting properties, click outside of the box to close the properties list.

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.
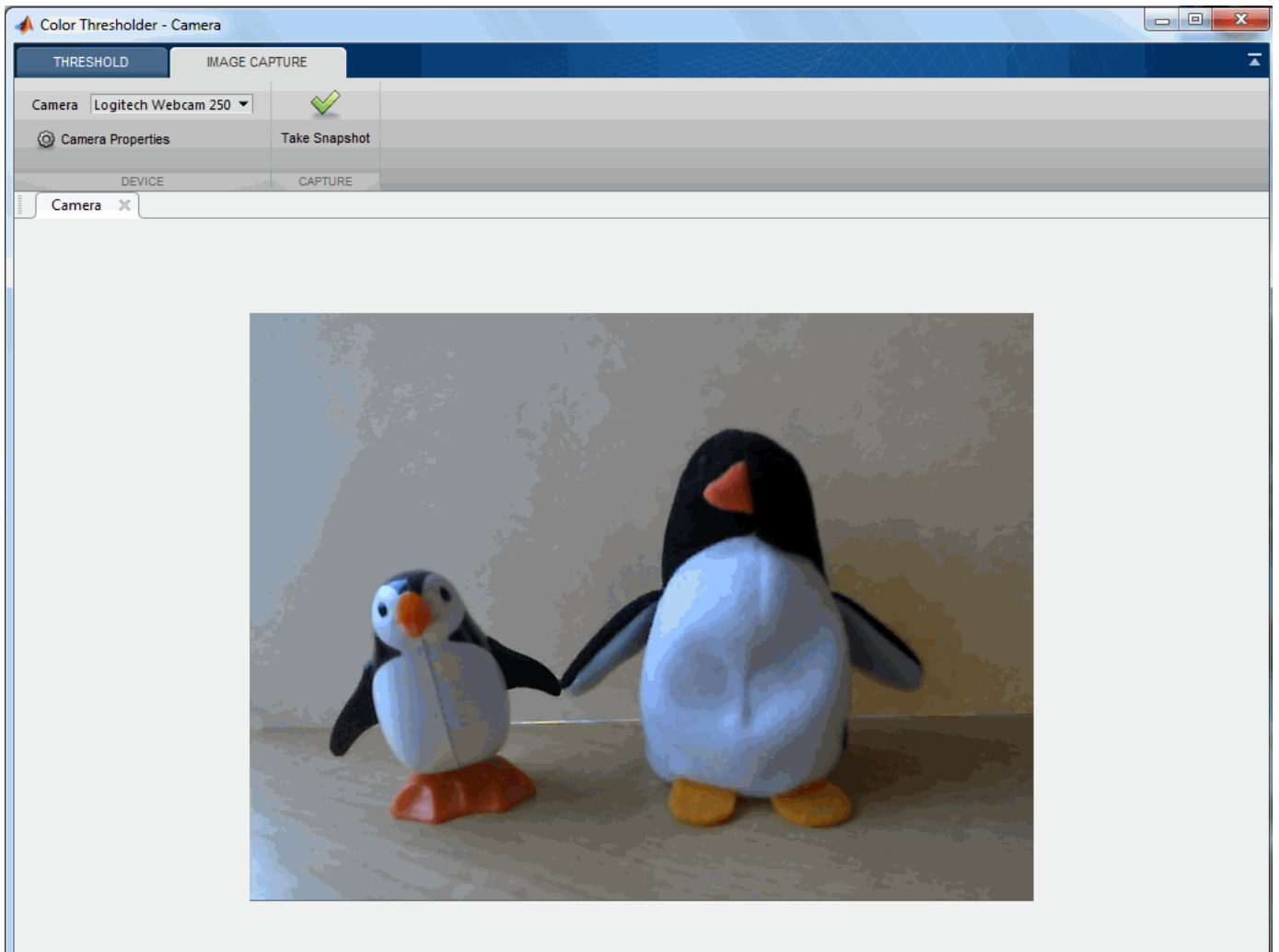
6 Click **Take Snapshot**. The **Choose a Color Space** dialog box opens and displays the four color space options: RGB, HSV, YCbCr, and L*a*b*.
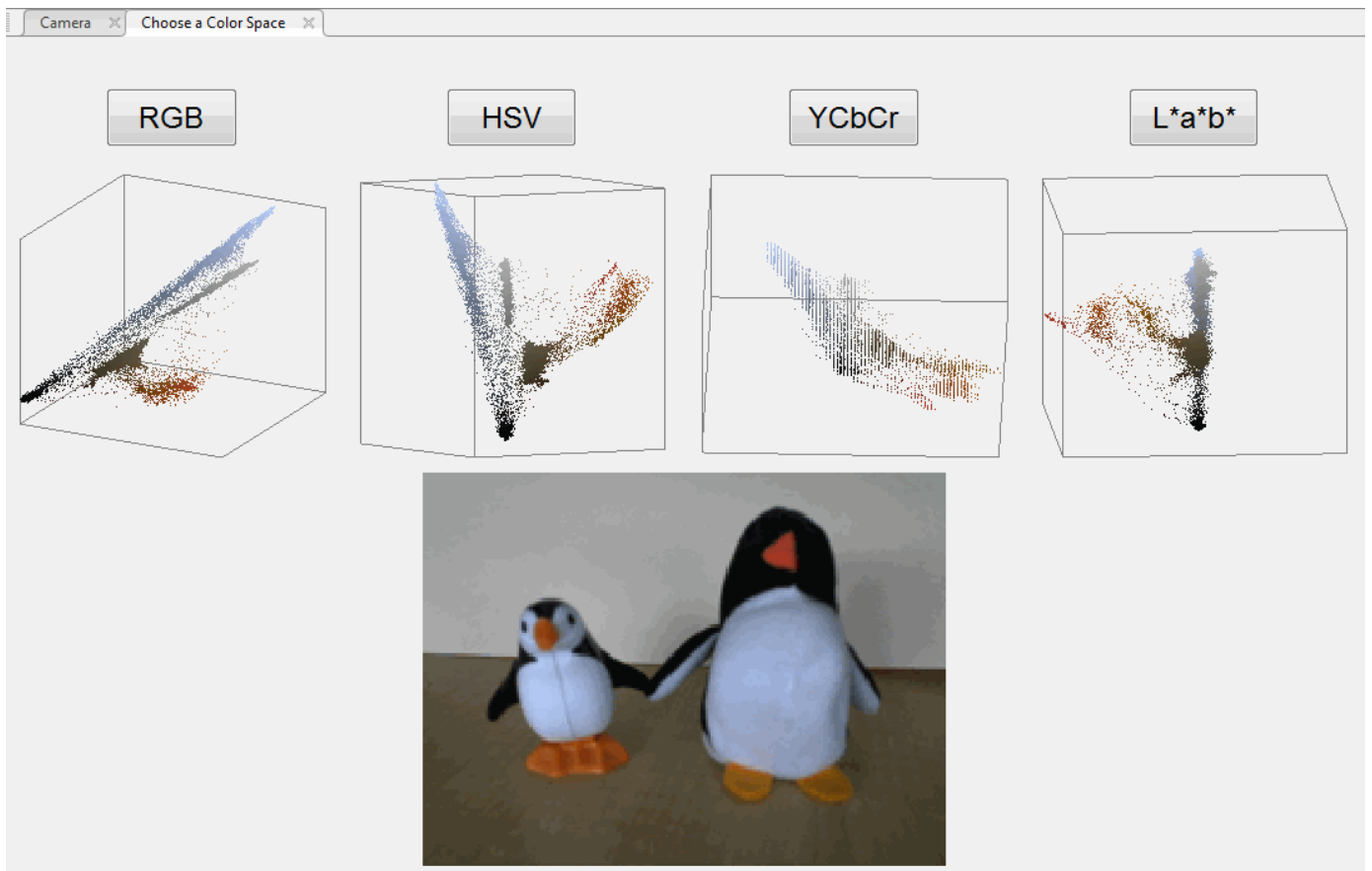
7 Choose a color space by clicking the button of your choice. Your image displays in a new tab using the selected color space.
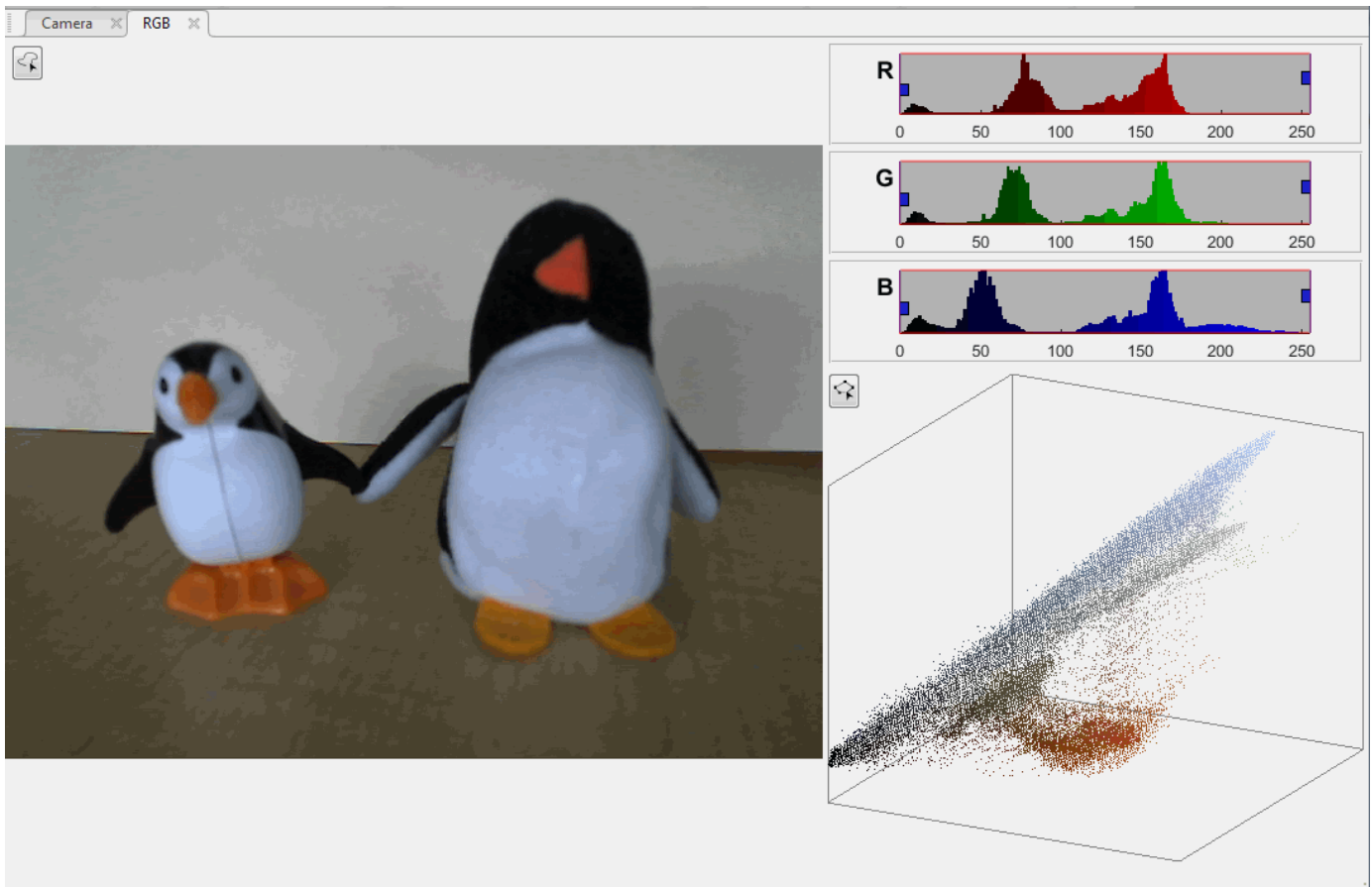
8 You can now perform color thresholding on the image. See "Segment Image and Create Mask Using Color Thresholder App" on page 13-42 for information about processing the image.

9 If you want to save the image that you captured, click **Export** and select **Export Images**.
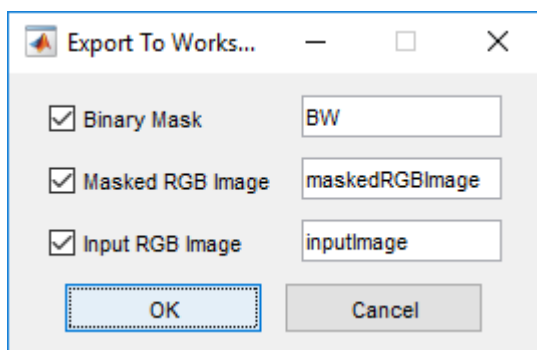
In the Export To Workspace dialog box, you can save the binary mask, segmented image, and the captured image. The **Input RGB Image** option saves the RGB image captured from the webcam.

10 You can take more snapshots, preview, or change camera properties at any time by clicking the **Camera** tab.

See Also
Color Thresholder

More About

- "Segment Image and Create Mask Using Color Thresholder App" on page 13-42

Image Segmentation Using Point Clouds in the Color Thresholder App

This example shows how to segment an image using the point cloud control in the Color Thresholder app. The example segments the human face from the background.

Load Image into 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. To work with the image in the Color Thresholder, you must convert the image into an RGB image, using the `demosaic` function. After converting the image, display it with the `imshow` function.

```
X = imread('mandi.tif');
rgb = demosaic(X,'bggr');
imshow(rgb)
```


Open the Image in the Color Thresholder

From the MATLAB® Toolstrip, open the Apps tab and under Image Processing and Computer Vision,

click the Color Thresholder icon . The Color Thresholder app opens.
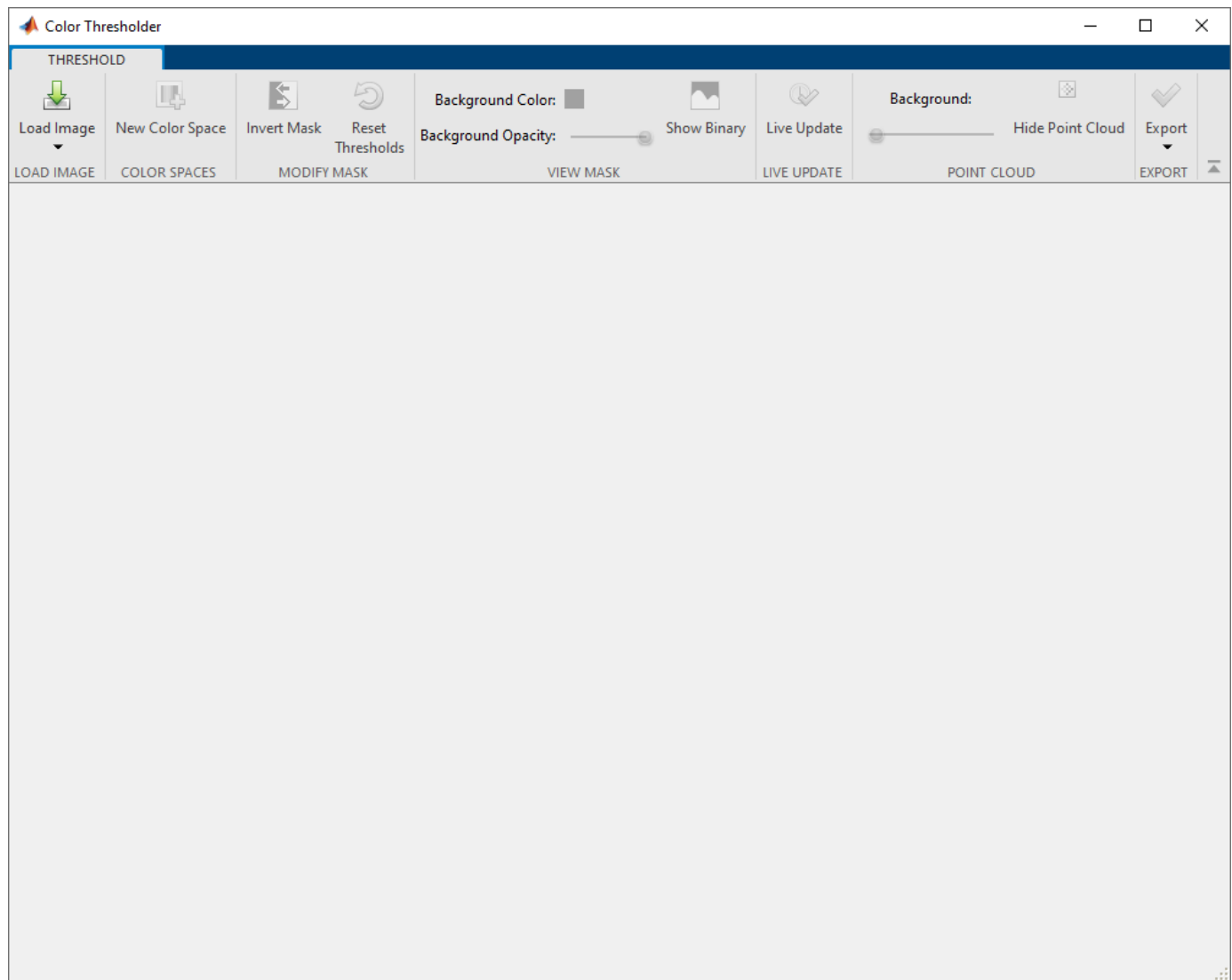
To bring an image into the Color Thresholder app, click **Load Image**. Because the image is already in the workspace, choose the **Load Image from Workspace** option. In the Import from Workspace dialog box, select the variable you created and click **OK.** You can also load an image by specifying its file name.
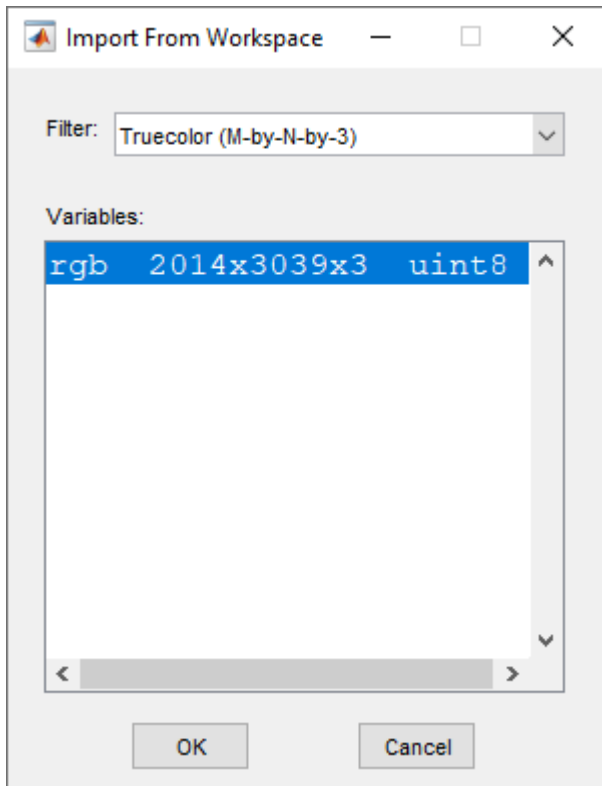
You can also open the app using the `colorThresholder` command, specifying the name of the image you want to open: `colorThresholder(rgb)`. For information about acquiring an image from a camera, see "Acquire Live Images in the Color Thresholder App" on page 13-55.

Choose a Color Space

When it opens, the Color Thresholder app displays the **Choose a color space** tab. This tab displays the image, and point cloud representations of the image, in several popular color spaces: RGB, HSV, YCbCr, and *L*a*b**.

Explore the point cloud representations of the image in each color space. Rotate the 3-D depiction in 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 to select the colors of the area you want to segment. For this example, choose the *L*a*b** color space.

Click one of the color space buttons to
choose the color space representation.
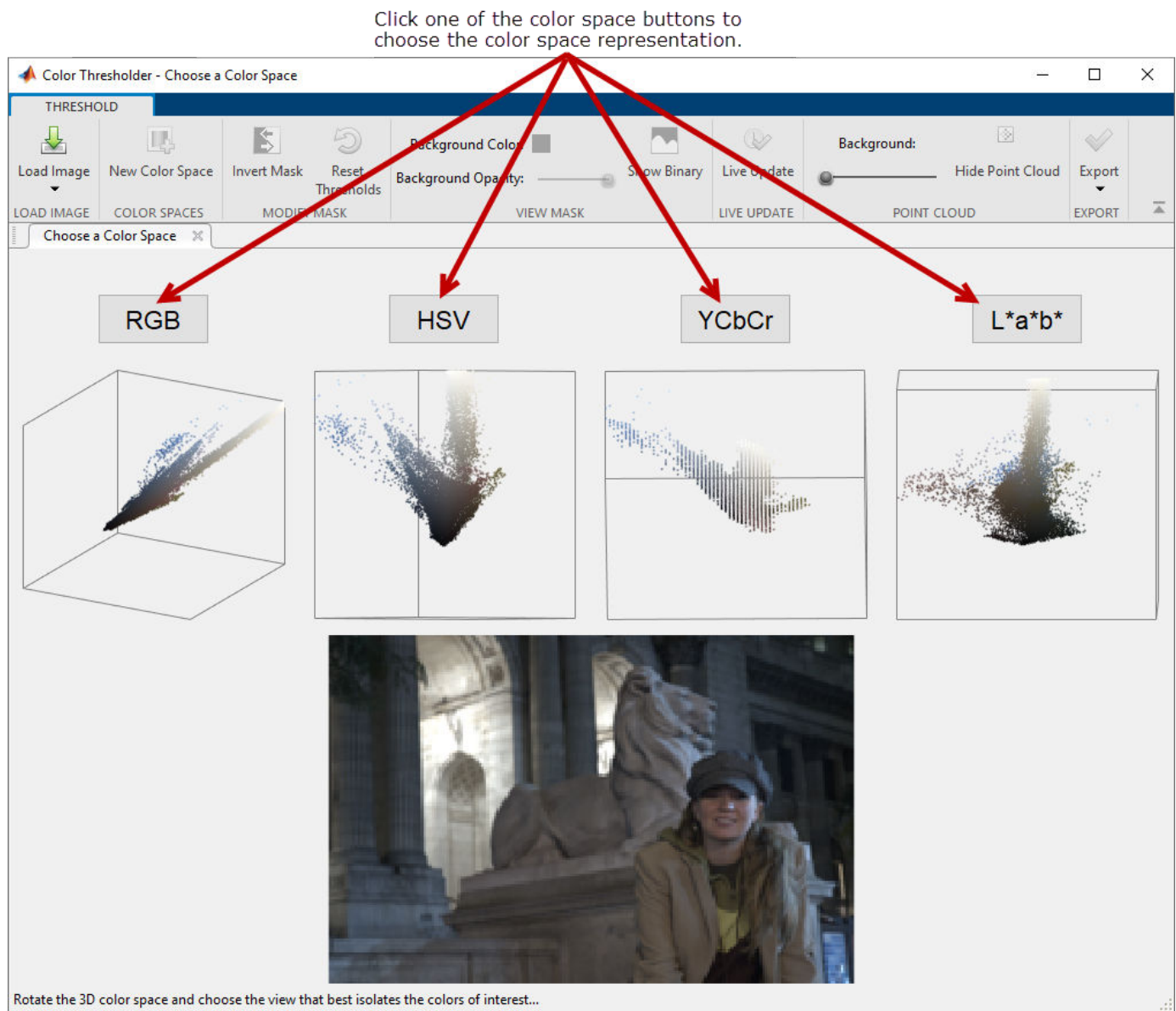
Segment the Image Using the Color Cloud

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. The tab
also includes a 3-D point cloud representation of the colors of the image in the color space. Other
color spaces use different types of controls.
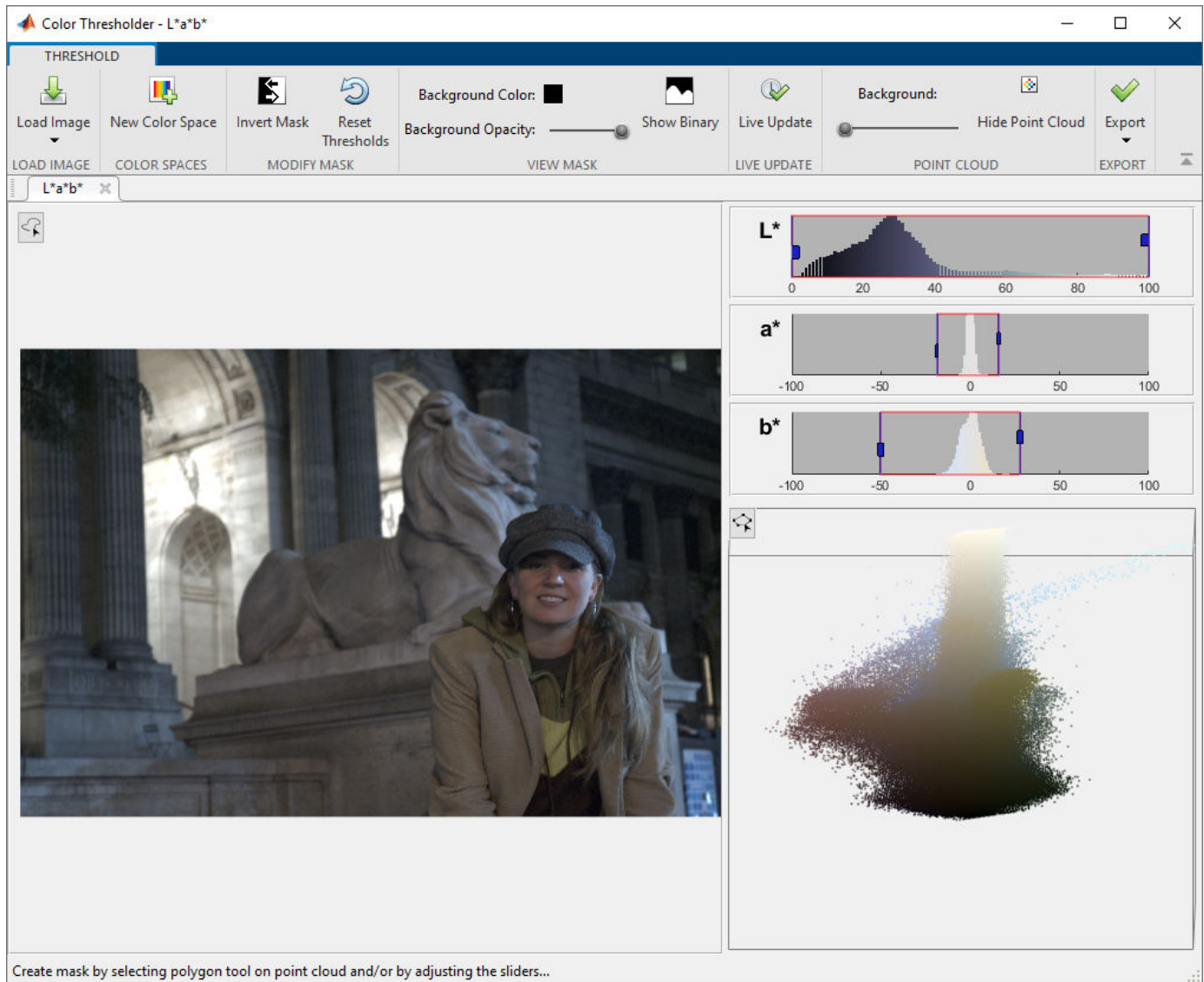
To explore the image, move the cursor over the image and use the pan and zoom controls.

Pan and zoom controls

Move the cursor over the image to see the pan and zoom controls

To segment the image, rotate the 3-D color cloud, using the mouse, to find a view of the color cloud that isolates the colors that you want to segment. To select the colors in the image, click the drawing tool in the upper-left corner of the point cloud. Then, using the mouse, draw a polygon around the colors you want to segment. 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.
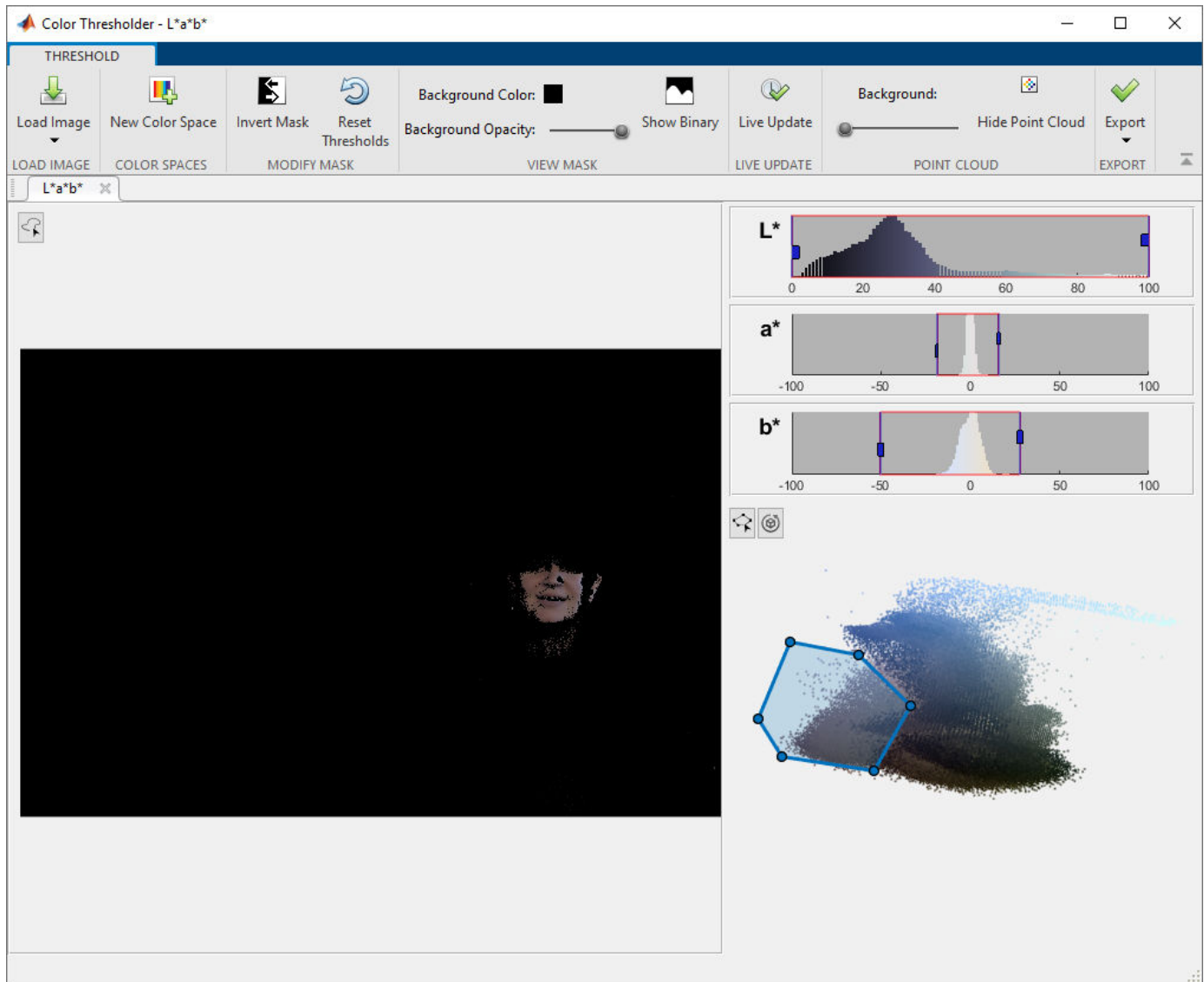
For information about creating a mask and saving it, see "Segment Image and Create Mask Using Color Thresholder App" on page 13-42.

See Also

Color Thresholder

More About

- "Segment Image and Create Mask Using Color Thresholder App" on page 13-42

Getting Started with 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.

Open Image Segmenter App and Load Data

Open the app and load an image to be segmented. **Image Segmenter** can open any file that can be read by `imread`.

You can open the **Image Segmenter** from the command line. Specify an image in the workspace or the name of a file.

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

Alternatively, open the app from the **Apps** tab, under **Image Processing and Computer Vision**. Then, from the **Load** menu, choose the name of a workspace variable or the name of the file containing the image.

After you load an image, you can optionally load an existing binary mask. For example, you might have previously created a mask of an RGB image in the **Color Thresholder** app and you want to refine the segmentation. To load an existing mask, click **Load Mask**. The segmentation mask image must be a logical image of the same size as the image you are segmenting.

Create and Add Regions to Segmented Mask

To create an initial mask, use any of the tools in the **Create Mask** and **Add to Mask** menus. If you want to start a new segmentation after creating a mask, click **New Segmentation**. You can perform multiple segmentations using the app. Each segmentation appears, with a thumbnail, in the Data Browser.

To add segmented regions to an existing mask, use tools in the **Add to Mask** menu. The app displays the steps you take while creating the segmentation in the History panel of the Data Browser.

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 Image Using Thresholding in Image Segmenter" on page 13-71.
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 "Segment Image Using Graph Cut in Image Segmenter" on page 13-92.

Tool	Description
Auto Cluster	An automatic technique where the app groups image features into a binary segmentation. This option is only available if you have Statistics and Machine Learning Toolbox™. For more information, see "Segment Image Using Auto Cluster in Image Segmenter" on page 13-117.
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 "Segment Image Using Find Circles in Image Segmenter" on page 13-110
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 "Segment Image Using Local Graph Cut (Grabcut) in Image Segmenter" on page 13-101.
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 Image By Drawing Regions Using Image Segmenter" on page 13-75.

When using the **Auto Cluster**, **Graph Cut**, and **Flood Fill** segmentation tools, you can also include texture as an additional consideration in your segmentation. Texture filtering can help distinguish foreground from background. To turn the texture option on and off, click **Include Texture Features**. When enabled, **Image Segmenter** uses Gabor filters to analyze the texture of the image as a preprocessing step in the segmentation. For more information about Gabor filters, see "Texture Segmentation Using Gabor Filters" on page 13-2.

Refine Segmented Mask

Image Segmenter provides access to several tools that you can use to refine the mask you created.

Tool	Description
Morphology	Many morphological techniques, such as dilation and erosion. For an example, view "Refine Segmentation Using Morphology in Image Segmenter" on page 13-87.
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 "Segment Image Using Active Contours in Image Segmenter" on page 13-81.
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 "Refine Segmentation Using Morphology in Image Segmenter" on page 13-87.

Tool	Description
Invert mask	Sometimes the segmentation is easier to evaluate if you invert the foreground and background. For an example, view "Segment Image Using Auto Cluster in Image Segmenter" on page 13-117

Export Segmentation Results

When you find an acceptable segmentation, you can export to the workspace the final segmentation mask image and the segmented version of the original image. To export the mask and segmentation to the workspace, click **Export** and select **Export Images**.

You can also generate the code used to perform the segmentation (requires Statistics and Machine Learning Toolbox.) Use the code to apply the same segmentation algorithm to similar images. To get the code, click **Export** and select **Generate Function**. The app opens the MATLAB editor containing a function with the autogenerated code. To save the code, click **Save** in the MATLAB editor.

See Also
Image Segmenter

Segment Image Using Thresholding in Image Segmenter

This example shows how to segment an image in the **Image Segmenter** app by using thresholding. The **Image Segmenter** app supports three different types of thresholding: Global, Manual, and Adaptive.

The **Image Segmenter** app supports many different segmentation methods and using the app can be an iterative process. You might try several different methods until you achieve the results you want.

Load Image into the Image Segmenter

Open the **Image Segmenter** app and load an image to be segmented. The app can open any file that can be read by `imread`.

For this example, first read an image into the workspace. This example uses an MRI image of a knee. The goal is to create a mask image that segments the bone from the soft tissue in the image.

```
I = dicomread('knee1');
knee = mat2gray(I);
```

From the MATLAB® toolstrip, open the **Image Segmenter** app. On the **Apps** tab, in the **Image Processing and Computer Vision** section, click **Image Segmenter** .

On the app toolstrip, click **Load**, and then select **Load Image from Workspace**. In the Import from Workspace dialog box, select the image you read into the workspace. The **Image Segmenter** app displays the image you selected.

You can also open the image in the **Image Segmenter** app using the `imageSegmenter` command, as follows:

```
imageSegmenter(knee);
```

After you load an image, you can optionally load an existing binary mask. For example, you might have previously created a mask of an RGB image in the **Color Thresholder** app and you want to refine the segmentation. To load an existing mask, click **Load Mask**. The segmentation mask image must be a logical image of the same size as the image you are segmenting.

Use Thresholding to Segment Image

Click **Threshold** in the Create Mask section of the **Image Segmenter** app toolstrip. The app displays the thresholded image in the Threshold tab. By default, the app uses global thresholding.
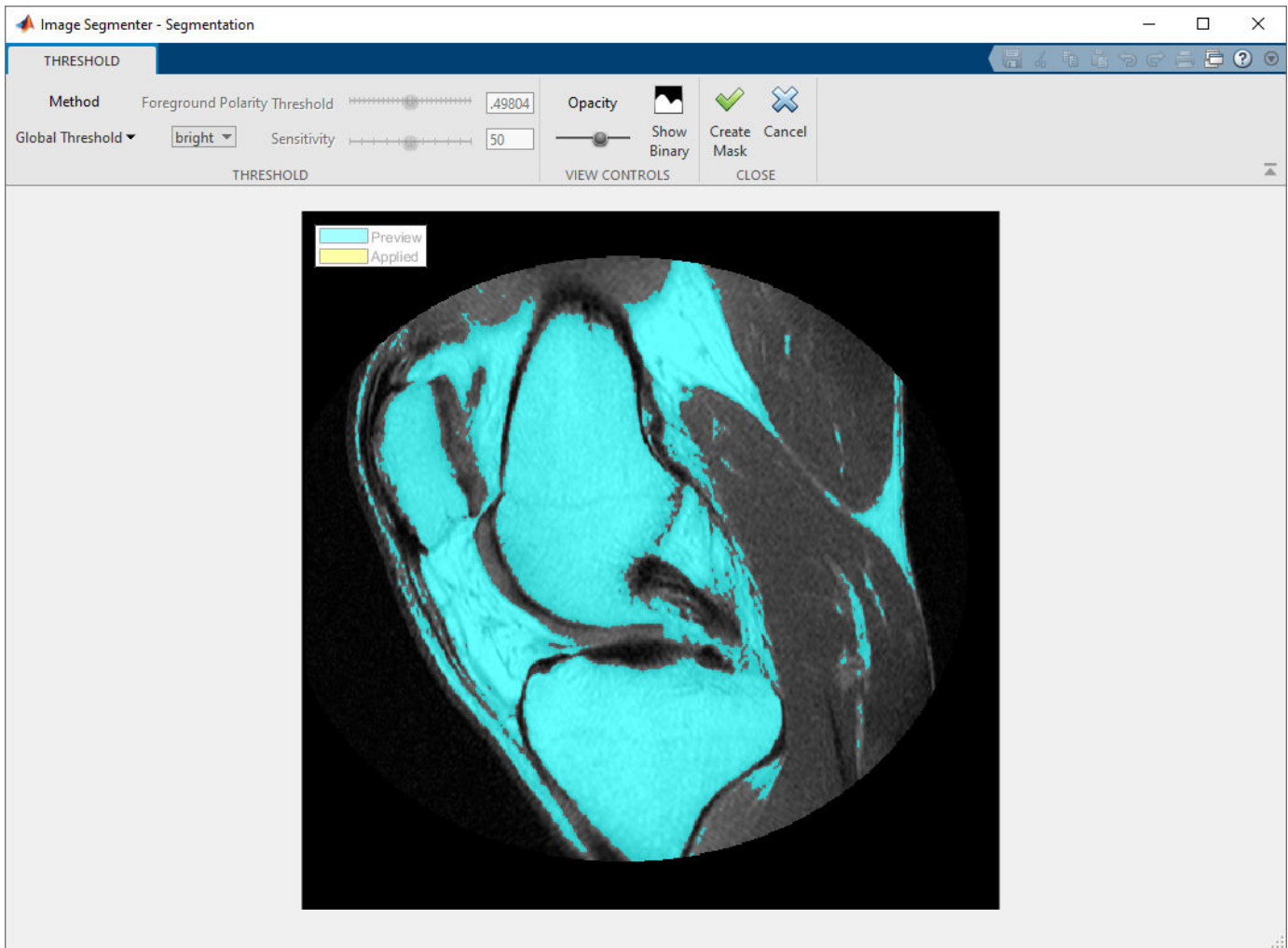
You can also choose **Manual** or **Adaptive** thresholding. Each thresholding option supports controls that you can use to fine-tune the thresholding. For example, with Manual thresholding, you can choose the threshold value using the slider. With Adaptive thresholding, you can choose the sensitivity using the slider. Try each option to see which thresholding method performs the best segmentation.

The following figure shows the results of using **Manual** thresholding.
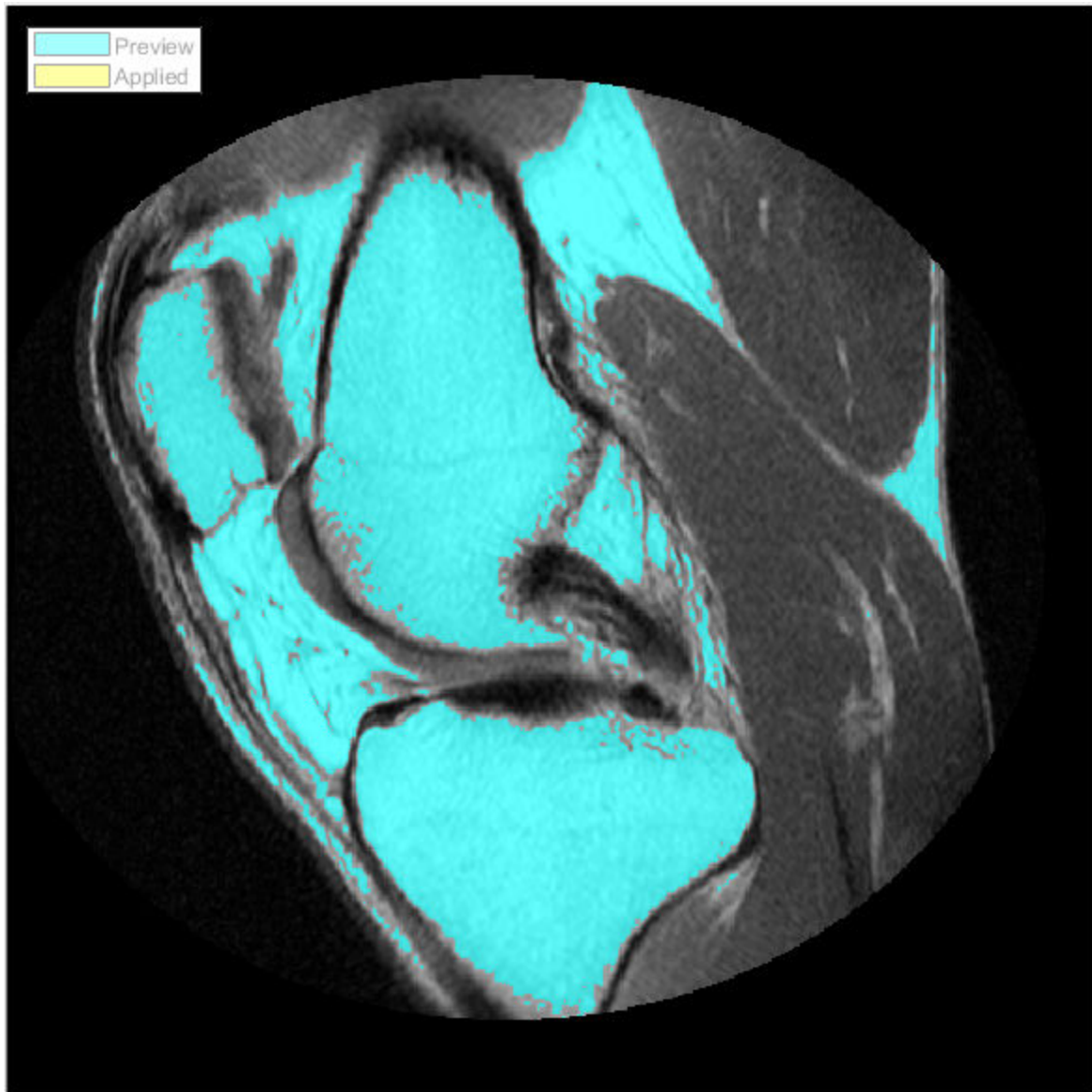
The knee image does not have well-defined pixel intensity differences between foreground and background and thresholding does not seem like the best choice to segment this image.

To save the segmentation, click **Create Mask**. If you want to try another segmentation method in the **Image Segmenter** app, click **Cancel** to return to the main segmentation app window.

See Also
Image Segmenter

Related Examples
- "Getting Started with Image Segmenter App" on page 13-68

Segment Image By Drawing Regions Using Image Segmenter

This example shows how to segment an image in the **Image Segmenter** app by drawing regions of interest. The **Image Segmenter** app offers many different ROI shapes including polygons, rectangles, ellipses, and circles. In addition, you can draw freehand shapes or freehand shapes that help you by following the underlying shape of the objects in the image.

The **Image Segmenter** app offers many different segmentation methods and using the app can be an iterative process. You might try several different methods until you achieve the results you want.

Load an Image in the Image Segmenter

Open the **Image Segmenter** app and load an image to be segmented. The app can open any file that can be read by `imread`.

For this example, read an image into the workspace. This example uses an MRI image of a knee. The goal is to create a mask image that segments the bone from the soft tissue in the image.

```
I = dicomread('knee1');
knee = mat2gray(I);
```

Open the **Image Segmenter** app from the MATLAB® toolstrip. On the **Apps** tab, in the **Image Processing and Computer Vision** section, click **Image Segmenter** .

On the app toolstrip, click **Load**, and then select **Load image from Workspace**. In the Import from Workspace dialog box, select the image you read into the workspace. The **Image Segmenter** app displays the image you selected.
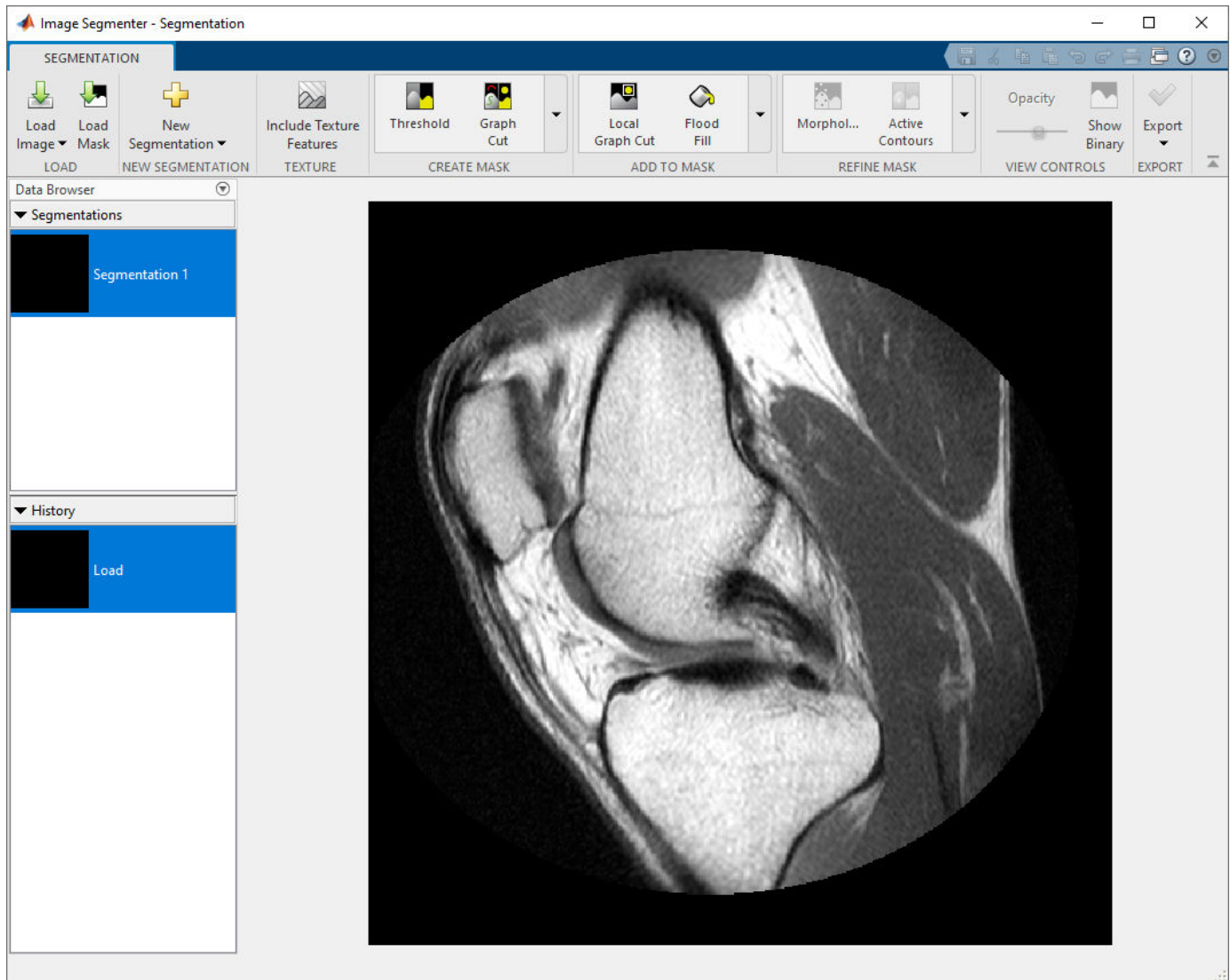
You can also open the image in the Image Segmenter app using the `imageSegmenter` command, as follows:

`imageSegmenter(knee);`

After you load an image into the app, you can optionally load an existing binary mask. For example, you might have previously created a mask of an RGB image in the **Color Thresholder** app and you want to refine the segmentation. To load an existing mask, click **Load Mask**. The segmentation mask image must be a logical image of the same size as the image you are segmenting.

Use ROI Tools to Draw Regions for Segmentation

Expand the Add to Mask group and click **Draw ROIs**. The app opens the ROI tab.
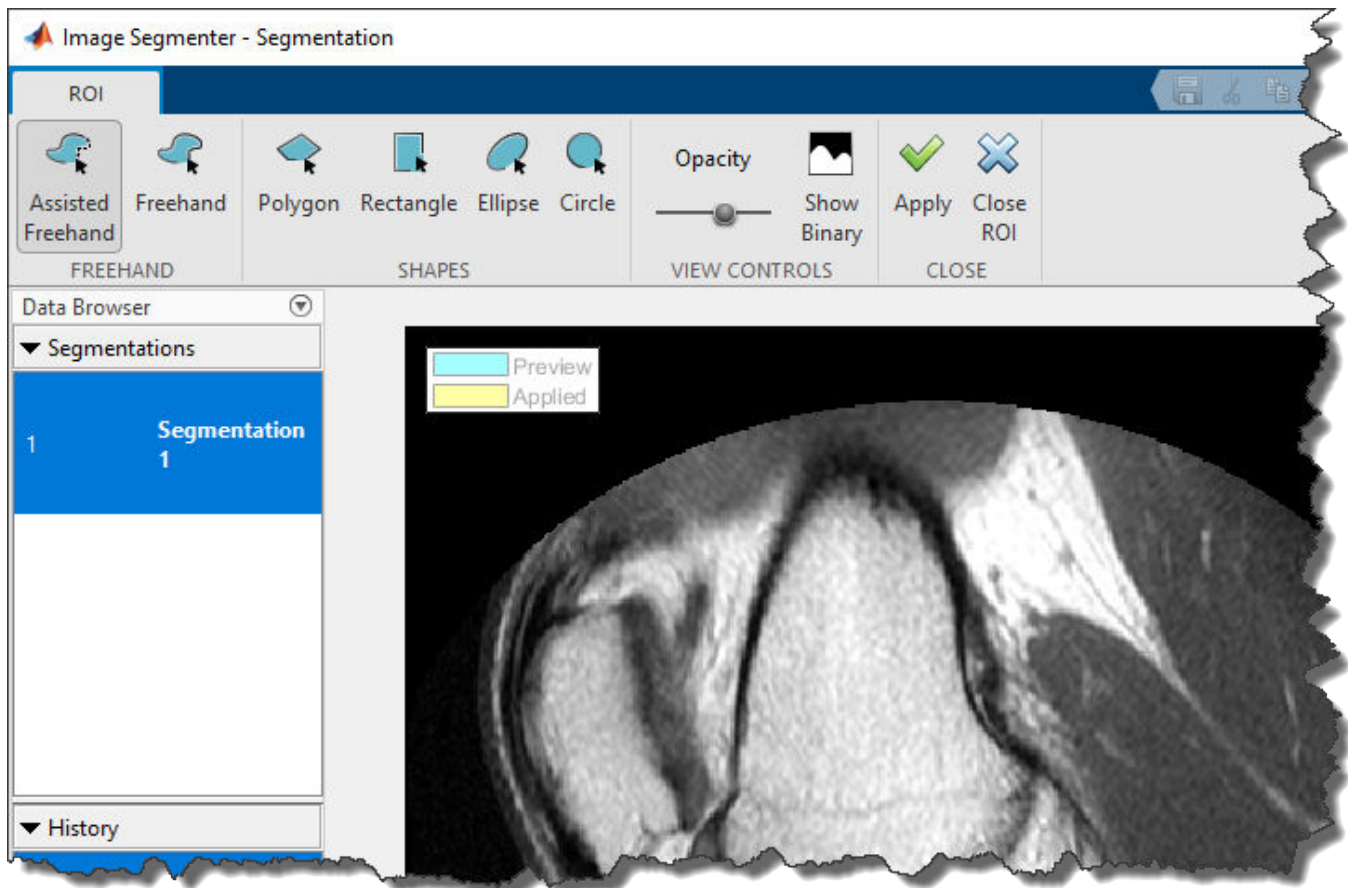
Select the type of ROI you want to draw. For this example, choose **Assisted Freehand**. As you move the cursor over the image, it changes to the crosshairs shape. Press the mouse button, and begin drawing a freehand shape over the area of the image that you want to segment. With the Assisted Freehand 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. Waypoints can help you make fine adjustments to the shape after you finish drawing. To add additional waypoints after you finish drawing, double-click on the ROI edge .
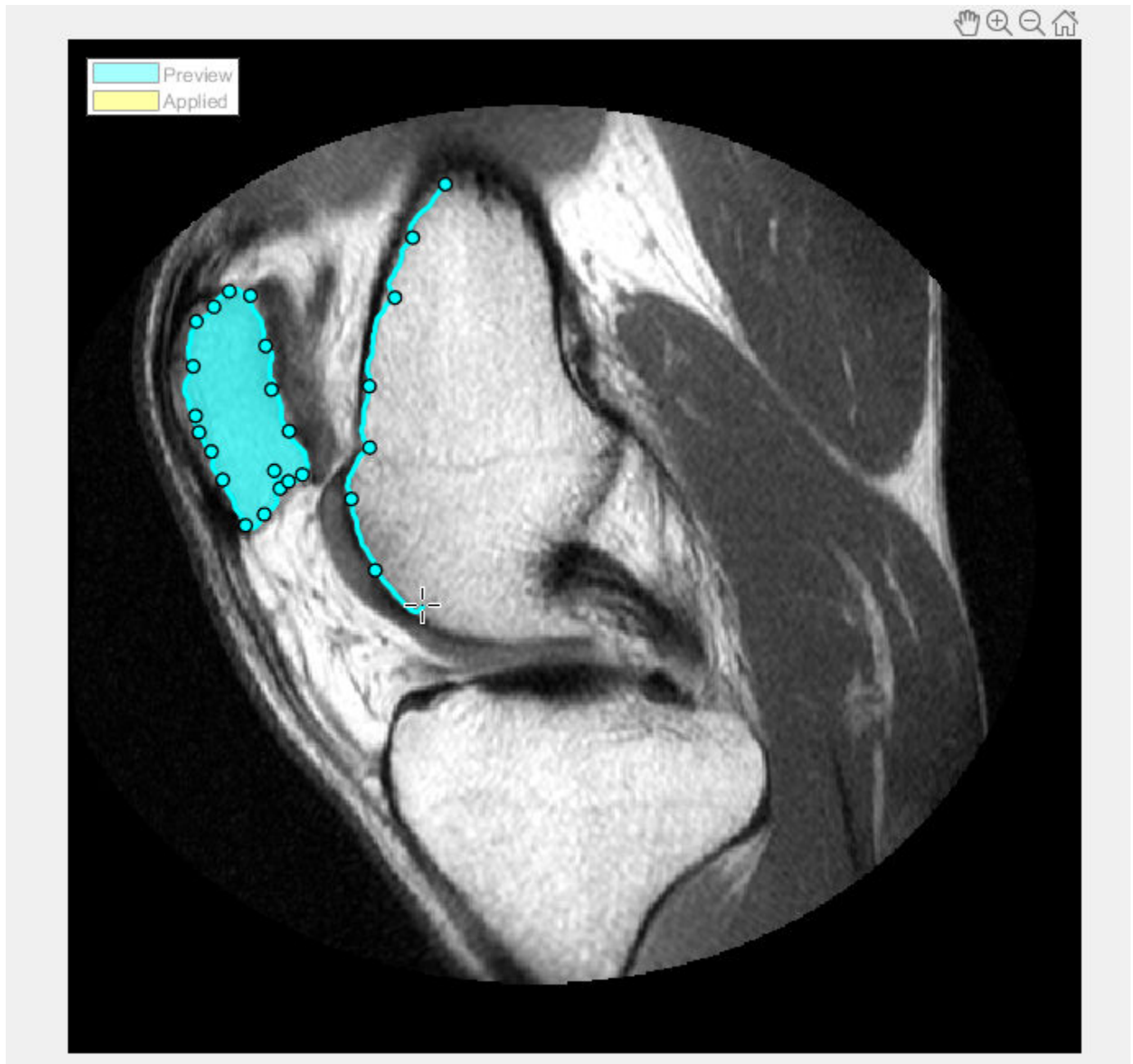
Continue drawing shapes until all the areas you want to segment are identified. To save the regions your have drawn, Click **Apply** (their color changes to yellow). To return to the Segmentation tab, click **Close ROI.**
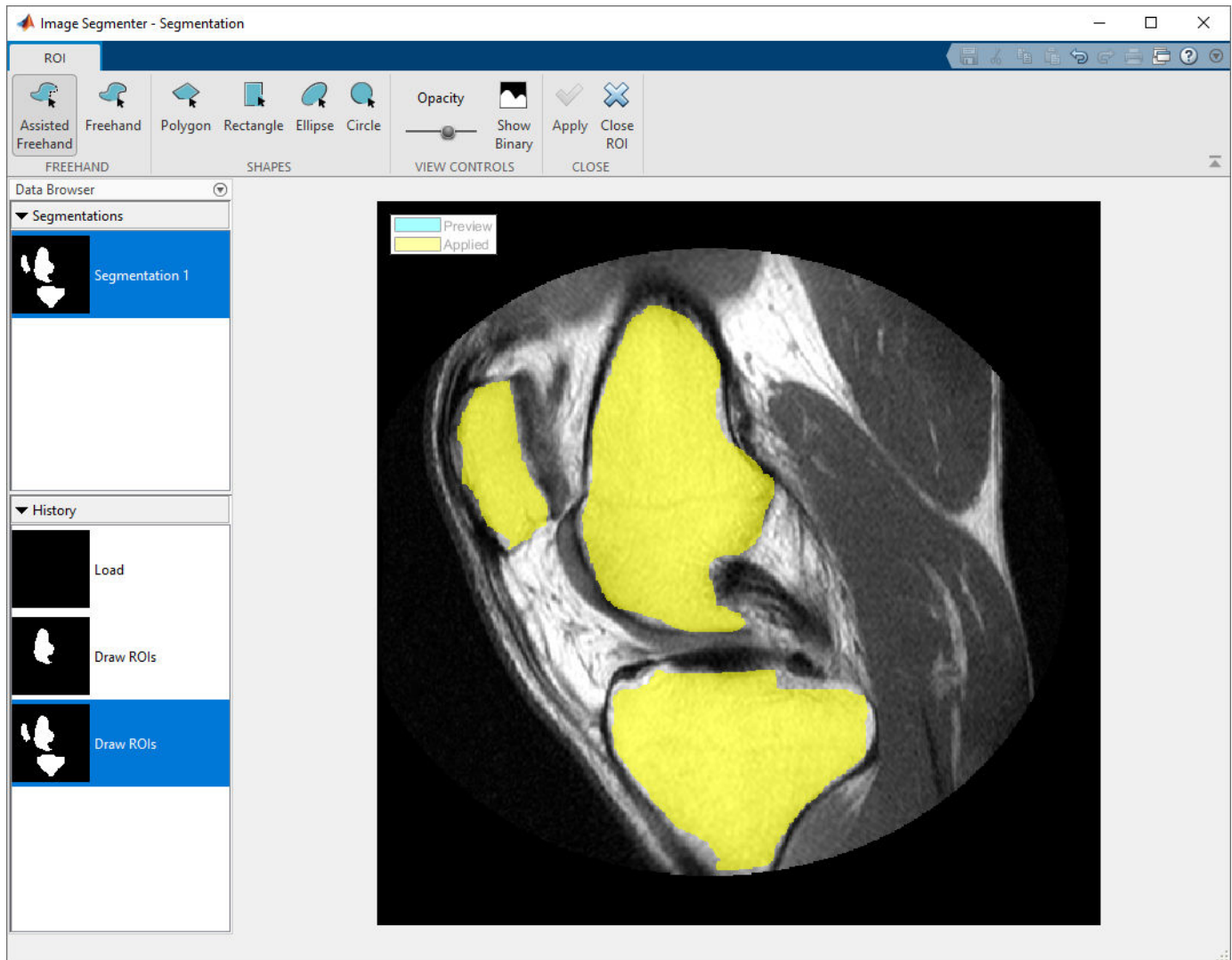
To view the mask image, click **Show Binary** on the Segmentation tab. To refine the mask image, use the tools in the Refine Mask section of **Image Segmenter** app toolstrip, such as **Clear Borders** or **Fill Holes**. When you are done, click **Export** to save the mask image to the workspace.
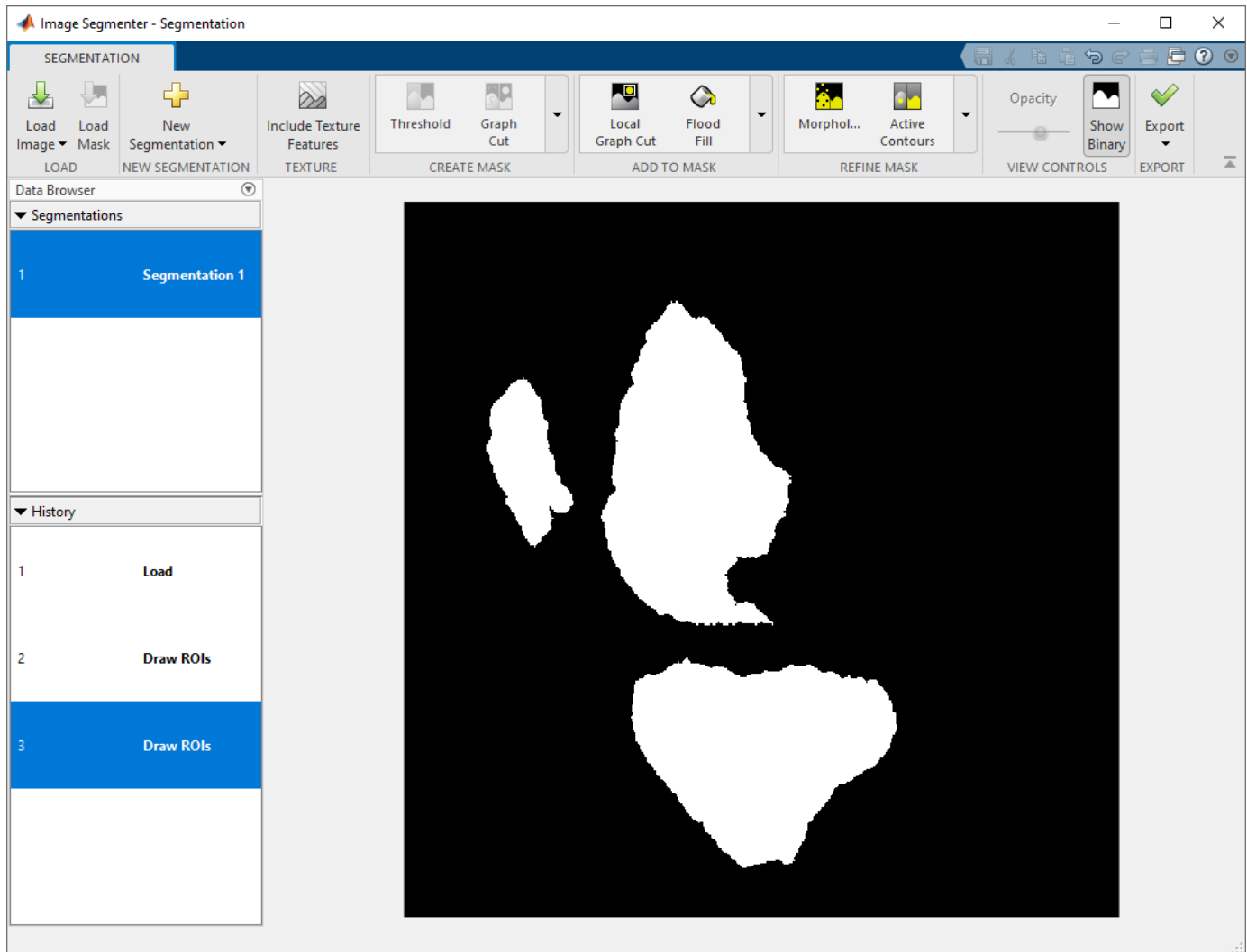
See Also
Image Segmenter

Related Examples

- "Getting Started with Image Segmenter App" on page 13-68

Segment Image Using Active Contours in Image Segmenter

This example shows how to segment an image in the **Image Segmenter** app by using active contours (also called *snakes*). Active contours is an automatic, iterative method where you mark locations in the image by drawing regions (called a *seed mask*). Active contours grows (or shrinks) these seed shapes to fill the borders of the region in the image. The accuracy of this initial seed mask can impact the final result. You can also use the **Include Texture Features** option with Active Contours.

The **Image Segmenter** app offers many different segmentation methods and using the app can be an iterative process. You might try several different methods until you achieve the results you want.

Load an Image in the Image Segmenter

Open the **Image Segmenter** app and load an image to be segmented. The app can open any file that can be read by `imread`.

For this example, read an image into the workspace. This example uses an MRI image of a knee. The goal is to create a mask image that segments the bone from the soft tissue in the image.

```
I = dicomread('knee1');
knee = mat2gray(I);
```

From the MATLAB® toolstrip, open the **Image Segmenter** app. On the **Apps** tab, in the **Image**

Processing and Computer Vision section**,** click **Image Segmenter** .

On the app toolstrip, click **Load**, and then select **Load Image from Workspace**. In the Import from Workspace dialog box, select the image you read into the workspace. The **Image Segmenter** app displays the image you selected.

You can also open the image in the Image Segmenter app using the `imageSegmenter` command, as follows:

```
imageSegmenter(knee);
```

After you load an image, you can optionally load an existing binary mask. For example, you might have previously created a mask of an RGB image in the **Color Thresholder** app and you want to refine the segmentation. To load an existing mask, click **Load Mask**. The segmentation mask image must be a logical image of the same size as the image you are segmenting.

Use Active Contours to Segment an Image

To segment an image using Active Contours, you must first create a rough estimation of the segmentation. For example, you can use the ROI tools to create a rough segmentation of the image (see "Segment Image By Drawing Regions Using Image Segmenter" on page 13-75). You could also load an existing binary mask image.

For this example, use the ROI tools to create seed shapes in the areas you want to segment. When you are done drawing the regions, click **Apply** and then click **Close ROI** to return to the Segmentation tab.
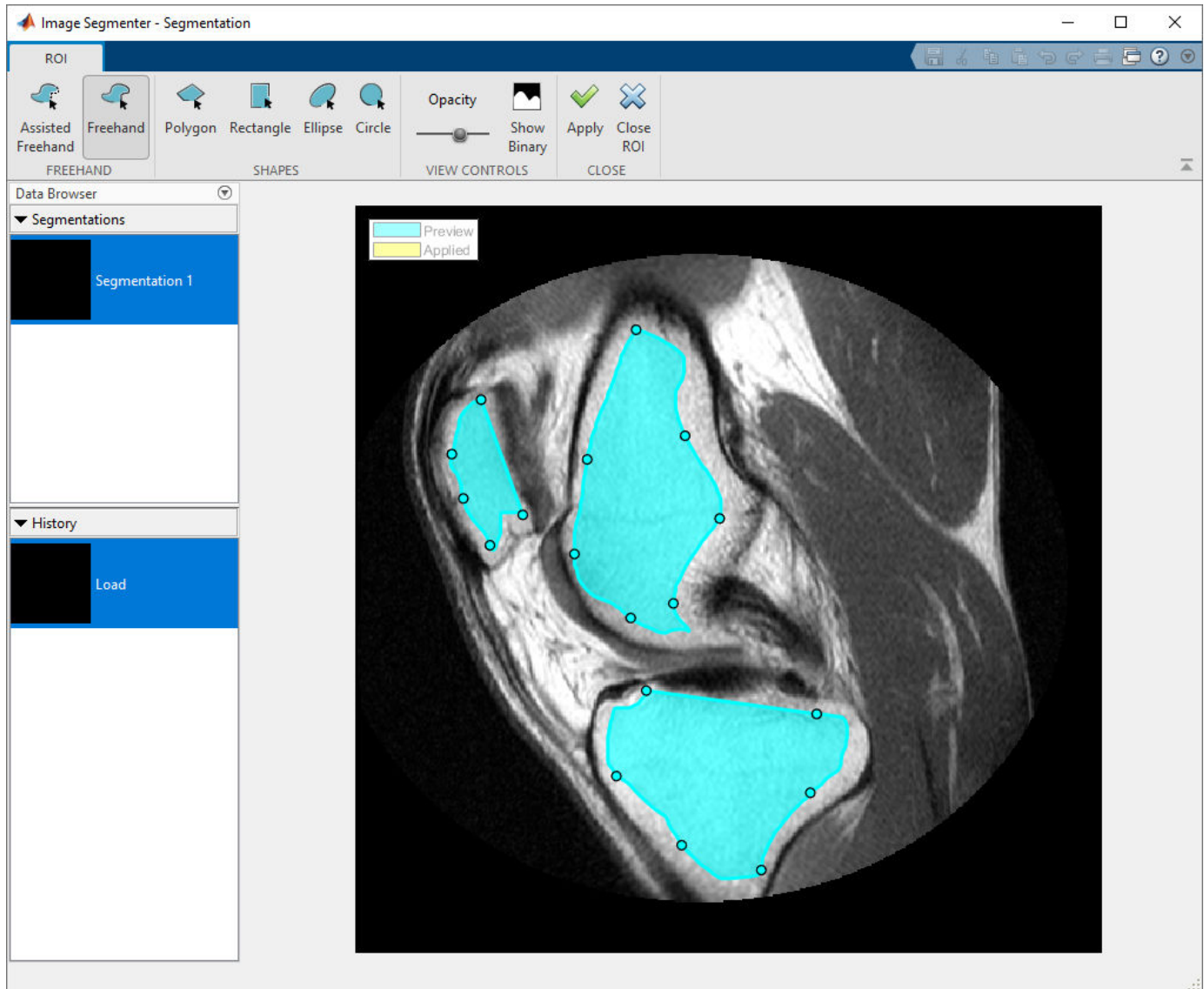
On the Segmentation tab, in the Refine Mask section of the toolstrip and click the **Active Contours**. The **Image Segmenter** app opens the Active Contours tab.

To use active contours, click **Evolve**. The app starts performing iterations to grow the seed masks 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 well 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.
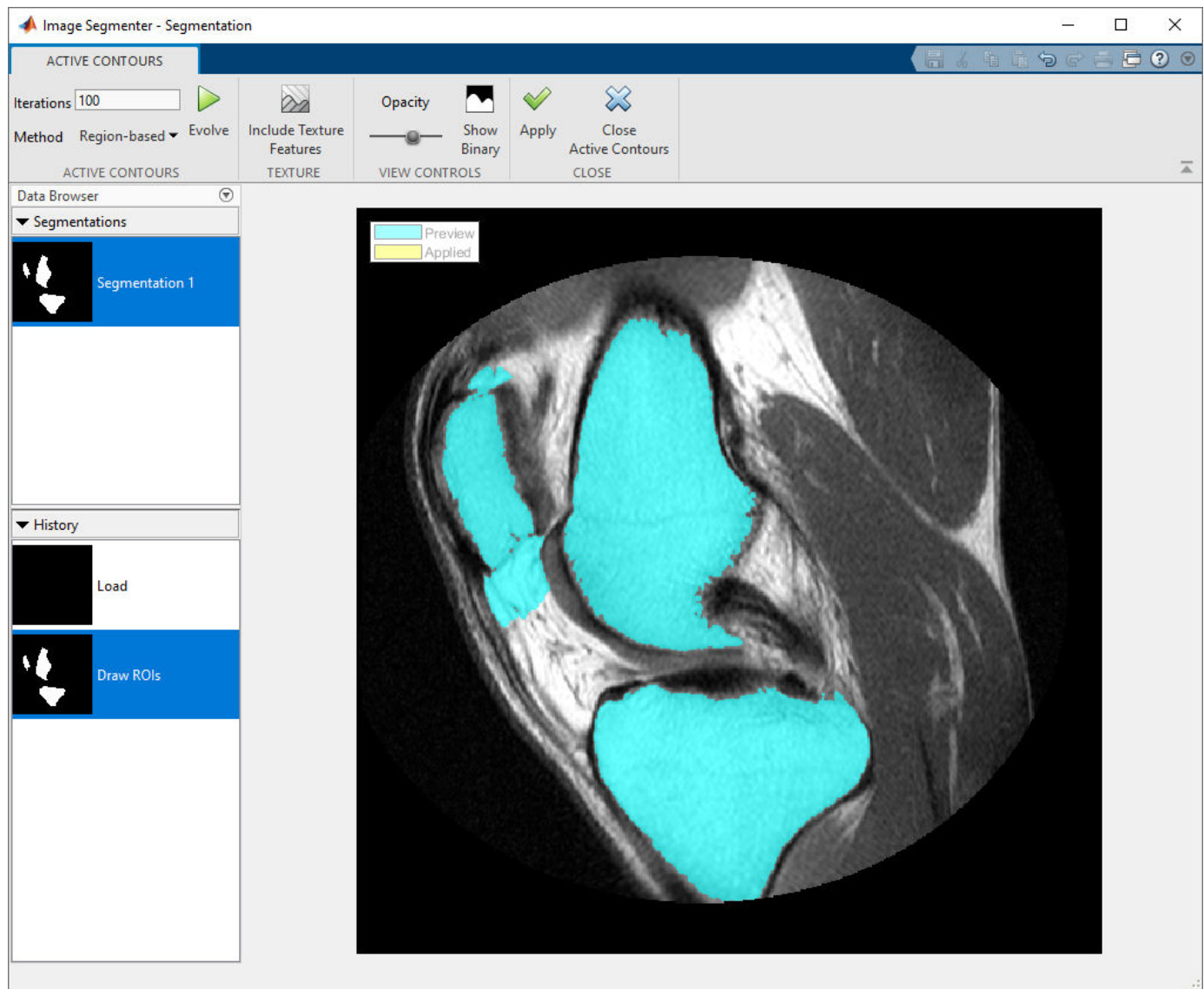
One way to get a better segmentation is to repeat active contours, reducing the number of iterations. Change the number of iterations in the iterations box, specifying 35, and click **Evolve** again. This time, the segmentation does not bleed into the background.
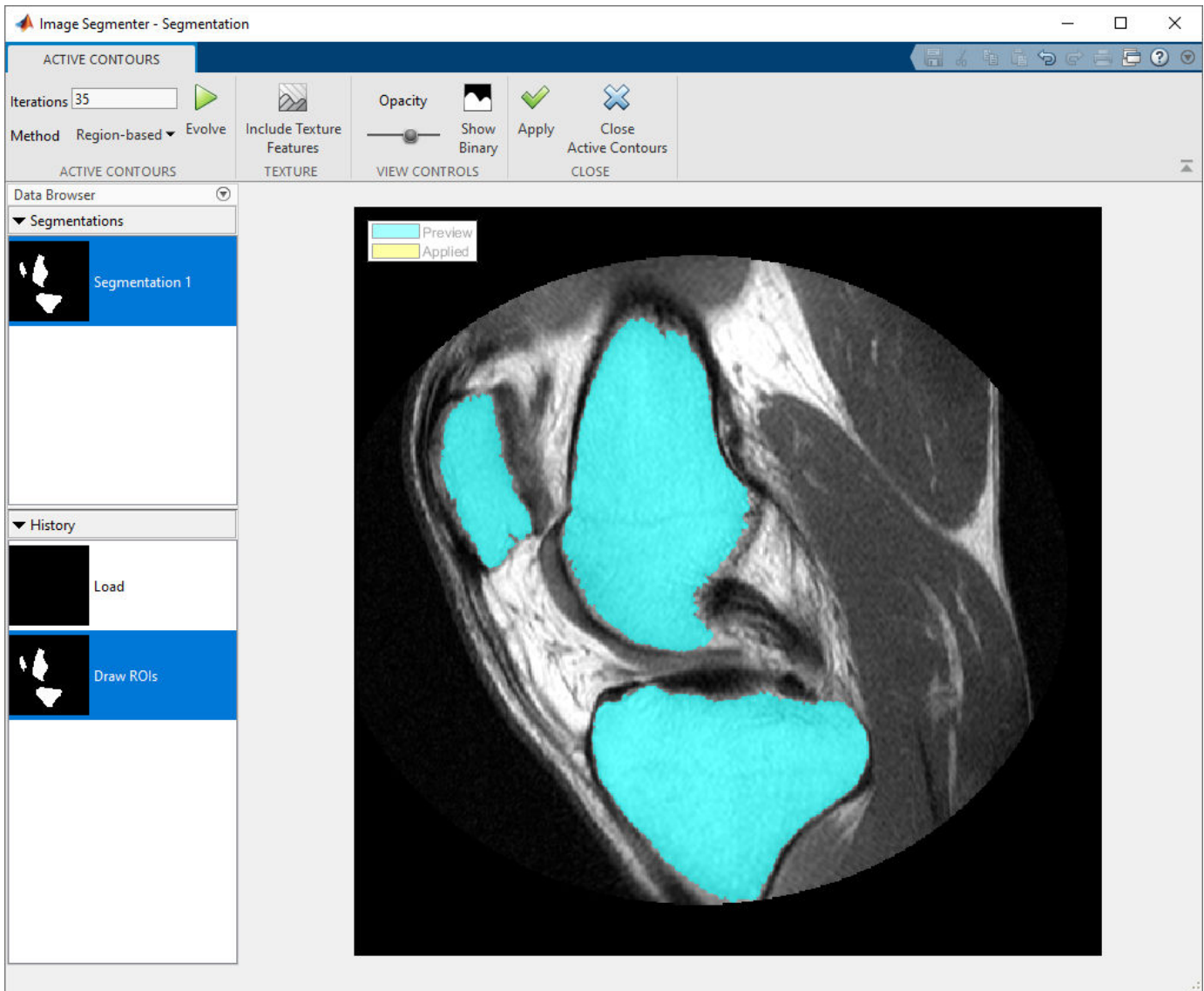
To save the segmentation, click **Apply**. To return to the Segmentation tab, click **Close Active Contours**.

To view the mask image, click **Show Binary** on the Segmentation tab. You can use other tools in the **Image Segmenter app** to refine the mask image, such as **Clear Borders** or **Fill Holes**. To save the mask image to the workspace, click **Export** .
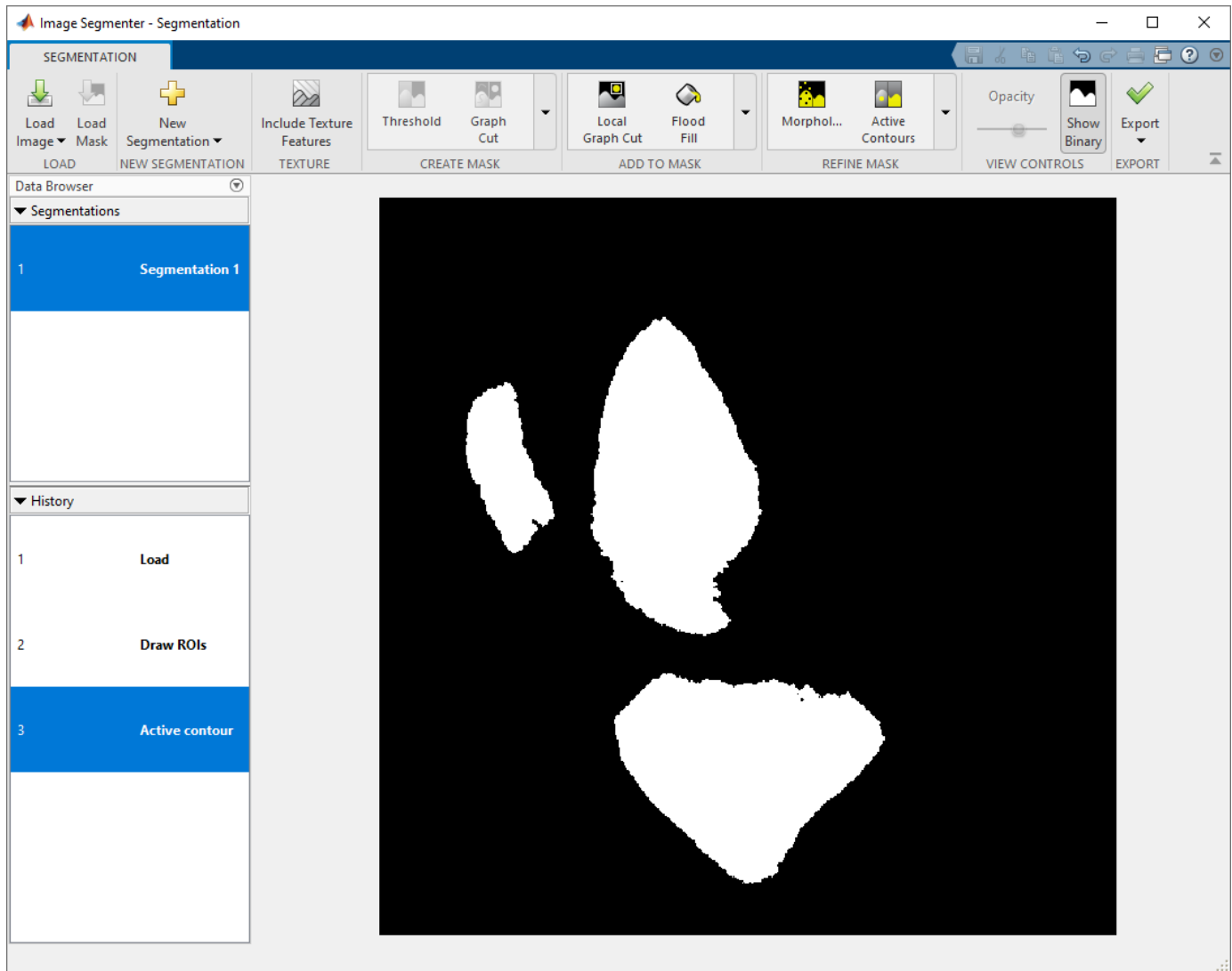
See Also

Image Segmenter | `activecontour`

Related Examples

*   "Getting Started with Image Segmenter App" on page 13-68

Refine Segmentation Using Morphology in Image Segmenter

This example shows how to use the capabilities of the **Image Segmenter** app to polish the appearance of the mask image you created with the app. The Image Segmenter app includes several morphological operations that you can use to fix small imperfections in the mask image.

This example creates a mask image using hand-drawn ROIs and active contours (see "Segment Image Using Active Contours in Image Segmenter" on page 13-81).

Load an Image in the Image Segmenter

Open the **Image Segmenter** app and load an image to be segmented. The **Image Segmenter** can open any file that can be read by `imread`.

For this example, first read an image into the workspace. This example uses an MRI image of a knee. The goal is to create a mask image that segments the bone from the soft tissue in the image.

```
I = dicomread('knee1');
knee = mat2gray(I);
```

From the MATLAB® toolstrip, open the **Image Segmenter** app. On the **Apps** tab, in the **Image Processing and Computer Vision** section, click **Image Segmenter** .

On the app toolstrip, click **Load**, and then select **Load image from Workspace**. In the Import from Workspace dialog box, select the image you read into the workspace. The **Image Segmenter** app displays the image you selected.

You can also open the image in the **Image Segmenter** app using the `imageSegmenter` command, as follows:

```
imageSegmenter(knee);
```

After you load an image, you can optionally load an existing binary mask. For example, you might have previously created a mask by drawing ROIs. To load an existing mask, click **Load Mask**. The segmentation mask image must be a logical image of the same size as the image you are segmenting.

Create the Mask Image

Create a rough segmentation of the image using ROI drawing tools. Use active contours to finish the segmentation. For more details on this process, see "Segment Image Using Active Contours in Image Segmenter" on page 13-81.

After finishing the segmentation, click **Show Binary** on the Segmentation tab to view the mask image. Upon close examination, you can see several small holes in the mask image.

Small holes
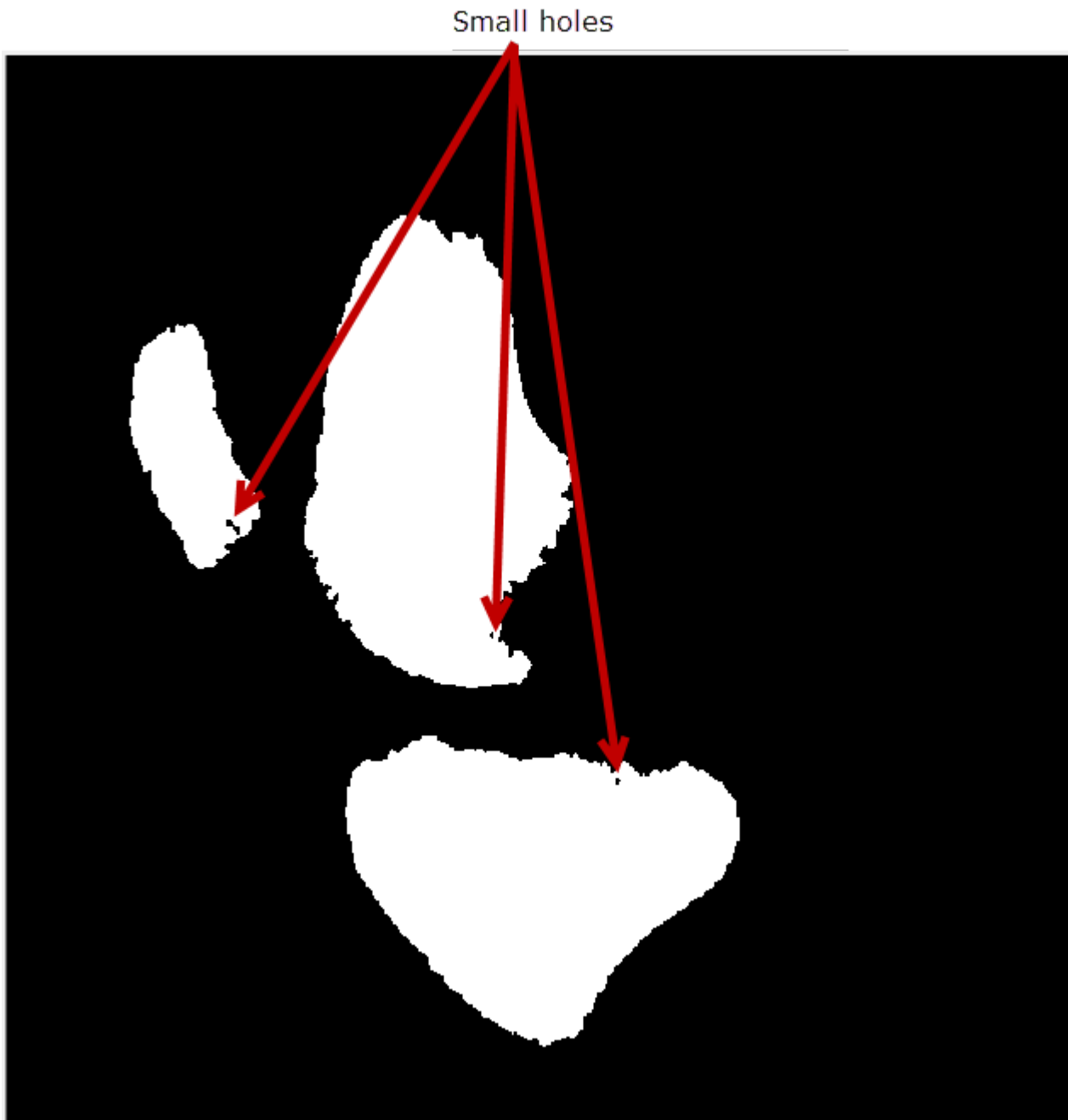
The Image Segmenter includes morphological tools to refine the binary mask. Expand the Refine Mask section of the app toolstrip and click **Fill Holes**.
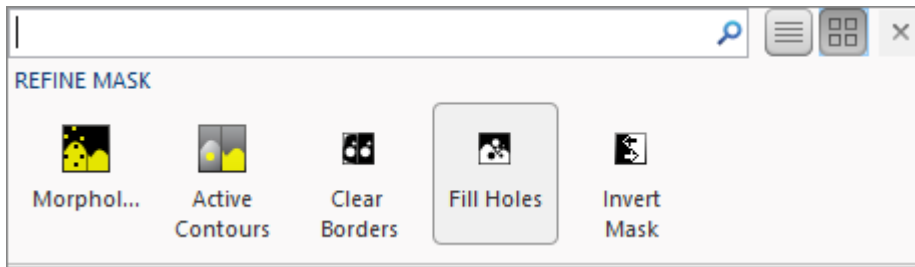
This removes the holes in the binary mask.

To save the binary mask, click **Export** and select **Export Images**.

See Also
Image Segmenter

Related Examples
- "Getting Started with Image Segmenter App" on page 13-68

Segment Image 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 technique 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 "Segment Image Using Local Graph Cut (Grabcut) in Image Segmenter" on page 13-101.

Load Image into the Image Segmenter App

Read an image into the workspace. For this example, read the sample image `baby.png` into the workspace.

```
b = imread('baby.jpg');
```

From the MATLAB® toolstrip, open the **Image Segmenter** app. On the **Apps** tab, in the Image

Processing and Computer Vision section, click **Image Segmenter** .
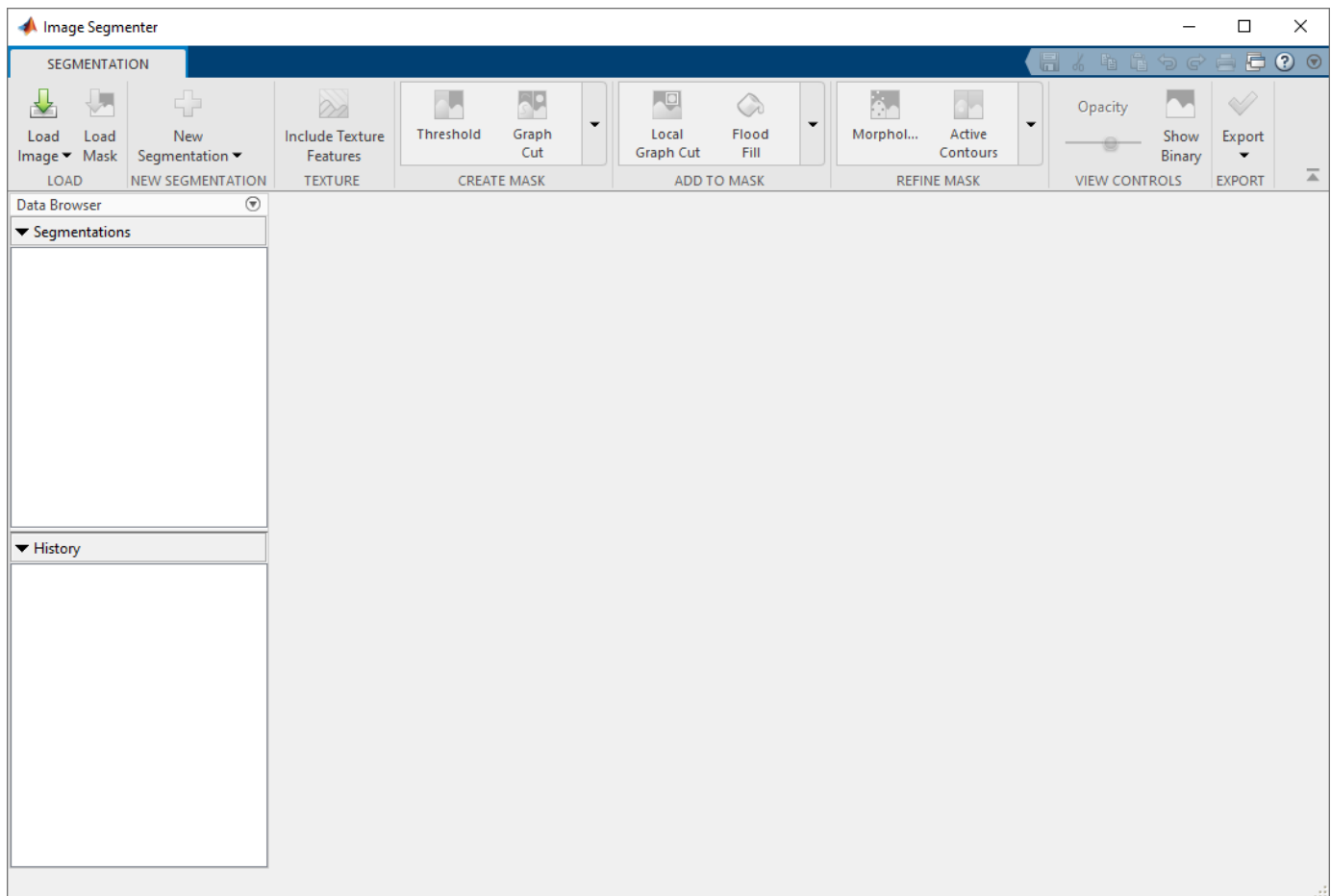
On the app toolstrip, click **Load**, and then select **Load Image from Workspace**. In the Import from Workspace dialog box, select the image you read into the workspace. The **Image Segmenter** app displays the image you selected.
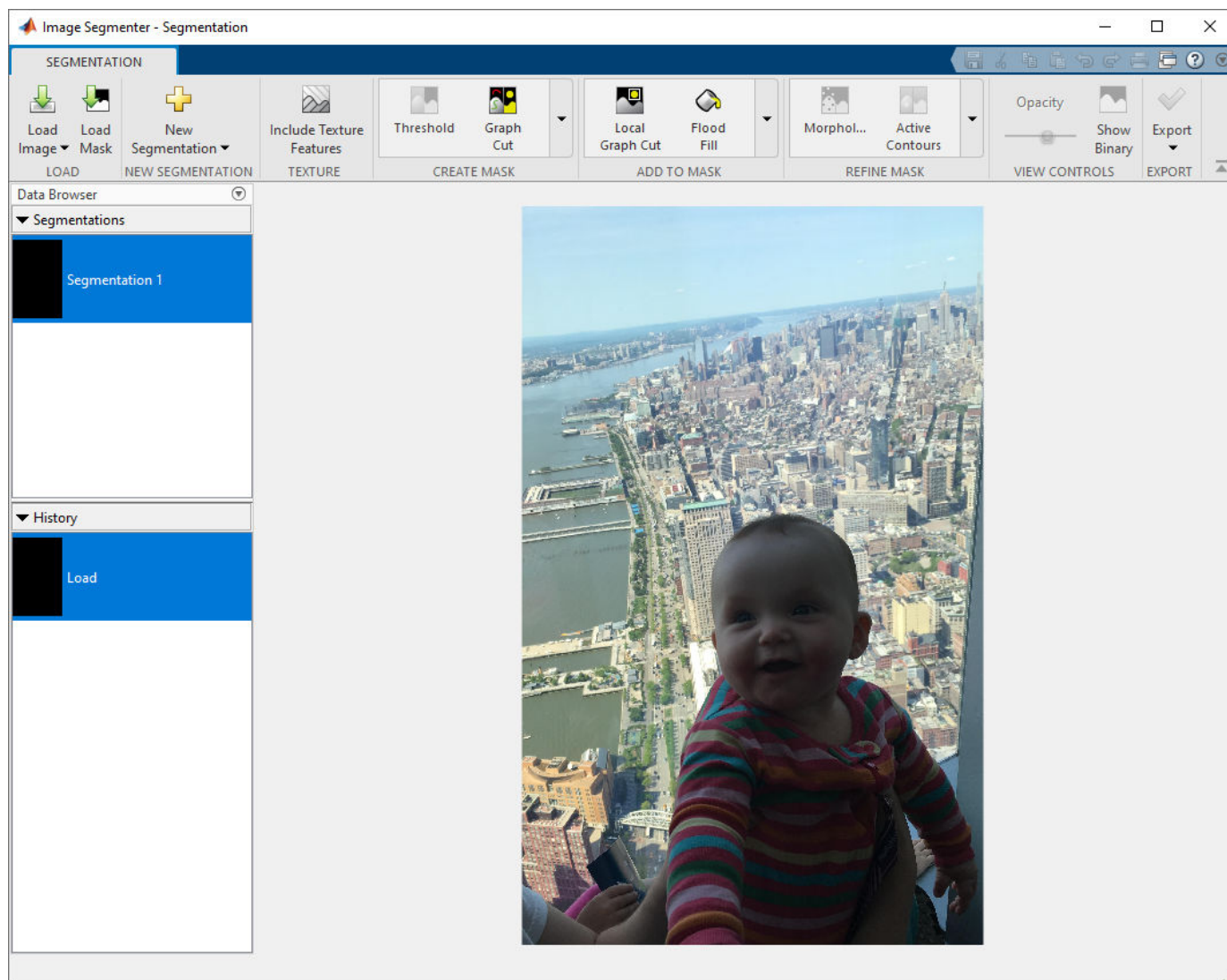
You can also open the app using the `imageSegmenter` command, specifying the image:

`imageSegmenter(b);`

Use Graph Cut to Segment Image

On the Image Segmenter app toolstrip, select **Graph Cut**.
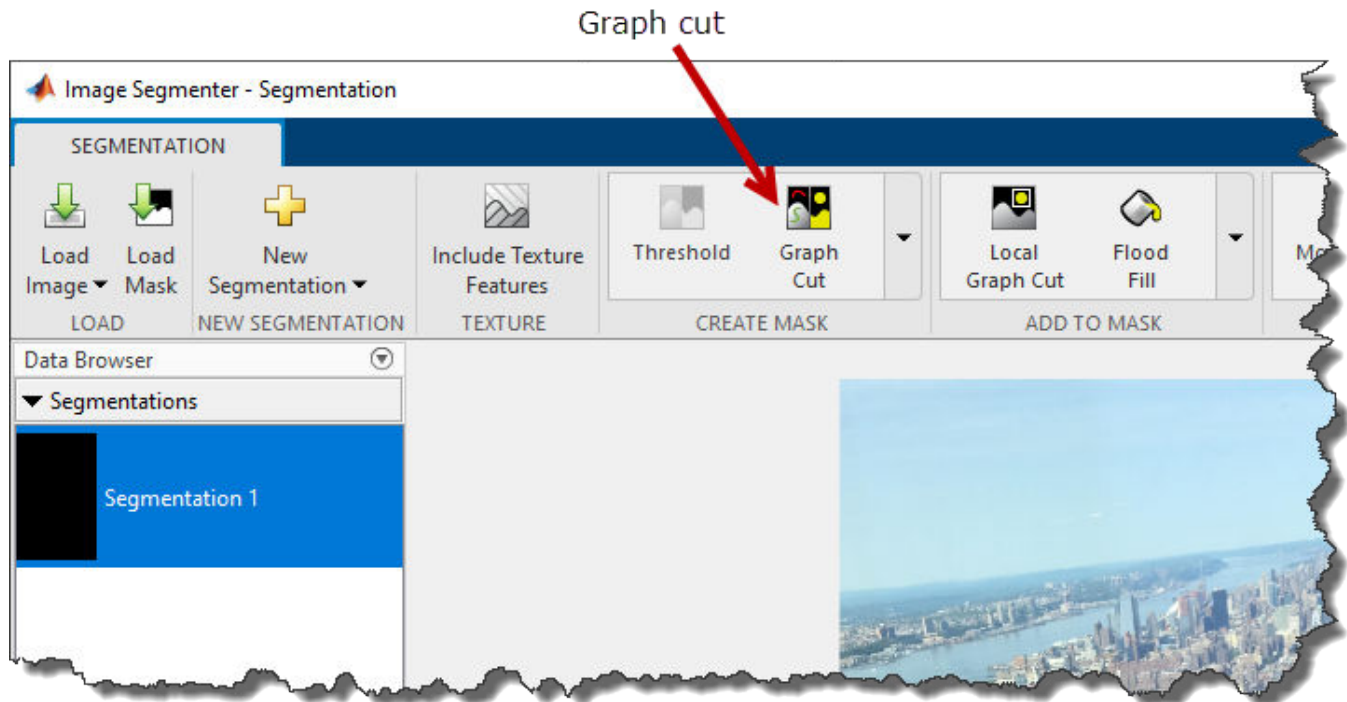
Graph cut

The Image Segmenter opens a new tab for Graph Cut segmentation. As a first step in Graph Cut segmentation, mark the elements of the image that you want to be in the foreground. When the Image Segmenter opens the Graph Cut tab, it preselects the **Mark Foreground** option. To mark an object as foreground, draw 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 want to segment. 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 want to start over, click **Clear Markings**.
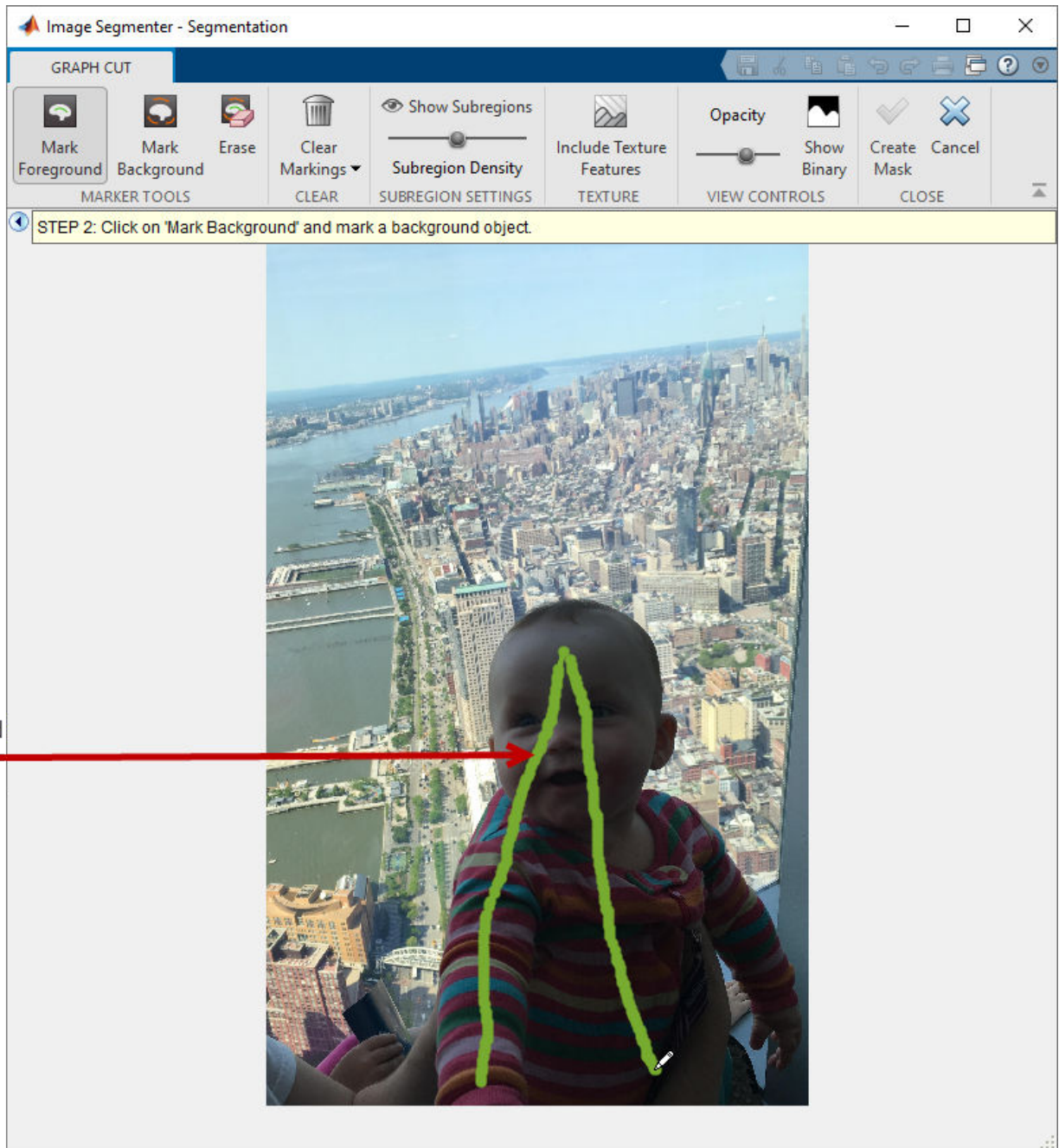
Foreground
Scribbles

Next, click **Mark Background** and draw scribbles to mark the elements of the image you want to be the background. When you finish drawing the lines, the Image Segmenter immediately performs the segmentation (shown in blue).
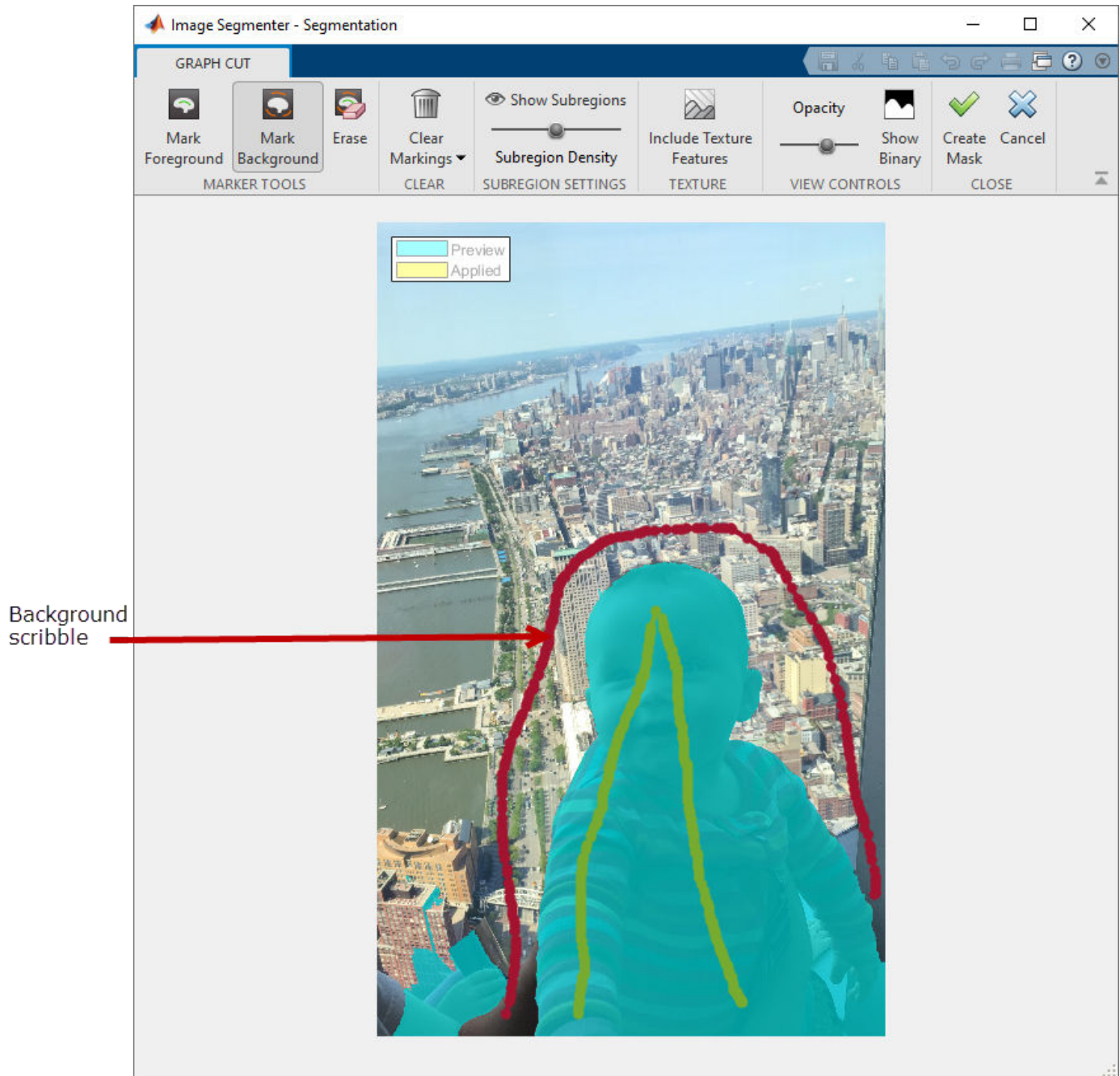
To refine the segmentation, continue drawing foreground and background lines. For example, there are several areas near the bottom of the image that need to be removed from the foreground. To fix these problems, draw additional background lines on these parts of the image.
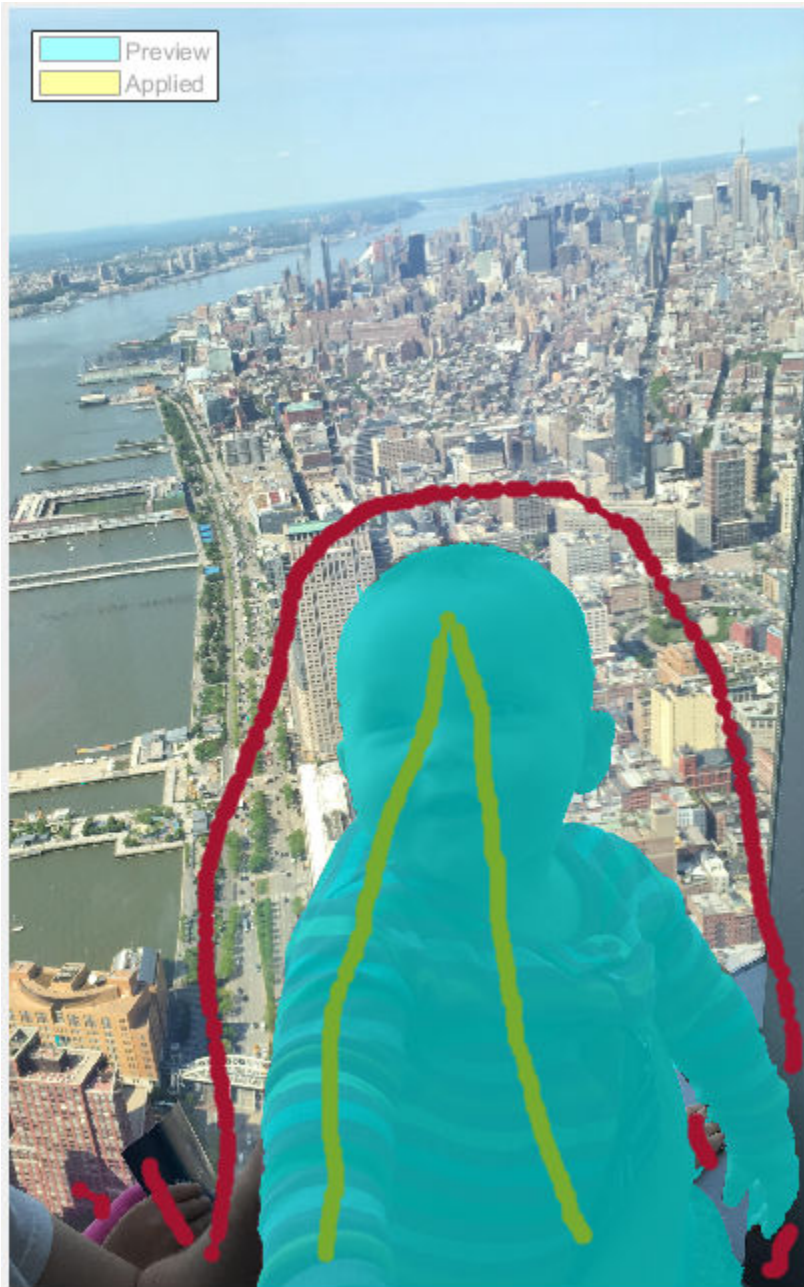
To get a better look at the segmentation, click **Show Binary**.

When you are satisfied with the segmentation, click **Create Mask** in the toolstrip on the Graph Cut tab. The app closes the Graph Cut tab and returns you to the Segmentation tab.

Save the Mask image to the Workspace

When you return to the main Segmentation tab, you can use tools to refine 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 the **Image Segmenter** app used to create the segmentation.
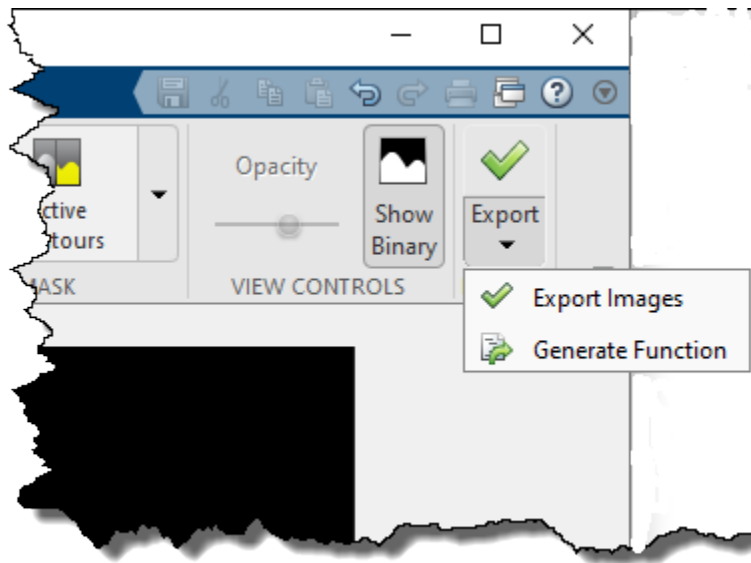
See Also

Image Segmenter | `lazysnapping`

Related Examples

- "Getting Started with Image Segmenter App" on page 13-68

Segment Image Using Local Graph Cut (Grabcut) in Image Segmenter

This example shows how to segment an image using Local Graph Cut (also known as *grabcut*) in the **Image Segmenter** app. Like Graph Cut, 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** app segments the image automatically based on the contents of the ROI.

Then, as with Graph Cut, you 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 "Segment Image Using Graph Cut in Image Segmenter" on page 13-92.

Load Image into the Image Segmenter App

Read an image into the workspace.

```
car = imread('car2.jpg');
```

From the MATLAB® Toolstrip, open the **Image Segmenter** app. On the **Apps** tab, in the **Image Processing and Computer Vision** section, click **Image Segmenter** .
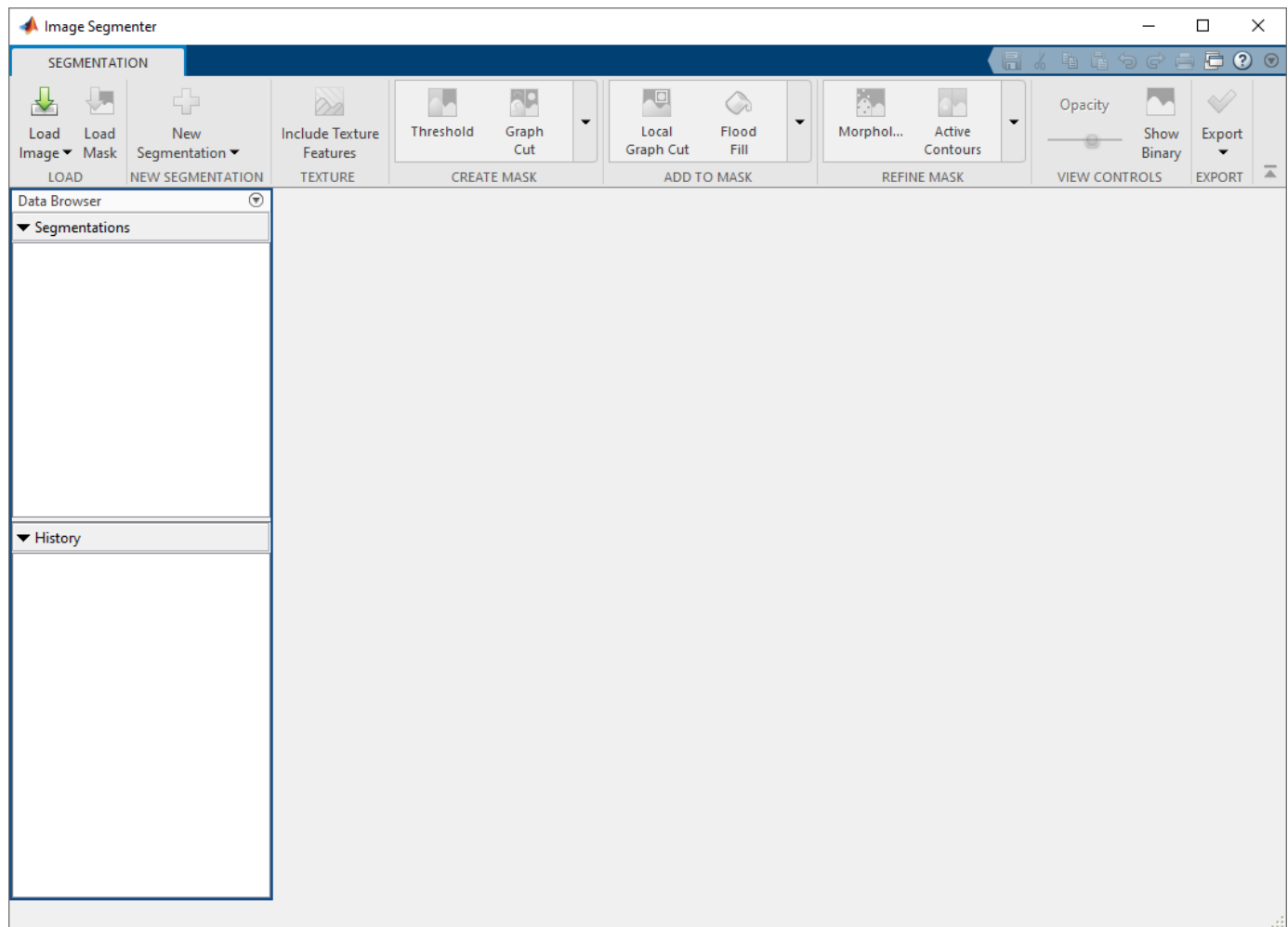
On the app toolstrip, click **Load Image**, and then select **Load Image from Workspace**. In the Import from Workspace dialog box, select the image you read into the workspace. The **Image Segmenter** app displays the image you selected.
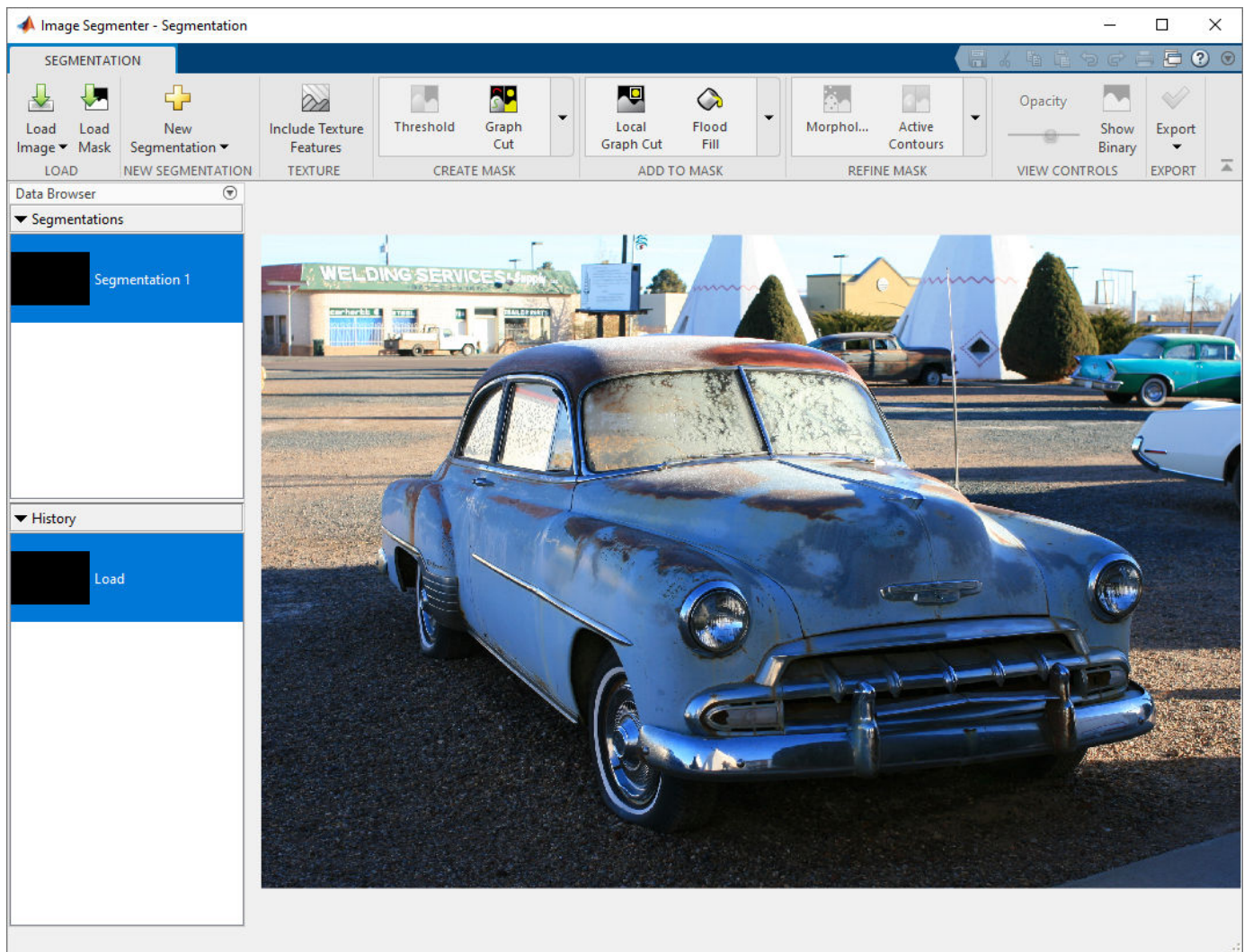
You can also open the app using the `imageSegmenter` command, specifying the name of the image variable.

```
imageSegmenter(b);
```

Use Local Graph Cut (Grabcut) to Segment Image

On the **Image Segmenter** app toolstrip, select **Local Graph Cut**.
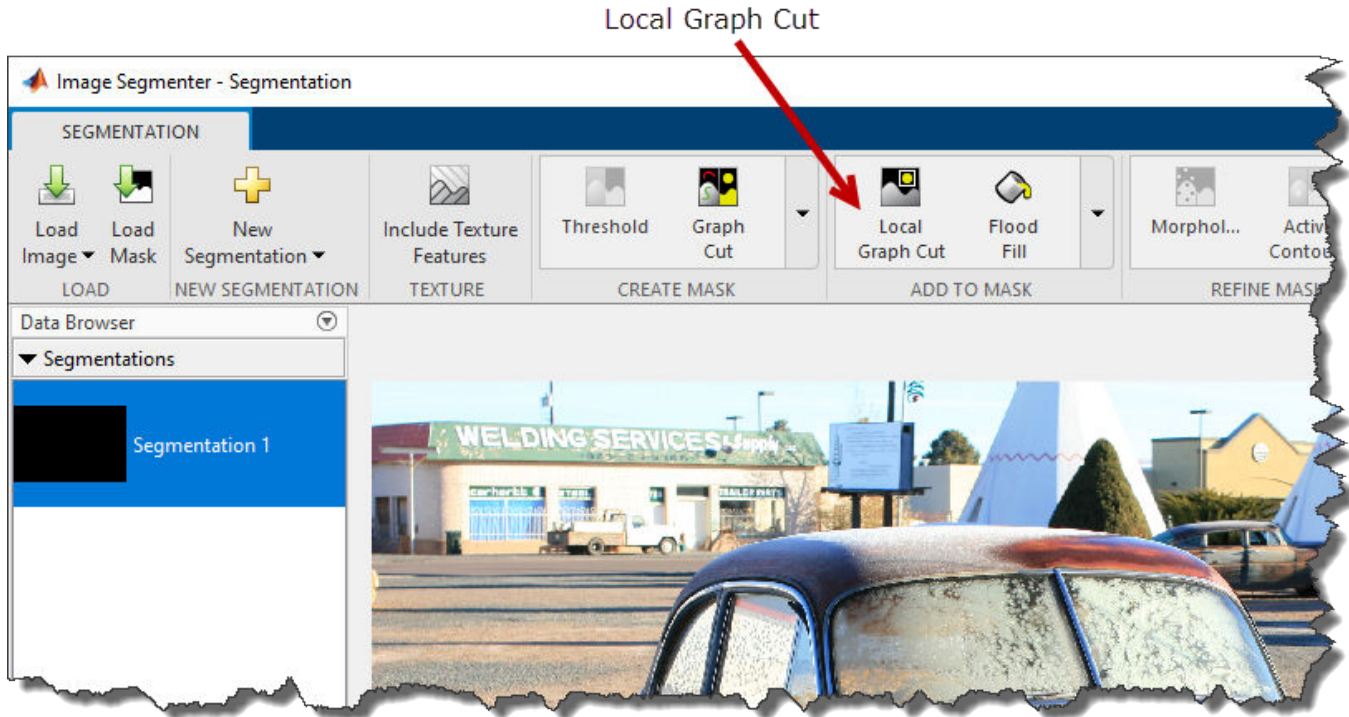
The **Image Segmenter** app opens a new tab for Local Graph Cut segmentation. As a first step in Local Graph Cut segmentation, draw an ROI around the object in the image that you want to segment. When the **Image Segmenter** app opens the Local Graph Cut tab, it preselects the **Draw ROI** button. Position 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 rectangular or polygonal 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, creating a vertex at each click. Double-click to finish the polygon. If you are not satisfied with the shape you drew, you can always edit it. Right-click the ROI and choose **Delete**.

When you finish the ROI, the **Image Segmenter** app automatically segments the object in the ROI. The blue shading indicates the segmented area.

ROI
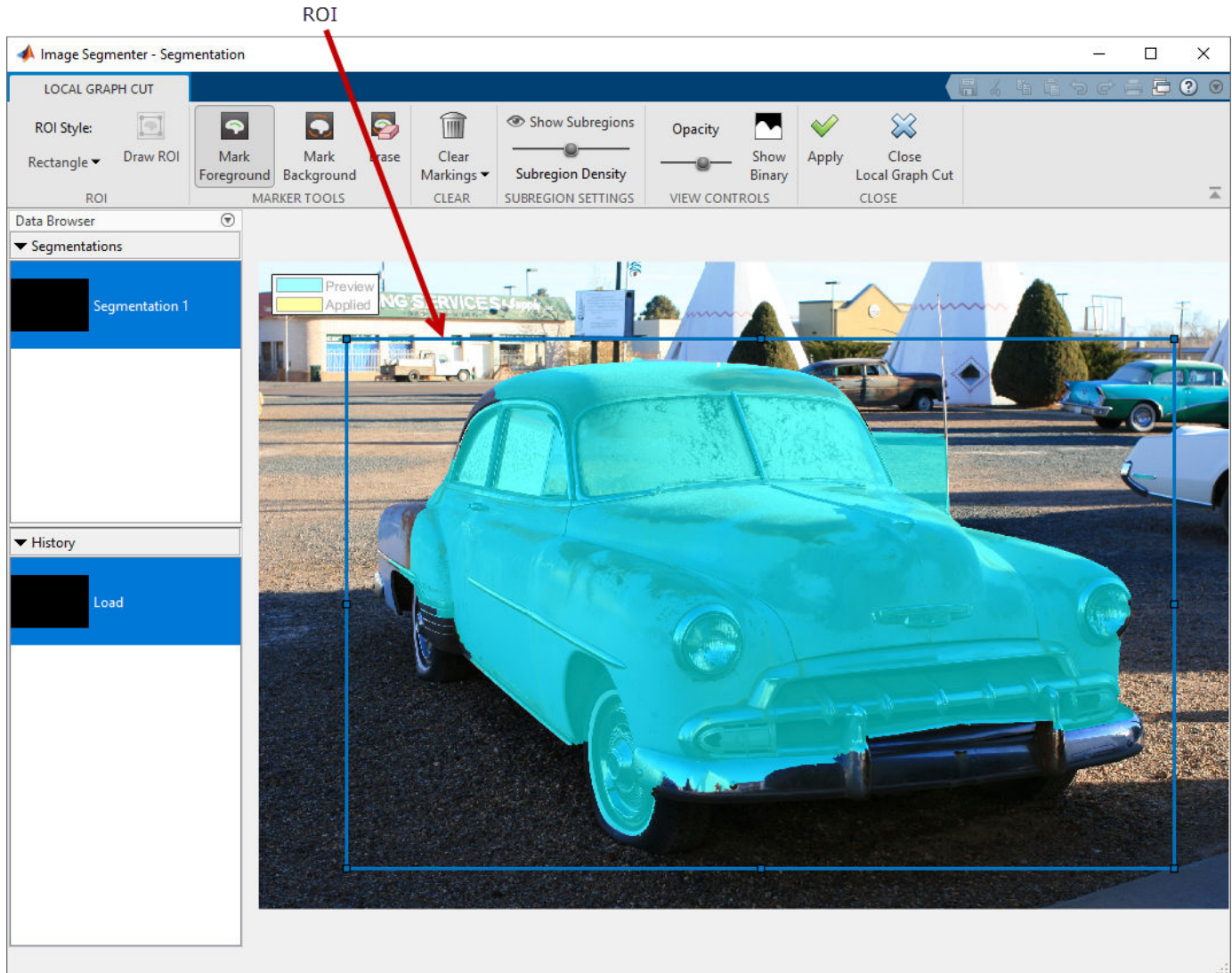
To refine the automatic segmentation, draw lines (scribbles) to mark any parts of the foreground that weren't included in the automatic segmentation. After you draw the ROI, the Image Segmenter selects the **Mark Foreground** button automatically.

Foreground scribbles
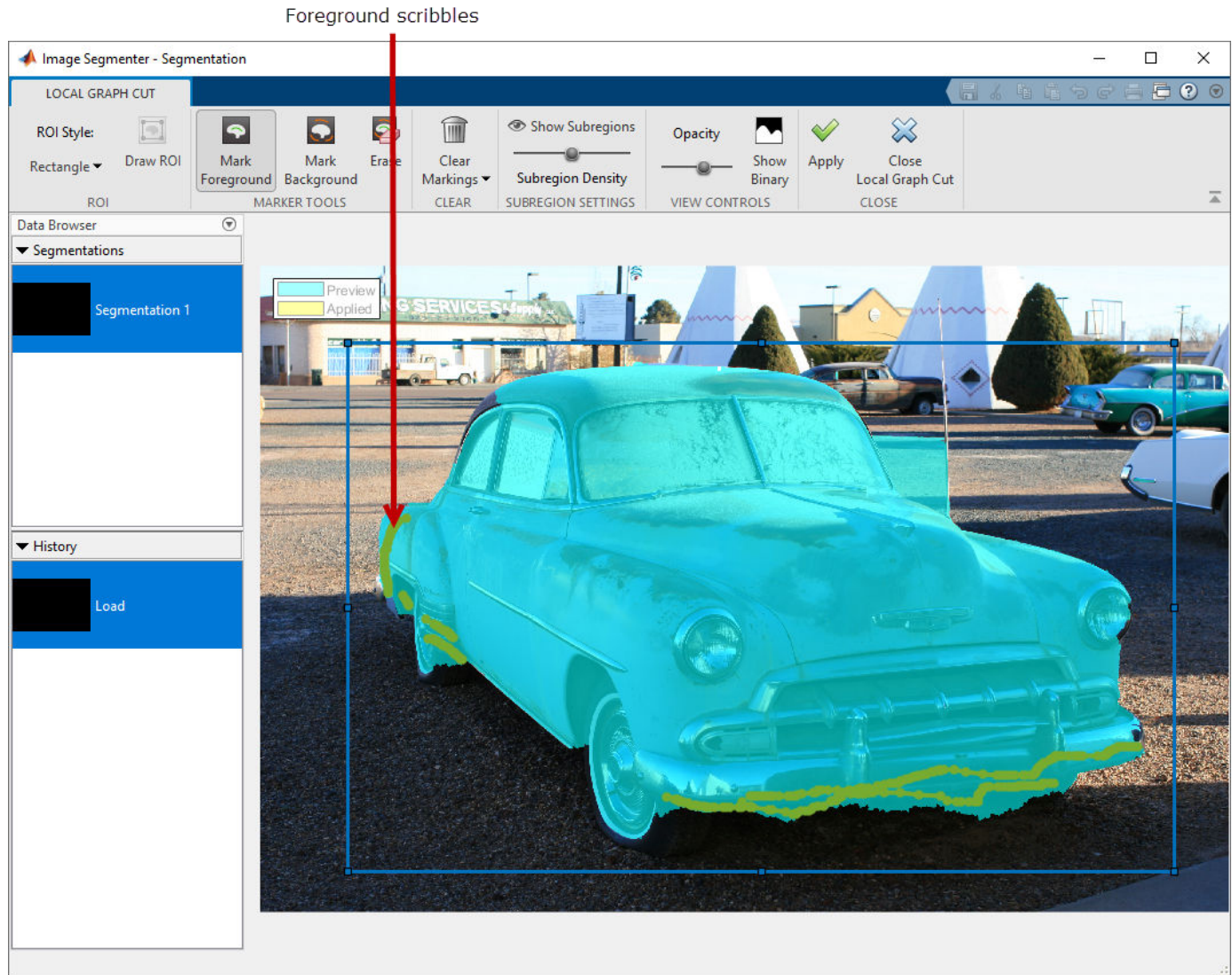
To remove areas from the segmentation that are not part of the foreground, mark those areas as background. Select the **Mark Background** option and draw lines inside the ROI to identify parts of the segmentation that should be in the background.

Background scribbles
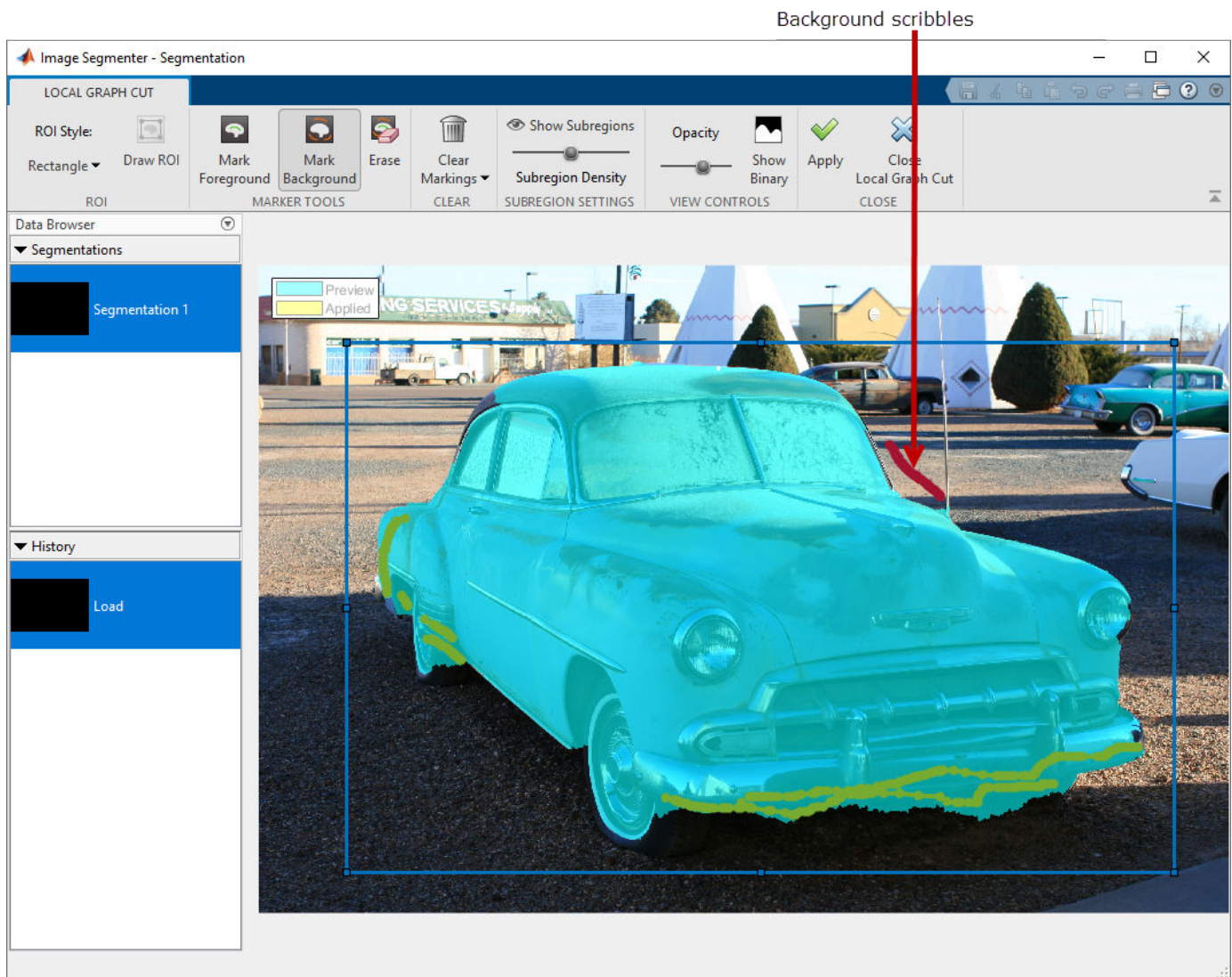
When you are satisfied with the segmentation, click **Apply**. The **Image Segmenter** app changes the color of the segmented part of the image to yellow.
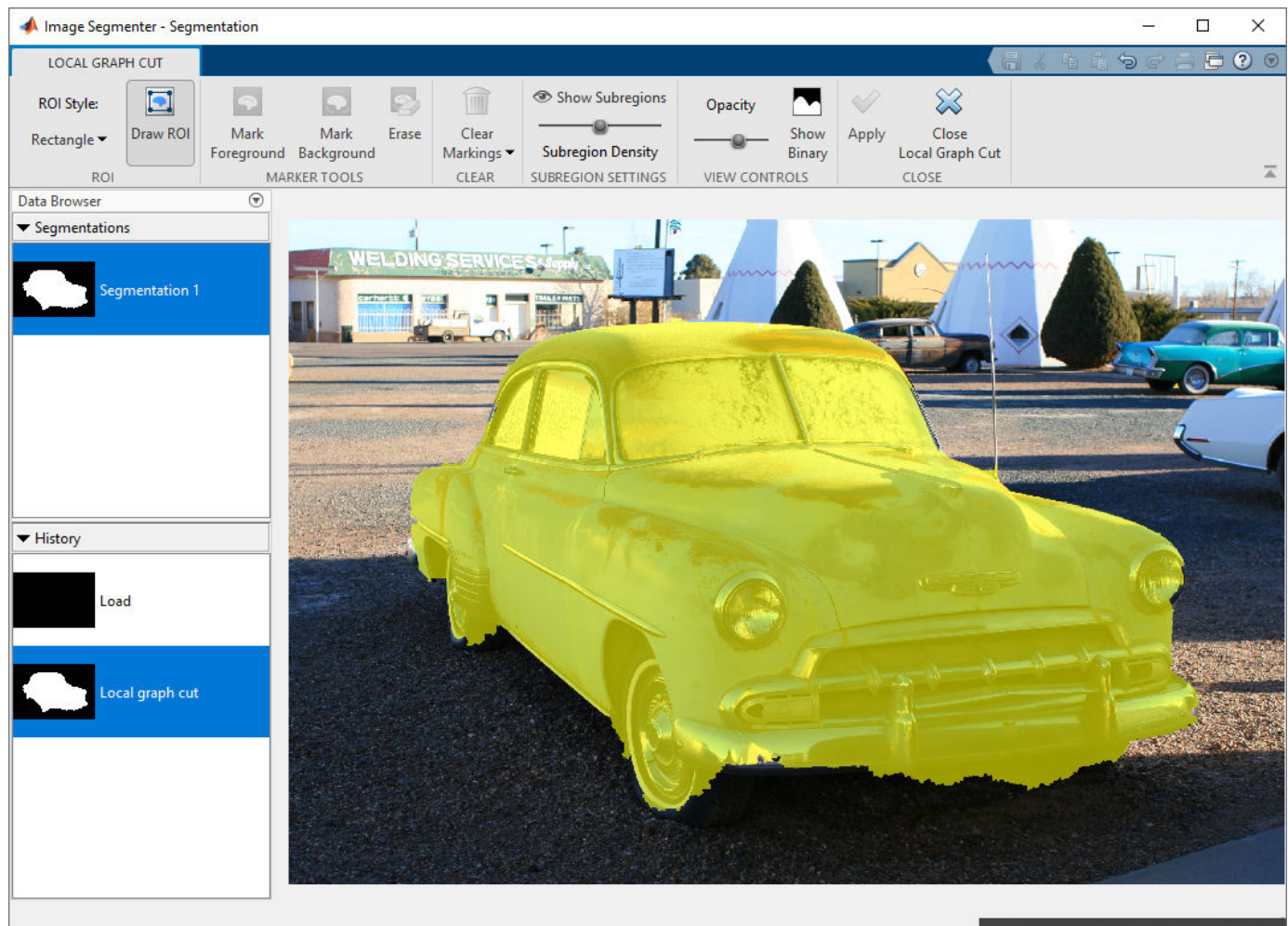
View Binary Image and Save Mask

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**.

Click **Show Binary** to view the mask image.
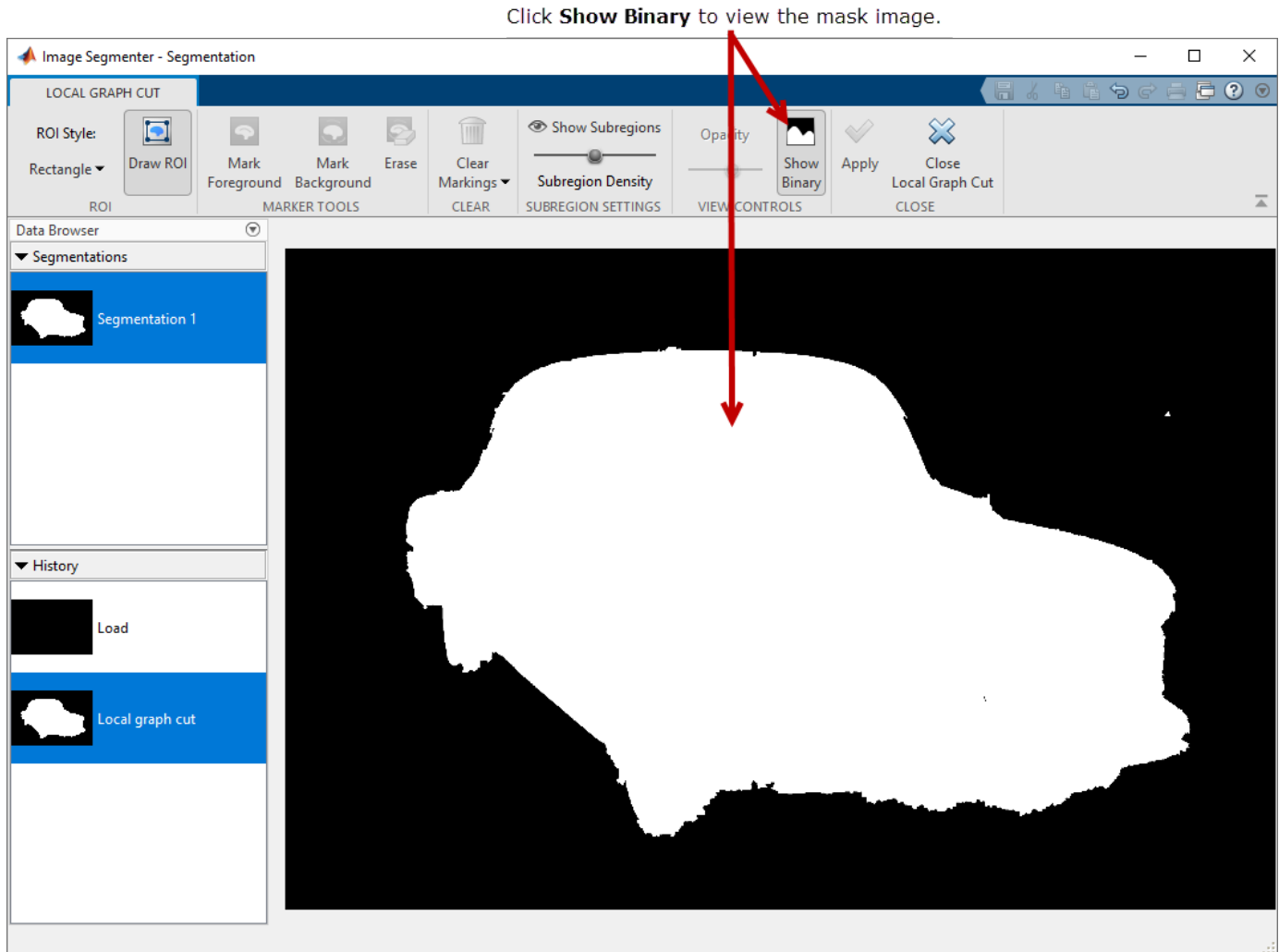
When you are done segmenting the image, you can save the binary mask, using the **Export** option. You can also obtain the code used for the segmentation.

See Also

Image Segmenter | `grabcut`

Related Examples

- "Getting Started with Image Segmenter App" on page 13-68

Segment Image 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.

Load Image into the Image Segmenter App

Read an image into the workspace.

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

From the MATLAB® Toolstrip, open the **Image Segmenter** app. On the **Apps** tab, in the **Image Processing and Computer Vision** section, click **Image Segmenter** .
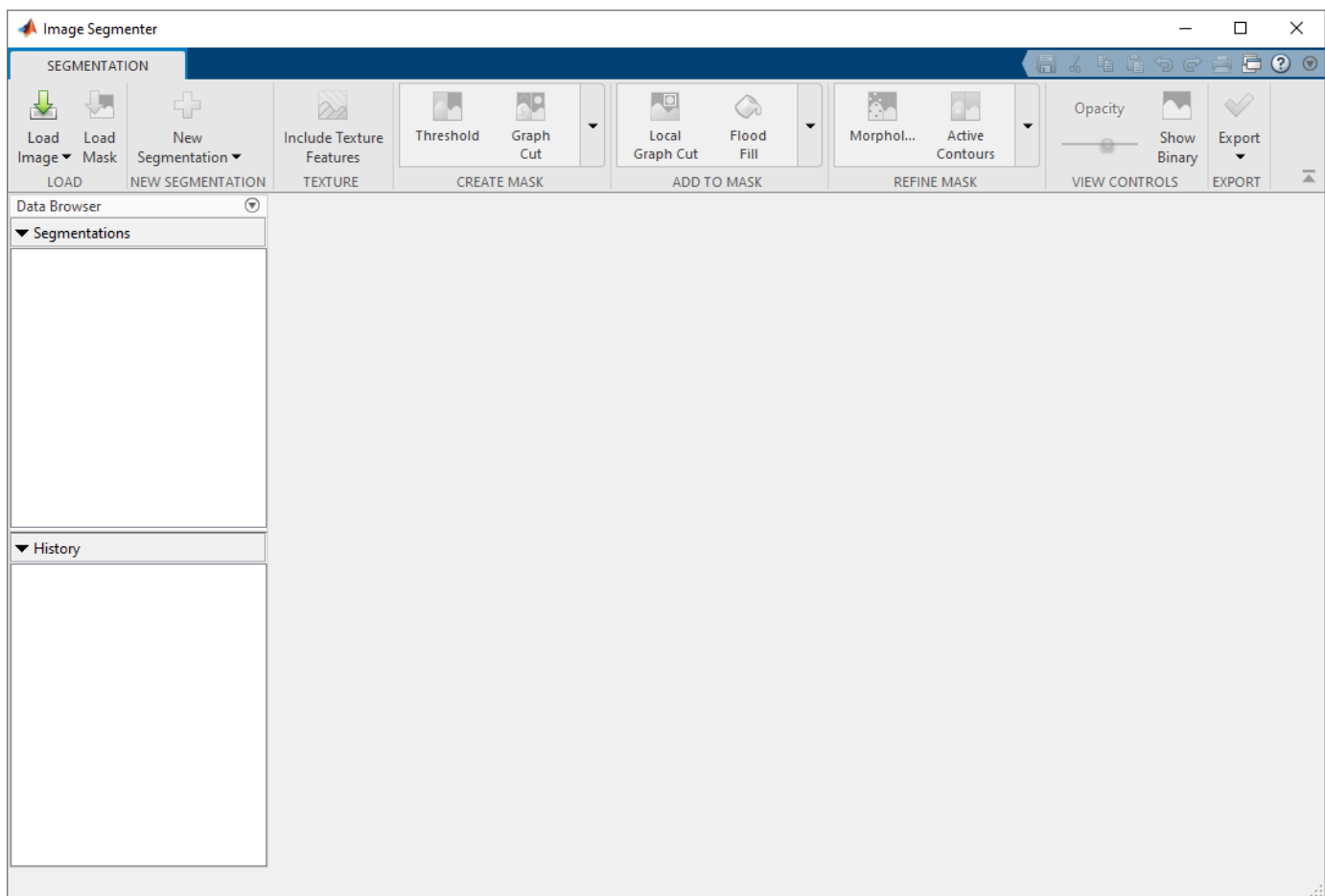
On the app toolstrip, click **Load Image**, and then select **Load Image from Workspace**. In the Import from Workspace dialog box, select the image you read into the workspace. The **Image Segmenter** app displays the image you selected.

You can also open the app using the `imageSegmenter` command, specifying the image:

```
imageSegmenter(coins);
```

Use Find Circles to Segment the Image

On the **Image Segmenter** app toolstrip, expand the Create Mask section and select **Find Circles**.

Expand the Create Mask toolstrip group and click **Find Circles**.
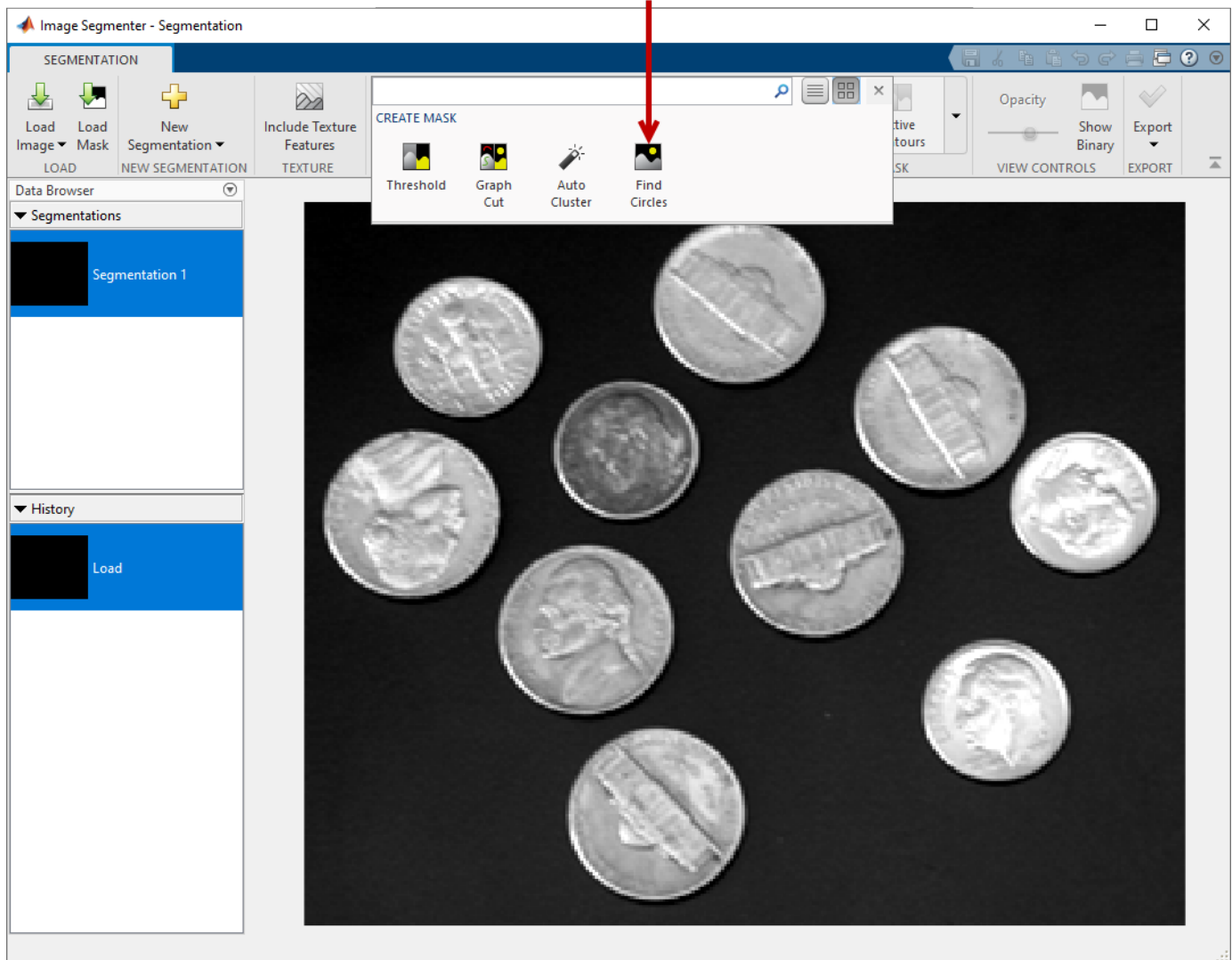
The **Image Segmenter** app opens a new tab for the Find Circles segmentation option.

In the Find Circles tab, first click **Ruler** and measure the diameters of some representative circles in the image to determine the range of sizes. To find circles, you must specify the lower and upper bounds on the diameters. Set the values in the **Min. Diameter** and **Max. Diameter** fields to values that you think include all the objects, 50 and 150.

Measure diameters
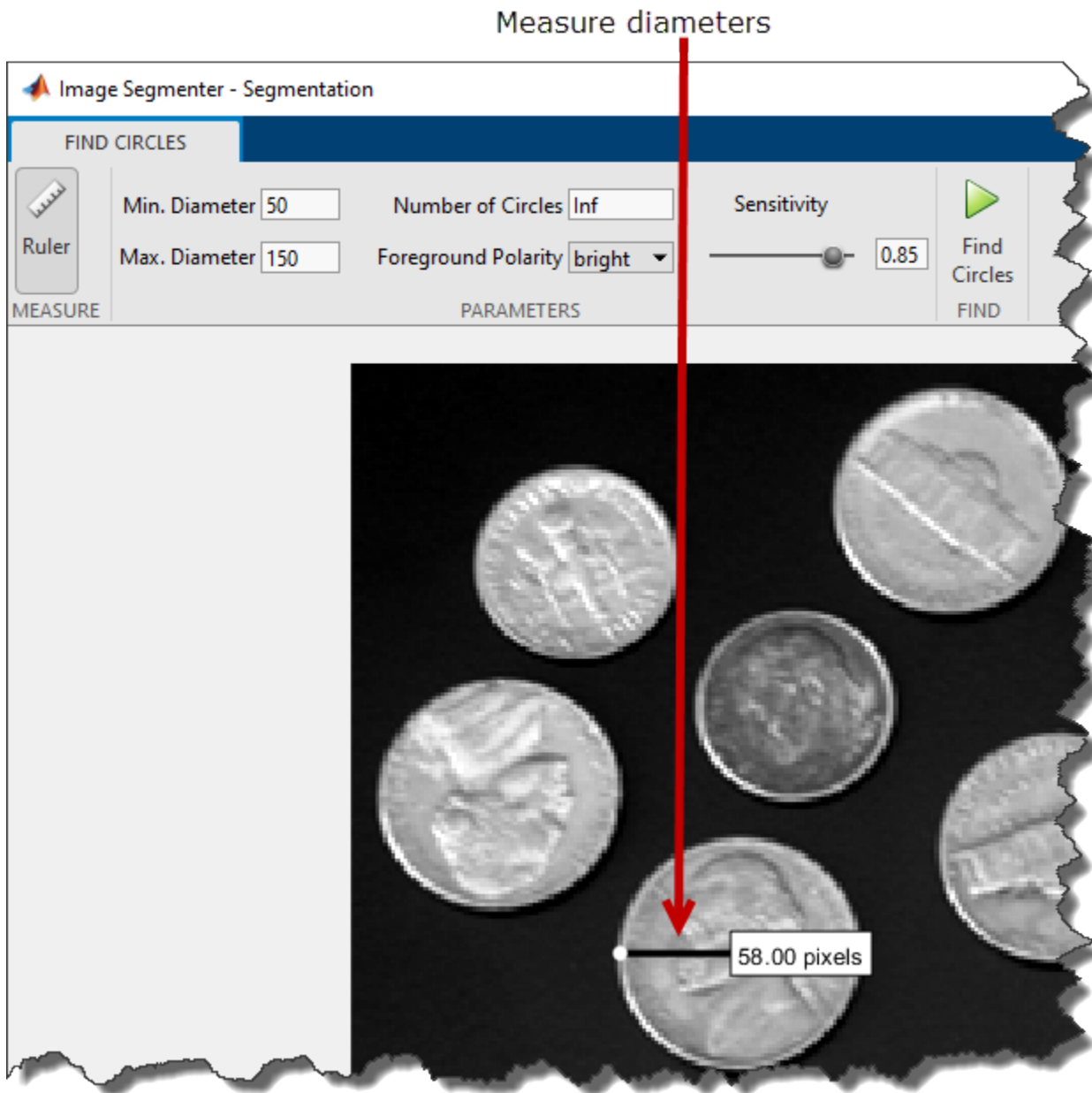
On the Find Circles tab, click **Find Circles**. The **Image Segmenter** app fills the circles it finds. However, find circles does not find two of the circles. Examining the image more closely, you discover that the diameter of these coins are slightly smaller than the specified minimum diameter.
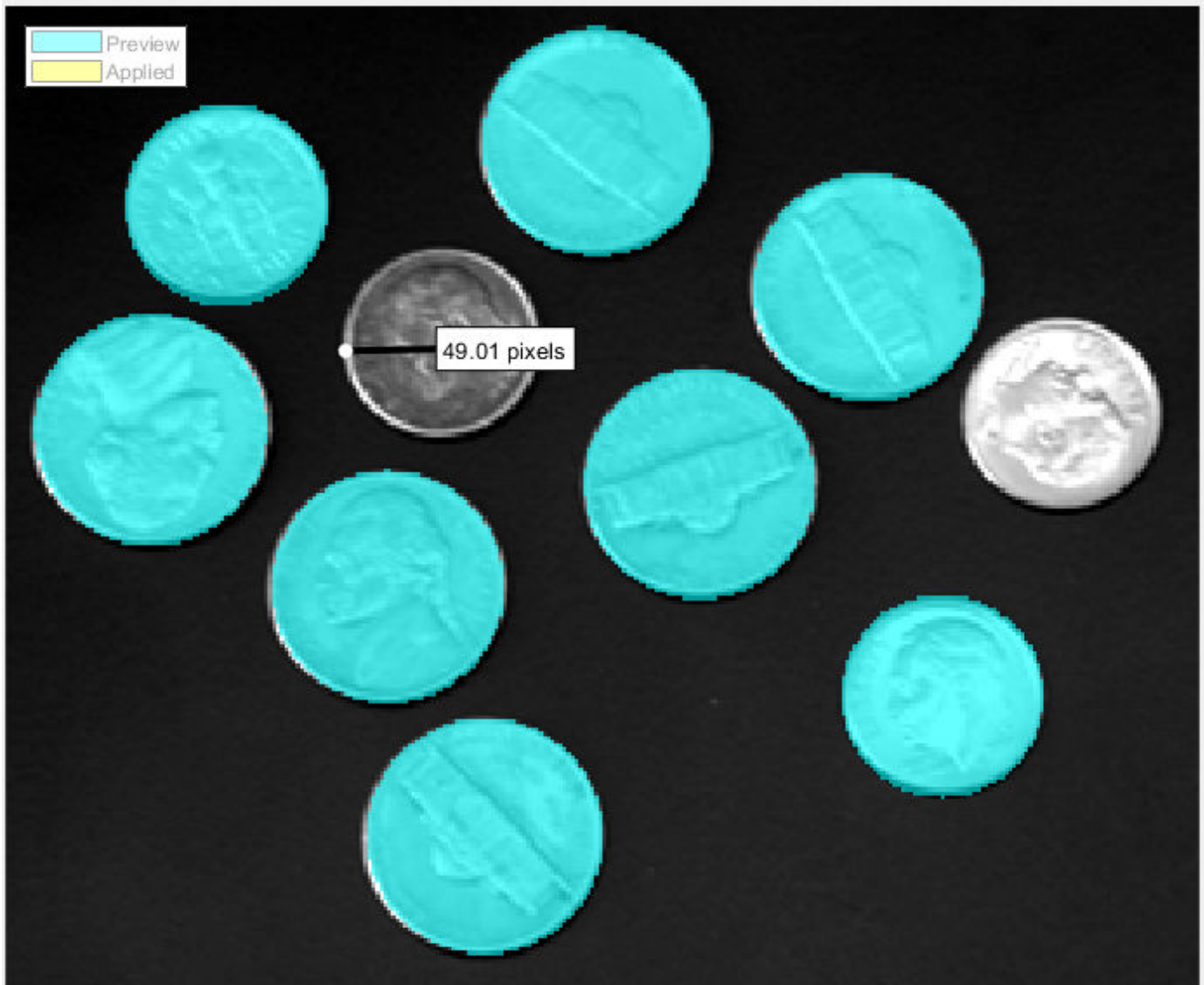
Change the minimum value to accommodate the sizes of the objects that were not segmented and run the find circles segmentation operation again. This time, **Find Circles** segments all the objects in the image.
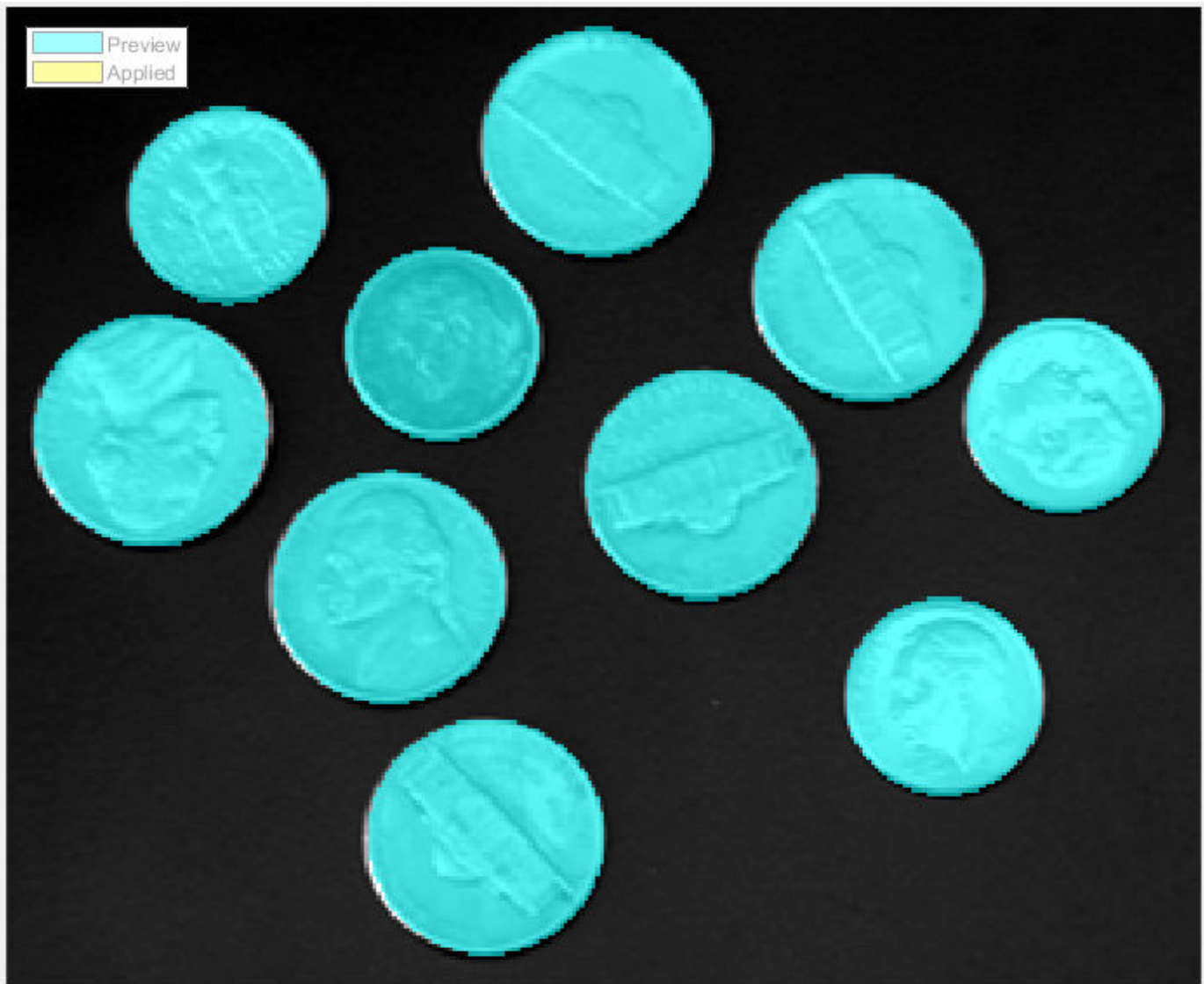
Save the Mask Image to the Workspace

When you are satisfied with the segmentation, click **Create Mask** on the Find Circles tab toolstrip and create the mask image. The **Image Segmenter** app closes the Find Circles tab and returns to the Segmentation tab. The color of the segmented circles changes to yellow. To view the mask image, click **Show Binary**.

When you are done segmenting the image, save the mask image by using the **Export** option. You can also obtain the code used for the segmentation.

Click **Show Binary** to view the segmented image.

Click **Export** to save the mask image.
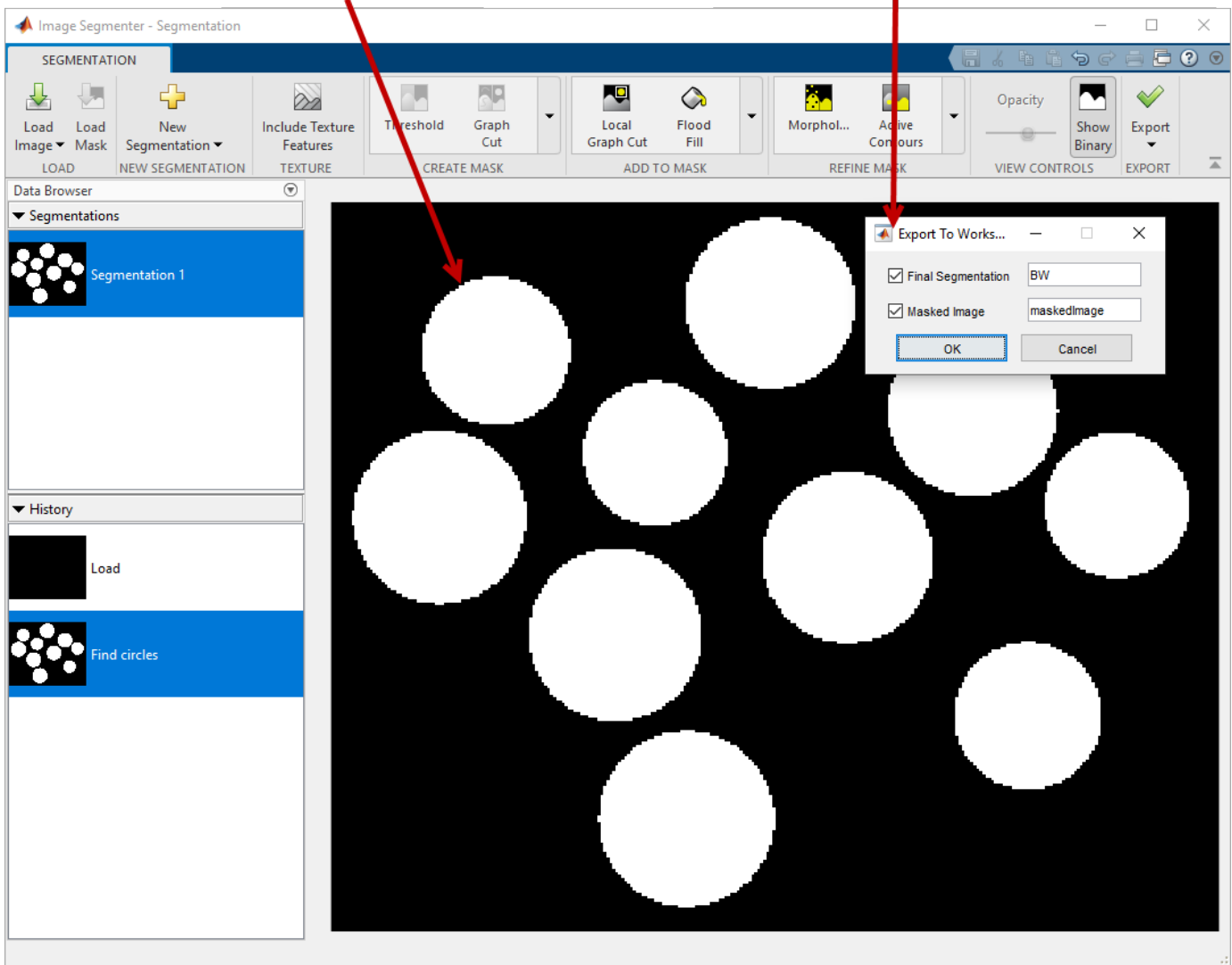
See Also

Image Segmenter | `imfindcircles`

Related Examples

- "Getting Started with Image Segmenter App" on page 13-68

Segment Image 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.

Load Image into the Image Segmenter App

Read an image into the workspace.

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

From the MATLAB® Toolstrip, open the **Image Segmenter** app. On the **Apps** tab, in the **Image Processing and Computer Vision** section, click **Image Segmente** .
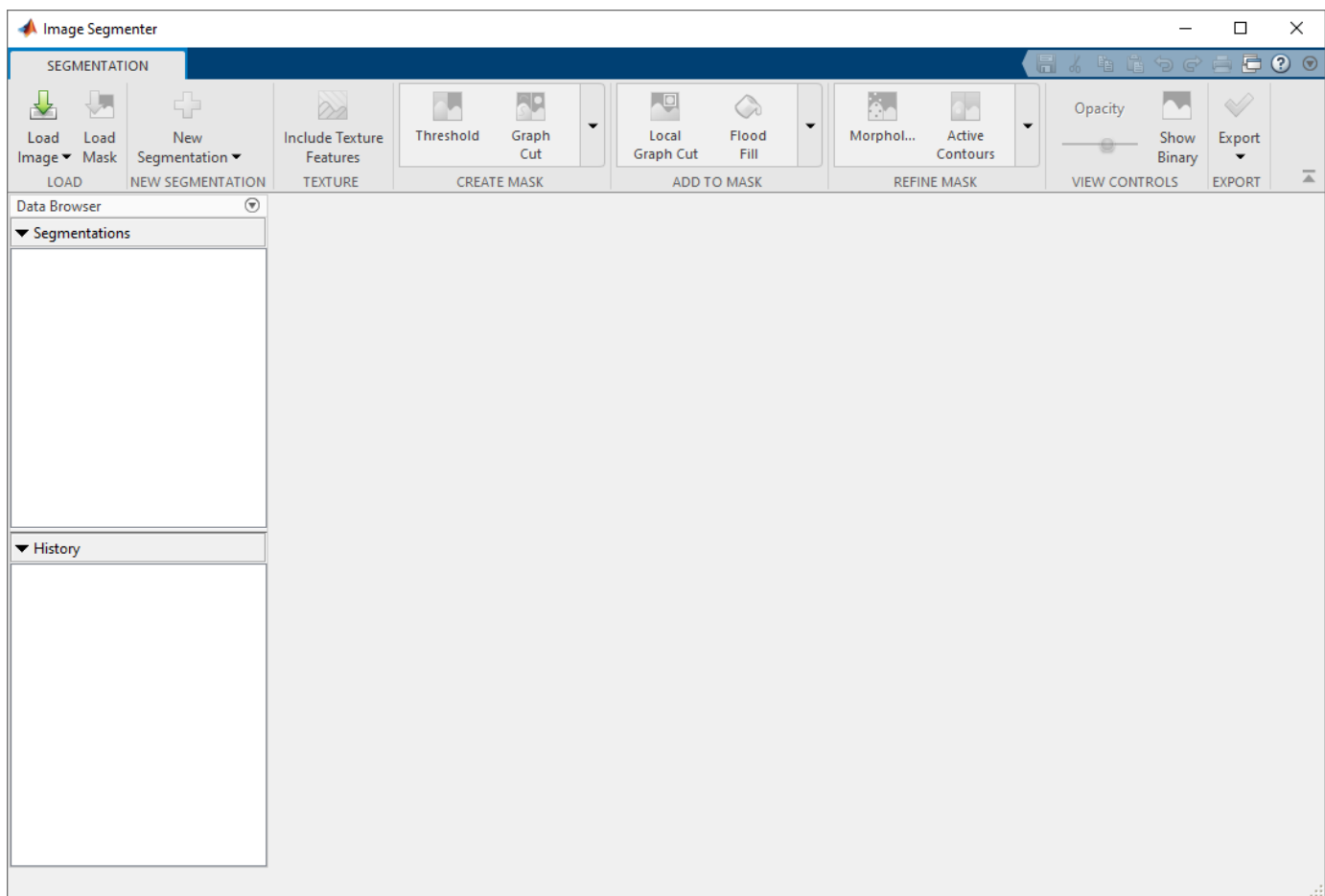
On the app toolstrip, click **Load Image**, and then select **Load Image from Workspace**. In the Import from Workspace dialog box, select the image you read into the workspace. The **Image Segmenter** app displays the image you selected.

You can also open the app using the `imageSegmenter` command, specifying the image:

`imageSegmenter(coins);`

Use Auto Cluster to Segment Image

On the **Image Segmenter** app toolstrip, expand the Create Mask section and choose **Auto Cluster**.
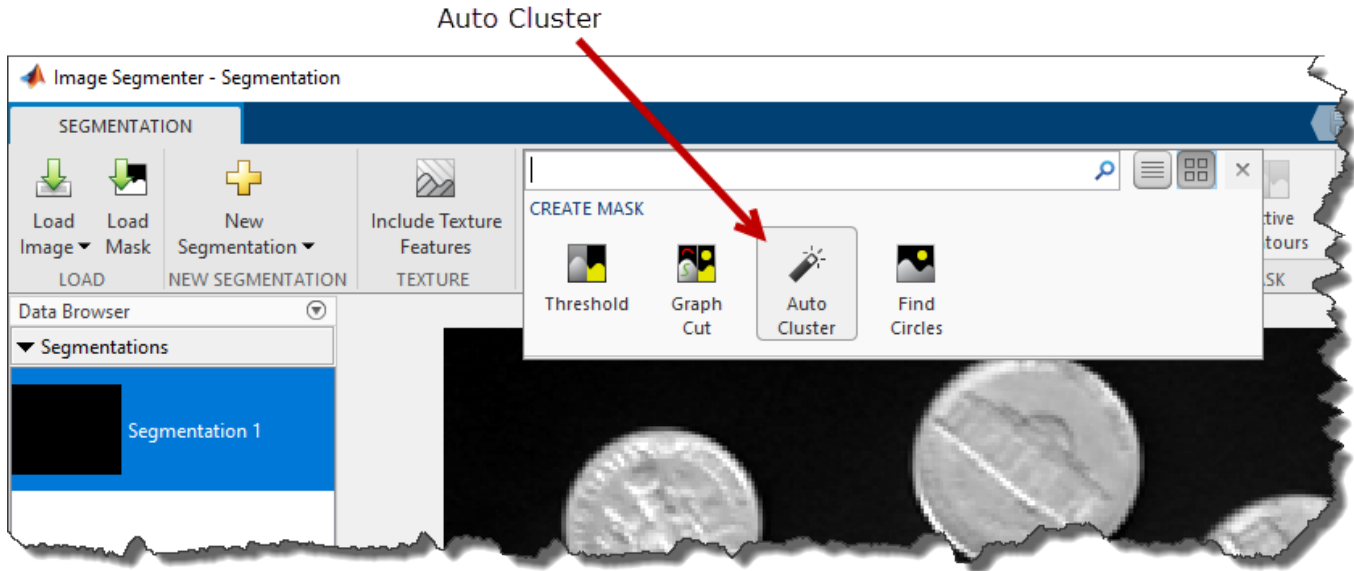
The Image Segmenter app automatically segments the image, displaying the result. The Auto Cluster option has correctly segmented all the circles. However, some of the circles have holes.
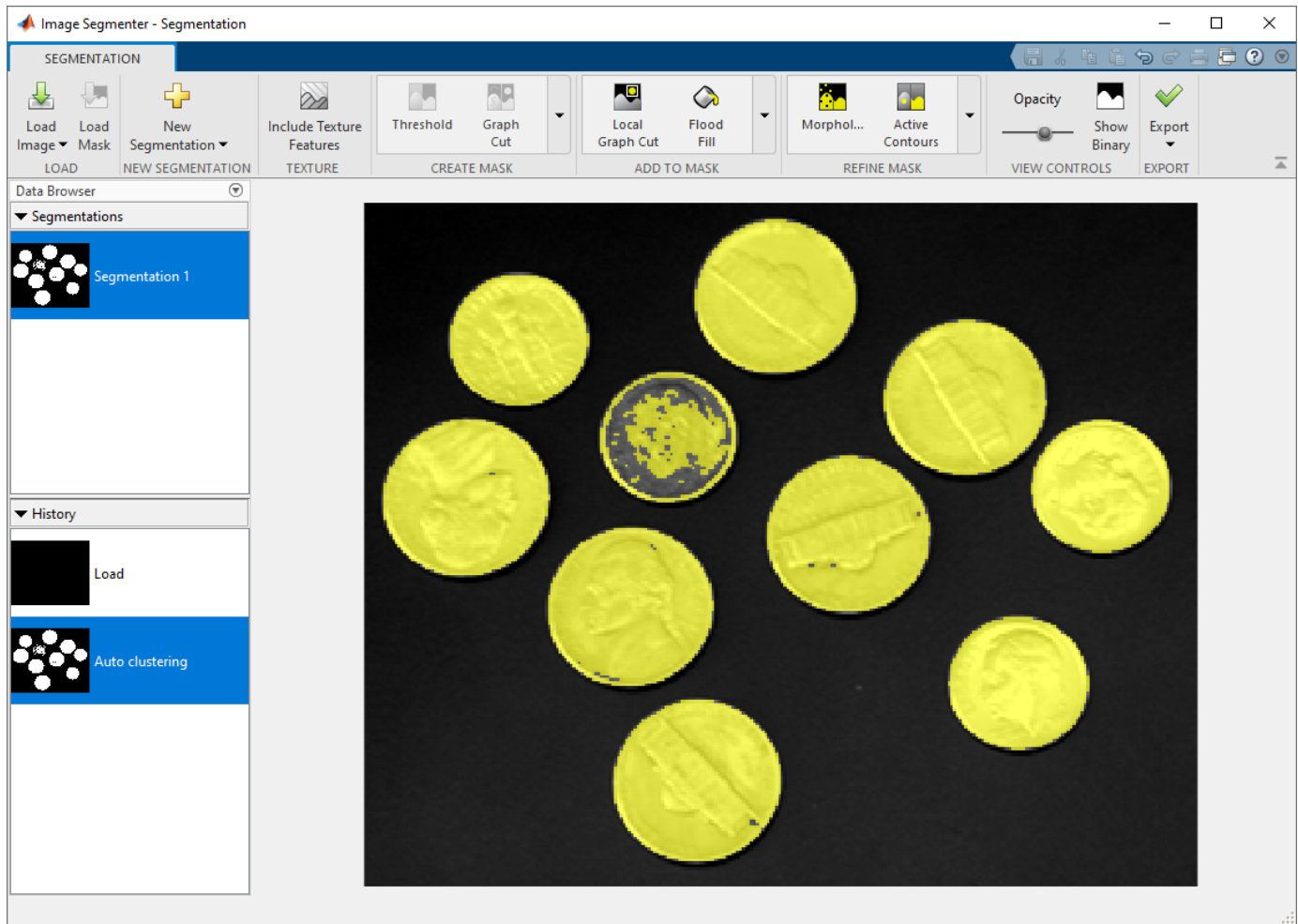
Clean up the holes in the segmented image using the **Fill Holes** option in the Refine Mask toolstrip group.
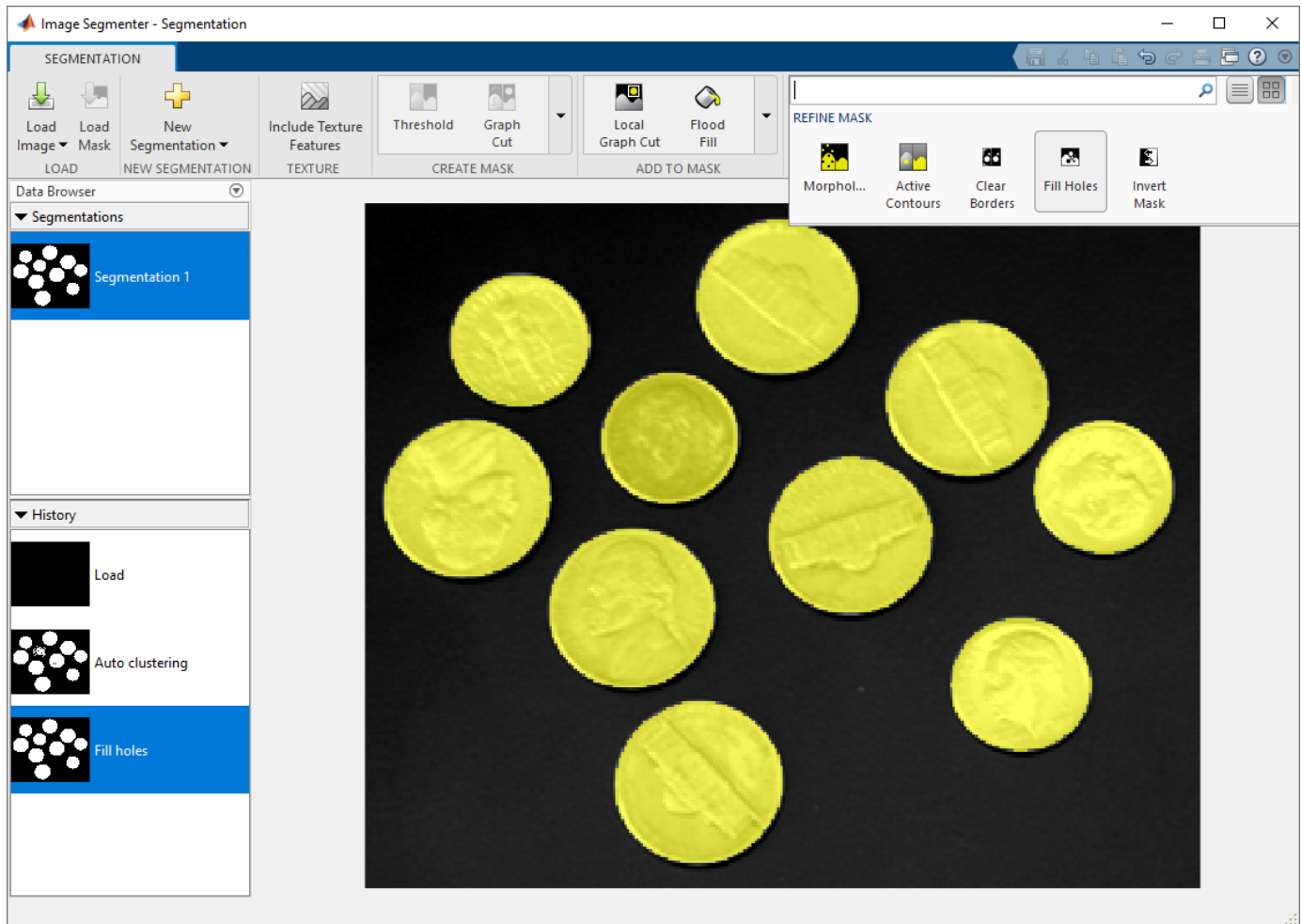
Save the Mask Image to the Workspace

When you are satisfied with the segmentation, click **Show Binary** to view the mask image. To save the binary mask, use the **Export** option. You can also obtain the code used for the segmentation.
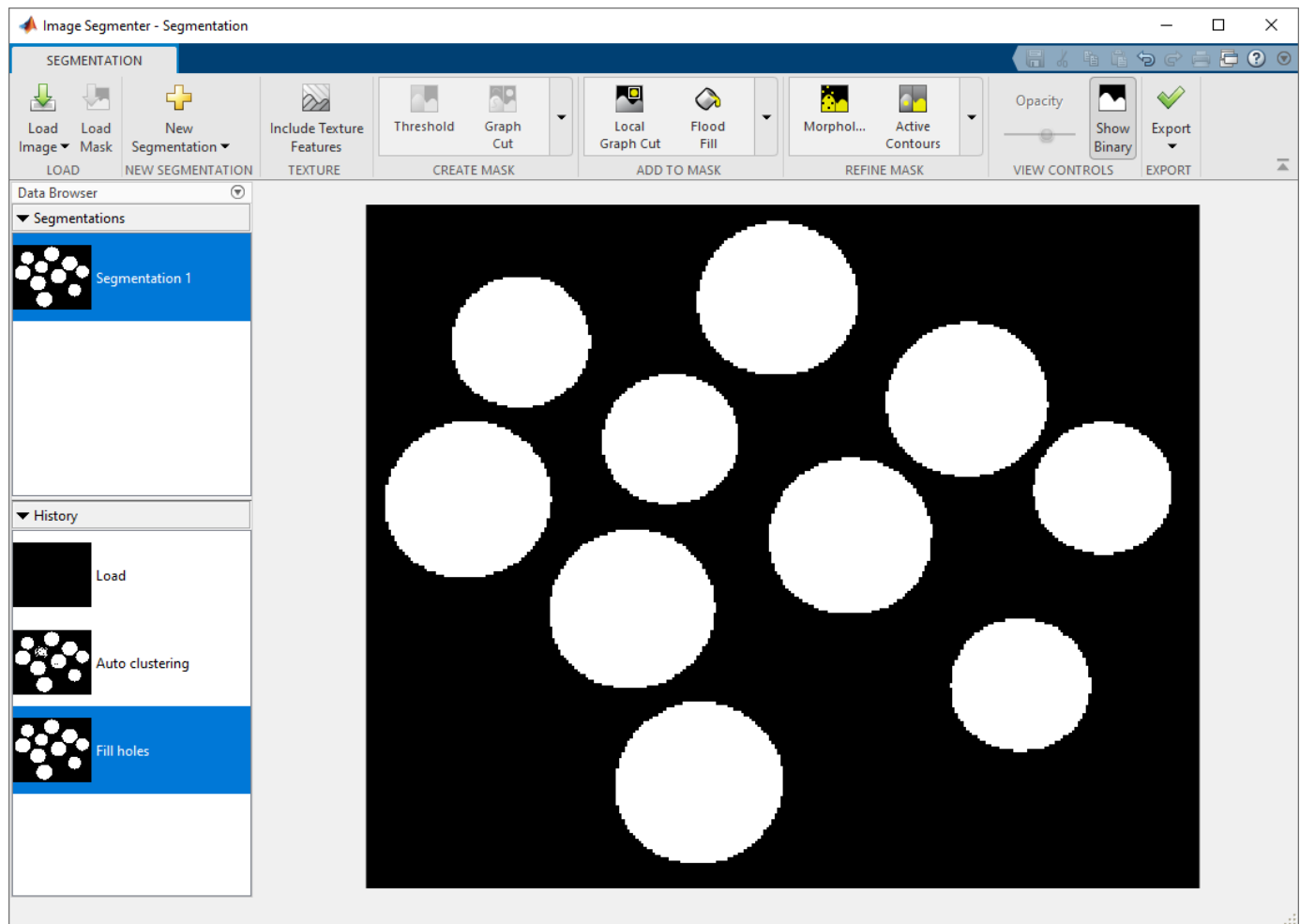
See Also

Image Segmenter | `imsegkmeans`

Related Examples

- "Getting Started with Image Segmenter App" on page 13-68

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.

Read an image into the workspace. For better performance, this example reduces the size of the image by half. Visually, there are four 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.

```
A = imread('http://eoimages.gsfc.nasa.gov/images/imagerecords/74000/74192/world.200411.3x5400x270
A = imresize(A,0.5);
imshow(A)
```
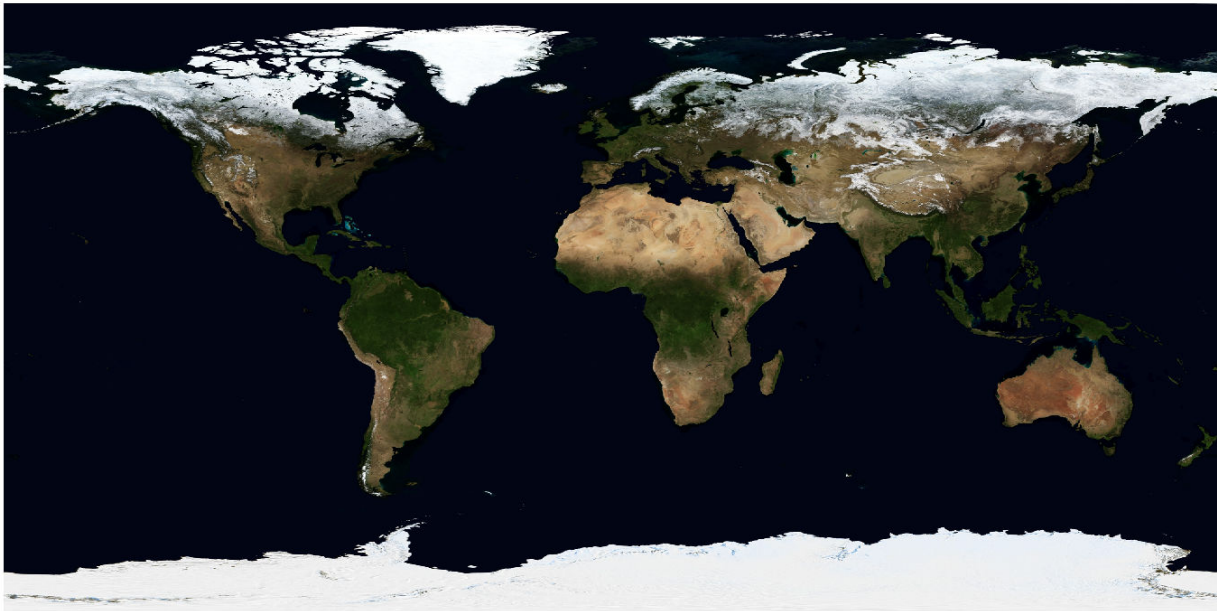

Convert the image to the *L*a*b** color space.

```
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'))
```
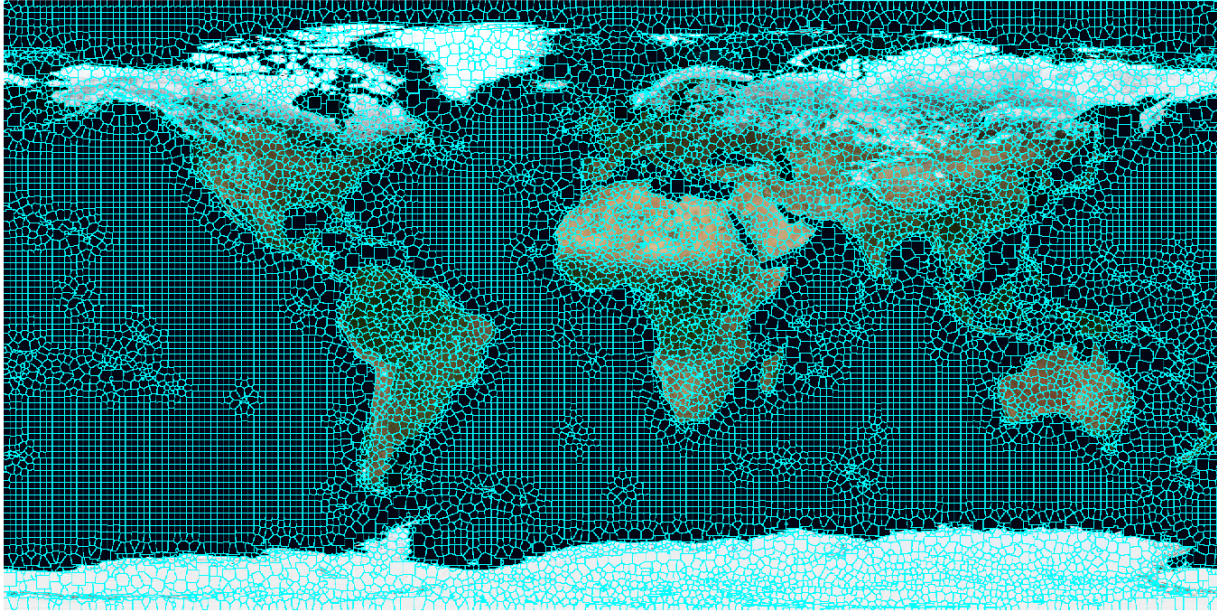
Create a cell array of the set of pixels in each region.

```
pixelIdxList = label2idx(L);
```

Determine the median color of each superpixel region in the *L*a*b** color space.

```
[m,n] = size(L);
meanColor = zeros(m,n,3,'single');
for i = 1:N
    meanColor(pixelIdxList{i}) = mean(Alab(pixelIdxList{i}));
    meanColor(pixelIdxList{i}+m*n) = mean(Alab(pixelIdxList{i}+m*n));
    meanColor(pixelIdxList{i}+2*m*n) = mean(Alab(pixelIdxList{i}+2*m*n));
end
```

Cluster the color feature of each superpixel by using the `imsegkmeans` function.

```
numColors = 4;
[Lout,cmap] = imsegkmeans(meanColor,numColors,'numAttempts',2);
cmap = lab2rgb(cmap);
imshow(label2rgb(Lout))
```
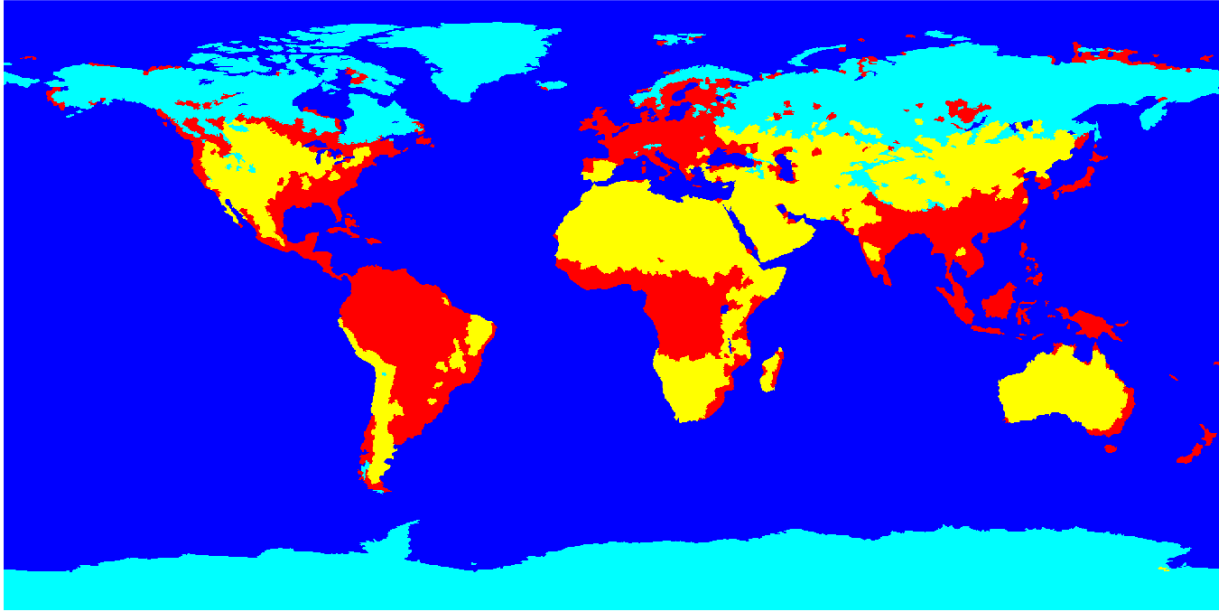
Use cluster centers as the colormap for a thematic map. The mean colors found during K-means clustering can be used directly as a colormap to give a more natural visual interpretation of the land classification assignments of forest, ice, dry land, and water.

```
imshow(double(Lout),cmap)
```
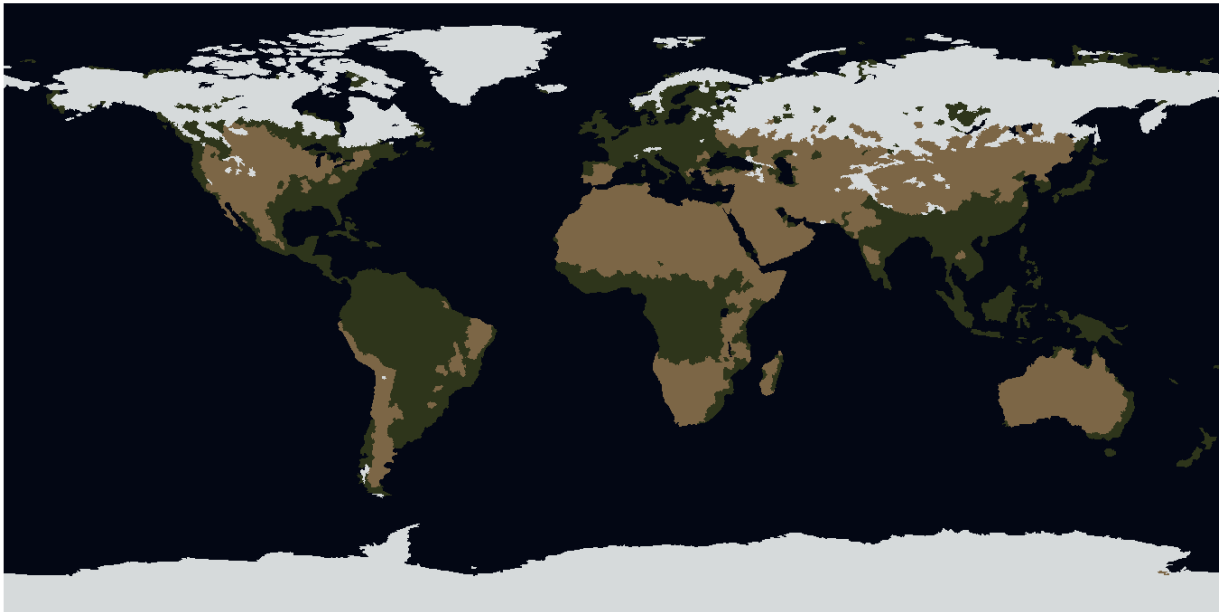
Segment Lungs from 3-D Chest Scan

This example shows how to perform a 3-D segmentation using active contours (snakes). You can view the results using the Volume Viewer app.

Prepare the Data

Load the human chest CT scan data into the workspace. To run this example, you must download the sample data from MathWorks™ using the Add-On Explorer. See "Install Sample Data Using Add-On Explorer" on page 11-173.

```
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 to the range [0, 1].

```
V = im2single(V);
```

View the chest scans using the Volume Viewer app. Open the app from the MATLAB® Apps toolstrip. You can also open the app by using the `volumeViewer` command and specifying the volume as an argument: `volumeViewer(V)`. Volume Viewer has preset alphamaps that are intended to provide the best view of certain types of data. To get the best view of the chest scans, select the **ct-bone** preset.

To get an easy-to-understand view of the chest scan, click **ct-bone**.
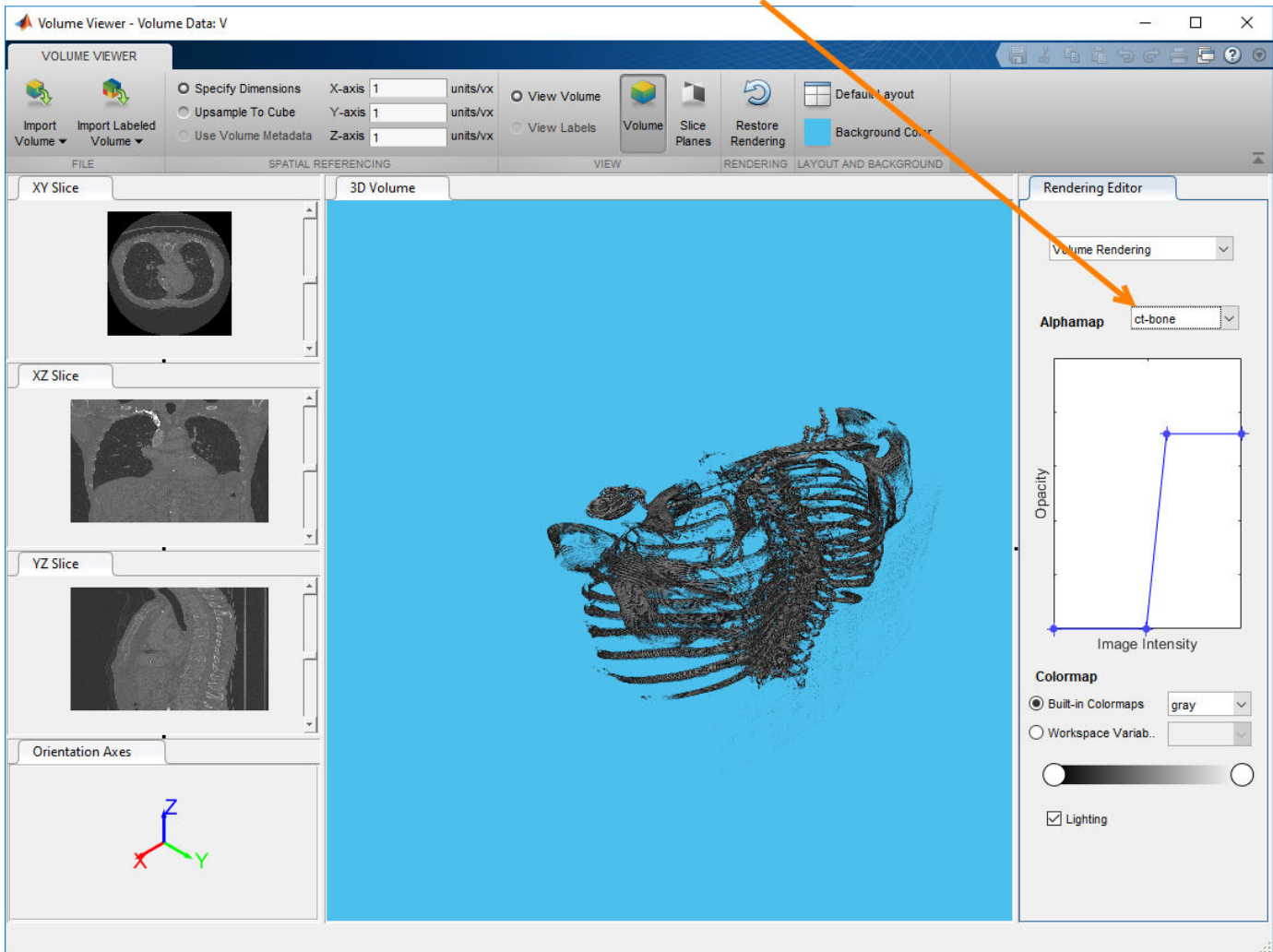
Segment the Lungs

Segment the lungs in the CT scan data using the active contour technique. Active contours 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,:,:));
```

View the 2-D slices using the `imshow` function.

```
figure
imshow(XY,[],'Border','tight');
```
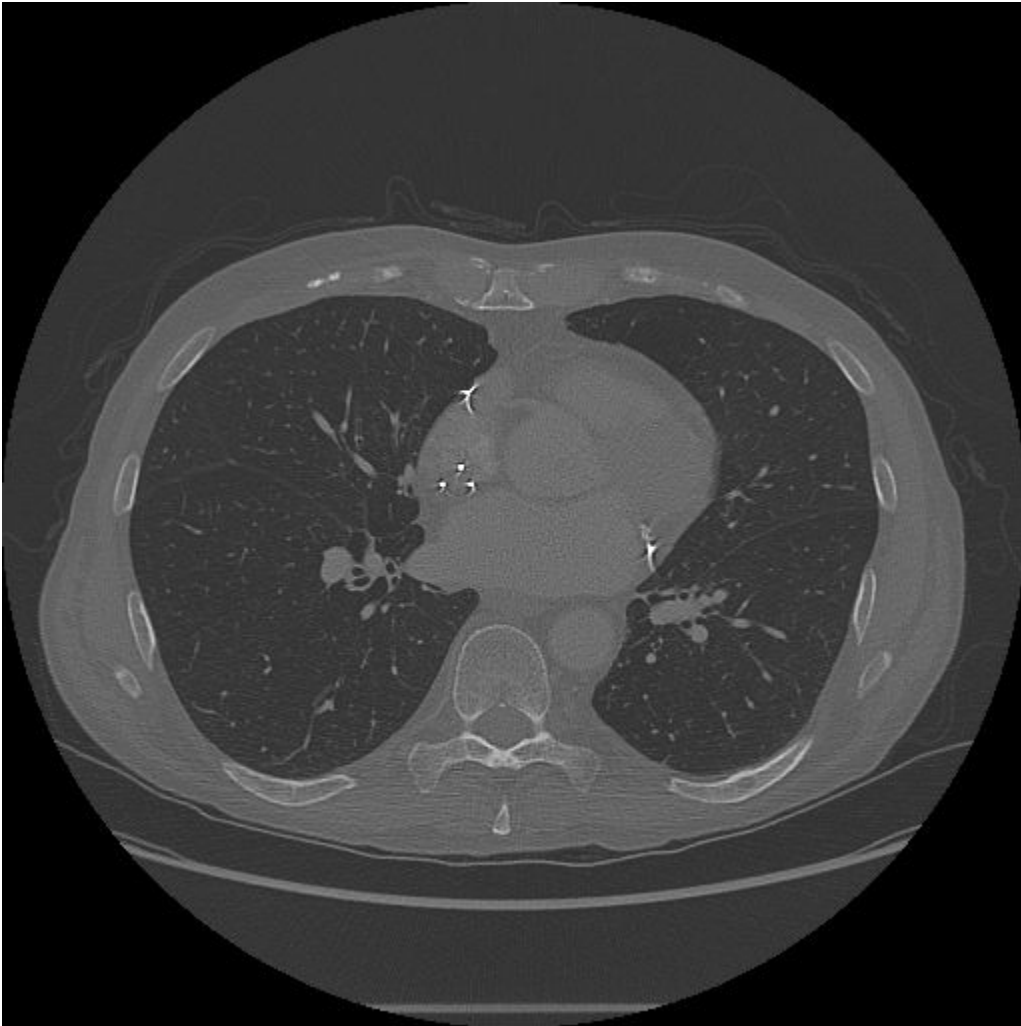
```
imshow(XZ,[],'Border','tight');
```
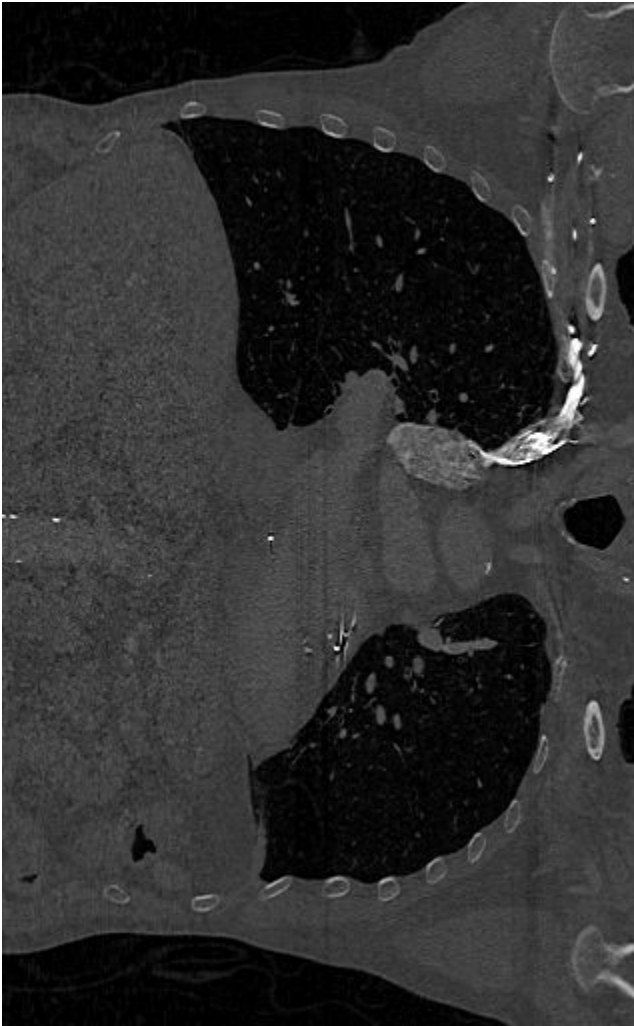
You can perform the segmentation in the Image Segmenter app. Open the app from the MATLAB Apps toolstrip or use the `imageSegmenter` command, specifying a 2-D slice as an argument, `imageSegmenter(XY)`.

To start the segmentation process, click **Threshold** to open the lung slice in the **Threshold** tab. On the Threshold tab, select the **Manual Threshold** option and move the Threshold slider to specify a threshold value that achieves a good segmentation of the lungs. Click **Create Mask** to accept the thresholding and return the Segmentation tab.
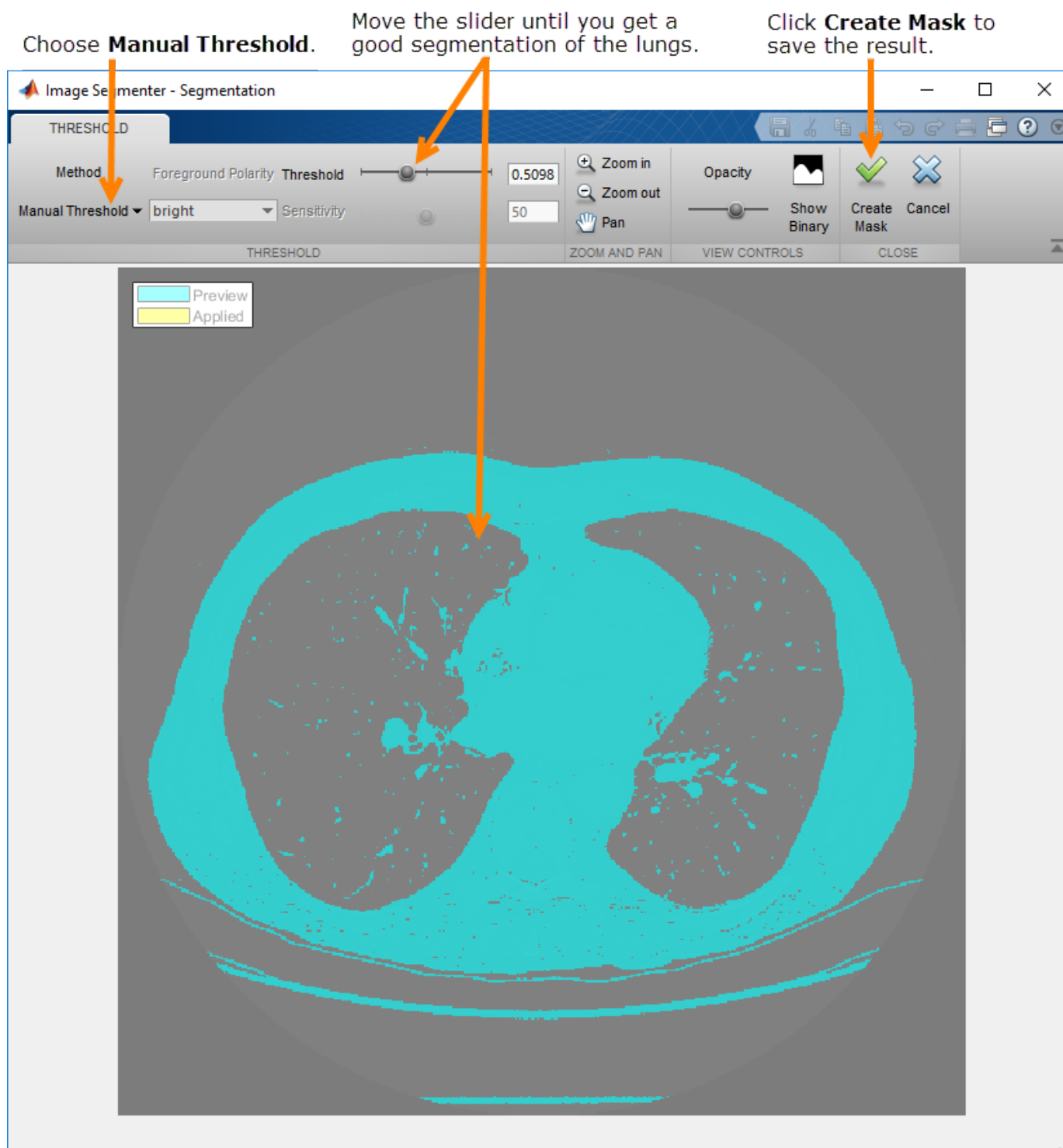
The app executes the following code to threshold the image.

```
BW = XY > 5.098000e-01;
```

After this initial lung segmentation, clean up the mask using options on the **Refine Mask** menu.

Expand the **Refine Mask** group.
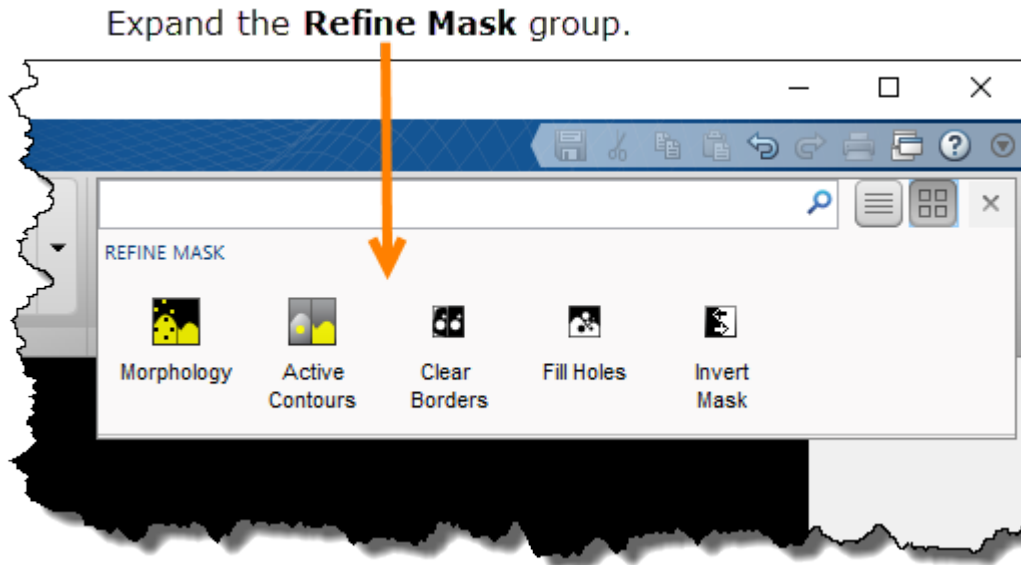
In the app, you can click each option to invert the mask image so that the lungs are in the foreground (**Invert Mask**), remove other segmented elements besides the lungs (**Clear Borders**), and fill holes inside the lung segmentation (**Fill Holes**). Finally, use the **Morphology** option to smooth the edges of the lung segmentation. On the Morphology tab, select the **Erode Mask** operation. After performing these steps, select **Show Binary** and save the mask image to the workspace.

The app executes the following code to refine the mask.

```
BW = imcomplement(BW);
BW = imclearborder(BW);
BW = imfill(BW, 'holes');
radius = 3;
decomposition = 0;
se = strel('disk',radius,decomposition);
BW = imerode(BW, se);
maskedImageXY = XY;
maskedImageXY(~BW) = 0;
imshow(maskedImageXY)
```

Perform the same operation on the XZ slice. Using Load Image, select the XZ variable. Use thresholding to perform the initial segmentation of the lungs. For the XZ slice, the **Global Threshold** option creates an adequate segmentation (the call to `imbinarize` in the following code). As with the XY slice, use options on the **Refine Mask** menu to create a polished segmentation of the lungs. In the erosion operation on the Morphology tab, specify a radius of 13 to remove small extraneous objects.

To segment the XZ slice and polish the result, the app executes the following code.

```
BW = imbinarize(XZ);
BW = imcomplement(BW);
BW = imclearborder(BW);
BW = imfill(BW,'holes');
radius = 13;
decomposition = 0;
```

```
se = strel('disk',radius,decomposition);
BW = imerode(BW, se);
maskedImageXZ = XZ;
maskedImageXZ(~BW) = 0;
imshow(maskedImageXZ)
```


Create Seed Mask and Segment Lungs Using activecontour

Create the 3-D seed mask that you can use with the `activecontour` function to segment the lungs.

Create a logical 3-D volume the same size as the input volume and insert `mask_XY` and `mask_XZ` at the appropriate spatial locations.

```
mask = false(size(V));
mask(:,:,160) = maskedImageXY;
mask(256,:,:) = mask(256,:,:)|reshape(maskedImageXZ,[1,512,318]);
```

Using this 3-D seed mask, segment the lungs in the 3-D volume using the active contour method. This operation 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);
```

You can view the segmented lungs in the Volume Viewer app by running the command `volumeViewer(segmentedImage)`. By manipulating the alphamap settings in the Rendering Editor, you can get a good view of just the lungs.

View segmented lungs as a volume.
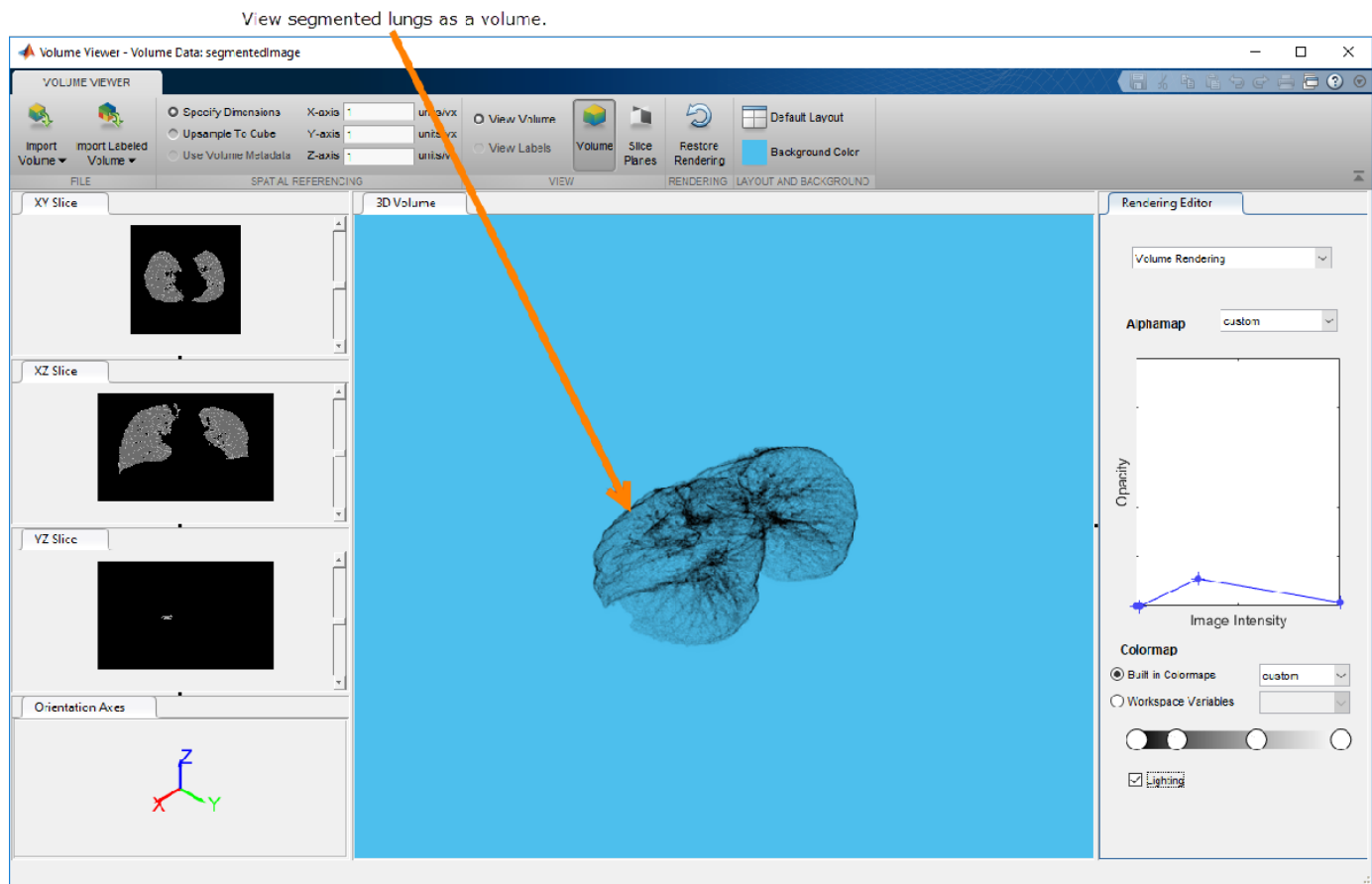
Compute the Volume of the Segmented Lungs

Use the `regionprops3` function with the `'volume'` option to calculate the volume of the lungs.

```
volLungsPixels = regionprops3(logical(BW),'volume');
```

Specify the spacing of the voxels in the x, y, and z dimensions, which was gathered from the original file metadata. The metadata is not included with the image data that you download from the Add-On Explorer.

```
spacingx = 0.76;
spacingy = 0.76;
spacingz = 1.26*1e-6;
unitvol = spacingx*spacingy*spacingz;

volLungs1 = volLungsPixels.Volume(1)*unitvol;
volLungs2 = volLungsPixels.Volume(2)*unitvol;
volLungsLiters = volLungs1 + volLungs2
```

```
volLungsLiters = 5.7726
```

See Also
activecontour | histeq | regionprops3

Create Binary Mask Using Volume Segmenter

This example shows how to segment a volume in the **Volume Segmenter** app. The **Volume Segmenter** app offers many ways to explore a volume and segment objects in the volume. For example, you can view the volume slice-by-slice or as a 3-D representation. To segment an object, you can draw a region of interest (ROI) using ROI drawing tools or a paint brush tool. This example creates a binary mask that isolates a small region of the brain.

Load Volumetric Data into the Workspace

Load a volume into the workspace. This example uses a stack of MRI brain images, stored in the MAT-file vol_001.mat.

```
load(fullfile(toolboxdir('images'),'imdata',...
    'BrainMRILabeled','images','vol_001.mat'));
```

This command loads a 240-by-240-by-155 volume named vol into workspace.

```
whos vol
```

```
  Name        Size                    Bytes  Class     Attributes

  vol       240x240x155            17856000  uint16
```

Open the Volume Segmenter

Open the **Volume Segmenter** app. Click the **Apps** tab on the MATLAB® toolstrip. In the **Image Processing and Computer Vision** section, click **Volume Segmenter**.

Load the Volume into the Volume Segmenter

To load the volume in the **Volume Segmenter** app, click **Open Volume** in the app toolstrip. For this example, select **Open from Workspace**. In the Import Volume dialog box, select the volume you loaded into the workspace, `vol`, and click **OK**. (You can also specify a volume when you open the app by using the `volumeSegmenter` command: `volumeSegmenter(vol)`.)

The **Volume Segmenter** app displays a 3-D representation of the volume in the **3-D Display** pane and displays individual slices of the data set in the Slice pane.

By default, the Slice pane displays the first slice of your data. The app displays the number of the slice displayed at the top of the image, for example, 1/155. In this dataset, the first few slices do not contain images of the brain.

The app also automatically creates a label for the segmentation in the **Labels** pane, using the default name `Label1`. You can define multiple labels in the **Labels** pane. However, to create a binary mask, you must use only one label.

To change the name of the label, double-click the label name. To change the color associated with the label, double-click the color square displayed in the **Labels** pane. You can optionally load an existing set of labels into the app using the **Open Labels** button.

Explore the Volume

To determine what you want to segment, explore the volume using the **3-D Display** pane and the **Slice** pane.

In the **3-D Display** pane, you can rotate the volume to examine the data from every angle, using the mouse. You can also customize the display of the volume in the **3-D Display** tab in the app toolstrip.

For example, if you have metadata that descibes the relative size of the voxels, you can specify it in the **Spatial Referencing** part of the **3-D Display** tab in the app toolstrip. To improve your view of the data, you can change the background color used in the 3-D display, modify the threshold and opacity of the display, and include orientation axes with the display, as shown in this figure. With the brain MRI data, you can see the tumor in the temporal lobe that you want to segment.

You can also view each slice of the volume in the **Slice** pane. Use the slider at the bottom of the pane to move from slice to slice. You can see the tumor on slice 35 through slice 88. By default, the **Slice** pane displays the volume oriented along the X-Y axis, but you can change this using buttons in the Orientation section of the toolstrip on the **Segmenter** tab. The **Slice** pane is also where you use drawing tools to define the mask.

Use Drawing Tools to Define the Mask

Once you have identified the object you want to segment, you can use the tools on the **Draw** tab in the app toolstrip to define the region. Select the drawing tool you want to use from the ROI tools: **Freehand**, **Assisted Freehand**, and **Polygon**, and a **Paint Brush** tool.

In the **Slice** pane, navigate to the slice where the object first appears, slice 35, and draw an outline around the object. For this example, use the **Polygon** drawing tool. Click to create a vertex, then move the cursor and click again to create a second vertex with a straight line connecting them. Continue this process to create a connected line. To add additional vertices after you finish drawing, double-click on the ROI edge.

Using Interpolation to Speed Object ROI Creation

You could move through the volume, slice-by-slice, and draw an ROI on each slice where the object appears. However, the **Volume Segmenter** app provides several automated interpolation tools that can help with segmenting an object across slices.

To use interpolation, you must first manually define the region on two slices. You have already defined the region on the first slice where the object appears, slice 35. Use the same process to define the region on the last slice where it appears, slice 88. The app places two bars on top of the slider, using the color associated with the label, to indicate the slices with ROIs.

With the object defined on two slices, click **Auto Interpolate**. The app automatically defines the ROI on all the intervening slides. The app uses blue bars to indicate all the slices that have ROIs, which now appear like a solid bar fro slice 35 to slice 88.

Alternatively, after defining an ROI on two slices, you can click **Manually Interpolate**. With this option, the app opens the Manually Interpolate dialog box. You select the two regions from which you want to interpolate, Region One and Region Two. To select the first region, use the slider at the bottom of the dialog box to navigate to the first slice with an ROI, slice 35, and then click inside the ROI displayed. To select the second region, click Region Two, navigate to slide 88, and click inside the ROI displayed. After selecting both regions, click **Run** to interpolate the ROI on all intervening slices.

Refine the Interpolated ROIs

After using interpolation, check the individual slices to see if the interpolation created satisfactory ROIs. Note that the ROI on slice 71 does not fill the entire object that you want to segment. You can manually adjust the ROI using the **Paint Brush** tool. Alternatively, you can use one of the tools on **Automate** tab. For example, you can use **Active Contours** to grow the ROIs on the slices where it does not fill the full size of the tumor. You can also use the **Add Algorithm** to specify your own algorithm to operate on the ROIs.

Perform Custom Processing

You can also your own algorithm to operate on the ROIs. On the **Automate** tab, click **Add Algorithm**. Choose whether you want your processing to operate on each 2-D slice (Slice-based) or on the entire 3-D volume (Volume-based).

Click **Add Algorithm.**

Select the Automate tab.

Specify existing function or create a new one.

For this example, under Slice-Based, select the **New** option and click **Function Template** to create a new function that operates on each 2-D slice. The app opens the template in the MATLAB editor. Replace the sample code in the template with code that you want to use. Your function must accept two arguments: each slice as a separate image and a mask. Your function must also return a mask image.

When you are done editing the template, save the file. The Volume Segmenter app automatically creates a button in the **Automate** tab toolstrip for your function. To test your function on one slice, click **Run**. By default, the app applies the function to only the current slice.

Your function appears selected in the **Automate** toolstrip.

Click **Run** to test your function on the current slice.

After testing your function on a single slice, you can run it on all of the slices or a subset of the slices. You can run it from the current slice to the end (the highest numbered slice) or from the current slice back to the beginning (slice 1). You can also specify a range of slices by specifying the starting slice and the ending slice.

When you choose one of the directional options, the app updates the slice numbers in the display. You can use this display to view the progress of processing.

Create the Binary Mask Volume

To create the binary mask volume, click **Save Labels** on the **Segmenter** tab. You can save the mask to a MAT-file or to a workspace variable. For this example, click **Save As Workspace Variable**. In the Save to workspace dialog box, specify whether you want to save the segmentation as a logical or categorial mask. Choose logical,(the default when there is only one label), give the variable a name, `my_mask_volume`, and click **OK**. The app creates a 3-D volume of class `logical` with the same dimensions as the original volume.

To view the mask, use the `volshow` function: `volshow(my_mask_volume);`.
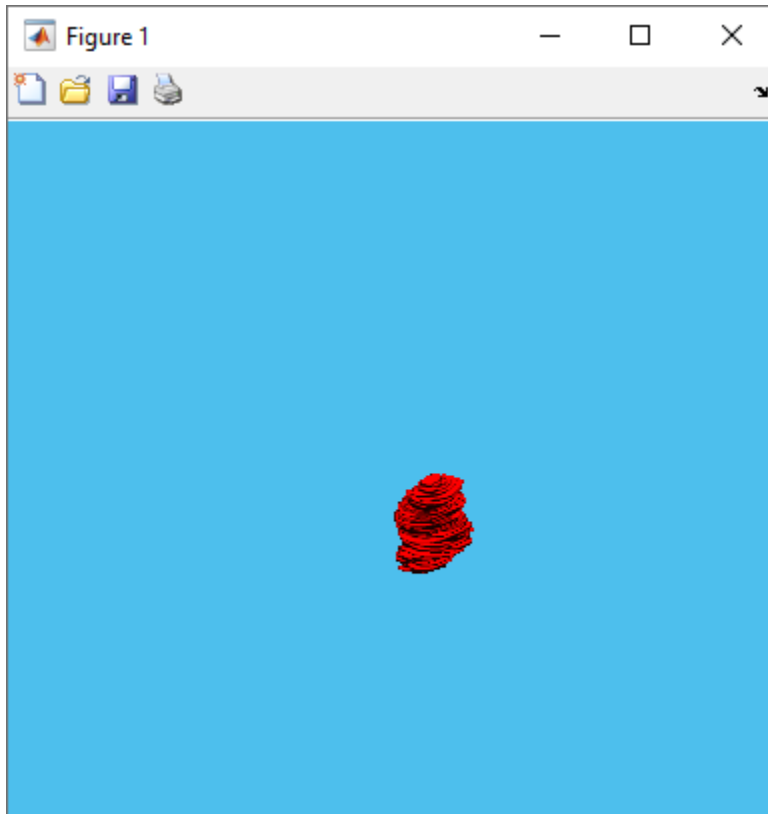
See Also
Volume Segmenter

Related Examples

- "Create Semantic Segmentation Using Volume Segmenter" on page 13-148

Create Semantic Segmentation Using Volume Segmenter

This example shows how to create a semantic segmentation of a volume using the **Volume Segmenter** app. The **Volume Segmenter** app offers many ways to explore a volume and segment objects in the volume. For example, you can view the volume slice-by-slice or as a 3-D representation. To segment an object, you can draw a region of interest (ROI) using ROI drawing tools or a paint brush tool. This example segments a stack of MRI images to label the brain and tumor regions. The example also labels the background.

Load Volumetric Data into the Workspace

Load a volume into the workspace. This example uses a stack of MRI brain images, stored in the MAT-file vol_001.mat.

```
load(fullfile(toolboxdir('images'),'imdata', ...
    'BrainMRILabeled','images','vol_001.mat'));
```

This command loads a 240-by-240-by-155 volume named vol into the workspace.

```
whos vol
```

Name	Size	Bytes	Class	Attributes
vol	240x240x155	17856000	uint16	

Open the Volume Segmenter

Open the **Volume Segmenter** app. Click the **Apps** tab on the MATLAB® toolstrip. In the **Image Processing and Computer Vision** section, click **Volume Segmenter**.

Load the Volume into the Volume Segmenter

To load the volume in the **Volume Segmenter** app, click **Open Volume** in the app toolstrip. For this example, select **Open from Workspace**. In the Import Volume dialog box, select the volume you loaded into the workspace, `vol`, and click **OK**. (You can also specify a volume when you open the app by using the `volumeSegmenter` command: `volumeSegmenter(vol)`.)

The **Volume Segmenter** app displays a 3-D representation of the volume in the **3-D Display** pane and displays individual slices of the data set in the Slice pane.

By default, the Slice pane displays the first slice of your data. The app displays the number of the slice displayed at the top of the image, for example, 1/155. In this dataset, the first few slices do not contain images of the brain.

The app also automatically creates a label for the segmentation in the **Labels** pane, using the default name `Label1`. You can define multiple labels in the **Labels** pane. However, to create a binary mask, you must use only one label.

To change the name of the label, double-click the label name. To change the color associated with the label, double-click the color square displayed in the **Labels** pane. You can optionally load an existing set of labels into the app using the **Open Labels** button.

Explore the Volume

To determine what you want to segment, explore the volume using the **3-D Display** pane and the **Slice** pane.

In the **3-D Display** pane, you can rotate the volume to examine the data from every angle, using the mouse. You can also customize the display of the volume in the **3-D Display** tab in the app toolstrip.

For example, if you have metadata that descibes the relative size of the voxels, you can specify it in the **Spatial Referencing** part of the **3-D Display** tab in the app toolstrip. To improve your view of the data, you can change the background color used in the 3-D display, modify the threshold and opacity of the display, and include orientation axes with the display, as shown in this figure. With the brain MRI data, you can see the tumor in the temporal lobe that you want to segment.

You can also view each slice of the volume in the **Slice** pane. Use the slider at the bottom of the pane to move from slice to slice. You can see the tumor on slice 35 through slice 88. By default, the **Slice** pane displays the volume oriented along the X-Y axis, but you can change this using buttons in the Orientation section of the toolstrip on the **Segmenter** tab. The **Slice** pane is also where you use drawing tools to define the mask.

Use Drawing Tools to Label Regions in Volume

Once you have identified the object you want to segment, you can use the tools on the **Draw** tab in the app toolstrip to define the region. Select the drawing tool you want to use from the ROI tools: **Freehand**, **Assisted Freehand**, and **Polygon**, and a **Paint Brush** tool.

Start by labeling the brain. When one object is nested in another object, as the tumor appears over the brain on slices, label the larger region first. The first step is to create a label in the **Labels** pane. The app provides one label by default, named Label1. To change the name of the label to be more descriptive for your application, double-click in the label and type in the new name. To change the default color associated with the label, double-click the colored square in the label identifier and select a color from the Color dialog box.
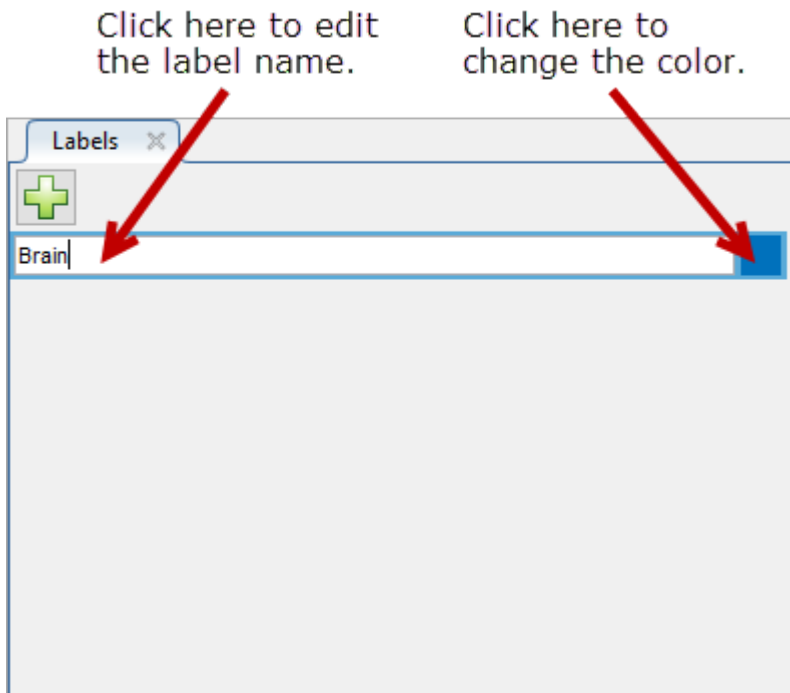
In the **Slice** pane, navigate to the slice where the object first appears and use a drawing tool to label the object. In the following figure, this example uses the Paint Brush tool to label the brain, but you can use any of the drawing tools.
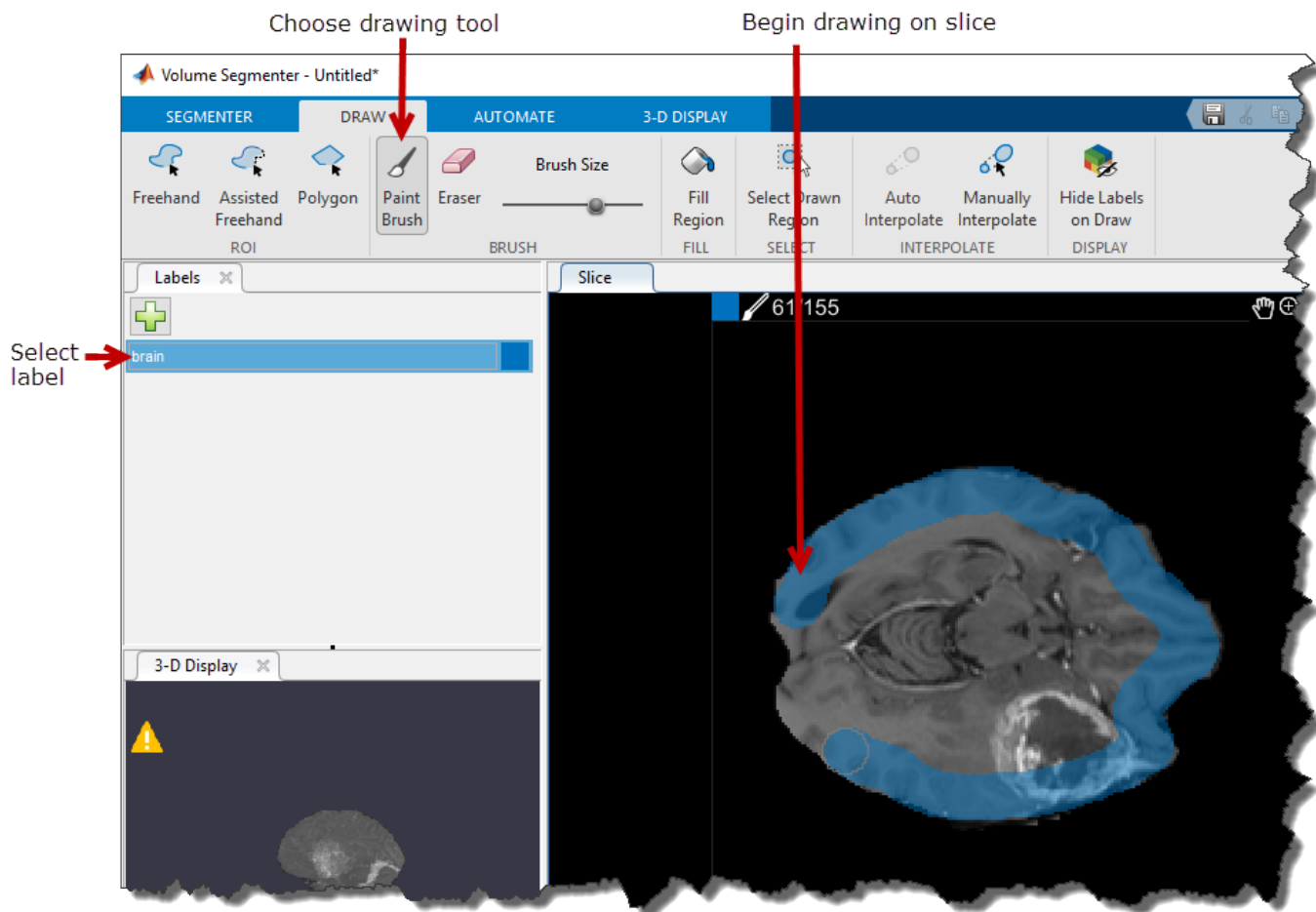
Using Interpolation to Speed Object ROI Creation

You could move through the volume, slice-by-slice, and draw an ROI on each slice where the object appears. However, the **Volume Segmenter** app provides several automated interpolation tools that can help with segmenting an object across slices.

To use interpolation, you must first manually define the region on two slices. You have already defined the region on the first slice where the object appears, slice 35. Use the same process to define the region on the last slice where it appears, slice 88. The app places two bars on top of the slider, using the color associated with the label, to indicate the slices with ROIs. The app places two bars on the slider, using the color associated with the label, to indicate the slices with ROIs.

With the object defined on two slices, click **Auto Interpolate**. The app automatically defines the ROI on all the intervening slides. The app uses blue bars to indicate all the slices that have ROIs, which now appear like a solid bar fro slice 35 to slice 88.

Alternatively, after defining an ROI on two slices, you can click **Manually Interpolate**. With this option, the app opens the Manually Interpolate dialog box. You select the two regions from which you want to interpolate, Region One and Region Two. To select the first region, use the slider at the bottom of the dialog box to navigate to the first slice with an ROI, slice 35, and then click inside the ROI displayed. To select the second region, click Region Two, navigate to slide 88, and click inside the ROI displayed. After selecting both regions, click **Run** to interpolate the ROI on all intervening slices.

Refine the Interpolated Labels

After using interpolation, check the individual slices to see if the interpolation created satisfactory ROIs. Note that the ROI on slice 71 does not fill the entire object that you want to segment. You can manually adjust the ROI using the **Paint Brush** tool. Alternatively, you can use one of the tools on **Automate** tab. For example, you can use **Active Contours** to grow the ROIs on the slices where it does not fill the full size of the tumor.

Perform Custom Processing

You can also your own algorithm to operate on the ROIs. On the **Automate** tab, click **Add Algorithm**. Choose whether you want your processing to operate on each 2-D slice (Slice-based) or on the entire 3-D volume (Volume-based).

Click **Add Algorithm.**

Select the Automate tab.

Specify existing function or create a new one.

For this example, under Slice-Based, select the **New** option and click **Function Template** to create a new function that operates on each 2-D slice. The app opens the template in the MATLAB editor. Replace the sample code in the template with code that you want to use. Your function must accept two arguments: each slice as a separate image and a mask. Your function must also return a mask image.

When you are done editing the template, save the file. The Volume Segmenter app automatically creates a button in the **Automate** tab toolstrip for your function. To test your function on one slice, click **Run**. By default, the app applies the function to only the current slice.

Your function appears selected in the **Automate** toolstrip.

Click **Run** to test your function on the current slice.

After testing your function on a single slice, you can run it on all of the slices or a subset of the slices. You can run it from the current slice to the end (the highest numbered slice) or from the current slice back to the beginning (slice 1). You can also specify a range of slices by specifying the starting slice and the ending slice.

When you choose one of the directional options, the app updates the slice numbers in the display. You can use this display to view the progress of processing.

Create Additional Labels

After labeling the brain on each slice, label the tumor wherever it appears on a slice, repeating the process described previously.

First, define a new label in the **Labels** pane. Click the Plus sign in the **Labels** pane to create a new label.

Click the plus sign to
create a new label.

Rename the
label.
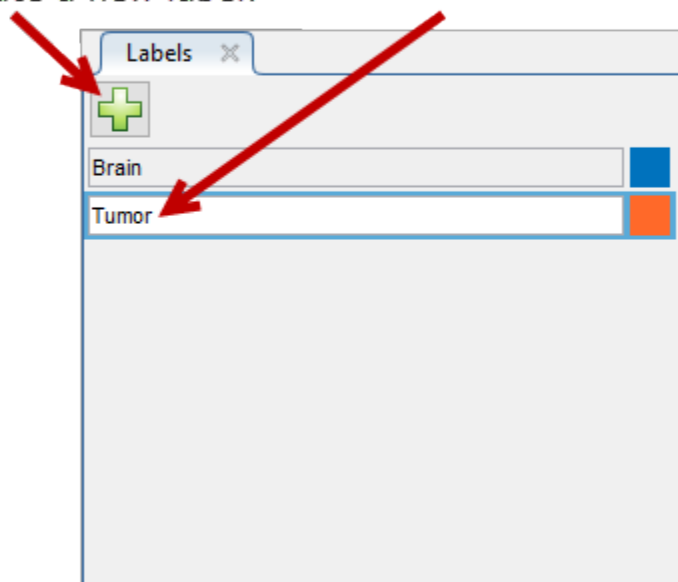
In the **Slice** pane, navigate to the slice where the object first appears and start labeling the object on each slice using a drawing tool. In the following figure, this example uses the Paint Brush tool to label the tumor. As previously, you can draw the object on each slice where it appears or use the interpolation tools to draw on multiple slices automatically. After interpolation, you can use drawing tools, such as the **Eraser**, to modify the automated segmentation on each slice.

Choose drawing tool

Draw a label over the tumor
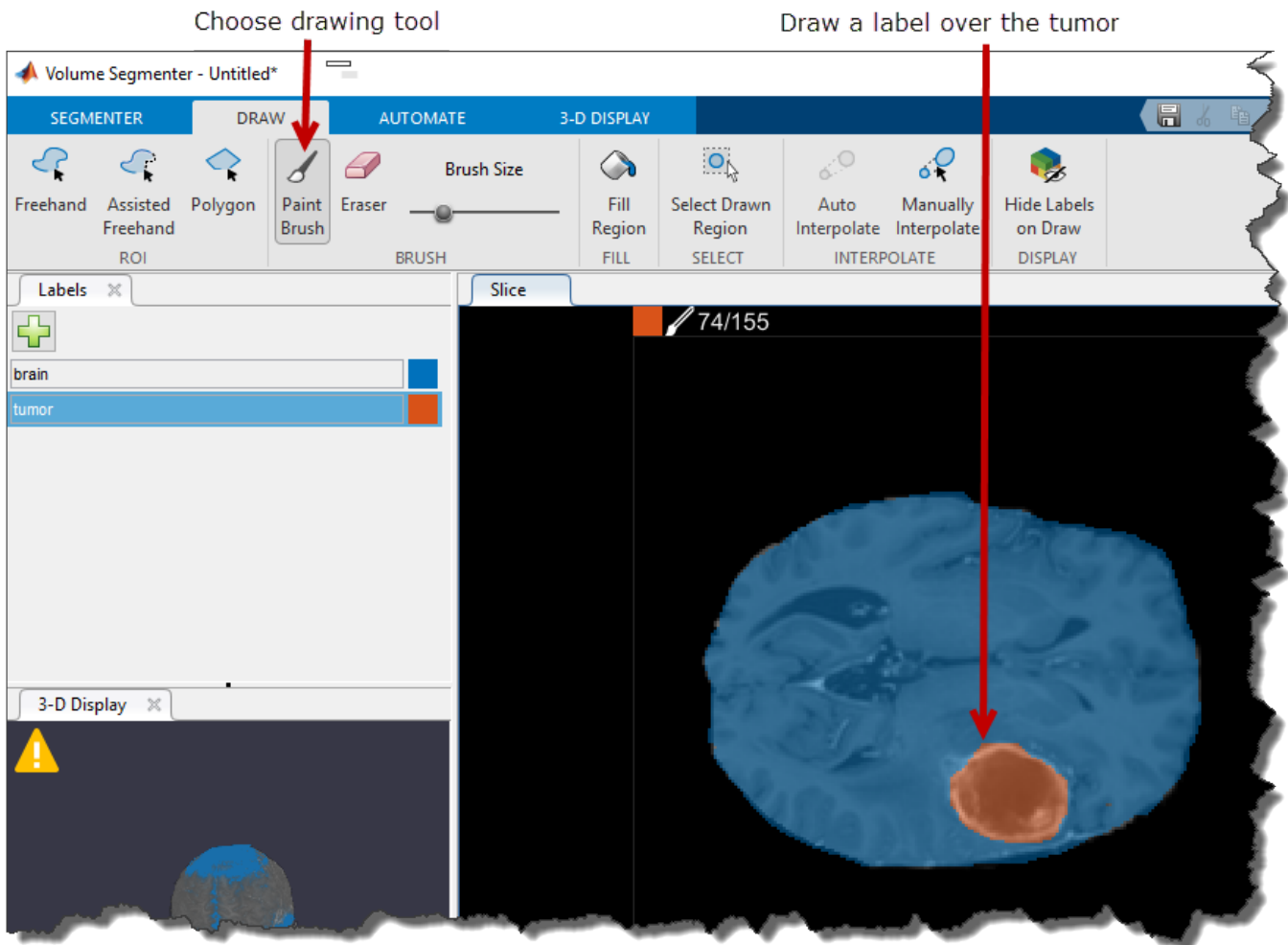
Save the Segmentation

When you complete labeling the brain and the tumor in the volume, save the segmentation. Click **Save Labels** on the **Segmenter** tab and choose from several options.You can save the labeled MRI data as a MAT-file or as a variable in the workspace. For this example, choose a workspace variable. When you define multiple labels, the Volume Segmenter app creates a categorical volume in the workspace that is the same size as the input volume. After you save the segmentation, you can optionally turn on **Autosave,** which periodically saves the segmentation automatically.

Choose **Save** option.
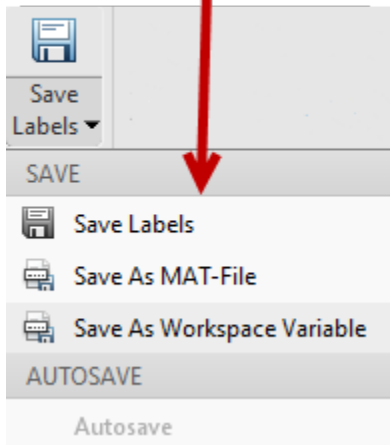
Make the Background a Separately Labelled Region

After labeling the brain and the tumor, and saving the segmentation to the workspace as a categorical volume, you might notice that the background voxels all have the value `<undefined>`. To label the background voxels so that they have a recognizable categorization as well, follow a similar process to that previously described:

1   Define a new label in the **Labels** pane, give the label a descriptive name, and select the color you want for the background.

2   Label the background on each slice. Navigate to a slice, select **Fill Region** on the **Draw** tab, and click anywhere in the background. Repeat this process on each slice.
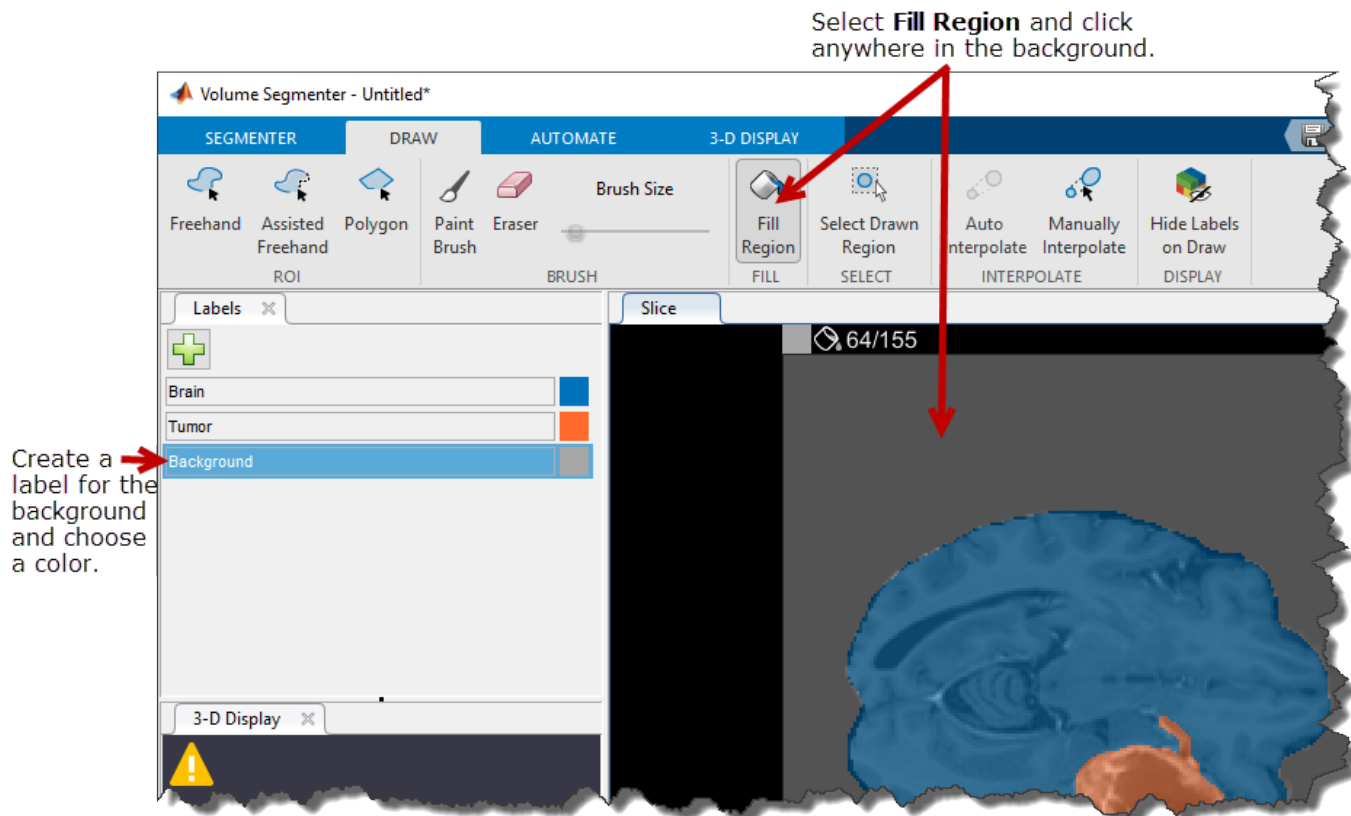
When you add a background, it can obscure the other labels in the visualization of the volume in the **3-D Display** pane. To view the other labeled regions in the **3-D Display** pane, disable the visibility of the background label. Click **Show Labels** in the **3-D Display** tab, click **Customize**, and deselect the visibility of the background label.

View the Labeled Volume

To view the mask, use the `labelvolshow` function: `labelvolshow(brain_labels);`.

See Also
Volume Segmenter

Related Examples
- "Create Binary Mask Using Volume Segmenter" on page 13-136

Image Deblurring

This chapter describes how to deblur an image using the toolbox deblurring functions.

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**.

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

Note The image **f** does not really exist. This image represents what you would have if you had perfect image acquisition conditions.

n	Additive noise, introduced during image acquisition, that corrupts the image

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 Images Using a Wiener Filter" on page 14-5.

```
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 Images Using a Wiener Filter" on page 14-5 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 Images Using a Regularized Filter" on page 14-12 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 "Adapt the Lucy-Richardson Deconvolution for Various Image Distortions" on page 14-22 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 "Adapt Blind Deconvolution for Various Image Distortions" on page 14-37 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 14-55 for more information.

- For information about creating your own deblurring functions, see "Create Your Own Deblurring Functions" on page 14-54.

Deblur Images Using a Wiener Filter

This example shows how to use Wiener deconvolution to deblur images. Wiener deconvolution can be used effectively when the frequency characteristics of the image and additive noise are known, to at least some degree.

Read Pristine Image

Read and display a pristine image that does not have blur or noise.

```
Ioriginal = imread('cameraman.tif');
imshow(Ioriginal)
title('Original Image')
```


Original Image

Simulate and Restore Motion Blur Without Noise

Simulate a blurred image that might result from camera motion. First, create a point-spread function, PSF, by using the `fspecial` function and specifying linear motion across 21 pixels at an angle of 11 degrees. Then, convolve the point-spread function with the image by using `imfilter`.

The original image has data type `uint8`. If you pass a `uint8` image to `imfilter`, then the function will quantize the output in order to return another `uint8` image. To reduce quantization errors, convert the image to `double` before calling `imfilter`.

```
PSF = fspecial('motion',21,11);
Idouble = im2double(Ioriginal);
blurred = imfilter(Idouble,PSF,'conv','circular');
imshow(blurred)
title('Blurred Image')
```

Blurred Image

Restore the blurred image by using the `deconvwnr` function. The blurred image does not have noise so you can omit the noise-to-signal (NSR) input argument.

```
wnr1 = deconvwnr(blurred,PSF);
imshow(wnr1)
title('Restored Blurred Image')
```

Restored Blurred Image

Simulate and Restore Motion Blur and Gaussian Noise

Add zero-mean Gaussian noise to the blurred image by using the `imnoise` function.

```
noise_mean = 0;
noise_var = 0.0001;
blurred_noisy = imnoise(blurred,'gaussian',noise_mean,noise_var);
imshow(blurred_noisy)
title('Blurred and Noisy Image')
```


Blurred and Noisy Image

Try to restore the blurred noisy image by using `deconvwnr` without providing a noise estimate. By default, the Wiener restoration filter assumes the NSR is equal to 0. In this case, the Wiener restoration filter is equivalent to an ideal inverse filter, which can be extremely sensitive to noise in the input image.

In this example, the noise in this restoration is amplified to such a degree that the image content is lost.

```
wnr2 = deconvwnr(blurred_noisy,PSF);
imshow(wnr2)
title('Restoration of Blurred Noisy Image (NSR = 0)')
```

Restoration of Blurred Noisy Image (NSR = 0)
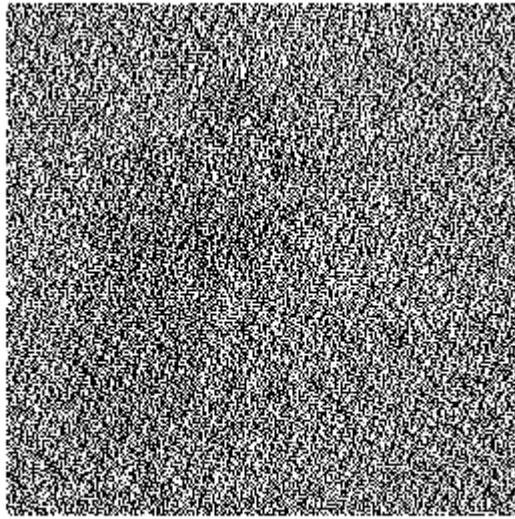
Try to restore the blurred noisy image by using `deconvwnr` with a more realistic value of the estimated noise.

```
signal_var = var(Idouble(:));
NSR = noise_var / signal_var;
wnr3 = deconvwnr(blurred_noisy,PSF,NSR);
imshow(wnr3)
title('Restoration of Blurred Noisy Image (Estimated NSR)')
```

Restoration of Blurred Noisy Image (Estimated NSR)

Simulate and Restore Motion Blur and 8-Bit Quantization Noise

Even a visually imperceptible amount of noise can affect the result. One source of noise is quantization errors from working with images in `uint8` representation. Earlier, to avoid quantization errors, this example simulated a blurred image from a pristine image in data type `double`. Now, to explore the impact of quantization errors on the restoration, simulate a blurred image from the pristine image in the original `uint8` data type.

```
blurred_quantized = imfilter(Ioriginal,PSF,'conv','circular');
imshow(blurred_quantized)
title('Blurred Quantized Image')
```

Blurred Quantized Image

Try to restore the blurred quantized image by using `deconvwnr` without providing a noise estimate. Even though no additional noise was added, this restoration is degraded compared to the restoration of the blurred image in data type `double`.

```
wnr4 = deconvwnr(blurred_quantized,PSF);
imshow(wnr4)
title('Restoration of Blurred Quantized Image (NSR = 0)');
```

Restoration of Blurred Quantized Image (NSR = 0)

Try to restore the blurred quantized image by using `deconvwnr` with a more realistic value of the estimated noise.

```
uniform_quantization_var = (1/256)^2 / 12;
signal_var = var(Idouble(:));
NSR = uniform_quantization_var / signal_var;
wnr5 = deconvwnr(blurred_quantized,PSF,NSR);
imshow(wnr5)
title('Restoration of Blurred Quantized Image (Estimated NSR)');
```

Restoration of Blurred Quantized Image (Estimated NSR)

See Also

deconvwnr | fspecial | imfilter | imnoise

More About

- "Image Deblurring" on page 14-2

Deblur Images Using a Regularized Filter

This example shows how to use regularized deconvolution to deblur images. Regularized deconvolution can be used effectively when limited information is known about the additive noise and constraints (such as smoothness) are applied on the recovered image. The blurred and noisy image is restored by a constrained least square restoration algorithm that uses a regularized filter.

Simulate Gaussian Blur and Gaussian Noise

Read and display a pristine image that does not have blur or noise.

```
I = im2double(imread('tissue.png'));
imshow(I);
title('Original Image');
text(size(I,2),size(I,1)+15, ...
    'Image courtesy of Alan Partin, Johns Hopkins University', ...
    'FontSize',7,'HorizontalAlignment','right');
```

Original Image
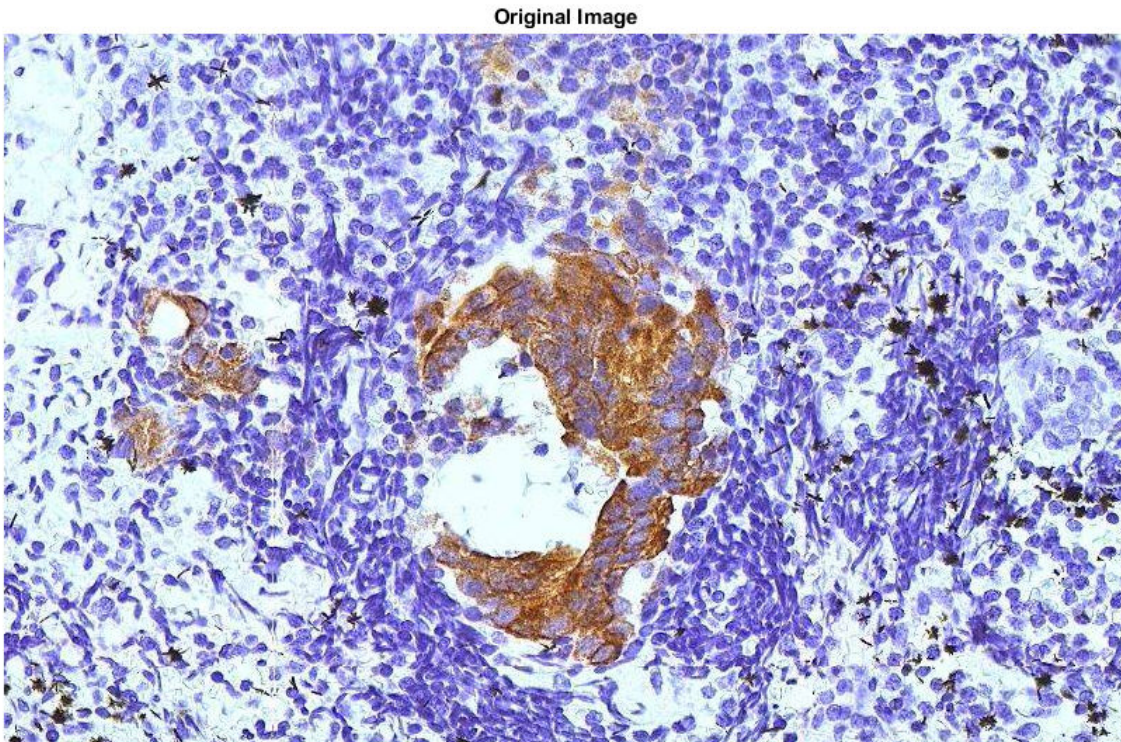
Image courtesy of Alan Partin, Johns Hopkins University

Simulate a blurred image that might result from an out-of-focus lens. First, create a point-spread function, PSF, by using the fspecial function and specifying a Gaussian filter of size 11-by-11 and standard deviation 5. Then, convolve the point-spread function with the image by using imfilter.

```
PSF = fspecial('gaussian',11,5);
blurred = imfilter(I,PSF,'conv');
```

Add zero-mean Gaussian noise to the blurred image by using the imnoise function.

```
noise_mean = 0;
noise_var = 0.02;
blurred_noisy = imnoise(blurred,'gaussian',noise_mean,noise_var);
```

Display the blurred noisy image.

```
imshow(blurred_noisy)
title('Blurred and Noisy Image')
```


Blurred and Noisy Image
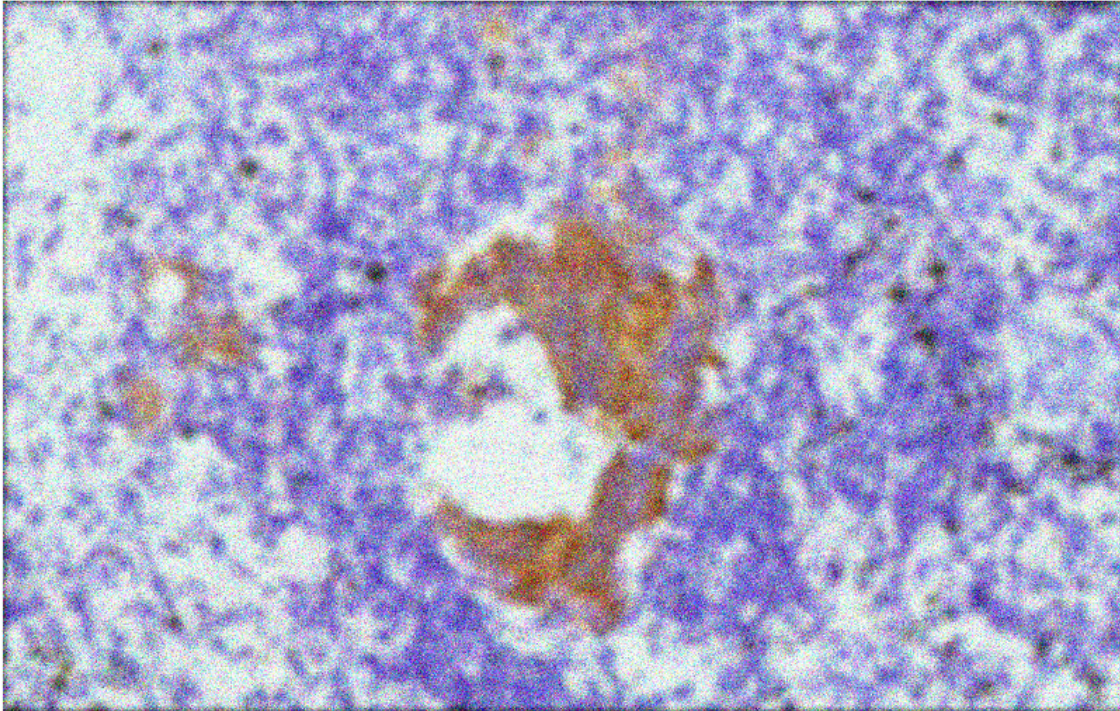
Restore Image Using Estimated Noise Power

Restore the blurred image by using the `deconvreg` function, supplying the noise power (NP) as the third input parameter. To illustrate how sensitive the algorithm is to the value of noise power, this example performs three restorations.

For the first restoration, use 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 regularized deconvolution has converged. This value is used later in the example.

```
NP = noise_var*numel(I);
[reg1,lagra] = deconvreg(blurred_noisy,PSF,NP);
imshow(reg1)
title('Restored with True NP')
```

Restored with True NP
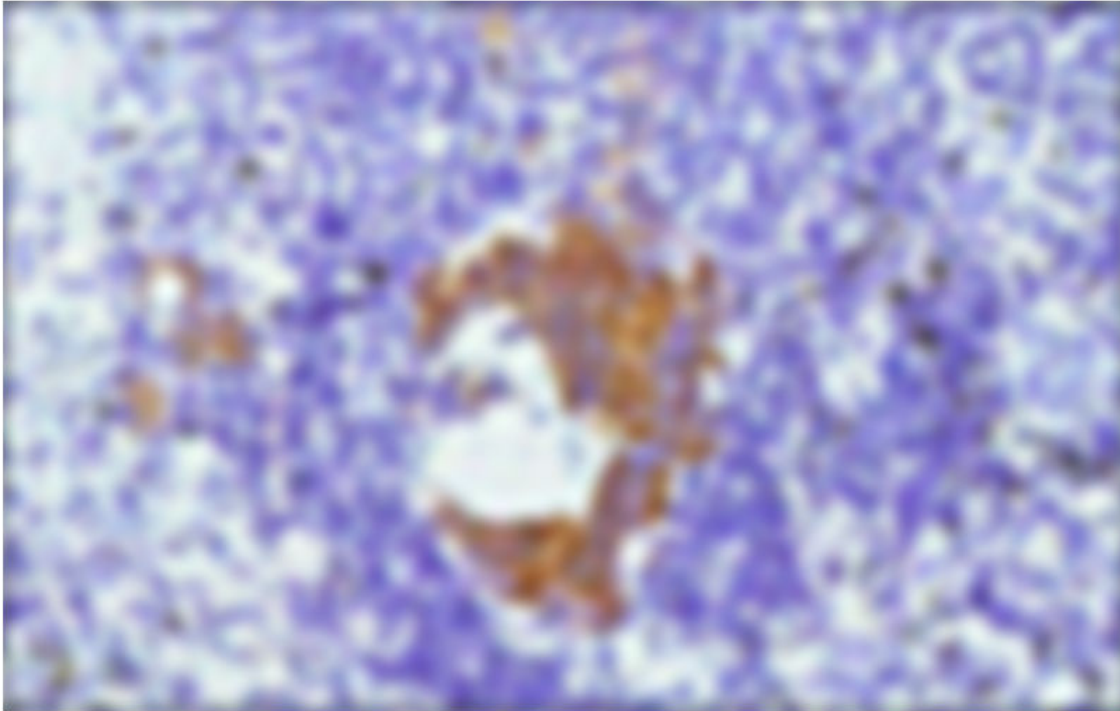
For the second restoration, use a slightly overestimated noise power. The restoration has poor resolution.

```
reg2 = deconvreg(blurred_noisy,PSF,NP*1.3);
imshow(reg2)
title('Restored with Larger NP')
```

Restored with Larger NP
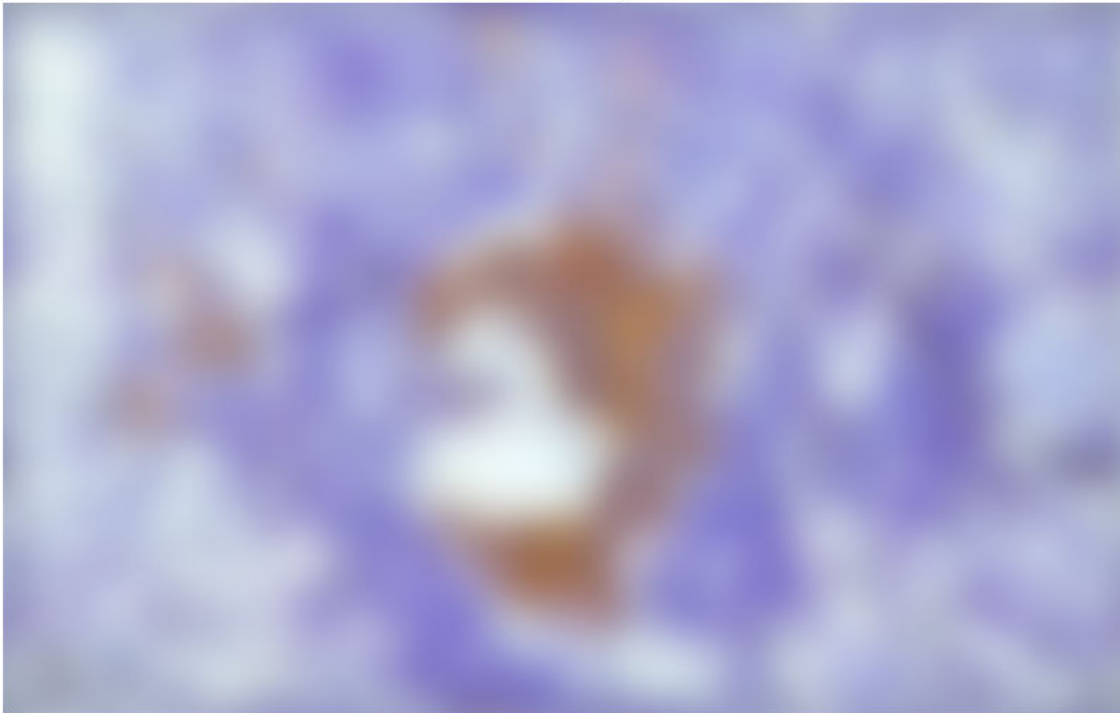
For the third restoration, use a slightly underestimated noise power. The restoration has overwhelming noise amplification and ringing from the image borders.

```
reg3 = deconvreg(blurred_noisy,PSF,NP/1.3);
imshow(reg3)
title('Restored with Smaller NP')
```

Restored with Smaller NP
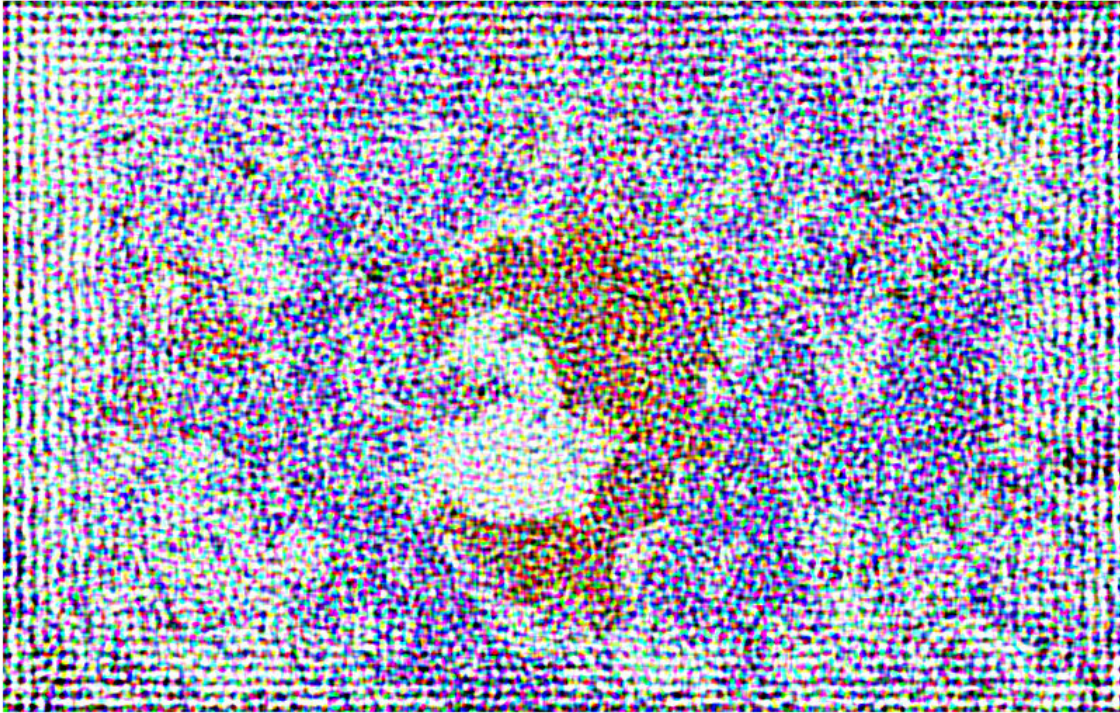
Reduce Noise Amplification and Ringing

You can reduce the noise amplification and ringing along the boundary of the image by calling the `edgetaper` function prior to deconvolution. The image restoration becomes less sensitive to the noise power parameter.

```
Edged = edgetaper(blurred_noisy,PSF);
reg4 = deconvreg(Edged,PSF,NP/1.3);
imshow(reg4)
title('Restored with Smaller NP and Edge Tapering')
```

Restored with Smaller NP and Edge Tapering
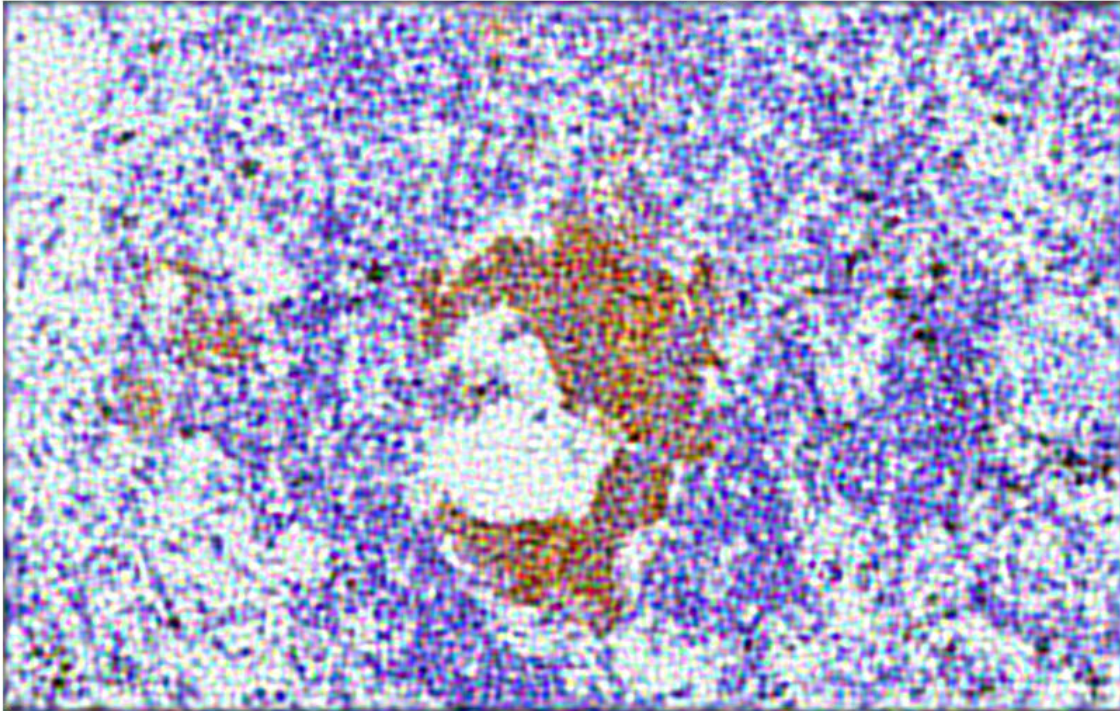
Use the Lagrange Multiplier

Restore the blurred and noisy image, assuming that the optimal solution is already found and the corresponding Lagrange multiplier is known. In this case, any value passed for noise power, NP, is ignored.

To illustrate how sensitive the algorithm is to the Lagrange multiplier, this example performs three restorations. The first restoration uses the `lagra` output from the `reg1` restoration performed earlier.

```
reg5 = deconvreg(Edged,PSF,[],lagra);
imshow(reg5)
title('Restored with LAGRA')
```

Restored with LAGRA
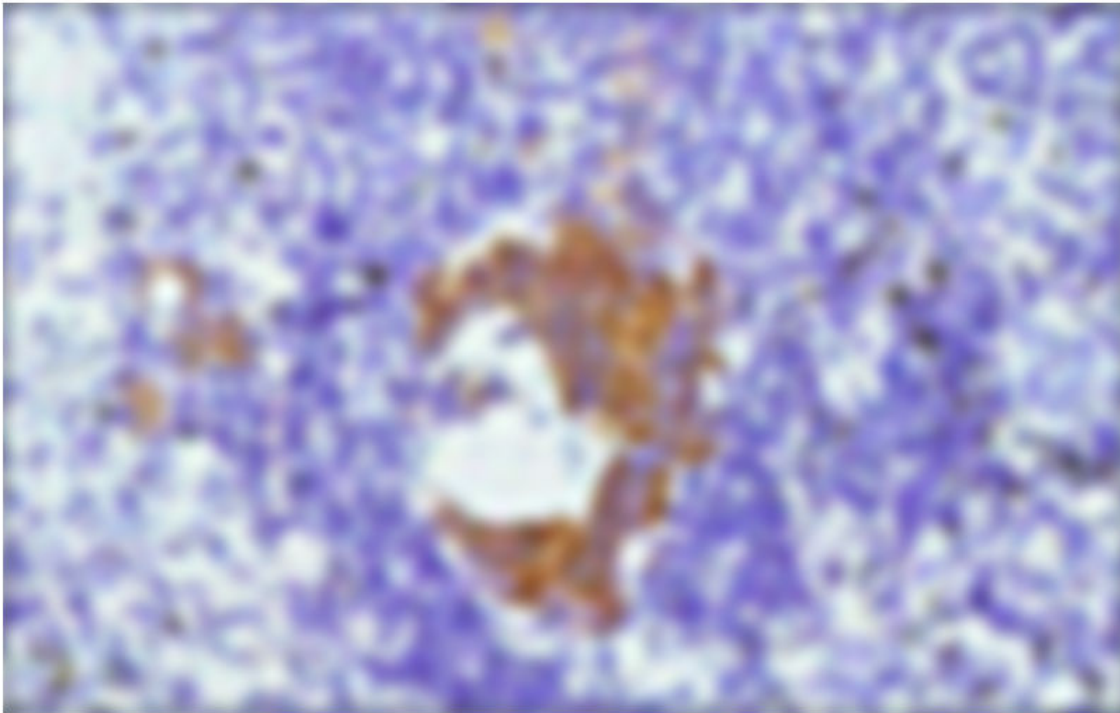
The second restoration uses 100*`lagra` which increases the significance of the constraint. By default, this leads to oversmoothing of the image.

```
reg6 = deconvreg(Edged,PSF,[],lagra*100);
imshow(reg6)
title('Restored with Large LAGRA')
```

Restored with Large LAGRA
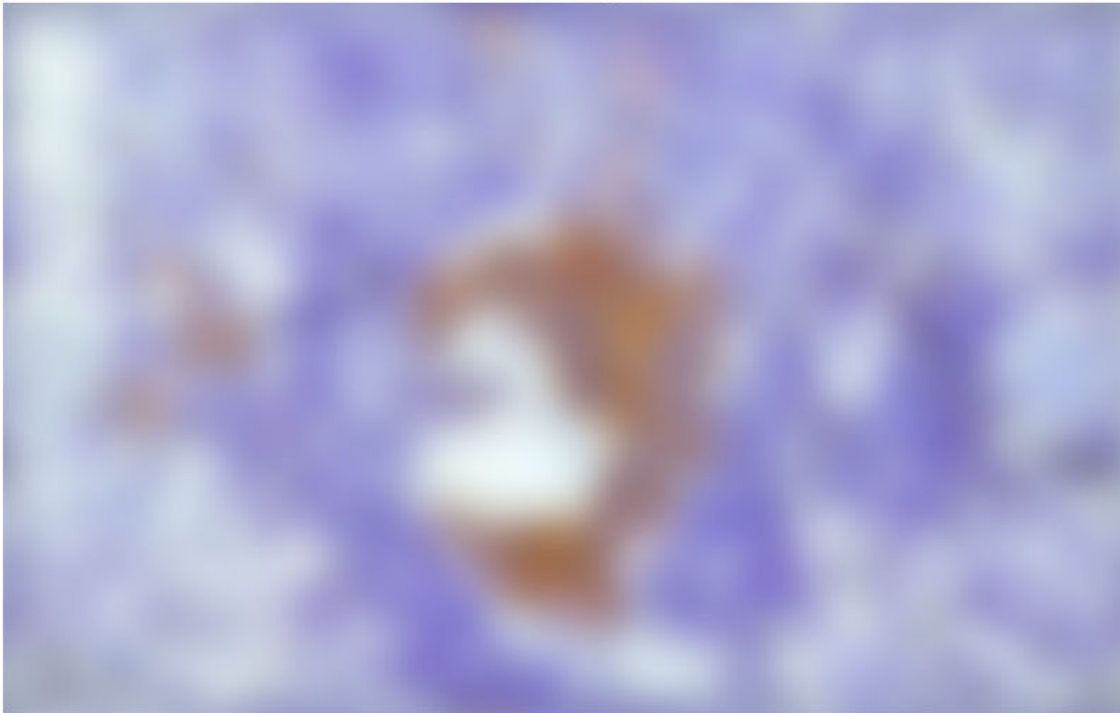
The third restoration uses `lagra`/100 which weakens the constraint (the smoothness requirement set for the image). It amplifies the noise. For the extreme case when the Lagrange multiplier equals 0, the reconstruction is a pure inverse filtering.

```
reg7 = deconvreg(Edged,PSF,[],lagra/100);
imshow(reg7)
title('Restored with Small LAGRA')
```

Restored with Small LAGRA
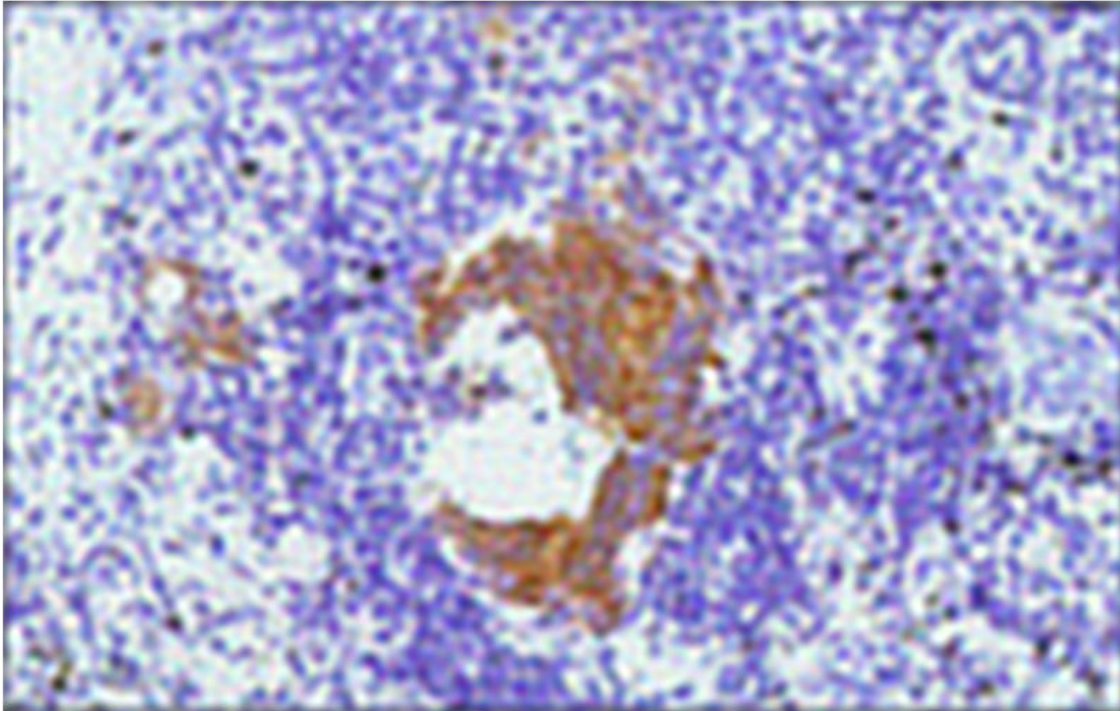
Use a Different Smoothness Constraint

Restore the blurred and noisy image using a different constraint for the regularization operator.
Instead of using the default Laplacian constraint on image smoothness, constrain the image
smoothness only in one dimension (1-D Laplacian).

```
regop = [-1 2 -1];
reg8 = deconvreg(blurred_noisy,PSF,[],lagra,regop);
imshow(reg8)
title('Constrained by 1-D Laplacian')
```

Constrained by 1-D Laplacian

See Also

deconvreg | fspecial | imfilter | imnoise

More About

- "Image Deblurring" on page 14-2

Adapt the Lucy-Richardson Deconvolution for Various Image Distortions

In this section...
"Reduce the Effect of Noise Amplification" on page 14-22
"Account for Nonuniform Image Quality" on page 14-22
"Handle Camera Read-Out Noise" on page 14-23
"Handling Undersampled Images" on page 14-23
"Refine the Result" on page 14-23

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.

Reduce 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 14-55 for more information.

Account 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` argument 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.

Handle 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 argument 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, such as the number of counts from the background radiation. The value of the `READOUT` argument 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.

Refine 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

See Also
`deconvblind` | `deconvlucy`

Related Examples

More About

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
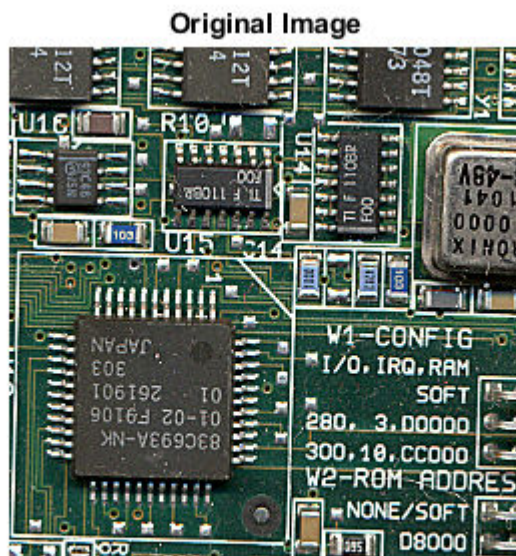
Image courtesy of courtesy of Alexander V. Panasyuk, Ph.D.
Harvard-Smithsonian Center for Astrophysics

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');
```
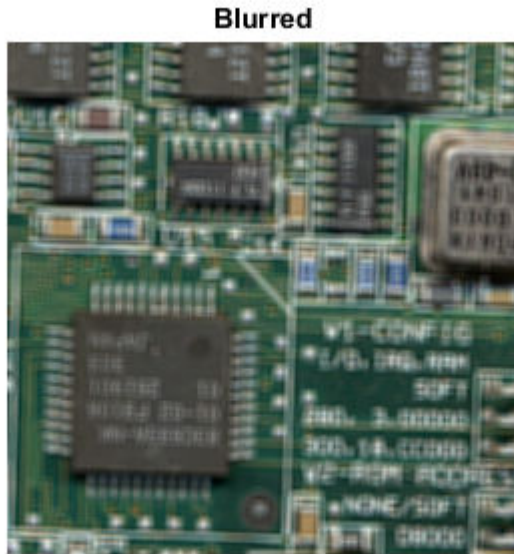
Blurred

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');
```

Blurred & Noisy
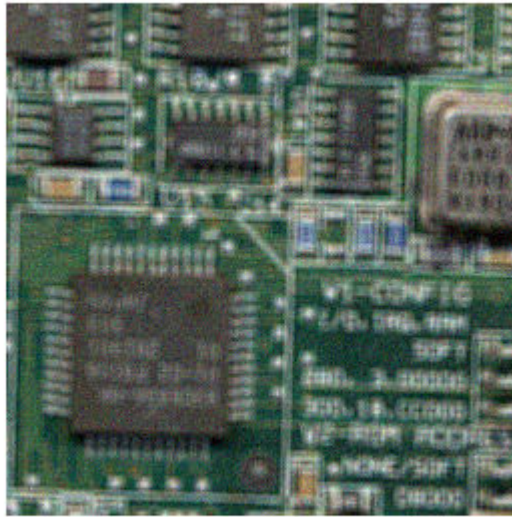
Step 3: Restore the Blurred and Noisy Image

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');
```

Restored Image, NUMIT = 5
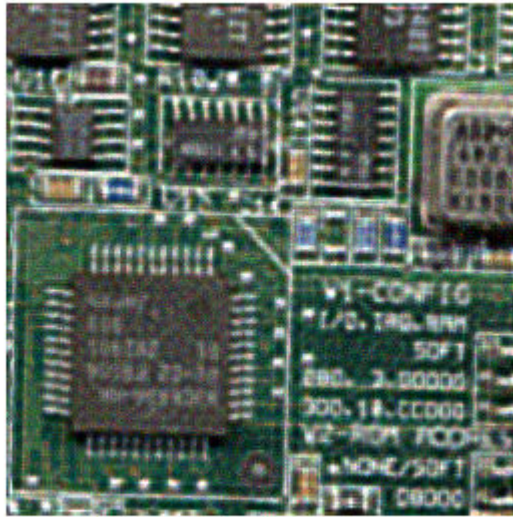
Step 4: Iterate to Explore the Restoration

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');
```

Restored Image, NUMIT = 15
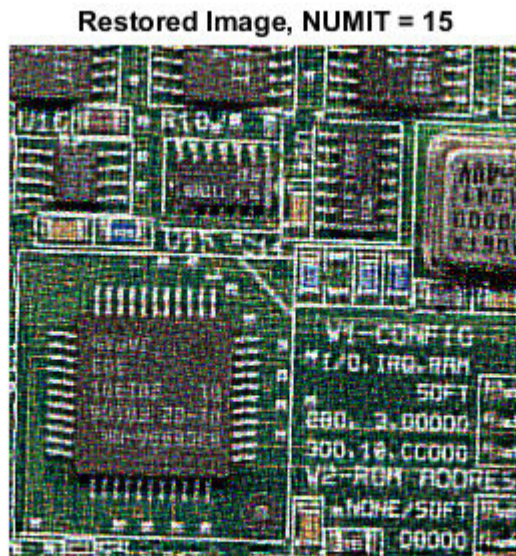
Step 5: Control Noise Amplification by Damping

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');
```

Restored Image with Damping, NUMIT = 15
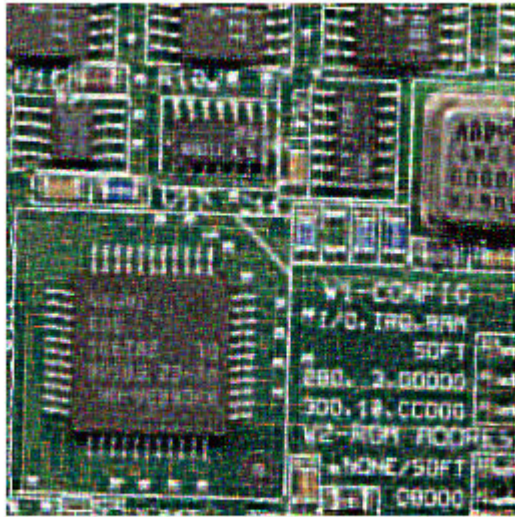
The next part of this example explores the `WEIGHT` and `SUBSMPL` input parameters of the deconvlucy function, using a simulated star image (for simplicity & speed).

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');
```

Data
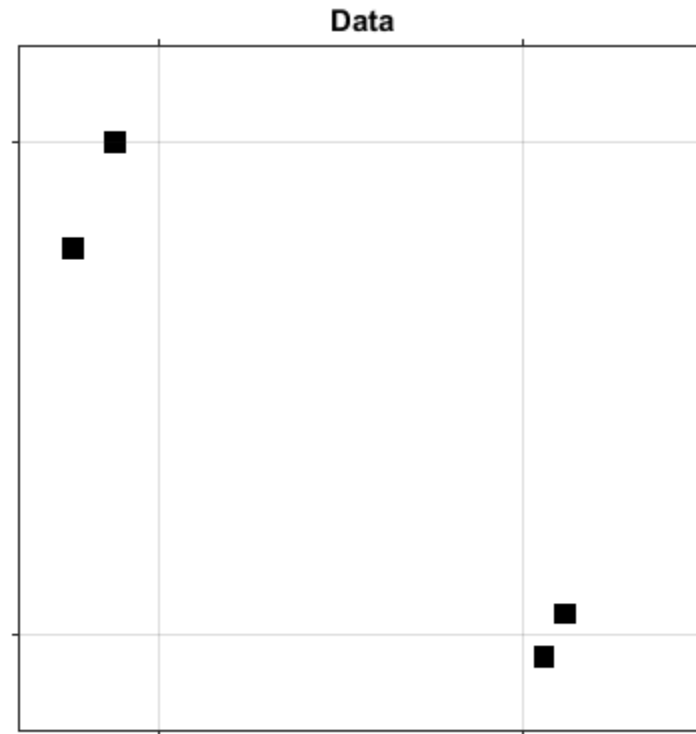
Step 7: Simulate a Blur

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');
```


Observed
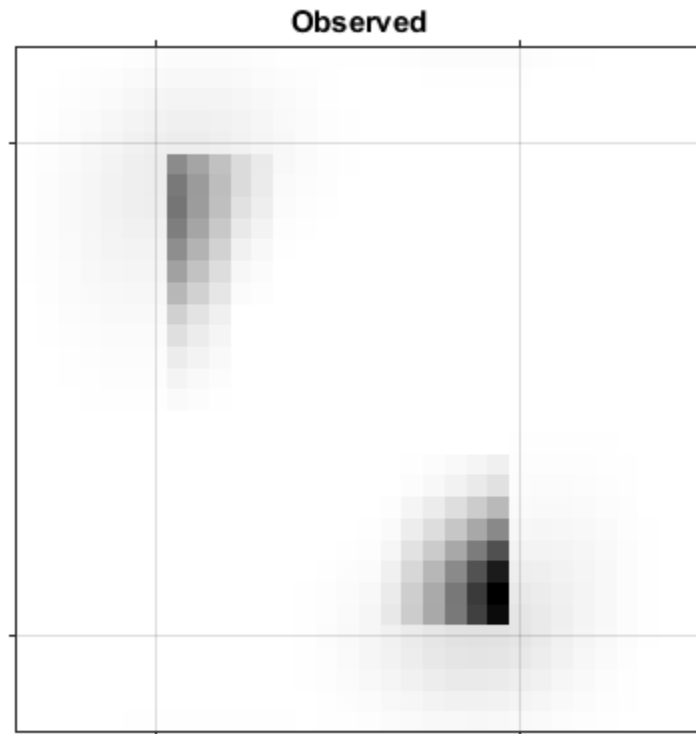
Step 8: Provide the WEIGHT Array

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');
```
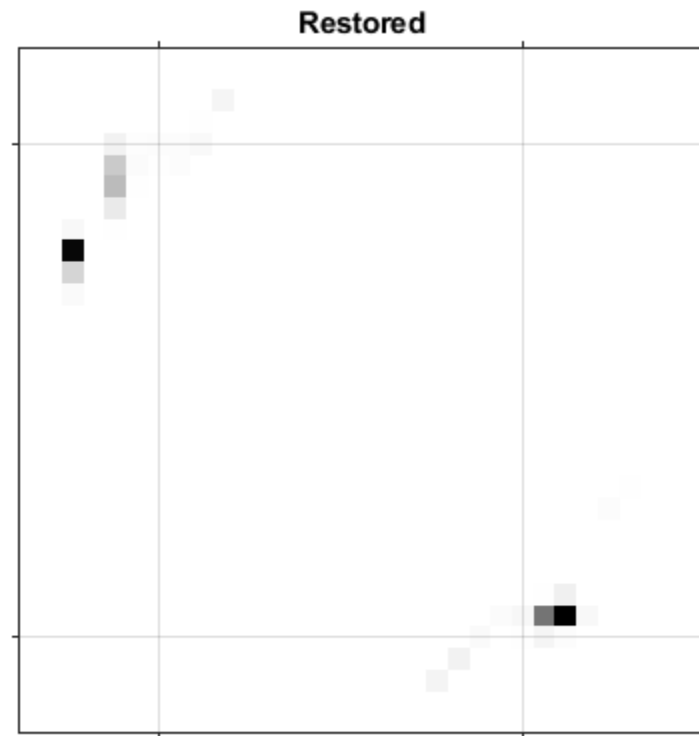
Step 9: Provide a finer-sampled PSF

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');
```

Binned Observed
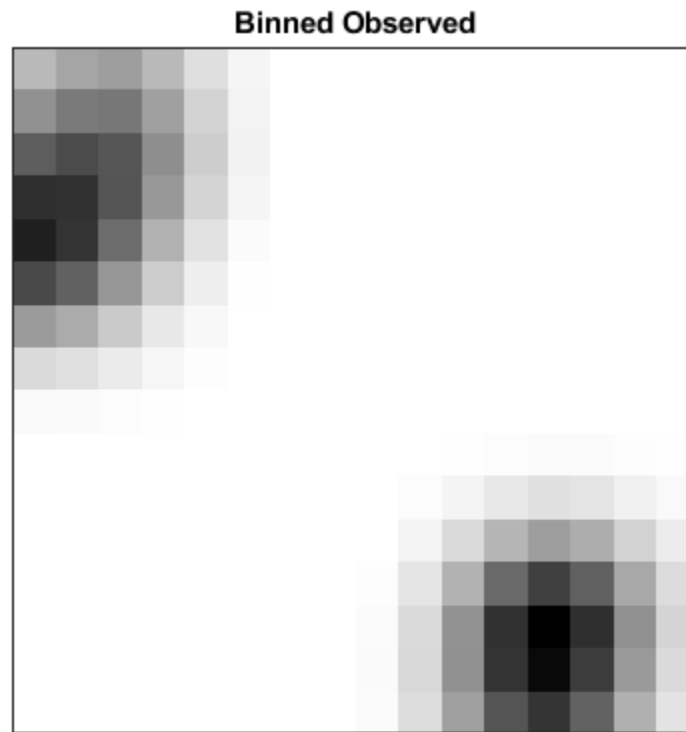
Restore the undersampled image, `BinnedImage`, using the undersampled PSF, `BinnedPSF`. Notice that the `luc5` image distinguishes only 3 stars.

```
luc5 = deconvlucy(BinnedImage,BinnedPSF,100);
figure;
imshow(1-luc5,[],'InitialMagnification','fit');
ax = gca;
ax.Visible = 'on';
ax.XTick = [];
ax.YTick = [];
title('Poor PSF');
```

Poor PSF
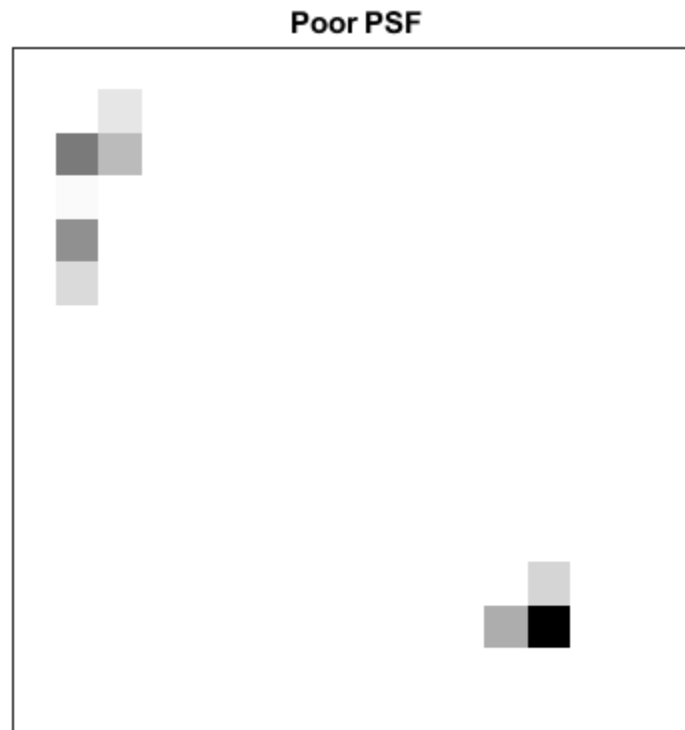
The next example restores the undersampled image (`BinnedImage`), this time using the finer PSF (defined on a SUBSMPL-times finer grid). The reconstructed image (`luc6`) resolves the position of the stars more accurately. Note how it distributes power between the two stars in the lower right corner of 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');
```

Fine PSF
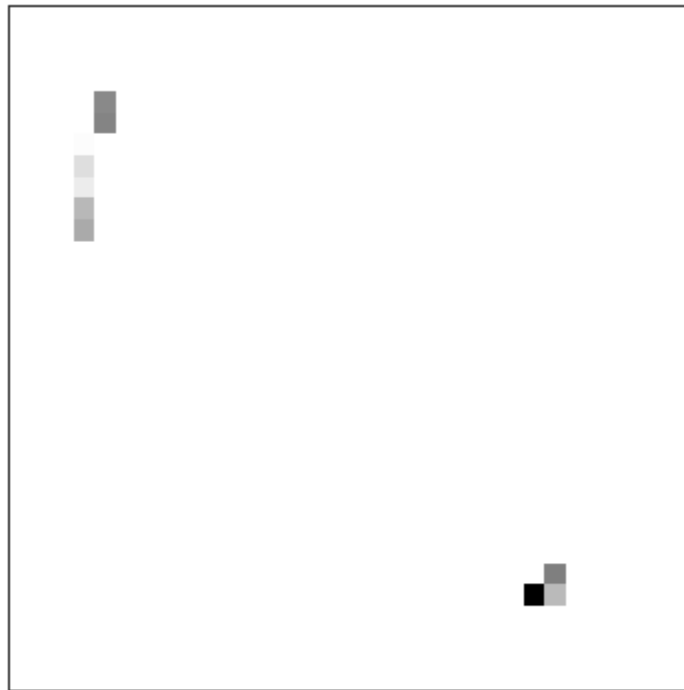
See Also

deconvblind | deconvlucy | deconvreg | deconvwnr

More About

* "Image Deblurring" on page 14-2
* "Adapt the Lucy-Richardson Deconvolution for Various Image Distortions" on page 14-22

Adapt Blind Deconvolution for Various Image Distortions

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 (such as bad pixels)
- Handle camera read-out noise

For more information about these adaptations, see "Adapt the Lucy-Richardson Deconvolution for Various Image Distortions" on page 14-22. The `deconvblind` function also supports PSF constraints that you can provide 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')
```

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')
```

Original PSF
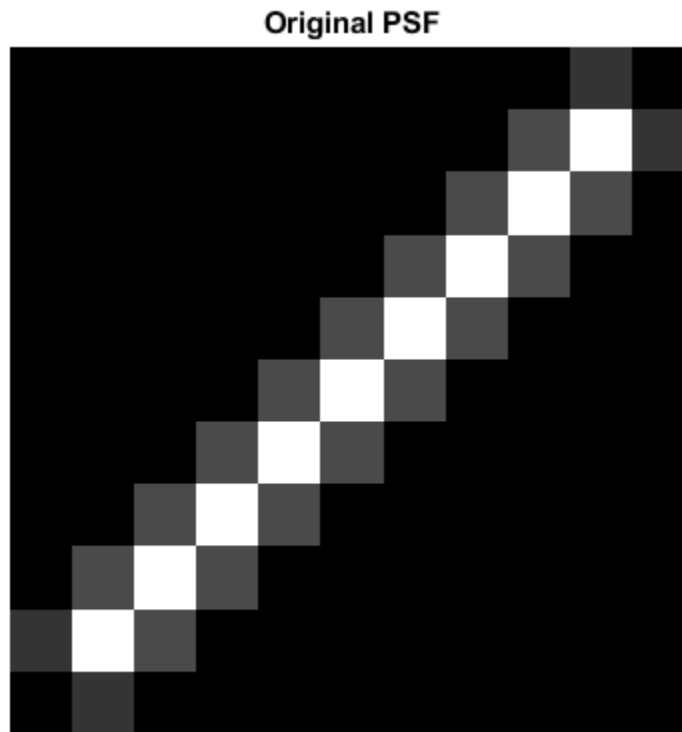
Create a simulated blur in the image, using the PSF, and display the blurred image.

```
Blurred = imfilter(I,PSF,'circ','conv');
figure
imshow(Blurred)
title('Blurred Image')
```

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')
```

Restored Image


```
figure
imshow(P,[],'InitialMagnification','fit')
title('Restored PSF')
```

Restored PSF
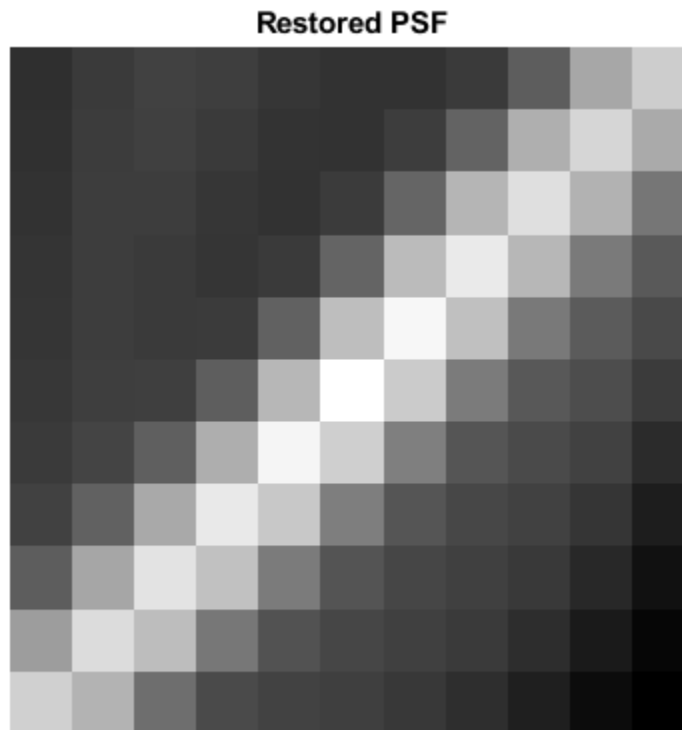
One way to improve the result is to create a weight array to exclude areas of high contrast from the deblurring operation. This can reduce contrast-related ringing in the result.

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')
```

Weight Array
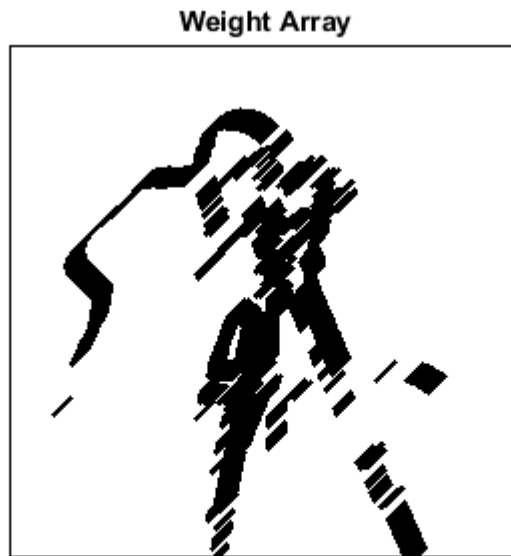
Refine the guess at the PSF. The reconstructed PSF returned by the first pass at deconvolution, `P` , shows a clear linearity. For this second pass, the example uses a new PSF which is the same as the returned PSF but with the small amplitude pixels set to 0.

```
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');
```

Newly Deblurred Image

```
figure, imshow(P2,[],'InitialMagnification','fit')
title('Newly Reconstructed PSF')
```


Newly Reconstructed PSF
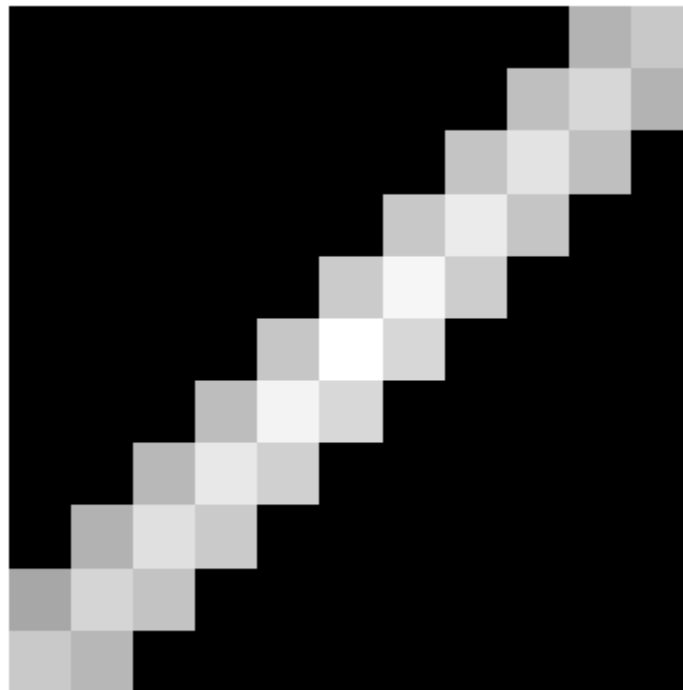
Refining the Result

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 `deconvblind` to know where to restart the process

The PSF output cell array contains similar elements.

See Also
`deconvblind` | `deconvlucy`

Related Examples
- "Deblurring Images Using the Blind Deconvolution Algorithm" on page 14-46

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');
```


Original Image

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')
```

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')
```

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')
```

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')
```


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')
```
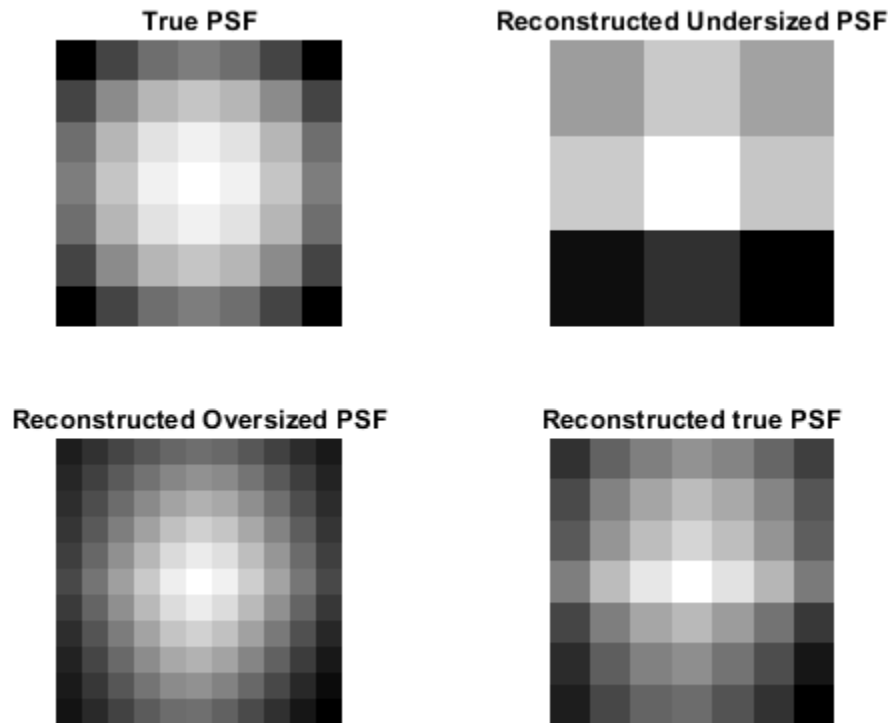
The PSF reconstructed in the first restoration, `P1`, obviously does not fit into the constrained size. It has a strong signal variation at the borders. The corresponding image, `J1`, does not show any improved clarity vs. the blurred image, `Blurred`.

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')
```
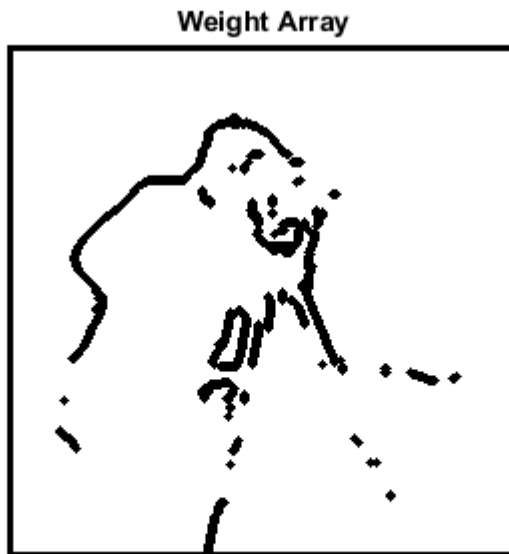

Weight Array

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')
```

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')
```

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

More About
- "Image Deblurring" on page 14-2
- "Adapt Blind Deconvolution for Various Image Distortions" on page 14-37

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.

To aid this conversion between PSFs and OTFs, use the padding function `padarray`.

See Also
`deconvblind`

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.
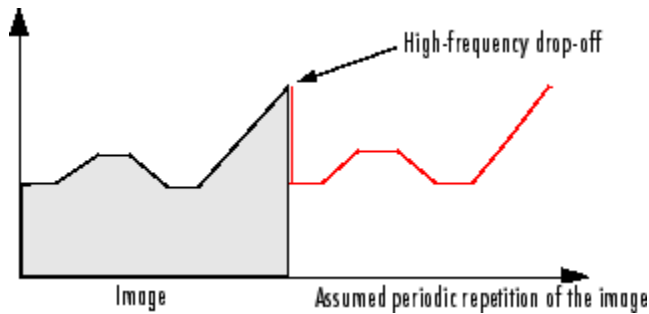
This high-frequency drop-off can create an effect called *boundary related ringing* in deblurred images. In this figure, note the horizontal and vertical patterns in the image.
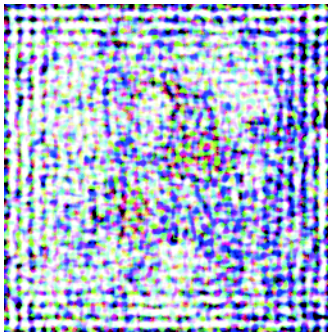
To avoid ringing, use the `edgetaper` function to preprocess your images before passing them to the deblurring functions. The `edgetaper` function removes the high-frequency drop-off at the edge of an image by blurring the entire image and then replacing the center pixels of the blurred image with the 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

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 (that is, 2^5) levels each of red, green, and blue. This supports 32,768 (that is, 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 (that is, 2^{16}).
24	24-bit displays use 8 bits for each of the three color components, resulting in 256 (that is, 2^8) levels each of red, green, and blue. This supports 16,777,216 (that is, 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".

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 15-3.

Reduce the Number of Colors in an Image

On systems with 24-bit color displays, truecolor images can display up to 16,777,216 (that is, 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, color maps 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 15-3
"Reduce Colors of Indexed Image Using imapprox" on page 15-7
"Reduce Colors Using Dithering" on page 15-7

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 15-3)
  - Uniform quantization
  - Minimum variance quantization
- Color map mapping (described in "Color map Mapping" on page 15-6)

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 15-7 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.
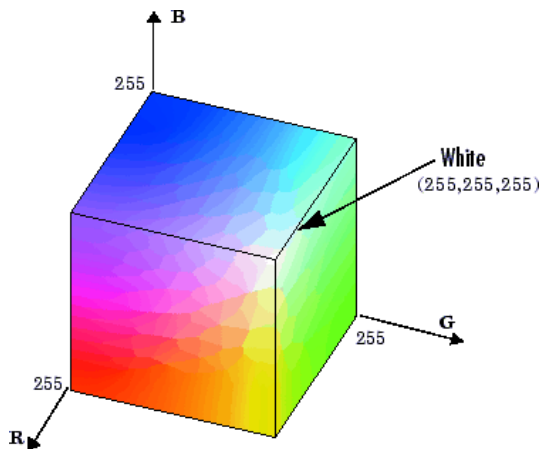
RGB Color Cube for uint8 Images

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`, then 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.
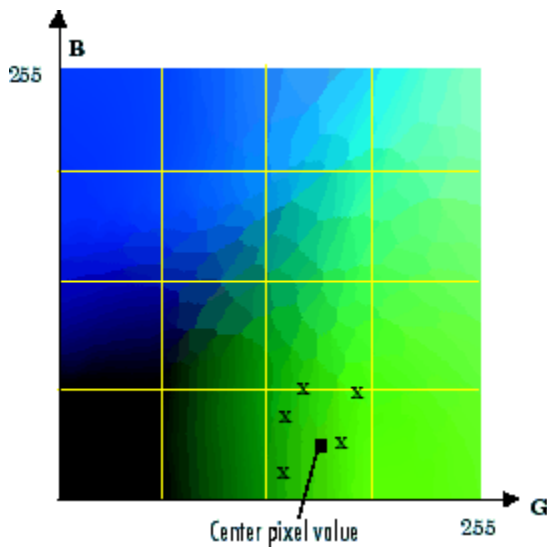
Uniform Quantization on a Slice of the RGB Color Cube

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 color map. As shown earlier, the maximum length of a color map created by uniform quantization can be predicted, but the color map 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 color map. 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 color map 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 color map 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.
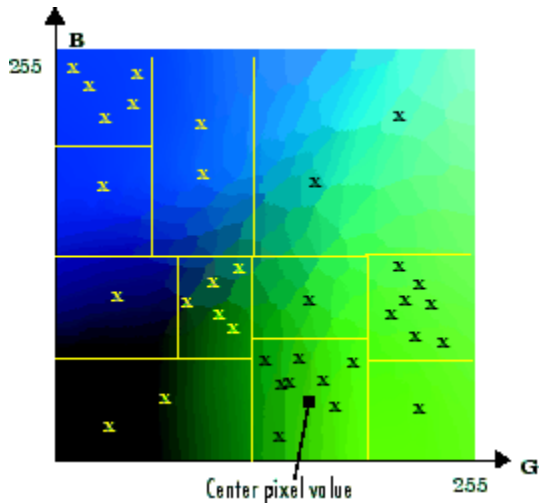
Minimum Variance Quantization on a Slice of the RGB Color Cube

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 color map 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 color map. Note that the computation for minimum variance quantization takes longer than that for uniform quantization.

Color map Mapping

If you specify an actual color map to use, `rgb2ind` uses *color map mapping* (instead of quantization) to find the colors in the specified color map that best match the colors in the RGB image. This method is useful if you need to create images that use a fixed color map. 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 color map. Color map mapping produces a good approximation if the specified color map has similar colors to those in the RGB image. If the color map 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 color map. The color map used for the two images is created on the fly using the MATLAB function `colorcube`, which creates an RGB color map containing the number of colors that you specify. (`colorcube` always creates the same color map for a given number of colors.) Because the color map 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 4-9 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 15-7 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 website, http://www.color.org.

Read ICC Profiles

To read an ICC profile into the 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 website, http://www.color.org.

Write ICC Profile Information to a File

To export ICC profile information from the workspace to a file, use the `iccwrite` function. This example reads a profile into the 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 website, http://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 workspace.

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

Read ICC profiles. Read the source and destination profiles into the 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*
*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 typically represents colors as red, green, and blue (RGB) numeric values. 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.

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 (shade of color), saturation (amount of gray or pure color), and luminance (intensity, or overall brightness).

The toolbox enables converting color data from one color space to another through mathematical transformations.

RGB

The RGB color space represents images as an *m*-by-*n*-by-3 numeric array whose elements specify the intensity values of the red, green, and blue color channels. The range of numeric values depends on the data type 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` arrays, RGB values range from [0, 65535].

The toolbox supports variations of the RGB color space.

RGB Color Space	Description
Linear RGB	Linear RGB values are raw data obtained from a camera sensor. The value of R, G, and B are directly proportional to the amount of light that illuminates the sensor. Preprocessing of raw image data, such as white balance, color balance, and chromatic aberration compensation, are performed on linear RGB values.

RGB Color Space	Description
sRGB	sRGB values apply a nonlinear function, called gamma correction on page 11-73, to linear RGB values. Images are frequently displayed in the sRGB color space because they appear brighter and colors are easier to distinguish. The parametric curve used to transform linear RGB values to the sRGB color space is: $f(u) = -f(-u), \qquad u < 0$ $f(u) = c \cdot u, \qquad 0 \le u < d$ $f(u) = a \cdot u^{\gamma} + b, \qquad u \ge d,$ where u represents one of the R, G, or B color values with these parameters: $a = 1.055$ $b = -0.055$ $c = 12.92$ $d = 0.0031308$ $\gamma = 1/2.4$
Adobe RGB (1998)	Adobe RGB (1998) RGB values apply gamma correction to linear RGB values using a simple power function: $v = u^{\gamma}, \qquad u \ge 0$ $v = -(-u)^{\gamma}, \qquad u < 0,$ with $\gamma = 1/2.19921875$

HSV

The HSV (Hue, Saturation, Value) color space corresponds better to how people experience color than the RGB color space does. For example, this color space is often used by people who are selecting colors, such as paint or ink color, from a color wheel or palette.

Attribute	Description
H	Hue, which corresponds to the color's position on a color wheel. H is in the range [0, 1]. As H increases, colors transition from red to orange, yellow, green, cyan, blue, magenta, and finally back to red. Both 0 and 1 indicate red.
S	Saturation, which is the amount of hue or departure from neutral. S is in the range [0, 1]. As S increases, colors vary from unsaturated (shades of gray) to fully saturated (no white component).

Attribute	Description
V	Value, which is the maximum value among the red, green, and blue components of a specific color. V is in the range [0, 1]. As V increases, the corresponding colors become increasingly brighter.
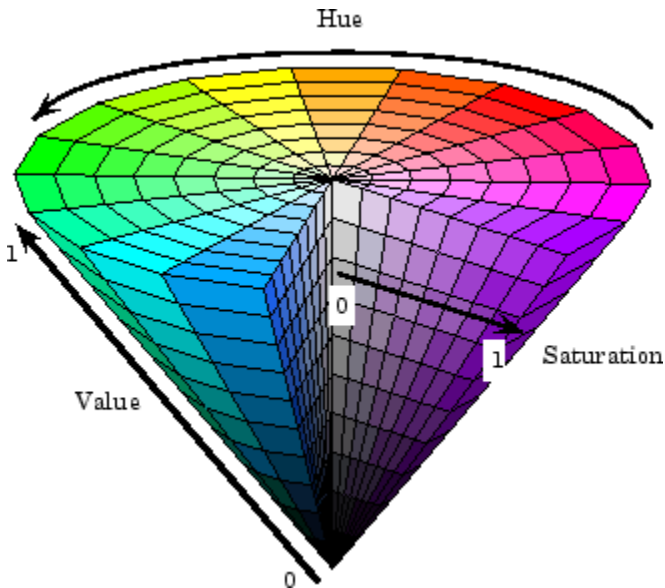
Illustration of the HSV Color Space

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 15-13.

Use the `rgb2hsv` and `hsv2rgb` functions to convert between the RGB and HSV color spaces.

CIE 1976 XYZ and CIE 1976 L*a*b*

CIE 1976 XYZ and CIE 1976 L*a*b* are device-independent color spaces developed by the International Commission on Illumination, known by the acronym CIE. These color spaces model colors according to the typical sensitivity of the three types of cone cells in the human eye.

The XYZ color space is the original model developed by the CIE. The Y channel represents the luminance of a color. The Z channel approximately relates to the amount of blue in an image, but the value of Z in the XYZ color space is not identical to the value of B in the RGB color space. The X channel does not have a clear color analogy. However, if you consider the XYZ color space as a 3-D coordinate system, then X values lie along the axis that is orthogonal to the Y (luminance) axis and the Z axis.

The L*a*b* color space provides a more perceptually uniform color space than the XYZ model. Colors in the L*a*b* color space can exist outside the RGB *gamut* (the valid set of RGB colors). For example, when you convert the L*a*b* value [100, 100, 100] to the RGB color space, the returned value is

[1.7682, 0.5746, 0.1940], which is not a valid RGB color. For more information, see "Determine If L*a*b* Value Is in RGB Gamut" on page 15-24.

Attribute	Description
*L**	Luminance or brightness of the image. Values are in the range [0, 100], where 0 specifies black and 100 specifies white. As *L** increases, colors become brighter.
*a**	Amount of red or green tones in the image. A large positive *a** value corresponds to red/magenta. A large negative *a** value corresponds to green. Although there is no single range for *a**, values commonly fall in the range [-100, 100] or [-128, 127].
*b**	Amount of yellow or blue tones in the image. A large positive *b** value corresponds to yellow. A large negative *b** value corresponds to blue. Although there is no single range for *b**, values commonly fall in the range [-100, 100] or [-128, 127].

Device-independent color spaces include the effect of the illumination source, called the reference white point. The source imparts a color hue to the raw image data according to the color temperature of the illuminant. For example, sunlight during sunrise or sunset imparts a yellow hue to an image, whereas sunlight around noontime imparts a blue hue.

Use the `rgb2xyz` and `xyz2rgb` functions to convert between the RGB and XYZ color spaces. Use the `rgb2lab` and `lab2rgb` functions to convert between the RGB and L*a*b* color spaces.

The toolbox supports several related color space specifications that are better suited to some purposes than XYZ. For more information see "Device-Independent Color Spaces" on page 15-13.

YCbCr

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 and Cr represent the difference between a reference value and the blue or red component, respectively. (YUV, another color space widely used for digital video, is very similar to YCbCr but not identical.)

Attribute	Description
Y	Luminance or brightness of the image. Colors increase in brightness as *Y* increases.
Cb	Chrominance value that indicates the difference between the blue component and a reference value.
Cr	Chrominance value that indicates the difference between the red component and a reference value.

The range of numeric values depends on the data type of the image. YCbCr does not use the full range of the image data type so that the video stream can include additional (non-image) information.

- For `single` or `double` arrays, *Y* is in the range [16/255, 235/255] and *Cb* and *Cr* are in the range [16/255, 240/255].
- For `uint8` arrays, *Y* is in the range [16, 235] and *Cb* and *Cr* are in the range [16, 240].

- For `uint16`, *Y* is in the range [4112, 60395] and *Cb* and *Cr* are in the range [4112, 61680].

Use the `rgb2ycbcr` and `ycbcr2rgb` functions to convert between the RGB and YCbCr color spaces.

YIQ

The National Television Systems Committee (NTSC) defines a color space known as YIQ. This color space is used in televisions in the United States. This color space separates grayscale information from color data, so the same signal can be used for both color and black and white television sets.

Attribute	Description
Y	Luma, or brightness of the image. Values are in the range [0, 1], where 0 specifies black and 1 specifies white. Colors increase in brightness as *Y* increases.
I	*In-phase*, which is approximately the amount of blue or orange tones in the image. *I* in the range [-0.5959, 0.5959], where negative numbers indicate blue tones and positive numbers indicate orange tones. As the magnitude of *I* increases, the saturation of the color increases.
Q	*Quadrature*, which is approximately the amount of green or purple tones in the image. *Q* in the range [-0.5229, 0.5229], where negative numbers indicate green tones and positive numbers indicate purple tones. As the magnitude of *Q* increases, the saturation of the color increases.

Use the `rgb2ntsc` and `ntsc2rgb` functions to convert between the RGB and YIQ color spaces.

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.

See Also

Related Examples

Convert Between RGB and HSV Color Spaces

This example shows how to adjust the saturation of a color image by converting the image to the HSV color space. The example then displays the separate HSV color planes (hue, saturation, and value) of a synthetic image.

Convert RGB Image to HSV Image

Read an RGB image into the workspace. Display the image.

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


Convert the image to the HSV color space.

```
HSV = rgb2hsv(RGB);
```

Process the HSV image. This example increases the saturation of the image by multiplying the S channel by a scale factor.

```
[h,s,v] = imsplit(HSV);
saturationFactor = 2;
s_sat = s*saturationFactor;
HSV_sat = cat(3,h,s_sat,v);
```

Convert the processed HSV image back to the RGB color space. Display the new RGB image. Colors in the processed image are more vibrant.

```
RGB_sat = hsv2rgb(HSV_sat);
imshow(RGB_sat)
```


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,"BackgroundColor",'w');
```
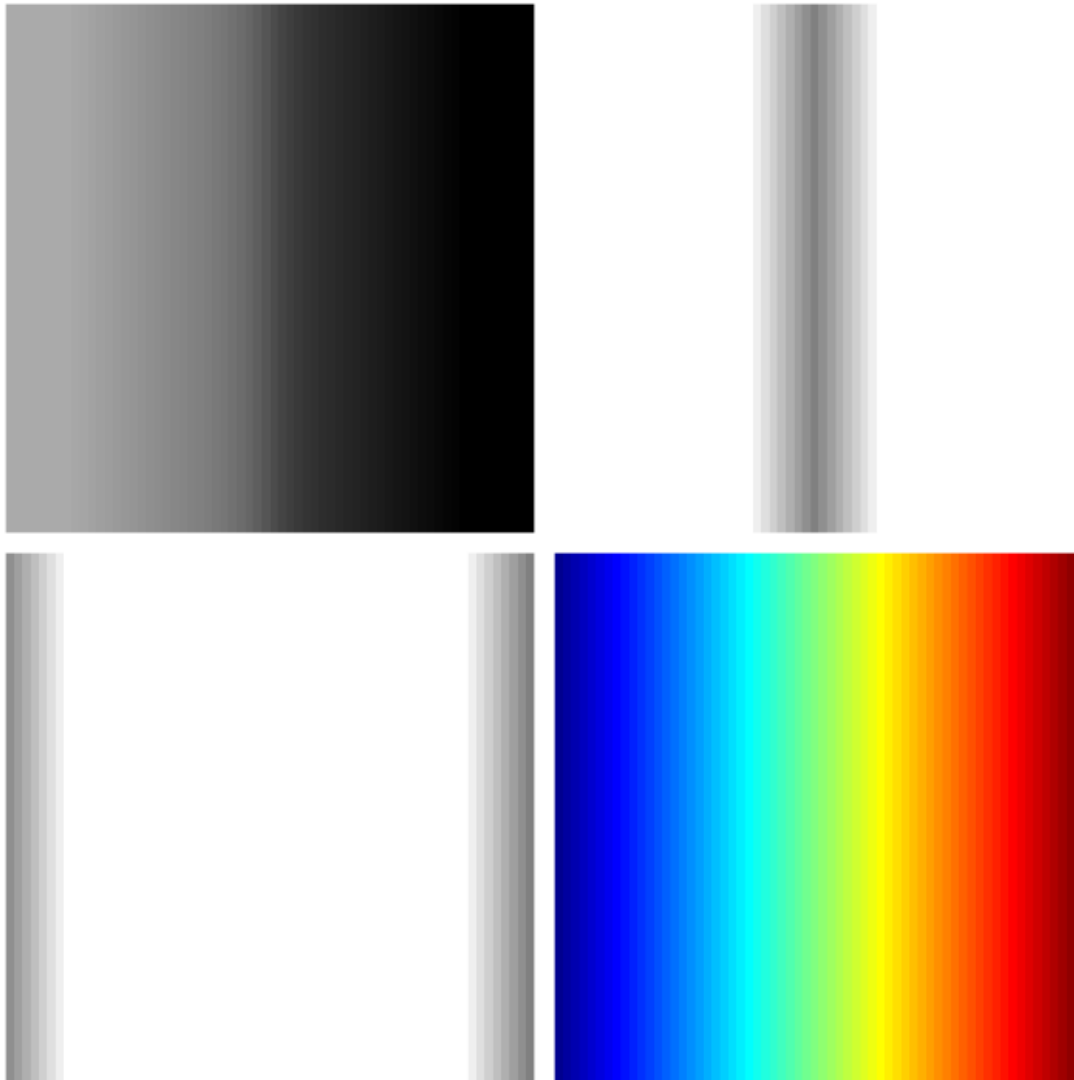
As the hue plane image in the preceding figure illustrates, hue values make a linear transition from high to low. If you compare the hue plane image against the original image, you can see that shades of deep blue have the highest values, and shades of deep red have the lowest values. (As stated previously, 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.

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

This example shows how to estimate illumination and perform white balance of a scene using three different illumination algorithms. The example compares the estimated illuminant to a ground truth illuminant calculated using an X-Rite® ColorChecker® chart.

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.

Automatic white balancing is done in two steps:

- Step 1: Estimate the scene illuminant.
- Step 2: Correct the color balance of the image.

Several different algorithms exist to estimate scene illuminant. The performance of each algorithm depends on the scene, lighting, and imaging conditions. This example judges the quality of three algorithms for illuminant estimation for one specific image by comparing them to the ground truth scene illuminant:

- White Patch Retinex [1]
- Gray World [2]
- Cheng's Principal Component Analysis (PCA) method [3]

After the ambient light is known, then correcting the colors in the image (Step 2) is an easy and fixed process.

Read and Preprocess Raw Camera Data

AWB algorithms are usually applied on the raw image data after a minimal amount of preprocessing, before the image is compressed and saved to the memory card.

Read a 16-bit raw image into the workspace. `foosballraw.tiff` is an image file that contains 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 other preprocessing operations such as demosaicing, denoising, chromatic aberration compensation, tone adjustments, and gamma correction.

```
A = imread('foosballraw.tiff');
```

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, interpolate using the `demosaic` function. The Bayer pattern used by the camera with which the photo was captured (Canon EOS 30D) is RGGB.

```
A = demosaic(A,'rggb');
```

Gamma-Correct Image for Detection and Display

The image A contains linear RGB values. Linear RGB values are appropriate for estimating scene illuminant and correcting the color balance of an image. However, if you try to display the linear RGB

image, it will appear very dim, because of the nonlinear characteristic of display devices. Therefore, for display purposes, gamma-correct the image to the sRGB color space using the `lin2rgb` function.

```
A_sRGB = lin2rgb(A);
```

Display the demosiaced image before and after gamma correction.

```
montage({A,A_sRGB})
title('Original Image Before and After Gamma Correction')
```


Original Image Before and After Gamma Correction

Measure Ground Truth Illuminant Using ColorChecker Chart

Calculate the ground truth illuminant using the X-Rite ColorChecker chart that is included in the scene. This chart consists of 24 neutral and color patches with known spectral reflectances.

Detect the chart in the gamma-corrected image by using the `colorChecker` function. The linear RGB image is too dark for `colorChecker` to detect the chart automatically.

```
chart_sRGB = colorChecker(A_sRGB);
```

Confirm that the chart is detected correctly.

```
displayChart(chart_sRGB)
```

Get the coordinates of the registration points at the four corners of the chart.

```
registrationPoints = chart_sRGB.RegistrationPoints;
```

Create a new `colorChecker` object from the linear RGB data. Specify the location of the chart using the coordinates of the registration points.

```
chart = colorChecker(A,"RegistrationPoints",registrationPoints);
```

Measure the RGB values of all color patches using the `measureColor` function.

```
colors = measureColor(chart)
```

colors=*24×9 table*

ROI	Color	Measured_R	Measured_G	Measured_B	Reference_L	Reference
1	{'DarkSkin' }	1773	2284	1106	37.54	14.37
2	{'LightSkin' }	5691	7627	4245	64.66	19.27
3	{'BlueSky' }	1919	5214	4677	49.32	-3.82
4	{'Foliage' }	1576	3376	1216	43.46	-12.74
5	{'BlueFlower' }	2997	6253	5997	54.94	9.61
6	{'BluishGreen' }	3676	12020	7388	70.48	-32.26
7	{'Orange' }	6250	5276	1213	62.73	35.83
8	{'PurplishBlue'}	1226	3779	5302	39.43	10.75

15-27

9	{'ModerateRed' }	4581	3277	2074	50.57	48.64
10	{'Purple' }	1233	1767	1889	30.1	22.54
11	{'YellowGreen' }	4939	10555	2657	71.77	-24.13
12	{'OrangeYellow'}	7410	8522	1608	71.51	18.24
13	{'Blue' }	567	2300	3888	28.37	15.42
14	{'Green' }	2087	6486	2278	54.38	-39.72
15	{'Red' }	3702	1986	968	42.43	51.05
16	{'Yellow' }	8906	12782	2422	81.8	2.67

⋮

To measure the ground truth illuminant, only use the 6 neutral patches on the bottom row of the chart. Extract the RGB values of the six neutral (achromatic) patches on the bottom row.

```
grayPatchRGB = colors{19:end,{'Measured_R','Measured_G','Measured_B'}};
grayPatchRGB = im2double(grayPatchRGB);
```

The ground truth illuminant is the average color of the neutral patches.

```
illuminant_groundtruth = mean(grayPatchRGB)
```

illuminant_groundtruth = *1×3*

    0.0693   0.1423   0.0943

Create Mask of ColorChecker Chart

When testing the AWB algorithms, prevent the algorithms from unfairly taking advantage of the chart by masking out the chart.

Create a polygon ROI over the chart by using the `drawpolygon` function. Specify the vertices of the polygon as the registration points.

```
chartROI = drawpolygon("Position",registrationPoints);
```

Convert the polygon ROI to a binary mask by using the `createMask` function.

```
mask_chart = createMask(chartROI);
```

Invert the mask. Pixels within the chart are excluded from the mask and pixels of the rest of the scene are included in the mask.

```
mask_scene = ~mask_chart;
```

To confirm the accuracy of the mask, display the mask over the image. Pixels included in the mask have a blue tint.

```
imshow(labeloverlay(A_sRGB,mask_scene));
```

Angular Error

You can consider an illuminant as a vector in 3-D RGB color space. The magnitude of the estimated illuminant does not matter as much as its direction, because the direction of the illuminant is what is used to white balance an image.

To evaluate the quality of an estimated illuminant, compute the angular error between the estimated illuminant and the ground truth. Angular error is the angle (in degrees) formed by the two vectors. The smaller the angular error, the better the estimation is.

To better understand the concept of angular error, consider the following visualization of an arbitrary illuminant and the ground truth measured using the ColorChecker chart. The `plotColorAngle` helper function plots a unit vector of an illuminant in 3-D RGB color space, and is defined at the end of the example.

```
sample_illuminant = [0.066 0.1262 0.0691];

p = plot3([0 1],[0 1],[0,1],'LineStyle',':','Color','k');
ax = p.Parent;
hold on
plotColorAngle(illuminant_groundtruth,ax)
plotColorAngle(sample_illuminant,ax)
title('Illuminants in RGB space')
view(28,36)
legend('Achromatic Line','Ground Truth Illuminant','Sample Illuminant')
```

```
grid on
axis equal
```
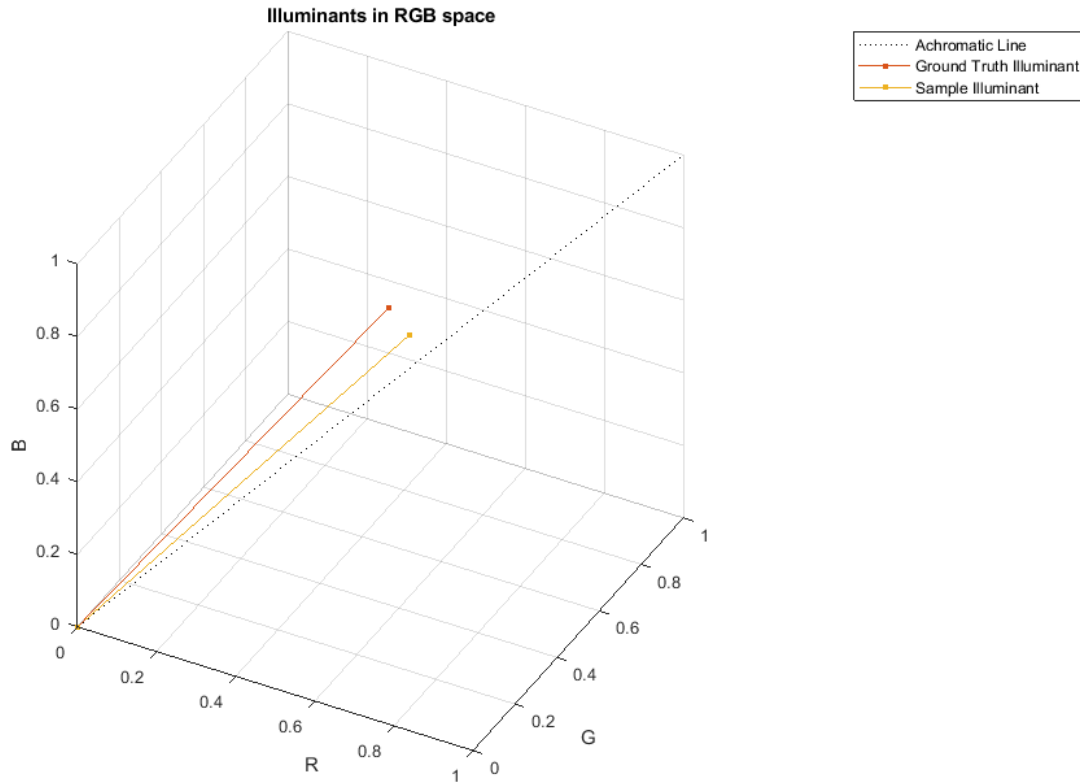

White Patch Retinex

The White Patch Retinex algorithm for illuminant estimation assumes that the scene contains a bright achromatic patch. This patch reflects the maximum light possible for each color band, which is the color of the scene illuminant. Use the `illumwhite` function to estimate illumination using the White Patch Retinex algorithm.

Include All Scene Pixels

Estimate the illuminant using all the pixels in the scene. Exclude the ColorChecker chart from the scene by using the `'Mask'` name-value pair argument.

```
percentileToExclude = 0;
illuminant_wp1 = illumwhite(A,percentileToExclude,'Mask',mask_scene);
```

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: 16.5377

White balance the image using the `chromadapt` function. Specify the estimated illuminant and indicate that color values are in the linear RGB color space.

```
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')
```


White-Balanced Image using White Patch Retinex with percentile=0

Exclude Brightest Pixels

The White Patch Retinex algorithm does not perform well when pixels are overexposed. To improve the performance of the algorithm, exclude the top 1% of the brightest pixels.

```
percentileToExclude = 1;
illuminant_wp2 = illumwhite(A,percentileToExclude,'Mask',mask_scene);
```

Calculate the angular error for the estimated illuminant. The error is less than when estimating the illuminant using all pixels.

```
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: 5.032
```

White balance the image in the linear RGB color space using the estimated illuminant.

```
B_wp2 = chromadapt(A,illuminant_wp2,'ColorSpace','linear-rgb');
```

Display the gamma-corrected white-balanced image with the new illuminant.

```
B_wp2_sRGB = lin2rgb(B_wp2);
imshow(B_wp2_sRGB)
title('White-Balanced Image using White Patch Retinex with percentile=1')
```


White-Balanced Image using White Patch Retinex with percentile=1

Gray World

The Gray World algorithm for illuminant estimation assumes that the average color of the world is gray, or achromatic. Therefore, it calculates the scene illuminant as the *average* RGB value in the image. Use the `illumgray` function to estimate illumination using the Gray World algorithm.

Include All Scene Pixels

First, estimate the scene illuminant 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, specify the percentiles as 0.

```
percentileToExclude = 0;
illuminant_gw1 = illumgray(A,percentileToExclude,'Mask',mask_scene);
```

Calculate the angular error between the estimated illuminant and the ground truth illuminant.

```
err_gw1 = colorangle(illuminant_gw1,illuminant_groundtruth);
disp(['Angular error for Gray World with percentiles=[0 0]: ' num2str(err_gw1)])
```

15-33

Angular error for Gray World with percentiles=[0 0]: 5.0414

White balance the image in the linear RGB color space using 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]')
```

White-Balanced Image using Gray World with percentiles=[0 0]

Exclude Brightest and Darkest Pixels

The Gray World algorithm does not perform well when pixels are underexposed or overexposed. To improve the performance of the algorithm, exclude the top 1% of the darkest and brightest pixels.

```
percentileToExclude = 1;
illuminant_gw2 = illumgray(A,percentileToExclude,'Mask',mask_scene);
```

Calculate the angular error for the estimated illuminant. The error is less than when estimating the illuminant using all pixels.

```
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]: 5.1092

White balance the image in the linear RGB color space using the estimated illuminant.

```
B_gw2 = chromadapt(A,illuminant_gw2,'ColorSpace','linear-rgb');
```

Display the gamma-corrected white-balanced image with the new illuminant.

```
B_gw2_sRGB = lin2rgb(B_gw2);
imshow(B_gw2_sRGB)
title('White-Balanced Image using Gray World with percentiles=[1 1]')
```


White-Balanced Image using Gray World with percentiles=[1 1]

Cheng's Principal Component Analysis (PCA) Method

Cheng's illuminant estimation method 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 percentile. 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 illuminant. Use the `illumpca` function to estimate illumination using Cheng's PCA algorithm.

Include Default Bottom and Top 3.5 Percent of Pixels

First, estimate the illuminant using the default percentage value of Cheng's PCA method, excluding those corresponding to the ColorChecker chart.

```
illuminant_ch2 = illumpca(A,'Mask',mask_scene);
```

Calculate the angular error between the estimated illuminant and the ground truth illuminant.

```
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: 5.0159

White balance the image in the linear RGB color space using the estimated illuminant.

```
B_ch2 = chromadapt(A,illuminant_ch2,'ColorSpace','linear-rgb');
```

Display the gamma-corrected white-balanced image.

```
B_ch2_sRGB = lin2rgb(B_ch2);
imshow(B_ch2_sRGB)
title('White-Balanced Image using Cheng with percentile=3.5')
```


White-Balanced Image using Cheng with percentile=3.5

Include Bottom and Top 5 Percent of Pixels

Now, estimate the scene illuminant using the bottom and top 5% of pixels along the direction of the mean color. The second argument of the `illumpca` function specifies the percentiles of bottom and top values (ordered by brightness) to exclude.

```
illuminant_ch1 = illumpca(A,5,'Mask',mask_scene);
```

Calculate the angular error between the estimated illuminant and the ground truth illuminant. The error is less than when estimating the illuminant using the default percentage.

```
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: 4.7451

White balance the image in the linear RGB color space using the estimated illuminant.

```
B_ch1 = chromadapt(A,illuminant_ch1,'ColorSpace','linear-rgb');
```

Display the gamma-corrected white-balanced image.

```
B_ch1_sRGB = lin2rgb(B_ch1);
imshow(B_ch1_sRGB)
title('White-Balanced Image using Cheng with percentage=5')
```


White-Balanced Image using Cheng with percentage=5

Find Optimal Parameters

To find the best parameter to use for each algorithm, you 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_scene);
    err(k,1) = colorangle(illuminant_wp,illuminant_groundtruth);
    % Gray World
    illuminant_gw = illumgray(A,param_range(k),'Mask',mask_scene);
    err(k,2) = colorangle(illuminant_gw,illuminant_groundtruth);
    % Cheng
    if (param_range(k) ~= 0)
        illuminant_ch = illumpca(A,param_range(k),'Mask',mask_scene);
        err(k,3) = colorangle(illuminant_ch,illuminant_groundtruth);
    else
        % Cheng's algorithm is undefined for percentage=0.
        err(k,3) = NaN;
    end
end
```

Display a heatmap of the angular error using the `heatmap` function. Dark blue colors indicate a low angular error while yellow colors indicate a high angular error. The optimal parameter has the smallest angular error.

```
heatmap(err,'Title','Angular Error','Colormap',parula(length(param_range)), ...
    'XData',["White Patch" "Gray World" "Cheng's PCA"], ...
    'YLabel','Parameter Value','YData',string(param_range));
```
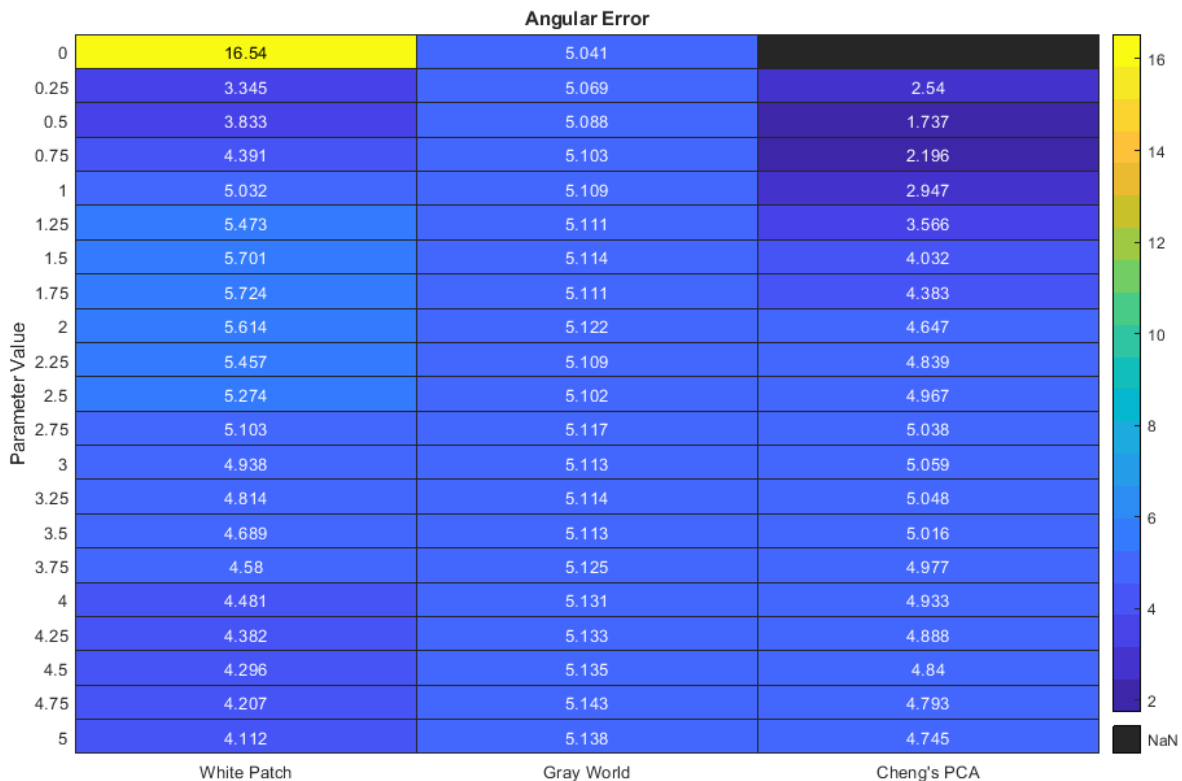

Find the best parameter for each algorithm.

```
[~,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 3.35 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 5.04 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 1.74 degrees

Calculate the estimated illuminant for each algorithm using the best parameter.

```
best_illum_wp = illumwhite(A,best_param_wp,'Mask',mask_scene);
best_illum_gw = illumgray(A,best_param_gw,'Mask',mask_scene);
best_illum_ch = illumpca(A,best_param_ch,'Mask',mask_scene);
```

Display the angular error of each best illuminant in the RGB color space.

```
p = plot3([0 1],[0 1],[0,1],'LineStyle',':','Color','k');
ax = p.Parent;
hold on
plotColorAngle(illuminant_groundtruth,ax)
plotColorAngle(best_illum_wp,ax)
plotColorAngle(best_illum_gw,ax)
plotColorAngle(best_illum_ch,ax)
title('Best Illuminants in RGB space')
view(28,36)
legend('Achromatic Line','Ground Truth','White Patch','Gray World','Cheng')
grid on
axis equal
```

15-39

Best Illuminants in RGB space
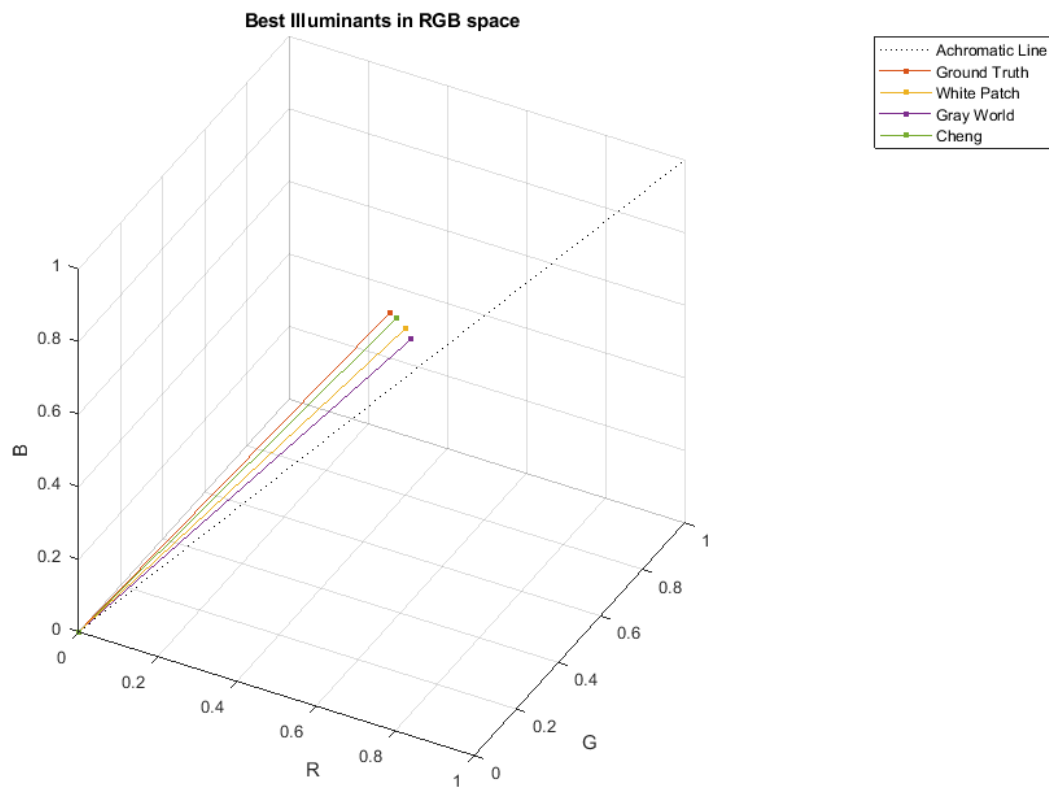
Calculate the optimal white-balanced images for each algorithm using the best illuminant.

```
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);
```

Display the optimal white-balanced images for each algorithm in a montage.

```
figure
montage({B_wp_best_sRGB,B_gw_best_sRGB,B_ch_best_sRGB},'Size',[1 3])
title('Montage of Best White-Balanced Images: White Point, Gray World, Cheng')
```

Montage of Best White-Balanced Images: White Point, Gray World, Cheng

Conclusion

This quick shootout between two classic illuminant estimation algorithms 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, the ground truth illuminant is estimated as the mean color of the neutral patches. It is common to use the median instead of the mean, which 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 algorithms.

Third, a full comparison of illuminant estimation algorithms should use a variety of images taken under different conditions. One algorithm might work better than the others for a particular image, but might perform poorly over the entire data set.

Supporting Function

The `plotColorAngle` function plots a unit vector of an illuminant in 3-D RGB color space. The input argument `illum` specifies the illuminant as an RGB color and the input argument `ax` specifies the axes on which to plot the unit vector.

```
function plotColorAngle(illum,ax)
    R = illum(1);
    G = illum(2);
    B = illum(3);
    magRGB = norm(illum);
    plot3([0 R/magRGB],[0 G/magRGB],[0 B/magRGB], ...
        'Marker','.','MarkerSize',10,'Parent',ax)
    xlabel('R')
    ylabel('G')
    zlabel('B')
    xlim([0 1])
    ylim([0 1])
```

```
    zlim([0 1])
end
```

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 | colorChecker | colorangle | illumgray | illumpca | illumwhite | lin2rgb | measureColor | rgb2lin

More About

- "Gamma Correction" on page 11-73

Calculate CIE94 Color Difference of Colors on Test Chart

This example shows how to calculate the color difference of measured and reference colors using the CIE94 standard. By default, the `measureColor` function calculates the color difference between measured and reference colors on a test chart using the CIE76 standard. You can use the `imcolordiff` function to calculate the color difference using the CIE94 or CIEDE2000 standard.

Read an image of an X-Rite® ColorChecker® chart into the workspace.

```
I = imread('colorCheckerTestImage.jpg');
```

Create a `colorChecker` object then display the chart with ROI annotations.

```
chart = colorChecker(I);
displayChart(chart)
```
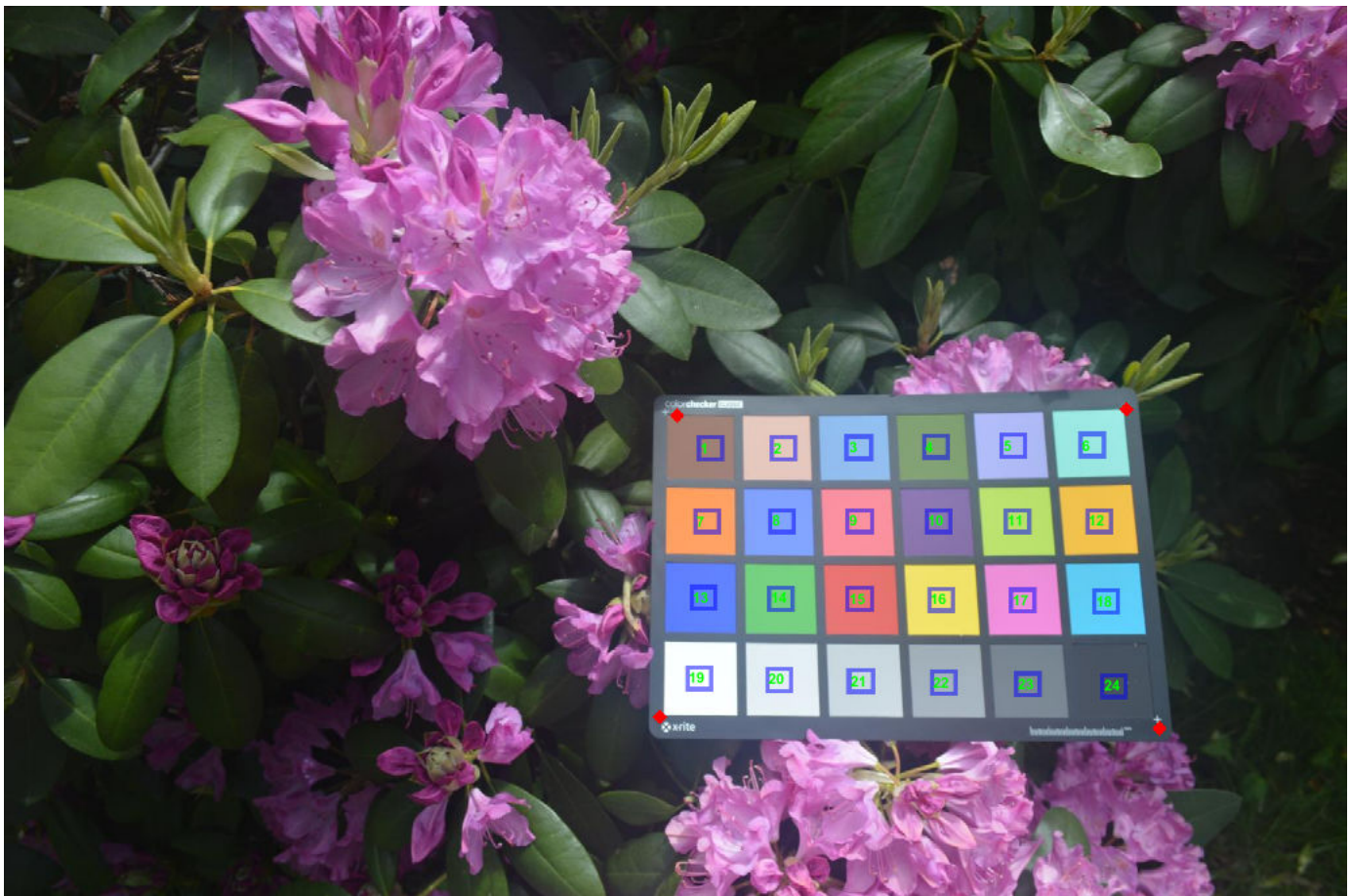

Measure the color in each color patch ROI and return the measurements in a table, `colorTable`. The color difference measurements in the `Delta_E` variable of the table follow the CIE76 standard.

```
colorTable = measureColor(chart);
```

On a color patch diagram, display the measured and reference colors with the corresponding CIE76 color difference superimposed on each patch.

```
displayColorPatch(colorTable)
```
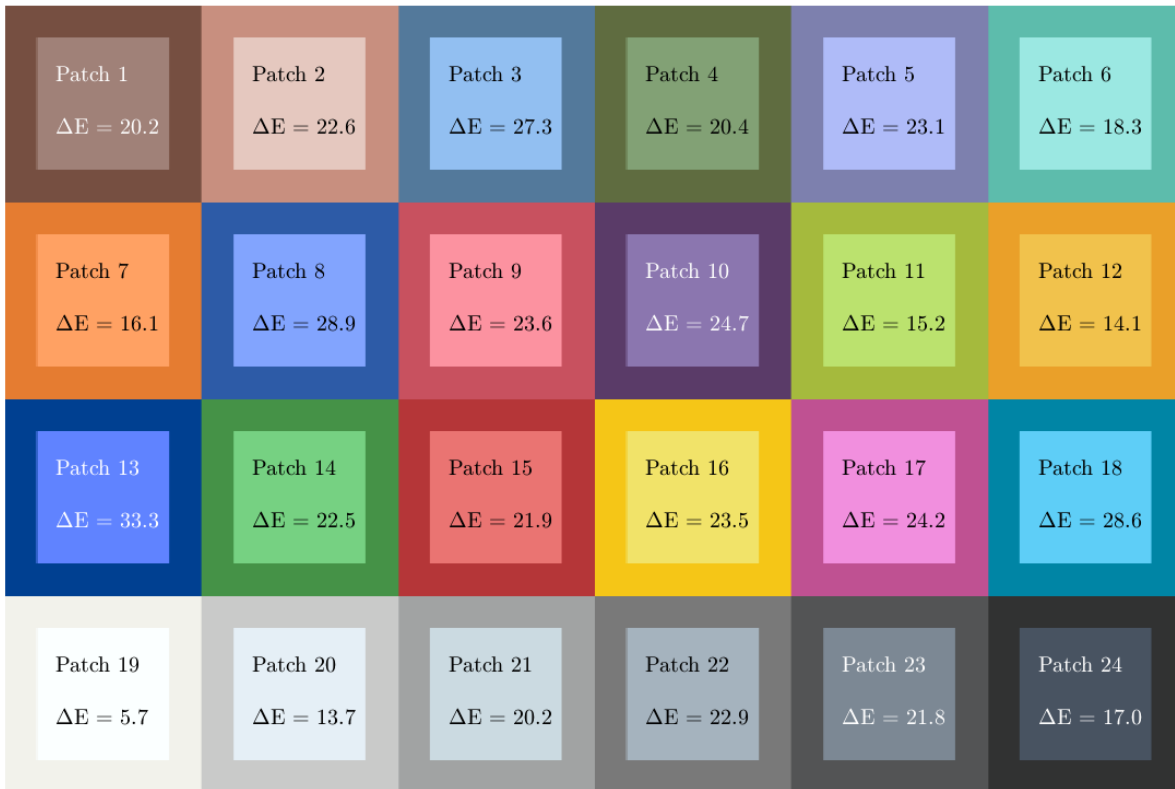
Extract the reference L*a*b* and measured RGB color values into a table.

```
referenceLab = colorTable{:,{'Reference_L','Reference_a','Reference_b'}};
measuredRGB = colorTable{:,{'Measured_R','Measured_G','Measured_B'}};
```

Convert the measured RGB colors to the L*a*b* color space, specifying a D50 white point.

```
measuredLab = rgb2lab(measuredRGB,"WhitePoint","d50");
```

Calculate the color difference using the `imcolordiff` function, specifying that the color measurements are in the L*a*b* color space. By default, this function calculates color differences using the CIE94 standard.

```
dE = imcolordiff(measuredLab,referenceLab,"isInputLab",true);
```

Create a new color table using the new color difference measurements.

```
colorTable94 = colorTable;
colorTable94{:,"Delta_E"} = dE;
```

On a color patch diagram, display the measured and reference colors with the corresponding CIE94 color difference superimposed on each patch.

```
displayColorPatch(colorTable94)
```

Patch 1 $\Delta E = 19.6$	Patch 2 $\Delta E = 20.1$	Patch 3 $\Delta E = 26.4$	Patch 4 $\Delta E = 20.0$	Patch 5 $\Delta E = 22.1$	Patch 6 $\Delta E = 17.1$
Patch 7 $\Delta E = 12.8$	Patch 8 $\Delta E = 28.4$	Patch 9 $\Delta E = 22.2$	Patch 10 $\Delta E = 23.9$	Patch 11 $\Delta E = 13.8$	Patch 12 $\Delta E = 10.7$
Patch 13 $\Delta E = 29.0$	Patch 14 $\Delta E = 22.4$	Patch 15 $\Delta E = 20.9$	Patch 16 $\Delta E = 10.7$	Patch 17 $\Delta E = 22.3$	Patch 18 $\Delta E = 28.2$
Patch 19 $\Delta E = 5.6$	Patch 20 $\Delta E = 13.4$	Patch 21 $\Delta E = 19.8$	Patch 22 $\Delta E = 22.2$	Patch 23 $\Delta E = 21.2$	Patch 24 $\Delta E = 15.5$
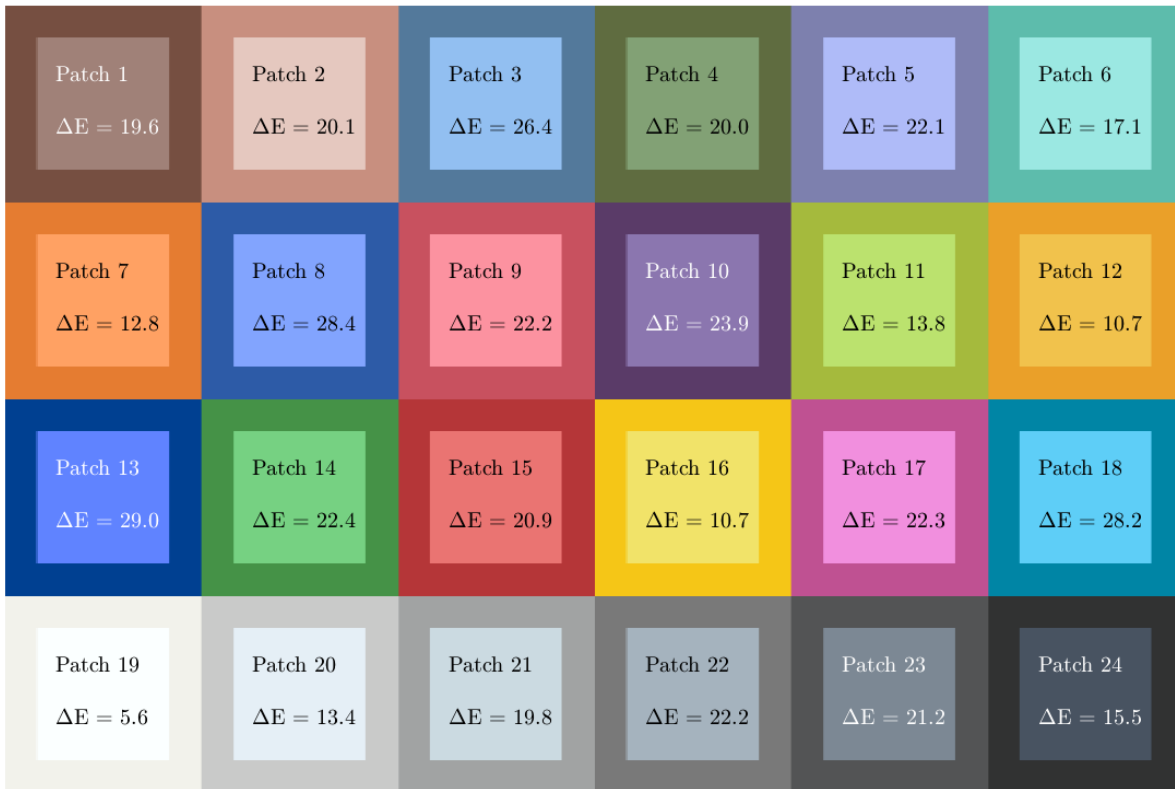
See Also

deltaE | displayChart | displayColorPatch | imcolordiff | plotChromaticity | rgb2lab

Related Examples

- "Correct Colors Using Color Correction Matrix" on page 11-166
- "Comparison of Auto White Balance Algorithms" on page 15-25

Big Image Processing

This topic describes how to work with big image data that does not fit in memory. Big images can have multiple resolution levels.

- "Set Spatial Referencing for Big Images" on page 16-2
- "Process Big Images Efficiently Using Partial Images or Lower Resolutions" on page 16-8
- "Process Big Images Efficiently Using Mask" on page 16-17
- "Explore Big Image Details with Interactive ROIs" on page 16-28
- "Warp Big Image" on page 16-35
- "Create Labeled bigimage from ROIs and Masks" on page 16-40

Set Spatial Referencing for Big Images

This example shows how to set and verify the spatial referencing information of a `bigimage` object.

Spatial Referencing in `bigimage`

`bigimage` works with multiresolution images, in which image data of a scene is stored as a set of images at different resolution levels. `bigimage` assumes that the spatial extents of each level are the same, in other words, that all levels cover the same physical area in the real world. The first step in working with a large multiresolution image is to validate this assumption.

Download Big Image Data

This example uses one image from the Camelyon16 data set. This data set contains 400 whole-slide images (WSIs) of lymph nodes, stored as multiresolution TIF files that are too large to be loaded into memory.

Create a directory to store the Camelyon16 image.

```
imageDir = fullfile(tempdir,'Camelyon16');
if ~exist(imageDir,'dir')
    mkdir(imageDir);
end
```

To download the image, go to the Camelyon17 website and click the first "CAMELYON16 data set" link. Open the "training" then "tumor" directory. Download the "tumor_091.tif" file and move the file to the directory specified by the `imageDir` variable.

Explore Default Spatial Referencing

Create a `bigimage` with the default spatial referencing information. By default, `bigimage` sets the spatial referencing of each level to have the same world extents as the finest layer. The finest layer is the layer that has the highest resolution and the most pixels.

```
bim = bigimage(fullfile(imageDir,'tumor_091.tif'));
```

Display the spatial referencing information at the finest level. The image size (specified by the `ImageSize` property) matches the extents in world coordinates. Notice that the default image coordinate system puts the center of the first pixel at (1,1). Because the pixel extents are 1 unit wide in each dimension, the left edge of the first pixel starts at (0.5,0.5).

```
finestLevel = bim.FinestResolutionLevel;
finestLevelInfo = bim.SpatialReferencing(finestLevel)


finestLevelInfo =

  imref2d with properties:

           XWorldLimits: [0.5000 6.1441e+04]
           YWorldLimits: [0.5000 5.3761e+04]
              ImageSize: [53760 61440]
    PixelExtentInWorldX: 1
    PixelExtentInWorldY: 1
    ImageExtentInWorldX: 61440
    ImageExtentInWorldY: 53760
```

```
        XIntrinsicLimits: [0.5000 6.1441e+04]
        YIntrinsicLimits: [0.5000 5.3761e+04]
```

Display the spatial referencing information at the coarsest level. The world extents are the same as the finest level, but the coarse image size is only 512-by-512 pixels. Effectively, each pixel in this coarse level corresponds to a 105-by-120 block of pixels in the finest resolution.

```
coarsestLevel = bim.CoarsestResolutionLevel;
disp(bim.SpatialReferencing(coarsestLevel))

  imref2d with properties:

           XWorldLimits: [0.5000 6.1441e+04]
           YWorldLimits: [0.5000 5.3761e+04]
              ImageSize: [512 512]
    PixelExtentInWorldX: 120
    PixelExtentInWorldY: 105
    ImageExtentInWorldX: 61440
    ImageExtentInWorldY: 53760
       XIntrinsicLimits: [0.5000 512.5000]
       YIntrinsicLimits: [0.5000 512.5000]
```

Verify Aspect Ratio

Display the image size and aspect ratio at each level. The aspect ratio is not consistent, which indicates that levels do not all span the same world area. Therefore, the default assumption is incorrect for this image.

```
t = table((1:8)',bim.LevelSizes(:,1),bim.LevelSizes(:,2), ...
    bim.LevelSizes(:,1)./bim.LevelSizes(:,2), ...
    'VariableNames',["Level" "Height" "Width" "Aspect Ratio"]);
disp(t)
```

Level	Height	Width	Aspect Ratio
1	53760	61440	0.875
2	27136	30720	0.88333
3	13824	15360	0.9
4	7168	7680	0.93333
5	3584	4096	0.875
6	2048	2048	1
7	1024	1024	1
8	512	512	1

Display Layers to Compare Spatial Extents

Display the `bigimage` by using the `bigimageshow` function. Display the coarsest resolution level on the left side of a figure window.

```
figure
subplot(1,2,1);
hl = bigimageshow(bim,'ResolutionLevel',coarsestLevel);
title('Coarsest Resolution Level (8)')
```
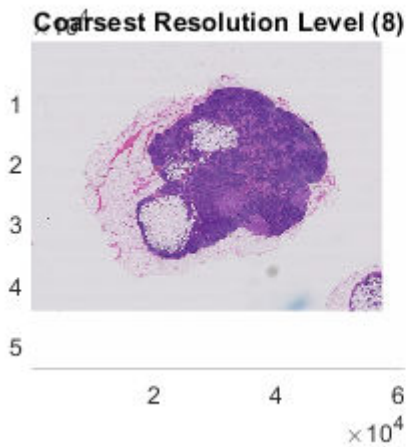
Display image data at the default resolution level on the right side of a figure window. By default, `bigimageshow` selects the level to display based on the screen resolution and the size of the displayed region.

```
subplot(1,2,2);
hr = bigimageshow(bim);
title('Default Resolution Level')
```
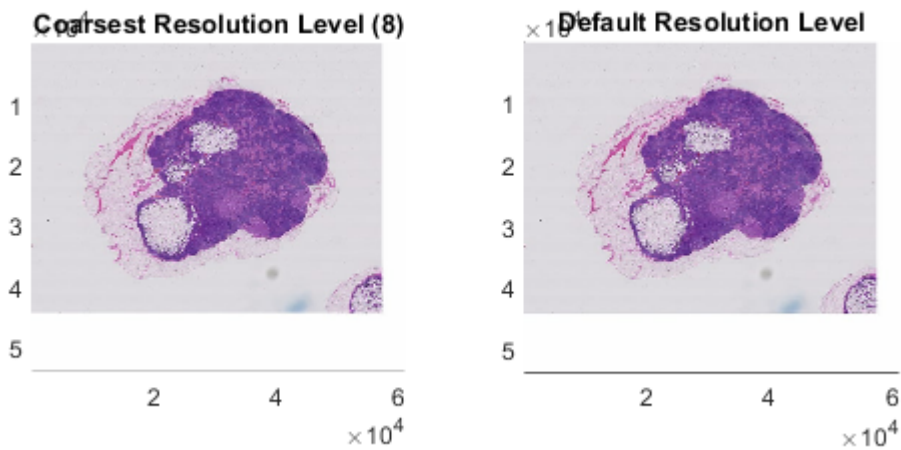

Ensure that both displays show the same extents.

```
linkaxes([hl.Parent,hr.Parent]);
```

Check Default Spatial Referencing

Zoom in on a feature.

```
xlim([45000 50000]);
ylim([12000 17000]);
```
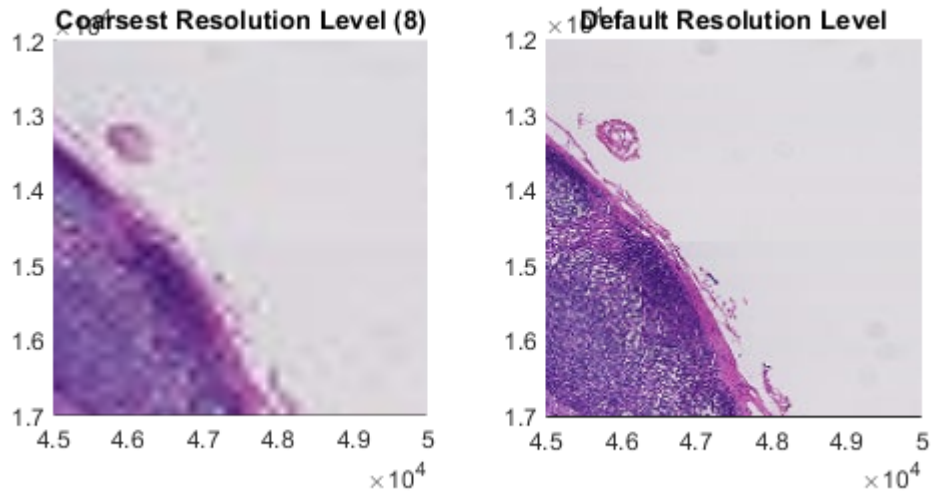
Change the resolution level of the image on the right side of the figure window. At level 6, the features look aligned with the coarsest level.

```
hr.ResolutionLevel = 6;
title('Level 6');
snapnow
```
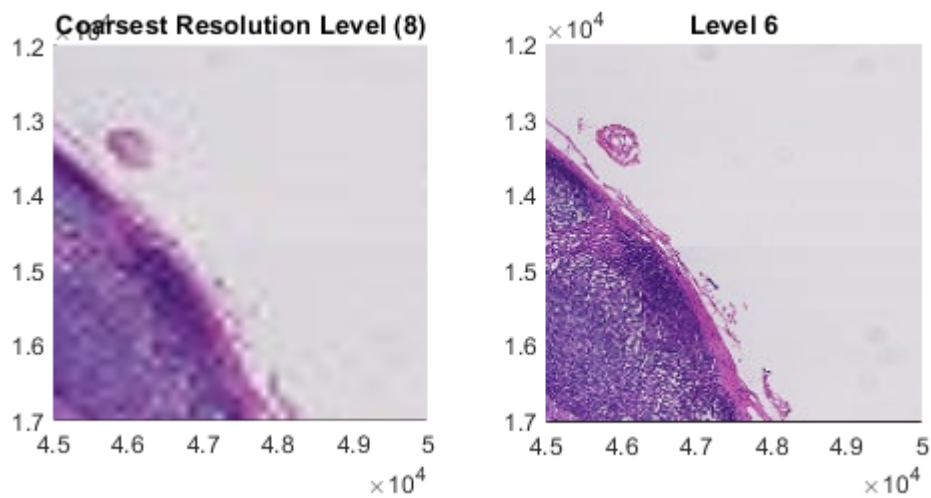
At level 1, the features are not aligned. Therefore, level 1 and level 8 do not span the same world extents.

```
hr.ResolutionLevel = 1;
title('Level 1');
snapnow
```
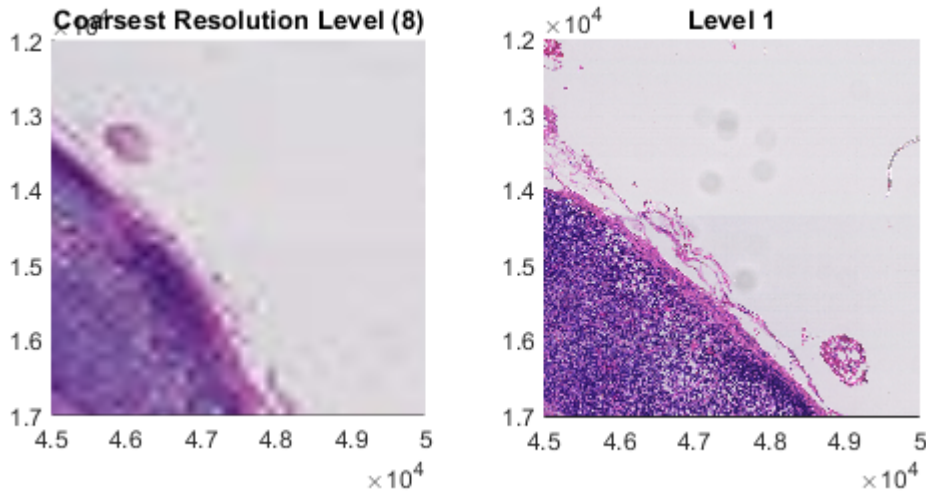
Get Spatial Extents from Big Image Metadata

Usually the original source of the data has spatial referencing information encoded in its metadata. For the Camelyon16 data set, spatial referencing information is stored as XML content in the `ImageDescription` metadata field at the finest resolution level. The XML content has a DICOM_PIXEL_SPACING attribute for each resolution level that specifies the pixel extents.

Get the `ImageDescription` metadata field at the finest resolution level of the `bigimage`.

```
binfo = imfinfo(bim.DataSource);
binfo = binfo(1).ImageDescription;
```

Search the content for the string "DICOM_PIXEL_SPACING". There are nine matches found. The second instance of the attribute corresponds to the pixel spacing of the finest level. The last instance of the attribute corresponds to the pixel spacing of the coarsest level.

```
indx = strfind(binfo,"DICOM_PIXEL_SPACING");
```

Store the pixel spacing at the finest level. To extract the value of the pixel spacing from the XML text, visually inspect the text following the second instance of the "DICOM_PIXEL_SPACING" attribute.

```
disp(binfo(indx(2):indx(2)+100))
pixelSpacing_L1 = 0.000227273;
```

```
DICOM_PIXEL_SPACING" Group="0x0028" Element="0x0030" PMSVR="IDoubleArray">&quot;0.000227273&quot
```

Similarly, store the pixel spacing at the coarsest level. To extract the value of the pixel spacing from the XML text, visually inspect the text following the last instance of the "DICOM_PIXEL_SPACING" attribute.

```
disp(binfo(indx(end):indx(end)+100))
pixelSpacing_L8 = 0.0290909;
```

```
DICOM_PIXEL_SPACING" Group="0x0028" Element="0x0030" PMSVR="IDoubleArray">&quot;0.0290909&quot; &
```

Set Spatial Extents

Calculate the relative pixel width between level 8 and level 1.

```
pixelDims = pixelSpacing_L8/pixelSpacing_L1;
```

The finest level has the reference spatial extent. Calculate the corresponding extents for level 8 with respect to the extents of level 1.

```
worldExtents = bim.SpatialReferencing(8).ImageSize.*pixelDims;
```

Update the spatial referencing of level 8.

```
bim.SpatialReferencing(8).XWorldLimits = [0.5 worldExtents(2)];
bim.SpatialReferencing(8).YWorldLimits = [0.5 worldExtents(1)];
```

Verify Alignment with Custom Spatial Referencing

Redisplay the data to confirm alignment of the key feature. Show level 8 on the left side and level 1 on the right side.

```
hl.CData = bim;
hl.ResolutionLevel = 8;
hr.CData = bim;
hr.ResolutionLevel = 1;
```
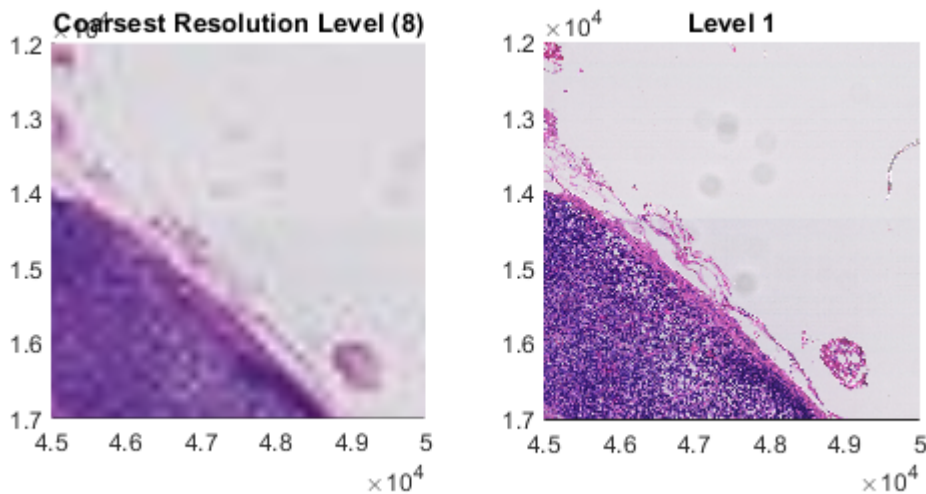

See Also

bigimage | bigimageshow

More About

•   "Define World Coordinate System of Image" on page 2-6

Process Big Images Efficiently Using Partial Images or Lower Resolutions

This example shows how to process a big image quickly using two strategies that enable computations on smaller representative samples of the high-resolution image.

Processing big images can be time consuming. This makes iterative development of algorithms prohibitively expensive. There are two common ways to shorten the feedback cycle: iterate on a lower resolution image or iterate on a partial region of the big image. This example demonstrates both of these approaches for creating a segmentation mask for a big image.

If you have Parallel Computing Toolbox™ installed, then you can further accelerate the processing by using multiple workers.

Create a `bigimage` using a modified version of image "tumor_091.tif" from the CAMELYON16 data set. The original image is a training image of a lymph node containing tumor tissue. The original image has eight resolution levels, and the finest level has resolution 53760-by-61440. The modified image has only three coarse resolution levels. The spatial referencing of the modified image has been adjusted to enforce a consistent aspect ratio and to register features at each level.

```
bim = bigimage('tumor_091R.tif');
```

Display the big image by using the `bigimageshow` function.

```
bigimageshow(bim);
```
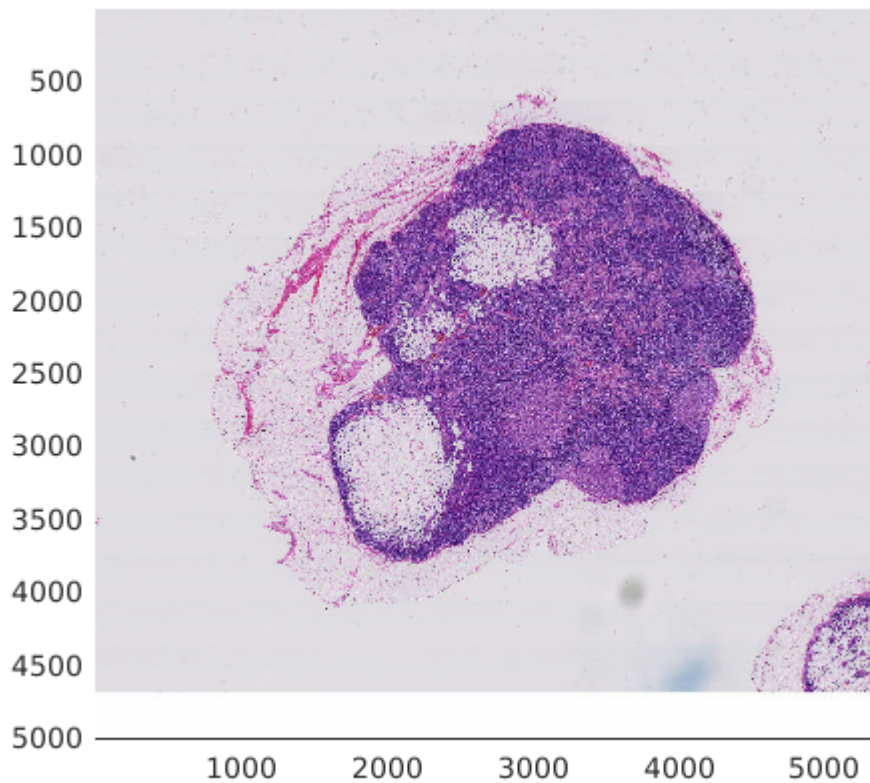
Accelerate Processing Using Lower Resolution Image

Many big images contain multiple resolution levels, including coarse lower resolution versions of the finest high-resolution image. In general, the distribution of individual pixel values should be roughly equal across all the levels. Leveraging this assumption, you can compute global statistics at a coarse level and then use the statistics to process the finer levels.

Extract the image at the coarsest level, then convert the image to grayscale.

```
imLowRes = getFullLevel(bim,bim.CoarsestResolutionLevel);
imLowResGray = rgb2gray(imLowRes);
```

Threshold the image into two classes and display the result.

```
thresh = graythresh(imLowResGray);
imLowResQuant = imbinarize(imLowResGray,thresh);
imshow(imLowResQuant)
```
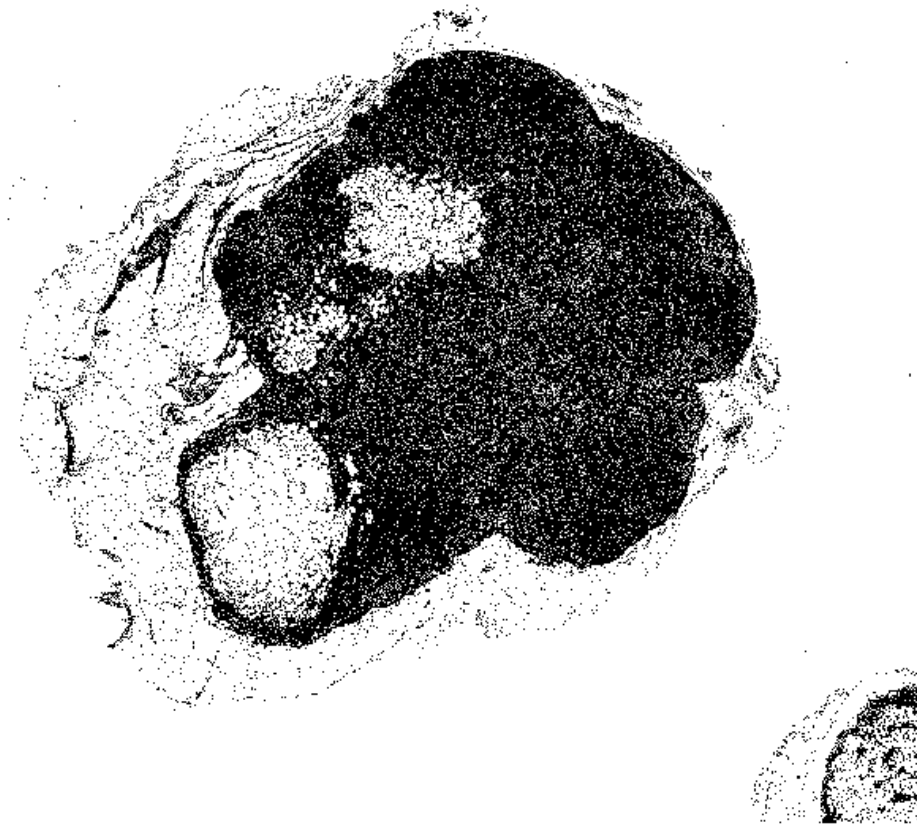
Validate on the largest image. Negate the result to obtain a mask for the stained region.

```
bq = apply(bim,bim.FinestResolutionLevel, ...
    @(im)~imbinarize(rgb2gray(im),thresh));
```

Visualize the result at the finest level.

```
bigimageshow(bq,'CDataMapping','scaled');
```
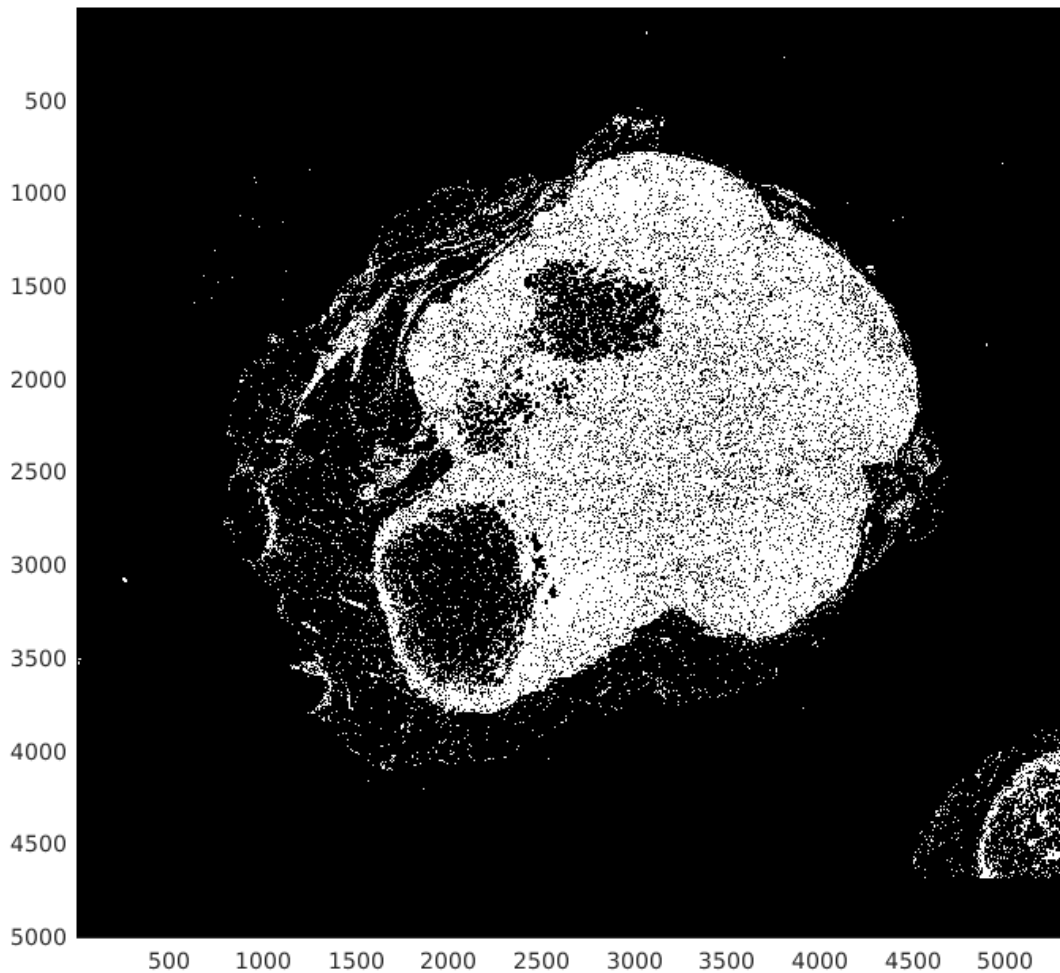
Accelerate Processing Using Using Partial Regions of Big Image

Another approach while working with large images is to extract a smaller region with features of interest. You can compute statistics from the ROI and then use the statistics to process the entire high-resolution image.

```
% Zoom in on a region of interest.
bigimageshow(bim);
xlim([2400,3300])
ylim([900 1700])
```
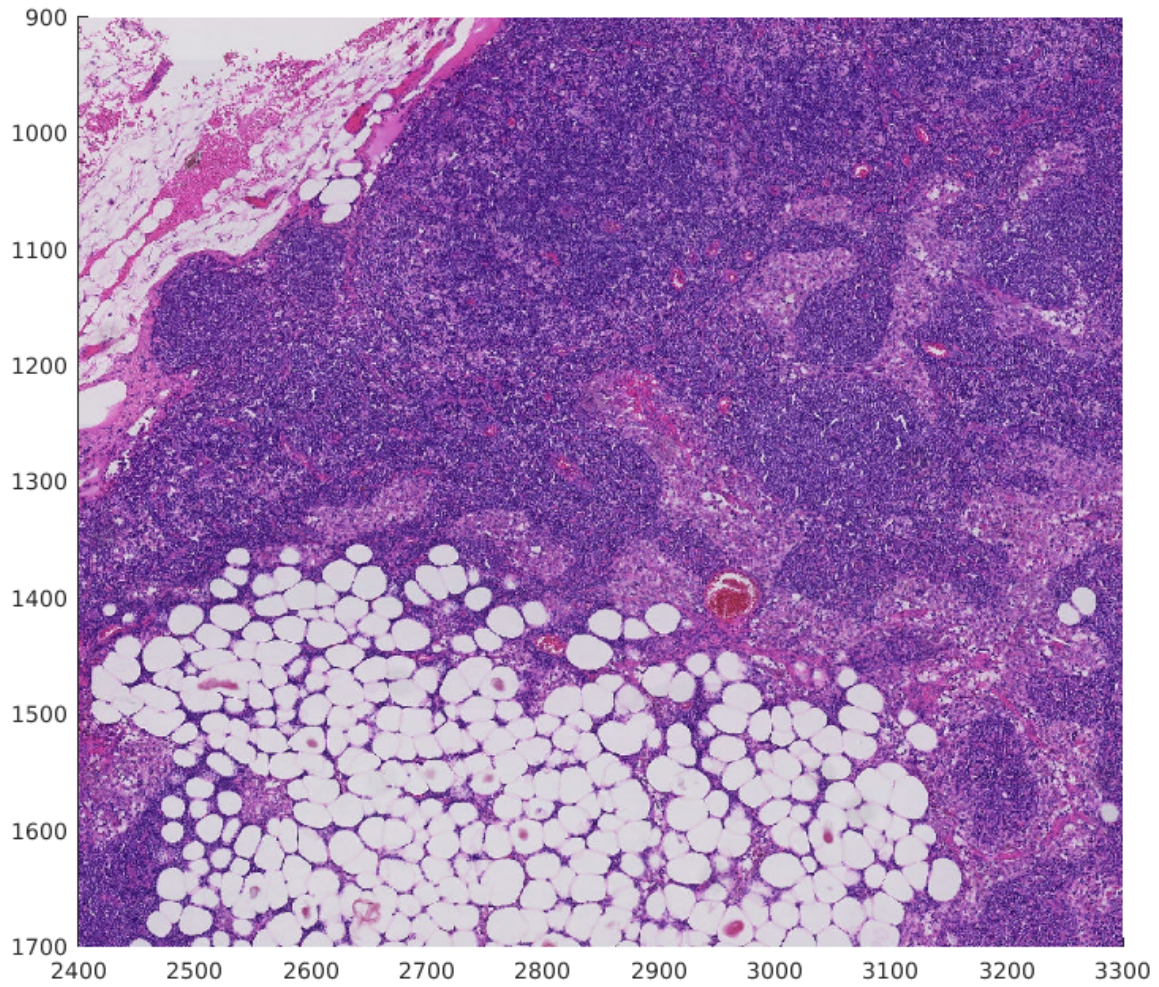
Extract the region being shown from the finest level.

```
xrange = xlim;
yrange = ylim;
imRegion = bim.getRegion(1,[xrange(1),yrange(1)],[xrange(2),yrange(2)]);
imshow(imRegion);
```
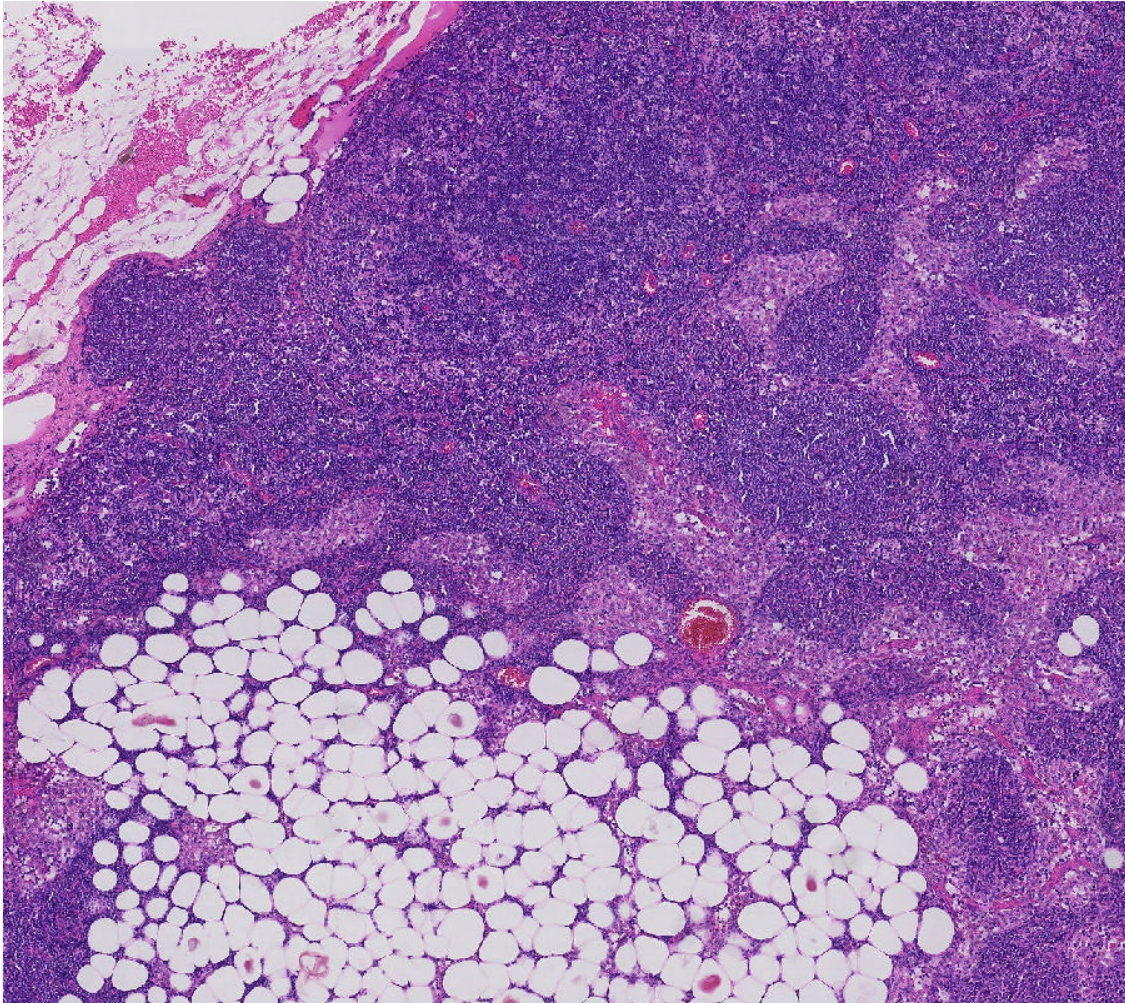
Prototype with this region then display the results.

```
imRegionGray = rgb2gray(imRegion);
thresh = graythresh(imRegionGray);
imLowResQuant = ~imbinarize(imRegionGray,thresh);

imshow(imLowResQuant)
```
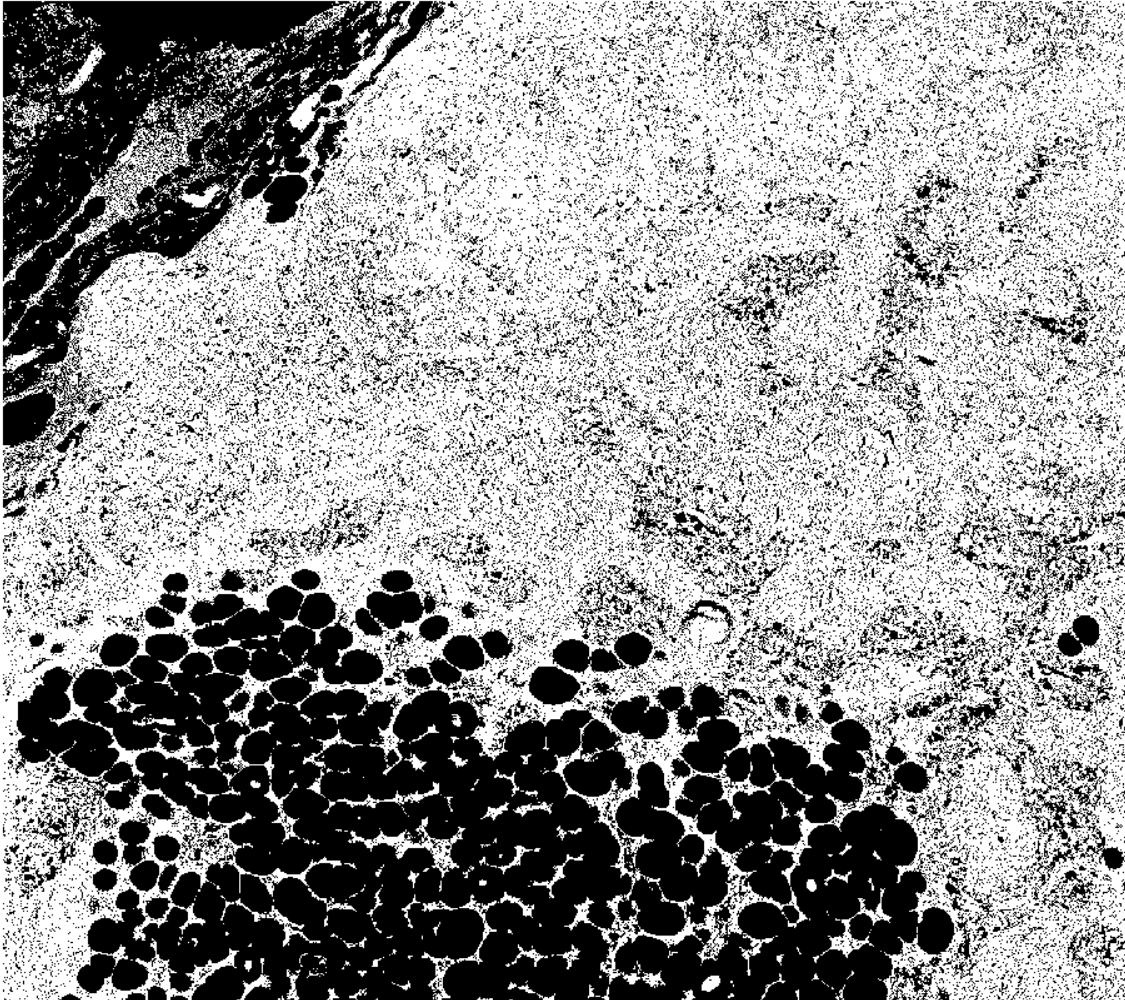
Validate on the full big image and display the results.

```
bq = apply(bim, bim.FinestResolutionLevel,...
    @(im)~imbinarize(rgb2gray(im), thresh));

bigimageshow(bq,'CDataMapping','scaled');
```
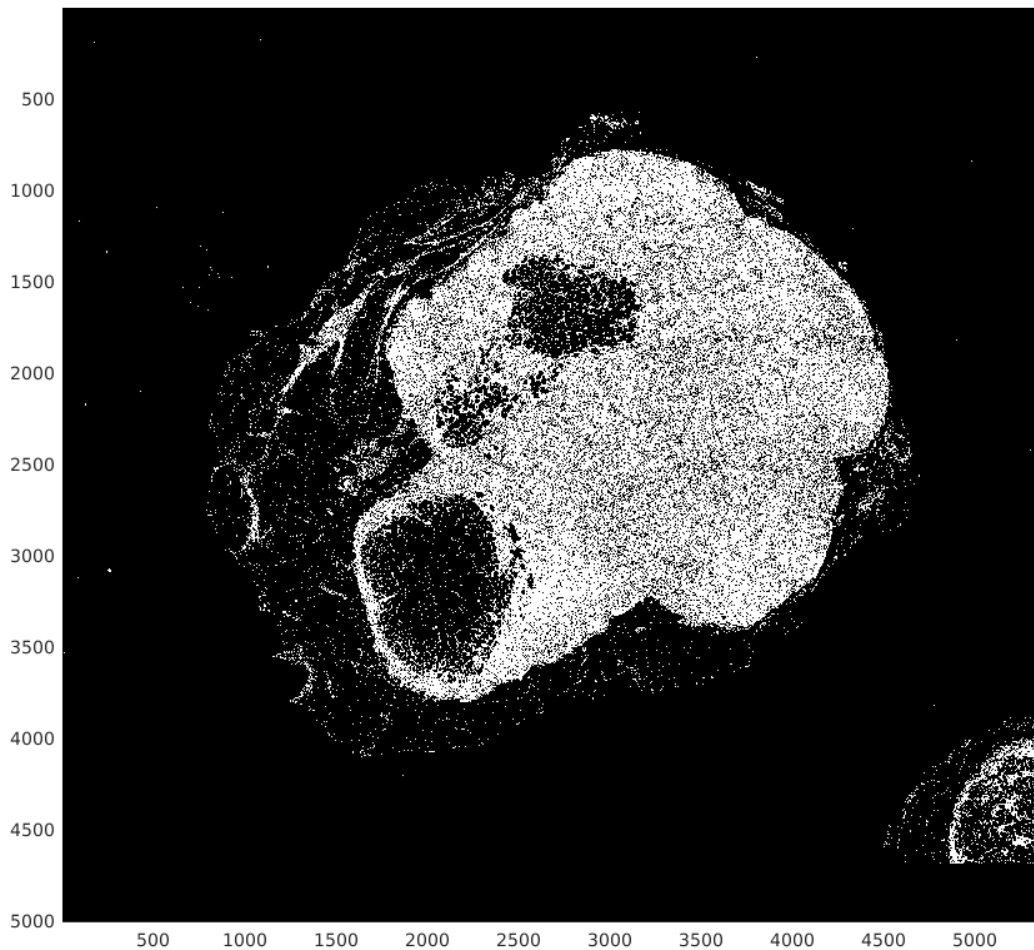
Accelerate Processing Using Parallel Computing Toolbox

If you have the Parallel Computing Toolbox™ installed, then you can distribute the processing across multiple workers to accelerate the processing. To try processing the image in parallel, set the `runInParallel` variable to `true`.

```
runInParallel = false;
if runInParallel
    % Open a pool
    p = gcp;
    % Ensure workers are on the same folder as the file to be able to
    % access it using just the relative path
    sourceDir = fileparts(which('tumor_091R.tif'));
    spmd
        cd(sourceDir)
    end
    % Run in parallel
    bq = apply(bim,bim.FinestResolutionLevel, ...
```

```
        @(im)~imbinarize(rgb2gray(im),thresh),'UseParallel',true);
end
```

See Also
apply | bigimage | bigimageshow

Process Big Images Efficiently Using Mask

This example shows how to process a big image efficiently by using a mask to isolate ROIs.

Some sources of large images have meaningful data in only a small portion of the image. You can improve the total processing time by limiting processing to the regions of interest (ROI) containing meaningful data. Use a mask to define ROIs. A mask is a logical image in which `true` pixels represent the ROI.

In the `bigimage` workflow, the mask represents the same spatial region as the image data but it does not need not have the size as the image. To further improve the efficiency of the workflow, create a mask from a coarse image, especially one that fits in memory. Then, use the coarse mask to process the finer images.

Create a `bigimage` using a modified version of image "tumor_091.tif" from the CAMELYON16 data set. The original image is a training image of a lymph node containing tumor tissue. The original image has eight resolution levels, and the finest level has resolution 53760-by-61440. The modified image has only three coarse resolution levels. The spatial referencing of the modified image has been adjusted to enforce a consistent aspect ratio and to register features at each level.

```
bim = bigimage('tumor_091R.tif');
```

Display the big image by using the `bigimageshow` function.

```
bigimageshow(bim);
```
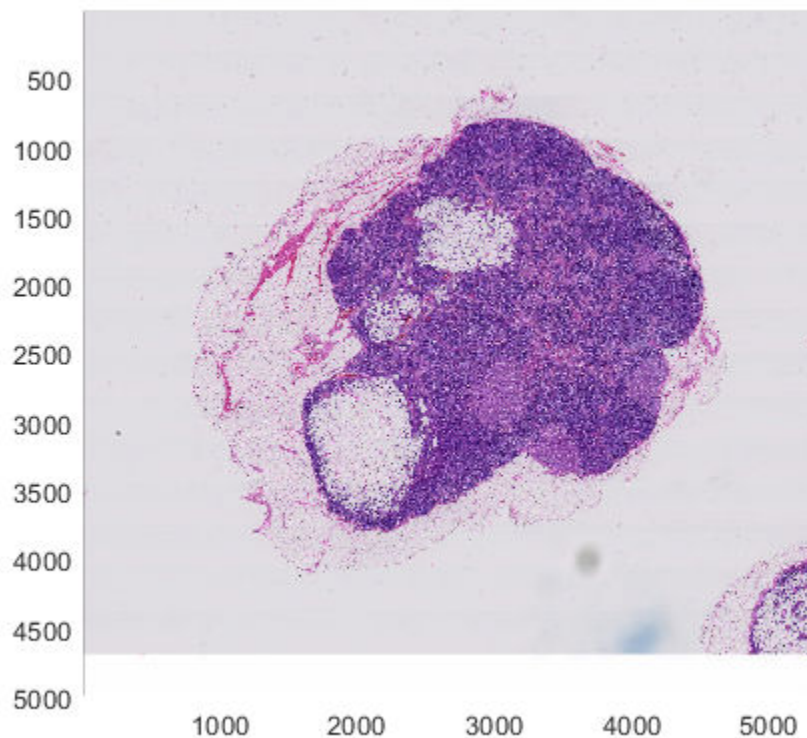
Create Mask

Determine the image size at the coarsest level.

```
coarseLevel = bim.CoarsestResolutionLevel;
coarseLevelSize = bim.LevelSizes(coarseLevel,:)
```

```
coarseLevelSize = 1×2

   625    670
```

Get the image at the coarsest resolution level.

```
imLowRes = bim.getFullLevel(bim.CoarsestResolutionLevel);
```

You can generate a mask from the coarse level using the Image Segmenter app. The app expects a grayscale input image, so get the lightness channel from the coarse image.

```
imLowResL = rgb2lightness(imLowRes);
```

To run Image Segmenter, enter this command in the command window:

```
imageSegmenter(imLowResL)
```

You can export the mask, BW, or code that reproduces the processing pipleline used to create a mask. This section of the example uses code that the app exports. Run this code to create a mask from the coarse input image.

```
%--------------------------------------------------
% Normalize input data to range in [0,1].
Xmin = min(imLowResL(:));
Xmax = max(imLowResL(:));
if isequal(Xmax,Xmin)
    imLowResL = 0*imLowResL;
else
    imLowResL = (imLowResL - Xmin) ./ (Xmax - Xmin);
end

% Threshold image - global threshold
BW = imbinarize(imLowResL);

% Invert mask
BW = imcomplement(BW);

% Open mask with square
width = 3;
se = strel('square', width);
BW = imopen(BW, se);
%--------------------------------------------------

imshow(BW)
```
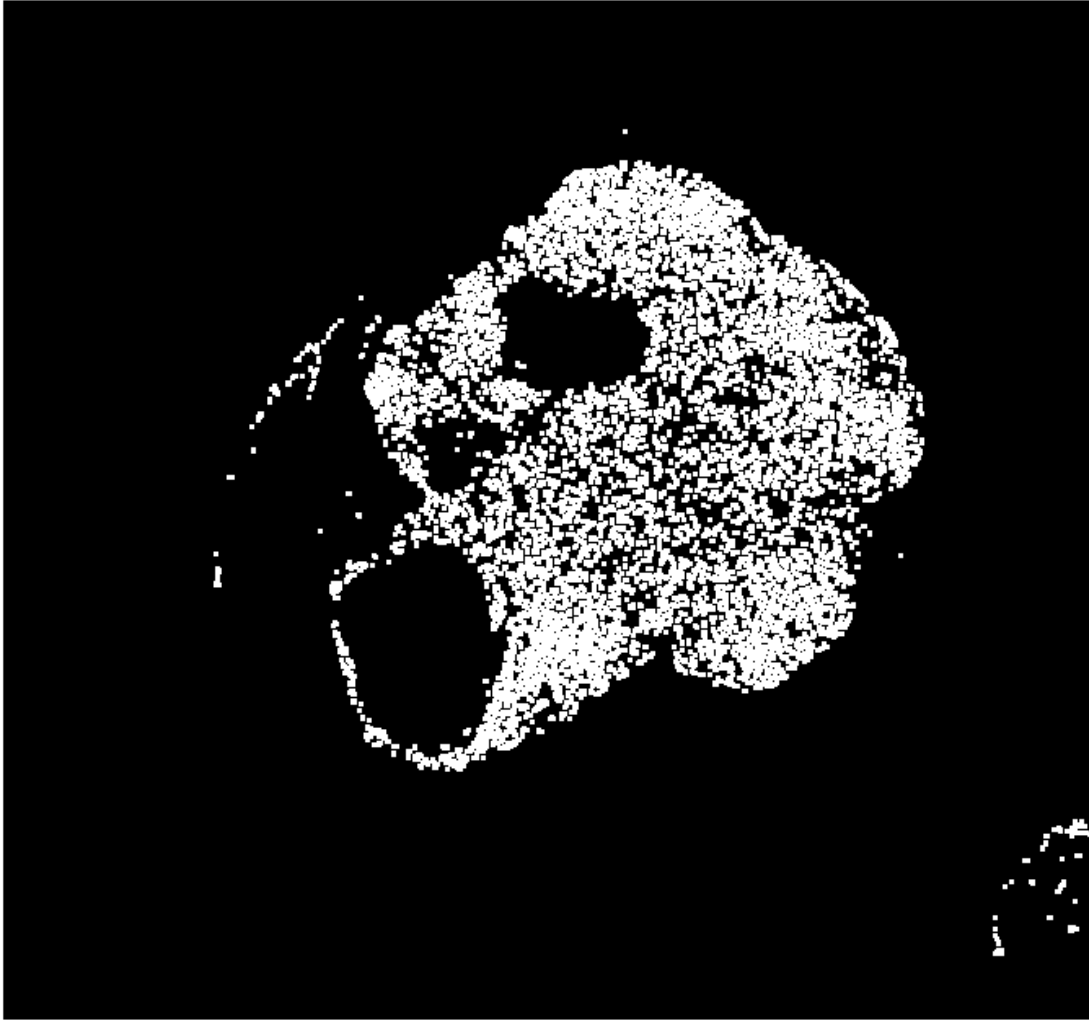
Create a `bigimage` from the mask with the same spatial referencing as the input mask.

```
bmask = bigimage(BW, ...
    'SpatialReferencing',bim.SpatialReferencing(coarseLevel));
```

Display the mask with a green background.

```
h = bigimageshow(bim);
h.Parent.Color = 'g';
h.Parent.Alphamap = [1 .5];
h.AlphaData = bmask;
h.AlphaDataMapping = 'direct';
```

16-19
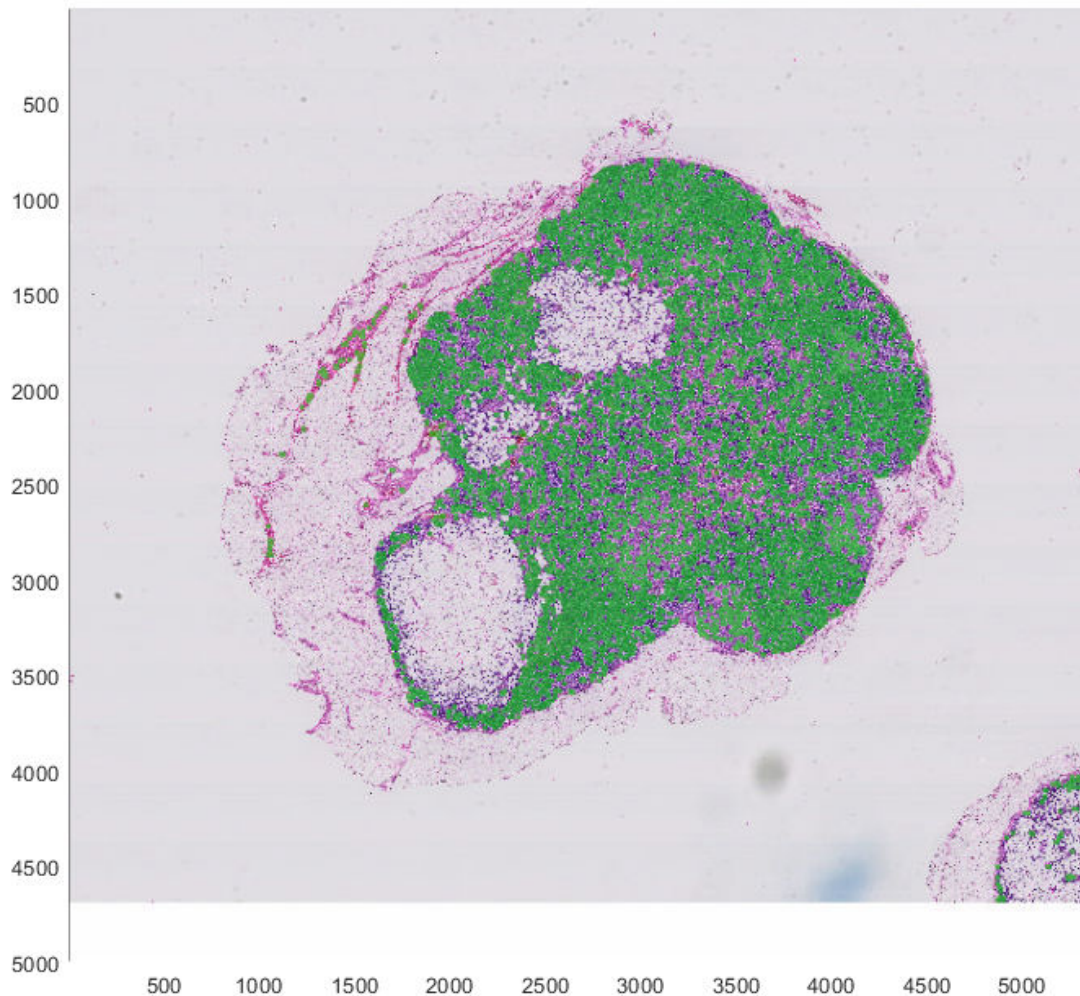
Adjust Inclusion Threshold to Cover Region of Interest

The `apply` function processes big images one block at a time. You can use the `'InclusionThreshold'` property with the mask to specify which blocks the `apply` function uses. The inclusion threshold specifies the percentage of mask pixels that must be `true` for `apply` to process the block.

Highlight the blocks that `apply` will process using the default inclusion threshold, `0.5`. Only center blocks, highlighted in green, will be processed.

```
h = bigimageshow(bim);
showmask(h,bmask,1);
title('Mask with Default Inclusion Threshold')
```

Mask with Default Inclusion Threshold
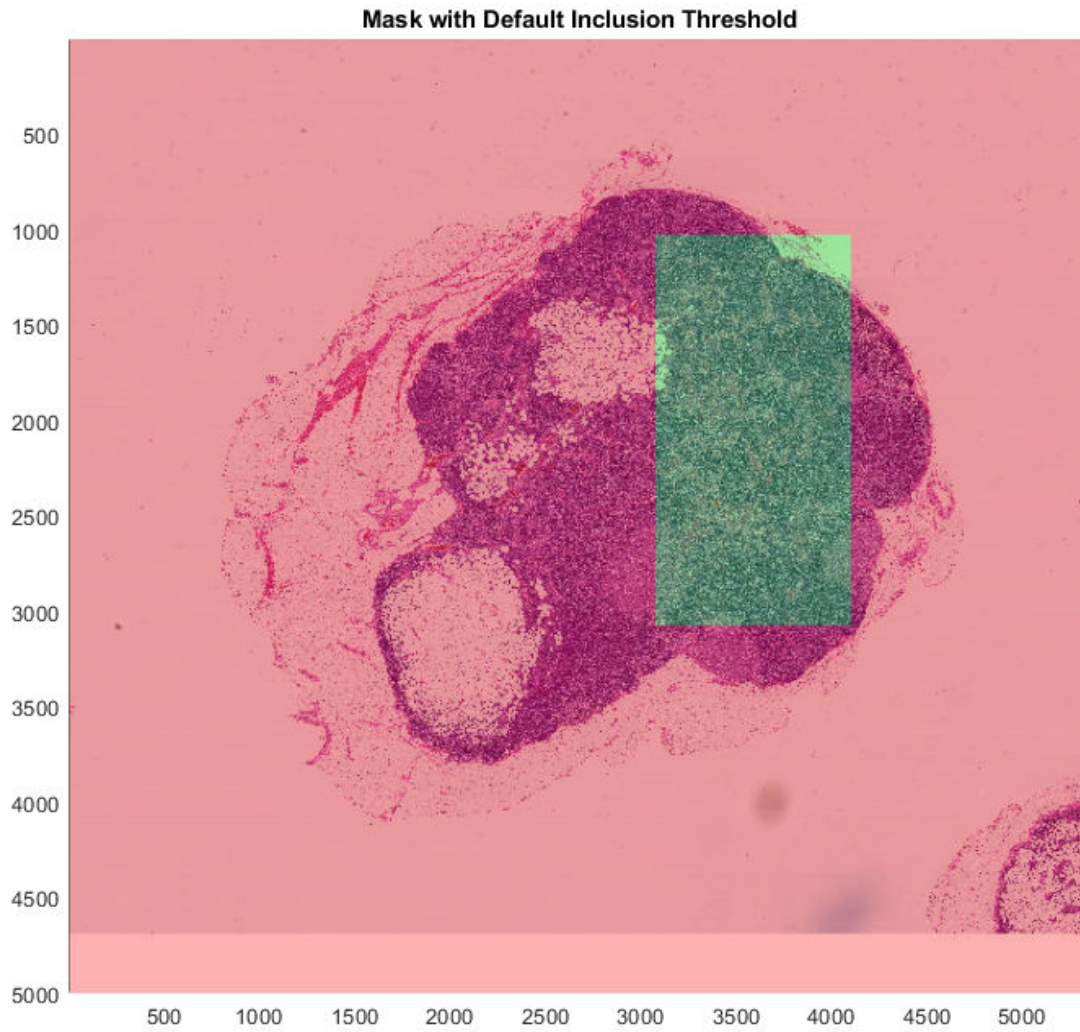
To process more blocks of the image, decrease the inclusion threshold.

```
showmask(h,bmask,1,'InclusionThreshold',0.4);
title('InclusionThreshold == 0.4')
```

16-21

InclusionThreshold == 0.4
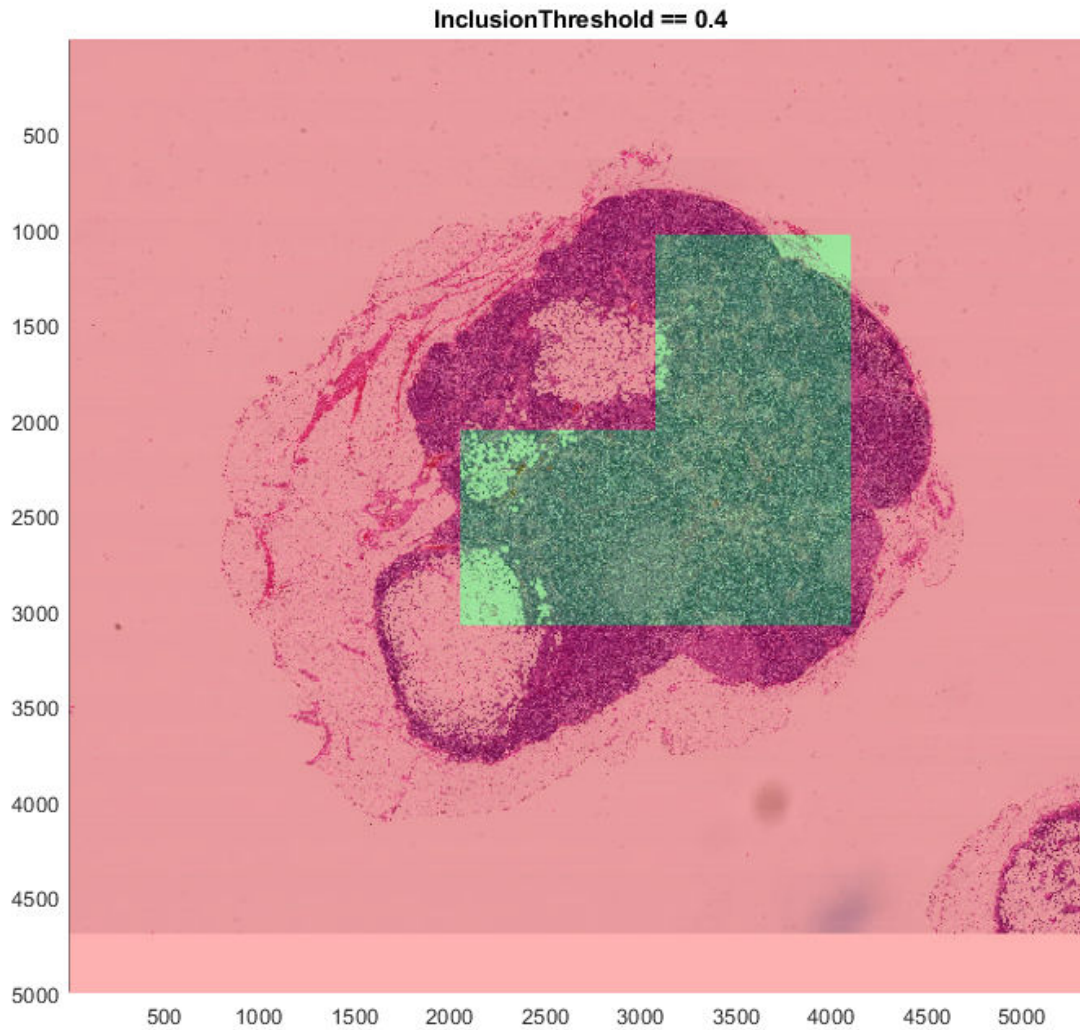
In the extreme case, process all blocks that have at least a single `true` pixel in the mask. To specify this option, set the `'InclusionThreshold'` property to `0`. Not all blocks of the image are included.

```
showmask(h,bmask,1,'InclusionThreshold',0);
title('InclusionThreshold == 0')
```
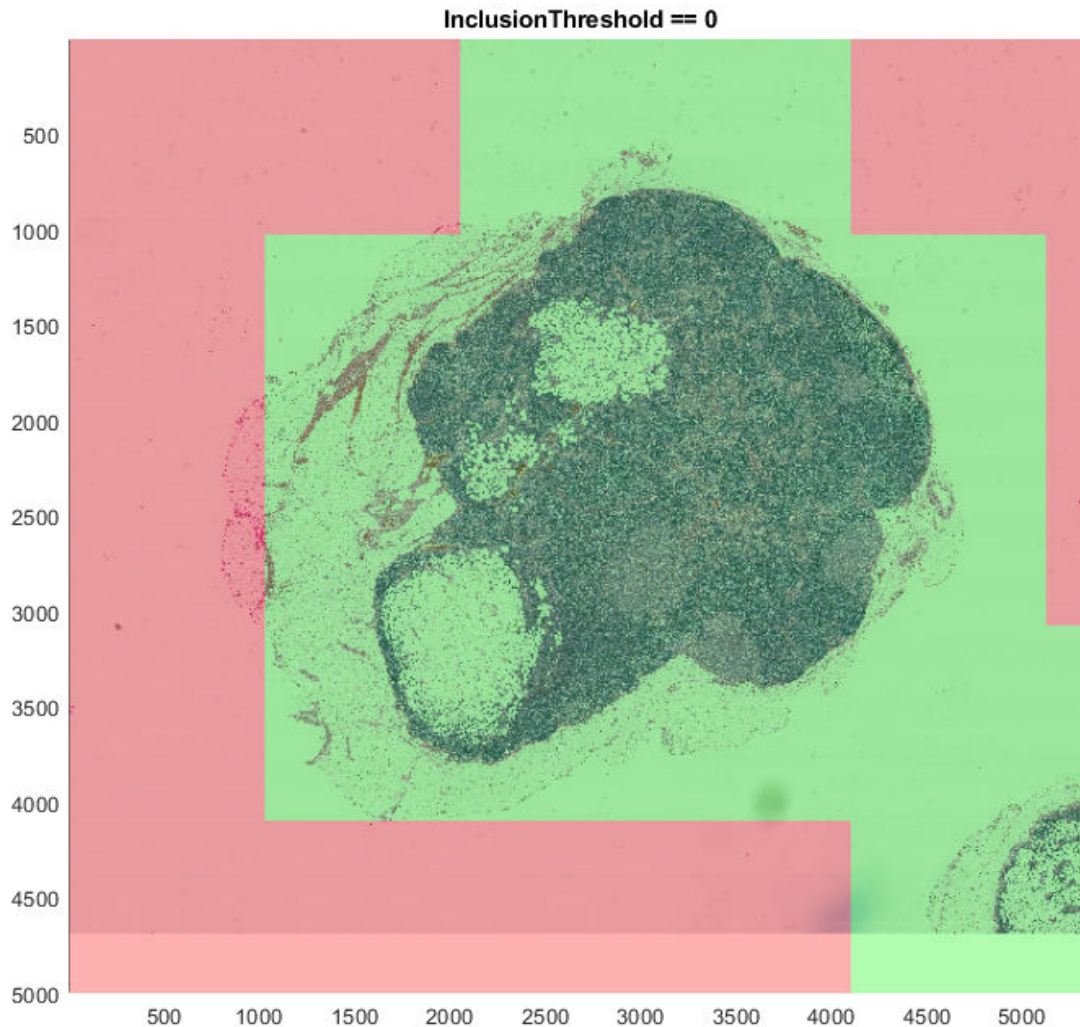
Using the mask with any value of `'InclusionThreshold'` will decrease the total execution time because `apply` will process fewer blocks than the full image. The benefit of using a mask becomes more significant as the image increases in resolution and as the processing pipelines increases in complexity.

Measure the execution time of filtering the full image.

```
tic
bout = apply(bim,1, ...
    @(block)imnlmfilt(block,'DegreeOfSmoothing',15));
tFullProcessing = toc;
```

Measure the execution time of filtering only blocks with the ROI.

```
tic
boutMasked = apply(bim,1, ...
```

```
      @(block)imnlmfilt(block,'DegreeOfSmoothing',15), ...
      'Mask',bmask,'InclusionThreshold',0);
tMaskedProcessing = toc;

bigimageshow(boutMasked)
defaultBlockSize = bim.BlockSize(1,:);
title(['Processed Image Using Mask with Default BlockSize == [' ...
      num2str(defaultBlockSize) ']']);
```
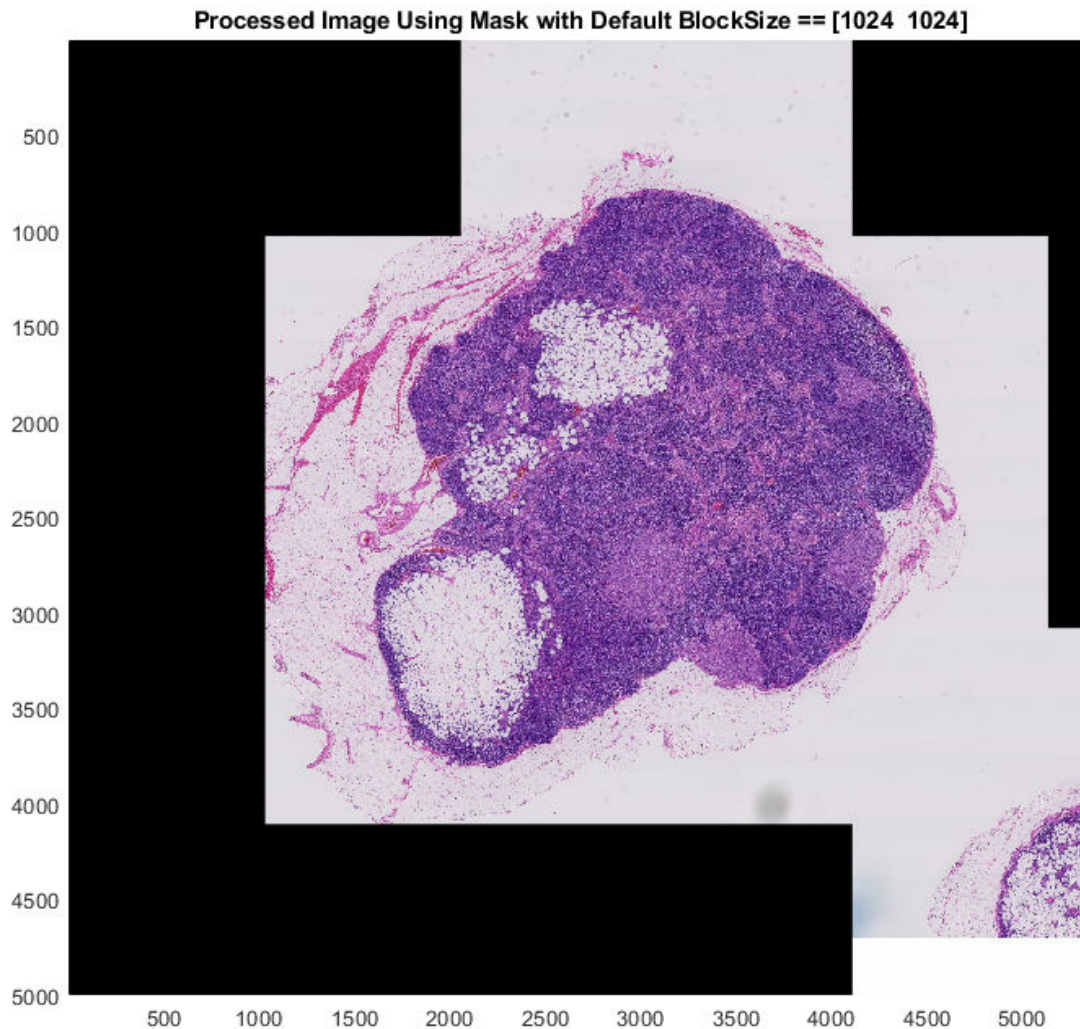

Compare the execution time of processing the full image compared to processing only blocks in the ROI.

```
disp(['Speedup using mask: ' ...
      num2str(tFullProcessing/tMaskedProcessing) 'x']);

Speedup using mask: 1.6817x
```

Adjust Block Size to Follow Contours of Region of Interest

You can decrease the block size to gain a tighter wrap around the ROI. For some block sizes, this will reduce the execution time because apply will process fewer pixels outside the ROI. However, if the block size is too small, then performance will decrease because the overhead of processing a larger number of block will offset the reduction in the number of pixels processed.

Highlight the blocks that apply will process using a smaller block size. To specify the block size, set the 'BlockSize' property.

```
blockSize = [512 512];
h = bigimageshow(bim);
showmask(h,bmask,1,'BlockSize',blockSize,'InclusionThreshold',0);
title(['BlockSize == [' num2str(blockSize) '], InclusionThreshold == 0'])
```
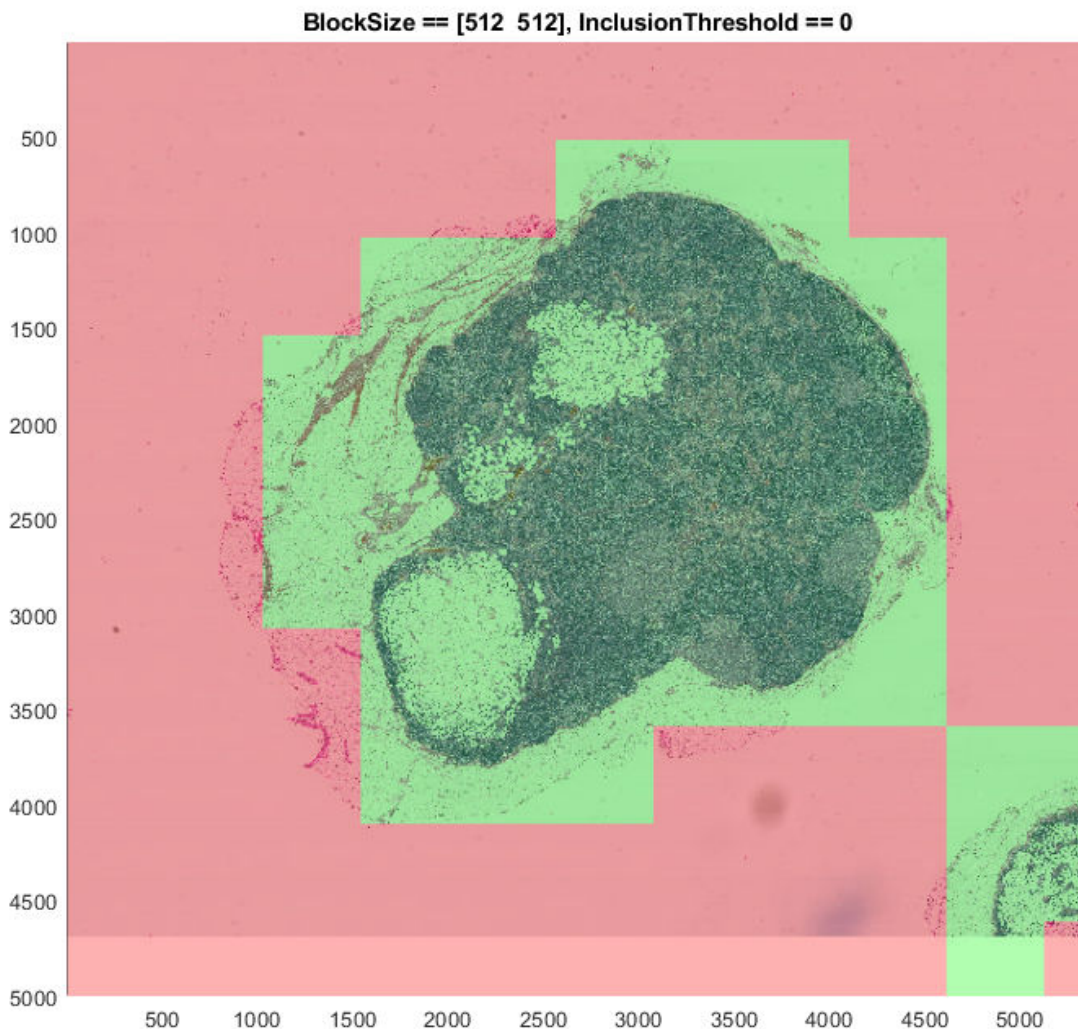

Measure the execution time of filtering all blocks within the ROI with a decreased block size.

16-25

```
tic
boutMasked = apply(bim,1, ...
    @(block)imnlmfilt(block,'DegreeOfSmoothing',15), ...
    'Mask',bmask,'BlockSize',blockSize,'InclusionThreshold',0);
tSmallerBlockProcessing = toc;

bigimageshow(boutMasked);
title(['Processed Image Using Mask with BlockSize == [' ...
    num2str(blockSize) ']']);
```
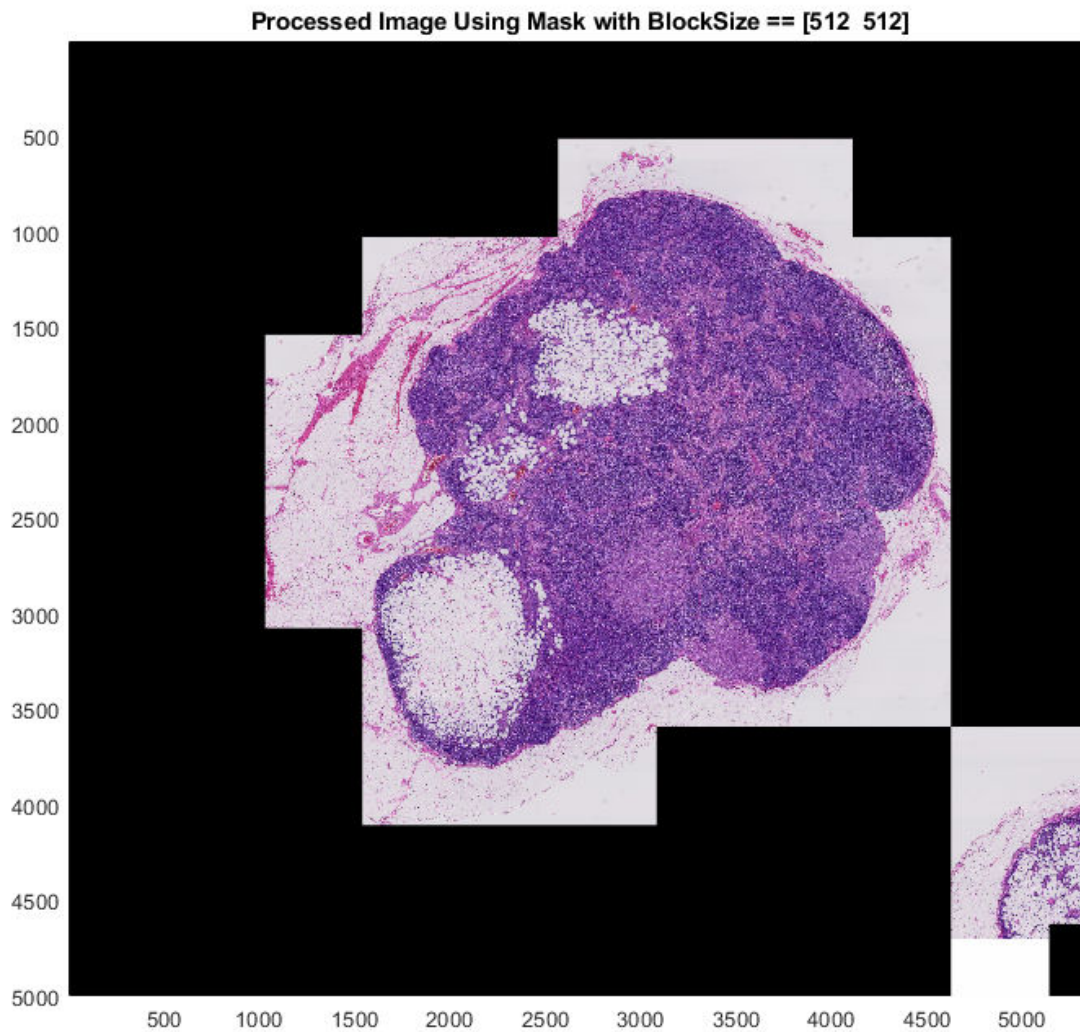

Compare the execution time of processing the entire ROI with smaller blocks compared to larger blocks.

```
disp(['Additional speedup using mask with decreased block size: ' ...
    num2str(tMaskedProcessing/tSmallerBlockProcessing) 'x']);
```

Additional speedup using mask with decreased block size: 1.3684x

See Also

apply | bigimage | bigimageshow

Explore Big Image Details with Interactive ROIs

This example shows how to display a detailed region of a big image selected interactively using ROI tools.

`bigimageshow` displays `bigimage` objects. If the `bigimage` object has multiple levels, then `bigimageshow` automatically picks the appropriate level based on the screen size and the view port. `bigimageshow` always works in a single 'world coordinate' and displays each level based on its spatial referencing information (the SpatialReferencing property). This allows two displays of the same `bigimage` object to show image detail at different levels, but share the same coordinate system.

Create a `bigimage` using a modified version of image "tumor_091.tif" from the CAMELYON16 data set. The original image is a training image of a lymph node containing tumor tissue. The original image has eight resolution levels, and the finest level has resolution 53760-by-61440. The modified image has only three coarse resolution levels. The spatial referencing of the modified image has been adjusted to enforce a consistent aspect ratio and to register features at each level.

```
bim = bigimage('tumor_091R.tif');
```

Display Overview Image and Detail View

Display the entire big image on the left side of a figure window by using the `bigimageshow` function. The resolution level of the displayed overview automatically changes depending on the size of the window and your screen size.

```
hf = figure;
haOView = subplot(1,2,1);
haOView.Tag = 'OverView';
hl = bigimageshow(bim,'Parent',haOView);
```
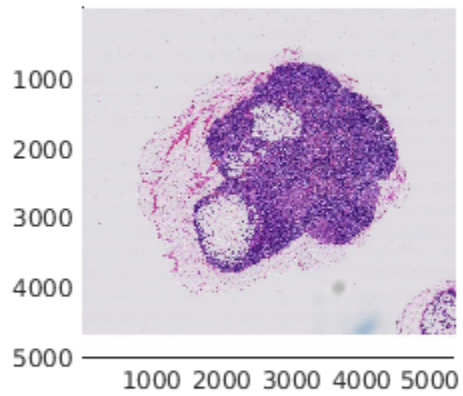
Fix the resolution level of the overview image as the coarsest resolution level.

```
hl.ResolutionLevel = bim.CoarsestResolutionLevel;
title('Overview');
```

Overview
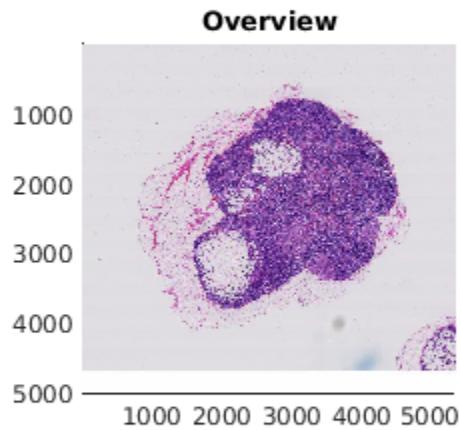
Display a detail view of the big image on the right side of the figure window. Allow `bigimageshow` to manage the level of the detail image automatically.

```
haDetailView = subplot(1,2,2);
haDetailView.Tag = 'DetailView';
hr = bigimageshow(bim,'Parent',haDetailView);
```
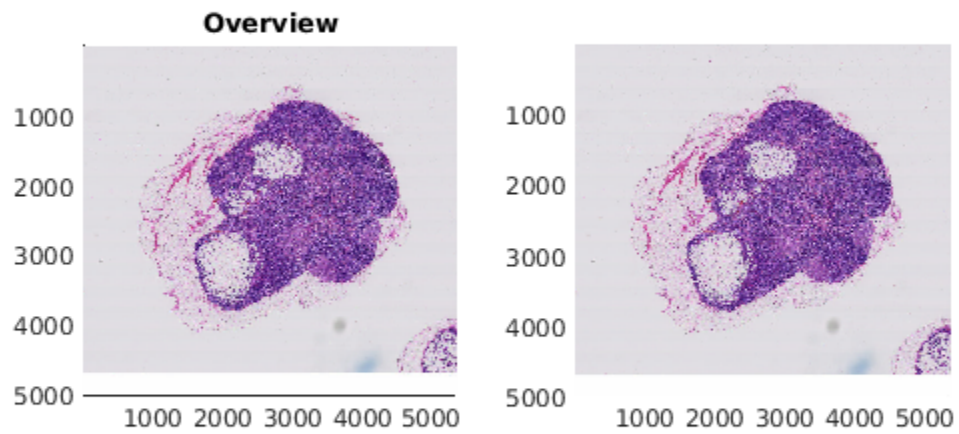
Zoom into the detailed view.

```
xlim([2800,3050])
ylim([500,750])
title('Detailed View');
```
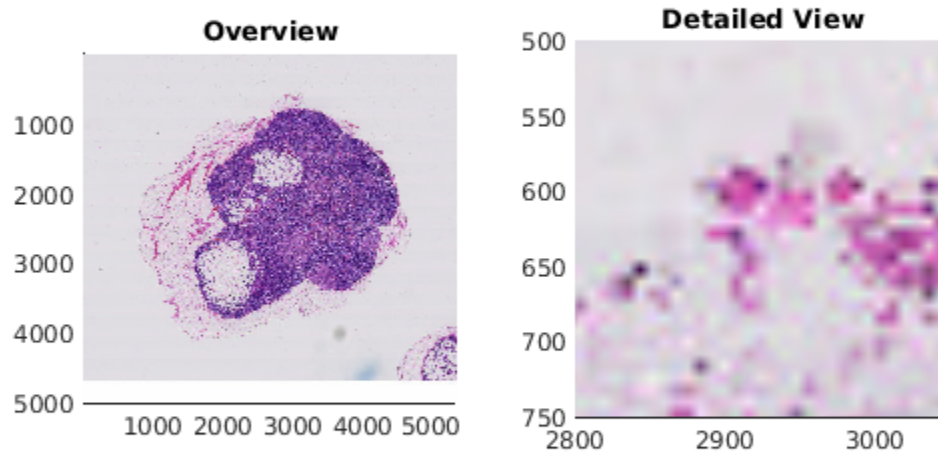
Add Interactive Rectangle ROI to Control Display View

In the overview image, draw a rectangle ROI. This example specifies the initial size and position of the rectangle programmatically by setting the `Position` property as a four-element vector of the form [xmin,ymin,width,height]. After the ROI appears on the overview, you can adjust the size and postion of the ROI interactively.

```
xrange = xlim;
yrange = ylim;
roiPosition = [xrange(1) yrange(1) xrange(2)-xrange(1) yrange(2)-yrange(1)];
hrOView = drawrectangle(haOView,'Position',roiPosition,'Color','r');
```
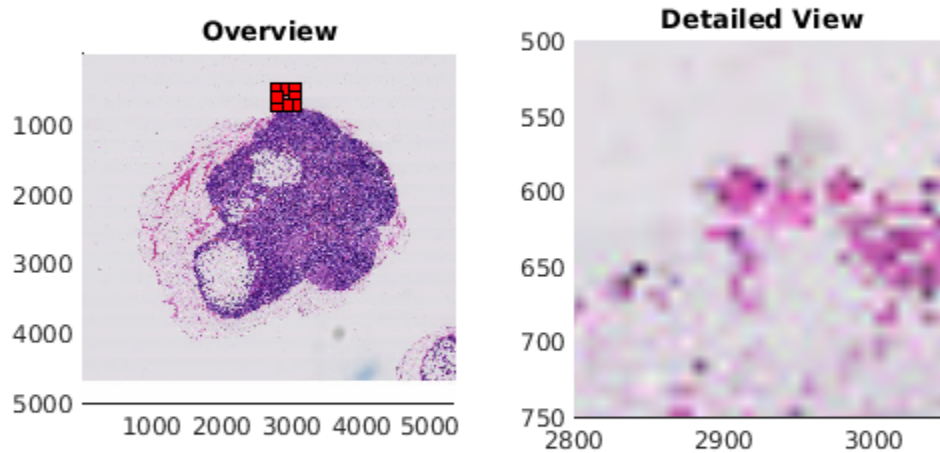
Save the handles of the rectangle to use when defining the interaction between the rectangle and the detail view.

```
hrOView.UserData.haDetailView = haDetailView;
haDetailView.UserData.hrOView = hrOView;
```

Add listeners to the detail view. These listeners detect changes in the spatial extents of the detail view. When the spatial extents change, the listeners call the `updateOverviewROI` helper function, which updates the extents of the ROI to match the extents of the detail view. The helper function is defined at the end of this example.

```
addlistener(haDetailView,'XLim','PostSet',@updateOverviewROI);
addlistener(haDetailView,'YLim','PostSet',@updateOverviewROI);
```

Add a listener to the rectangle ROI. These listeners detect changes in the spatial extent of the rectangle. When the limits change, the listeners call the `updateOverViewROI` helper function, which updates the extents of the detail image to match the extents of the ROI. The helper function is defined at the end of this example.

```
addlistener(hrOView,'MovingROI',@updateDetailView);
```

You can now change the size and position of the rectangle ROI interactively to adjust the display view. Similarly, when you zoom and pan the detail view, the size and position of the ROI updates.

This example changes the size and position of the ROI programmatically by setting the `Position` property.

```
hrOView.Position = [2230,1300,980,840];
evt.CurrentPosition = hrOView.Position;
updateDetailView(hrOView,evt);
```
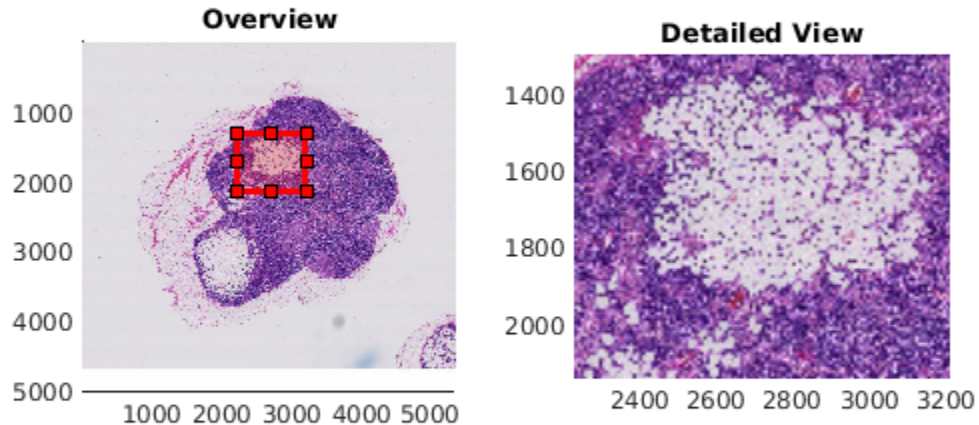

Define Callback Functions to Control Interactions

```
function updateOverviewROI(~,hEvt)
  % Update overview rectangle position whenever the right hand side
  % zooms/pans.
  ha = hEvt.AffectedObject;
  hr = hEvt.AffectedObject.UserData.hrOView;
  hr.Position = [ha.XLim(1),ha.YLim(1),diff(ha.XLim),diff(ha.YLim)];
end

function updateDetailView(hSrc,hEvt)
  % Update the right side detail view anytime the overview rectangle is
  % moved. bigimageshow automatically picks the appropriate image level.
  ha = hSrc.UserData.haDetailView;
  ha.XLim = [hEvt.CurrentPosition(1), ...
      hEvt.CurrentPosition(1)+hEvt.CurrentPosition(3)];
  ha.YLim = [hEvt.CurrentPosition(2), ...
      hEvt.CurrentPosition(2)+hEvt.CurrentPosition(4)];
end
```

See Also

bigimage | bigimageshow

Warp Big Image

This example shows how to apply a geometric transformation (warping) to a big image.

Applying a geometric transformation to an image is a key step in many image processing applications like image registration. You can use `imwarp` to warp coarse images that fit in memory. For high-resolution big images that do not fit in memory, warp blocks of the image. Set the spatial referencing of the warped image to preserve characteristics of the image such as pixel extents.

Create a `bigimage` using a modified version of image "tumor_091.tif" from the CAMELYON16 data set. The original image is a training image of a lymph node containing tumor tissue. The original image has eight resolution levels, and the finest level has resolution 53760-by-61440. The modified image has only three coarse resolution levels. The spatial referencing of the modified image has been adjusted to enforce a consistent aspect ratio and to register features at each level.

```
bim = bigimage('tumor_091R.tif');
```

Apply Geometric Transformation to Coarse Image

Create an `affine2d` object that stores information about an affine geometric transformation. This transformation applies translation and shear.

```
tform = affine2d([
    0.99 0.01 0
    0.17 0.98 0
    120 -30 1]);
```

Get the image at the coarsest resolution level.

```
imCoarse = getFullLevel(bim);
```

Warp the coarse image by using `imwarp`. Display the original image and the warped image in a montage.

```
imCoarseWarped = imwarp(imCoarse,tform);
montage({imCoarse,imCoarseWarped})
```
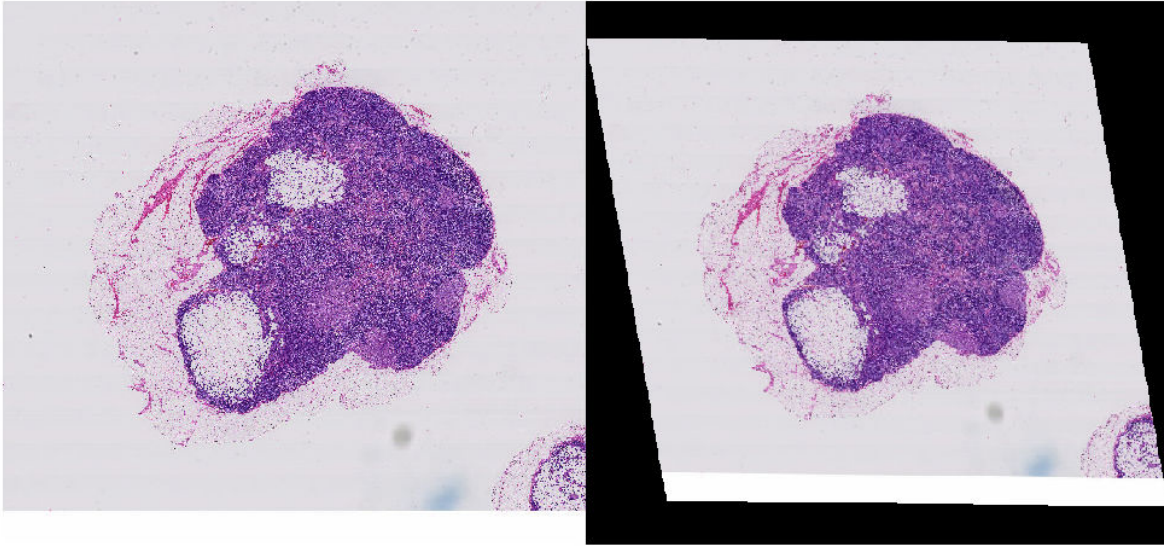
Create Spatial Referencing for Warped Fine Image

Before applying the geometric transformation to the image at a fine resolution level, calculate the spatial referencing of the big image after the warping. Use this spatial referencing when transforming blocks

Get the pixel extent of the original image from its spatial referencing information.

```
srcLevel = bim.FinestResolutionLevel;
inRef = bim.SpatialReferencing(srcLevel);
inPixelExtent = [inRef.PixelExtentInWorldX,inRef.PixelExtentInWorldY];
```

Calculate the output horizontal and vertical spatial limits when the transformation is applied.

```
[xout,yout] = outputLimits(tform,inRef.XWorldLimits,inRef.YWorldLimits);
```

Calculate the size of the output image that preserves the pixel extent. Specify the image size in the format [*numrows, numcols*].

```
outImgSize = [ceil(diff(yout)/inPixelExtent(2)),ceil(diff(xout)/inPixelExtent(1))];
```

Create an `imref2d` object that stores the spatial referencing information of the warped image. Set the world limits and image size of the warped image.

```
outRef = imref2d(outImgSize,xout,yout)

outRef =
  imref2d with properties:

           XWorldLimits: [120.5800 6275]
           YWorldLimits: [-29.5050 4.9241e+03]
              ImageSize: [4954 6155]
    PixelExtentInWorldX: 0.9999
    PixelExtentInWorldY: 0.9999
```

```
       ImageExtentInWorldX: 6.1544e+03
       ImageExtentInWorldY: 4.9536e+03
           XIntrinsicLimits: [0.5000 6.1555e+03]
           YIntrinsicLimits: [0.5000 4.9545e+03]
```

Specify the origin of the output as the top left pixel.

```
outOrigin = [xout(1),yout(1)];
```

Calculate the corresponding output pixel dimensions.

```
outPixelExtent = [outRef.PixelExtentInWorldX,outRef.PixelExtentInWorldY];
halfPixWidth = outPixelExtent/2;
```

Apply Block-Wise Warping to Fine Image

Create a writable `bigimage` by specifying the output spatial referencing information. Specify a block size that is large enough to use memory efficiently.

```
outBlockSize = [1024 1024];
bwarped = bigimage(outRef,bim.Channels,bim.ClassUnderlying, ...
    'BlockSize',outBlockSize);
```

Loop through the output image, one block at a time. For each output block:

1   Find the coordinates of the four corners of the output block.
2   Inverse map these coordinates back to the input to get the input (source) region.
3   Read the contents of the input region.
4   Create spatial referencing describing the input region.
5   Calculate the output block content by using `imwarp`.
6   Write the output block to the output image by using the `setBlock` function.

If you have Parallel Computing Toolbox™, then you can replace the outer `for` statement with a `parfor` statement to run the loop in parallel.

```
colStarts = 1:outBlockSize(2):outRef.ImageSize(2);
for cInd = 1:numel(colStarts)
    colStart = colStarts(cInd);
    for rStart = 1:outBlockSize(1):outRef.ImageSize(1)

        % Center of top left pixel of this block in world units
        xyStart = [colStart,rStart].* outPixelExtent;
        xyStart = xyStart + outOrigin;

        % Center of bottom right pixel of this block in world units
        bsize = fliplr(bwarped.BlockSize); % (r,c) -> (x,y)
        xyEnd = ([colStart,rStart] + (bsize-1)).*outPixelExtent;
        xyEnd = xyEnd + outOrigin;

        % Output bounding box
        outbbox = [xyStart
            xyStart(1) xyEnd(2)
            xyEnd(1) xyStart(2)
            xyEnd];
```

```matlab
        % Corresponding spatial reference which describes this rectangle
        outRegionRef = imref2d(bsize);
        outRegionRef.XWorldLimits = [xyStart(1)-halfPixWidth(1),xyEnd(1)+halfPixWidth(1)];
        outRegionRef.YWorldLimits = [xyStart(2)-halfPixWidth(2),xyEnd(2)+halfPixWidth(2)];

        % Get corresponding input region. The region is not rectangular if the transformation
        % includes shear
        inRegion = transformPointsInverse(tform,outbbox);

        % Clamp to image extents (working with pixel centers)
        xcenterLims = inRef.XWorldLimits + [halfPixWidth(1)-halfPixWidth(1)];
        ycenterLims = inRef.YWorldLimits + [halfPixWidth(2)-halfPixWidth(2)];
        inRegion(:,1) = max(inRegion(:,1),xcenterLims(1));
        inRegion(:,1) = min(inRegion(:,1),xcenterLims(2));
        inRegion(:,2) = max(inRegion(:,2),ycenterLims(1));
        inRegion(:,2) = min(inRegion(:,2),ycenterLims(2));

        % Find the input bounding box (still using pixel centers)
        inbboxStart = [min(inRegion(:,1)) min(inRegion(:,2))];
        inbboxEnd = [max(inRegion(:,1)) max(inRegion(:,2))];

        % Read corresponding input region
        inputRegion = getRegion(bim,srcLevel,inbboxStart,inbboxEnd);

        % Input region's spatial referencing
        inRegionRef = imref2d(size(inputRegion));
        % Convert to pixel edges from pixel centers
        inRegionRef.XWorldLimits = [inbboxStart(1)-halfPixWidth(1),inbboxEnd(1)+halfPixWidth(2)]
        inRegionRef.YWorldLimits = [inbboxStart(2)-halfPixWidth(1),inbboxEnd(2)+halfPixWidth(2)]

        warpedBlock = imwarp(inputRegion,inRegionRef,tform,'OutputView',outRegionRef);

        % Set the block data in the output bigimage
        setBlock(bwarped,1,xyStart,warpedBlock);
    end
end
```

Display the warped image.

```matlab
bigimageshow(bwarped)
```
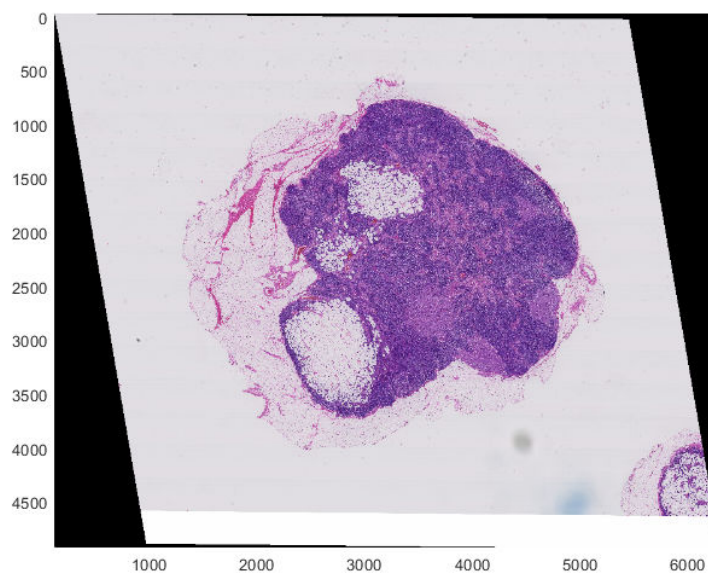
See Also

affine2d | bigimage | bigimageshow | getRegion | imref2d | setBlock | transformPointsInverse

Create Labeled bigimage from ROIs and Masks

This example shows how to create a labeled `bigimage` consisting of categorical labels.

There are several ways to specify categorical label data for an image. This example shows two approaches. One approach uses polygonal ROI objects that store the coordinates of the boundaries of tumor and normal tissue. The other approach uses a mask to indicate a binary segmentation of the image into tissue and background. The example combines the information in the polygon coordinates and mask representations to create a single labeled `bigimage`.

Create a `bigimage` using a modified version of image "tumor_091.tif" from the CAMELYON16 data set. The original image is a training image of a lymph node containing tumor tissue. The original image has eight resolution levels, and the finest level has resolution 53760-by-61440. The modified image has only three coarse resolution levels. The spatial referencing of the modified image has been adjusted to enforce a consistent aspect ratio and to register features at each level.

```
bim = bigimage('tumor_091R.tif');
```

Get the spatial referencing and pixel extent of the `bigimage` at the desired output level.

```
ref = bim.SpatialReferencing(bim.FinestResolutionLevel);
pixelExtent = [ref.PixelExtentInWorldX,ref.PixelExtentInWorldY];
```

Load Label Data

The CAMELYON16 data set provides labels of tumor and normal regions as a set of coordinates specifying manually annotated region boundaries with respect to the finest resolution level. When pixels exist within the boundaries of both a normal region and a tumor region, the correct label for those pixels is normal tissue.

Load label data for the big image. This example uses a modified version of labels of the "tumor_091.tif" image from the CAMELYON16 data set. The original labels are stored in XML format. The modified labels are resampled and saved as MAT files.

```
roiPoints = load('labelledROIs.mat')

roiPoints = struct with fields:
      cancerRegions: {[344×2 double]  [53×2 double]  [539×2 double]  [247×2 double]  [37×2 doub
   nonCancerRegions: {[46×2 double]}
```

Create polygonal ROI objects that store the coordinates of the tumor boundaries and normal tissue boundaries.

```
tumorPolys = cellfun(@(position) images.roi.Polygon( ...
    'Position',position,'Visible','on','Color','r'), ...
    roiPoints.cancerRegions);
normalPolys = cellfun(@(position) images.roi.Polygon( ...
    'Position',position,'Visible','on','Color','g'), ...
    roiPoints.nonCancerRegions);
```

Represent Labels as ROI Objects

Display the image overlaid with the annotated ROIs. The ROIs have the same coordinate system as the image, so changing the resolution levels of the displayed image still renders the ROIs accurately.

```
figure
h = bigimageshow(bim);
set(tumorPolys,'Parent',gca);
set(normalPolys,'Parent',gca);
title(['Resolution Level:' num2str(h.ResolutionLevel)]);
```
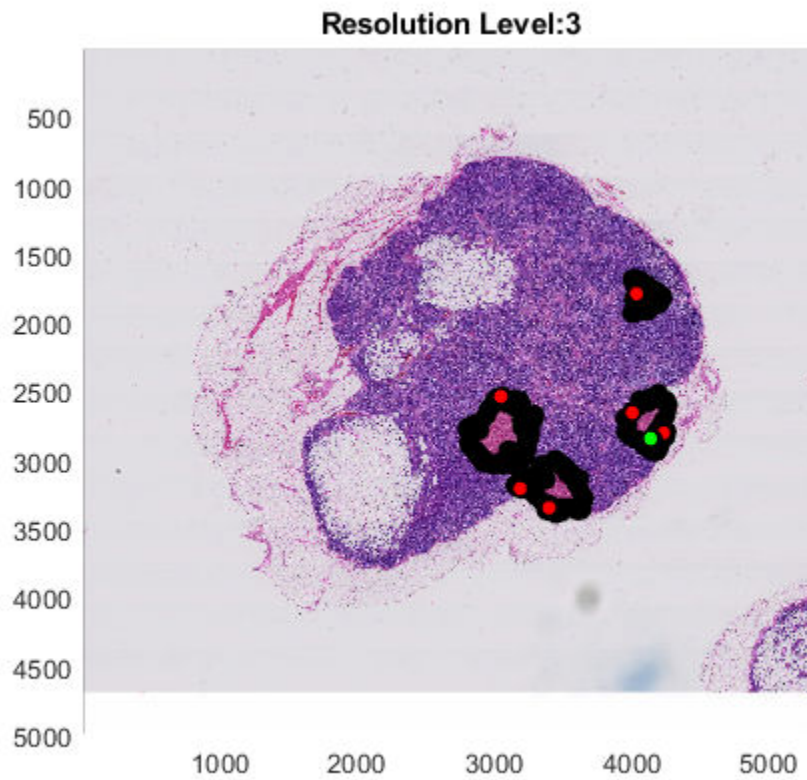

Zoom in to one ROI.

```
xlim([3940 4290])
ylim([2680 3010])
title(['Resolution Level:' num2str(h.ResolutionLevel)]);
```
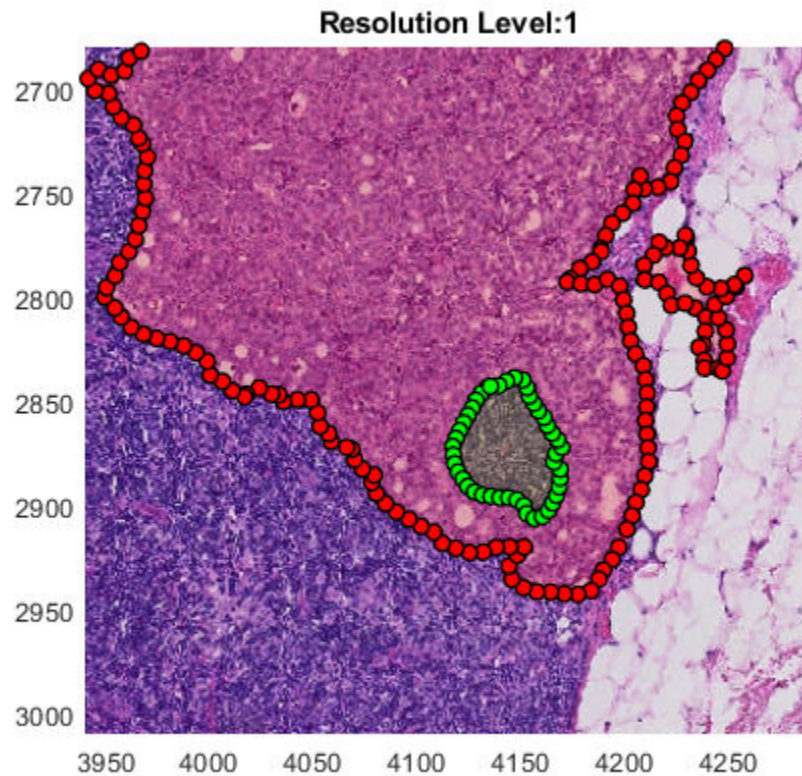
Resolution Level:1

Represent Labels as Masks

Create a mask at the coarsest resolution level for the stained region, which includes both tumor and normal tissue. The mask is `1` (`true`) for pixels whose grayscale value is less than 130. Fill small holes in the mask by performing morphological closing using the `bwmorph` function.

```
tissueMask = apply(bim,bim.CoarsestResolutionLevel, ...
    @(im)bwmorph(rgb2gray(im)<130,'close'));
bigimageshow(tissueMask);
```
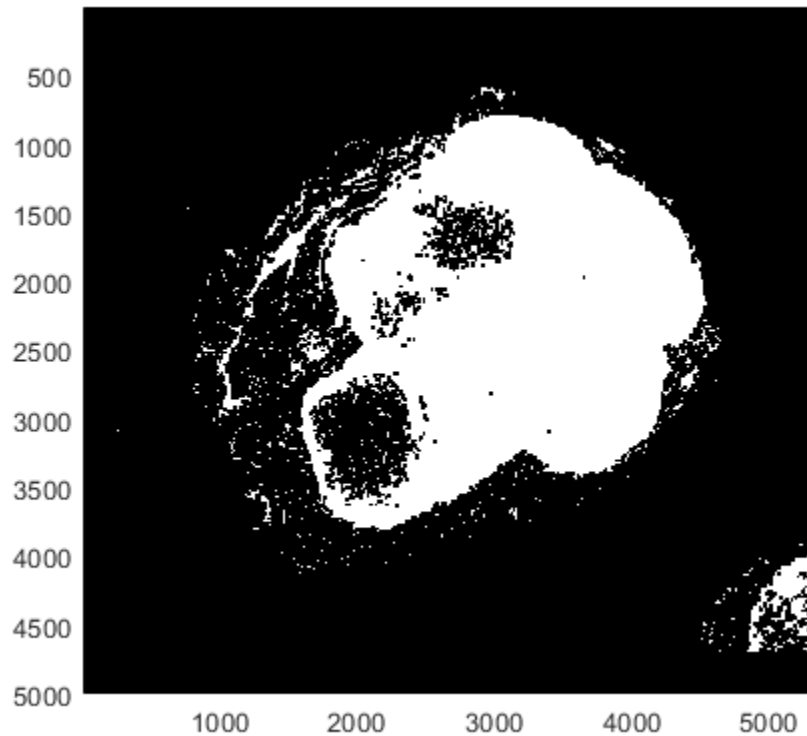
Create Combined Labeled bigimage

Combine the information in the polygon coordinates and mask representations to create a single labeled bigmage.

To store the labeled image, create a writeable bigimage with data type categorical. Specify the required class names and the corresponding numeric pixel label ID values. Assign the label 0 to the 'Background' class. Assign a value to the 'UndefinedID' label that does not equal any of the pixel label IDs.

```
blockSize = [512 512];
bLabeled = bigimage(ref,1,'categorical', ...
    'Classes',["Background","Normal","Tumor"], ...
    'PixelLabelIDs',[0 1 2], ...
    'UndefinedID',255, ...
    'BlockSize',blockSize);
```

Loop through each output bigimage, one block at a time. Determine the label of each block, then set the pixel data of the block accordingly.

To determine the label of each block, start with the mask of all tissue. Pixel values of 0 in the mask correspond to background, which matches the pixel label ID of the 'Background' class. Pixel values of 1 in the mask correspond to all tissue, which matches the pixel label ID of the 'Normal' class. Convert the polygon coordinates of tumor tissue to a mask by using the poly2mask function, then replace those pixels with the pixel label ID of the 'Tumor' class, 2.

If you have Parallel Computing Toolbox™, then you can run the loop in parallel by replacing the `for` statement with a `parfor` statement.

```matlab
for cStart = 1:bLabeled.BlockSize(2):ref.ImageSize(2)
    for rStart = 1:bLabeled.BlockSize(1):ref.ImageSize(1)
        % Find the center of top left pixel of the block in world units
        xyStart = [cStart, rStart].*pixelExtent;

        % Clamp block edges to the border of the image
        cEnd = min(cStart + bLabeled.BlockSize(2)-1,ref.ImageSize(2));
        rEnd = min(rStart + bLabeled.BlockSize(1)-1,ref.ImageSize(1));

        % Find the center of bottom right pixel of the block in world units
        xyEnd = [cEnd rEnd].*pixelExtent;

        % Calculate the block size
        bsize = [rEnd cEnd] - [rStart cStart] + 1;

        % Create a grid encompassing all pixels in the block.
        [xgrid,ygrid] = meshgrid(xyStart(1):ref.PixelExtentInWorldX:xyEnd(1), ...
            xyStart(2):ref.PixelExtentInWorldY:xyEnd(2));

        % Get the mask of all tissue in the block
        blockNumeric = getRegion(tissueMask,1,xyStart,xyEnd);
        % This tissue mask was created at a coarse level, so scale it up
        blockNumeric = imresize(blockNumeric,bsize);

        % Find the pixel coordinates of healthy tissue then convert the
        % polygon coordinates to a mask.
        tmask = false(bsize);
        for ind = 1:numel(roiPoints.cancerRegions)
            vertices = roiPoints.cancerRegions{ind};
            % Transform coordinates to local block pixel locations
            vertices = (vertices-xyStart)./pixelExtent+1;
            isTumor = poly2mask(vertices(:,1),vertices(:,2), ...
                bsize(1), bsize(2));
            tmask = tmask|isTumor;
        end

        % Some healthy tissue ROIs are enclosed within a tumor ROI. Find
        % the pixel coordinates of healthy tissue then convert the polygon
        % coordinates to a mask.
        for ind = 1:numel(roiPoints.nonCancerRegions)
            vertices = roiPoints.nonCancerRegions{ind};
            % Transform coordinates to local block pixel locations
            vertices = (vertices-xyStart)./pixelExtent+1;
            isHealthy = poly2mask(vertices(:,1),vertices(:,2), ...
                bsize(1),bsize(2));
            tmask = tmask & ~isHealthy;
        end

        % Set the value of pixels in tumor regions as the corresponding
        % pixel label ID, |2|.
        blockNumeric = uint8(blockNumeric);
        blockNumeric(tmask) = 2;

        % Create a categorical image from the block data.
        blockCategorical = categorical(blockNumeric,...
```

```
            bLabeled.PixelLabelIDs,bLabeled.Classes);

        % Set the block of the categorical |bigimage| as the categorical
        % block image.
        setBlock(bLabeled,1,xyStart,blockCategorical);
    end
end
```

Display Overlay of Labels and Original Data

Display the image data, then display the labeled image data in the same axes. The three labels (normal, tumor, and background) appear in three different colors. Make the labels partially transparent so that you can distinguish the image content underneath.

```
hbim = bigimageshow(bim);
hla = axes;
hbl  = bigimageshow(bLabeled,'Parent',hla);
hbl.AlphaData = 0.7;
hla.Visible = 'off';
```
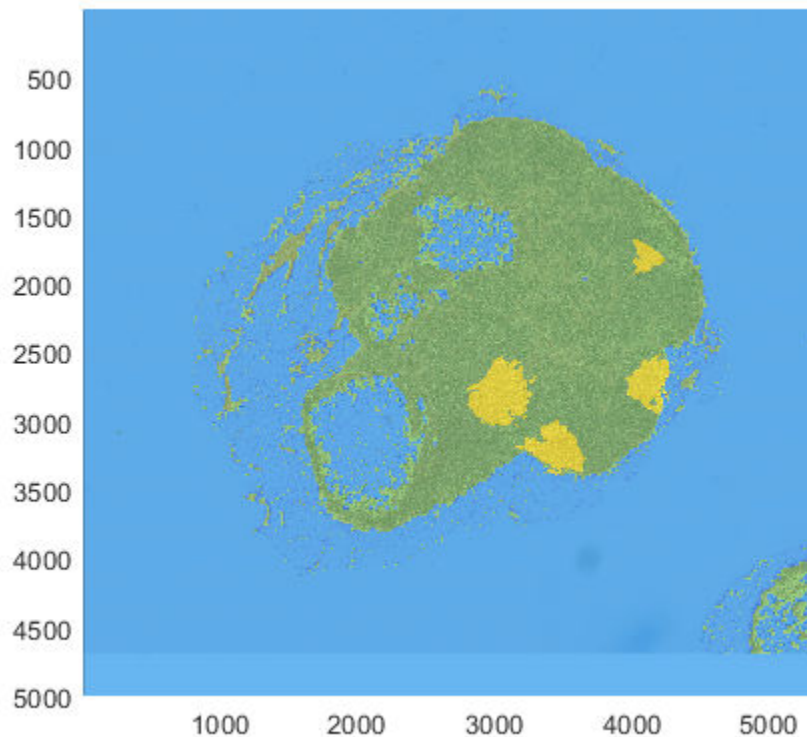

Zoom in to a ROI. Increase the label transparency so that you can more clearly distinguish the image content underneath.

```
linkaxes(findall(gcf,'Type','axes'));
xlim([3940 4290])
ylim([2680 3010])
hbl.AlphaData = 0.3;
```
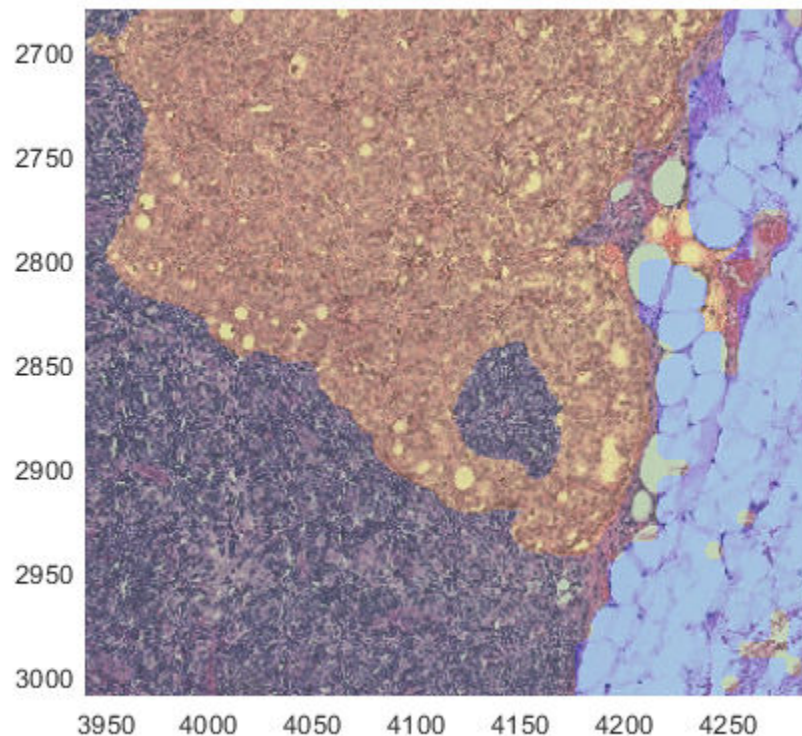
Save Labels for Use Later

For most data sets, you can create labels once and then reuse the labels for multiple training sessions. The labeled big image, `bLabeled`, is backed by temporary files that do not exist across MATLAB® sessions. To reuse the labels in a different session of MATLAB, write `bLabeled` to a persistent location.

```
imageDir = 'Labels';
if exist(imageDir,'dir')
    rmdir('Labels','s');
end
labelDir = fullfile(imageDir,'labelled');
write(bLabeled,labelDir);
```

In a fresh session of MATLAB, you can reload the labeled `bigimage` by creating a new `bigimage`. When loading a labeled `bigimage`, you must also specify 'Classes', 'PixelLabelIDs', and 'UndefinedID'.

```
bLabeled = bigimage(labelDir, ...
    'PixelLabelIDs',[ 0 1 2],'Classes',["Background","Normal","Tumor"], ...
    'UndefinedID',255);
```

See Also

bigimage | bigimageDatastore | bigimageshow

Related Examples

- "Deep Learning Classification of Large Multiresolution Images" (Deep Learning Toolbox)

Neighborhood and Block Operations

This chapter discusses these generic block processing functions. Topics covered include

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 17-3
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 17-6
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 17-20.

Sliding Neighborhood Operations

In this section...
"Determine the Center Pixel" on page 17-3
"General Algorithm of Sliding Neighborhood Operations" on page 17-4
"Border Padding Behavior in Sliding Neighborhood Operations" on page 17-4
"Implementing Linear and Nonlinear Filtering as Sliding Neighborhood Operations" on page 17-4

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 17-6 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 17-3.
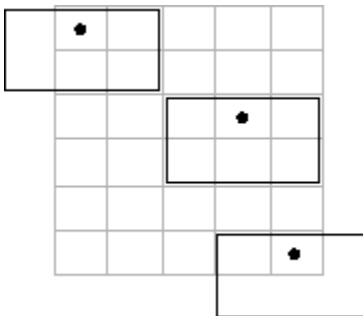
Neighborhood Blocks in a 6-by-5 Matrix

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 17-20.

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". For more information about anonymous functions, see "Anonymous Functions".)

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);
```
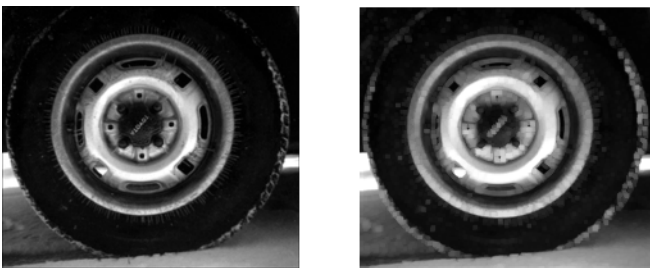

Each Output Pixel Set to Maximum Input Neighborhood Value

Distinct Block Processing

In this section...
"Implement Block Processing Using the blockproc Function" on page 17-6
"Apply Padding" on page 17-7

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 17-7. (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 17-3.)
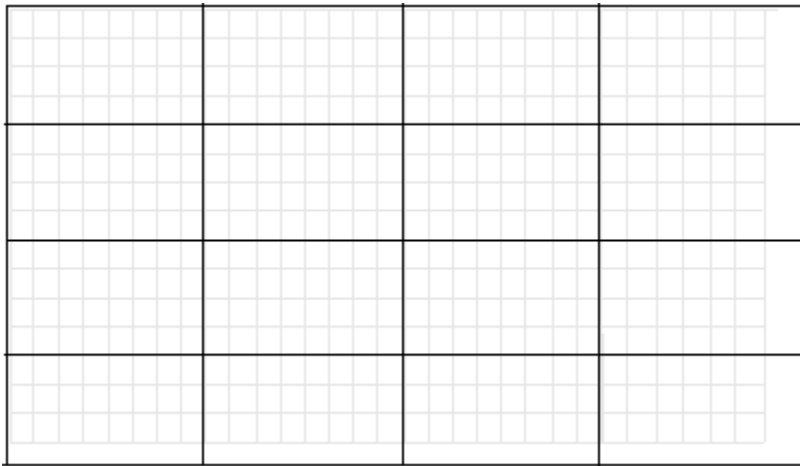
Image Divided into Distinct Blocks

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". For more information about anonymous functions, see "Anonymous Functions".)

```
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 17-20.

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 17-6, 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.
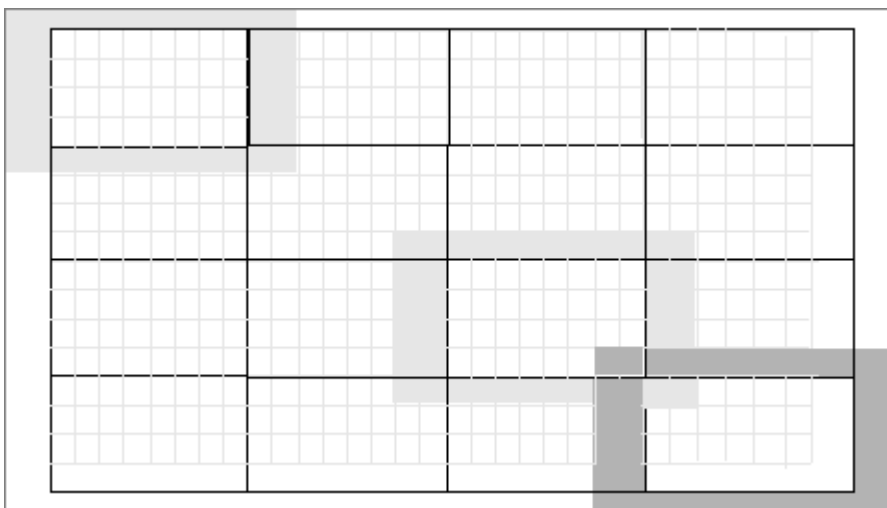
Image Divided into Distinct Blocks with Specified Borders

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.

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. (Stripped TIFF images are always organized in rows, never in columns.) 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.

Choose Block Size to Optimize blockproc Performance

This example shows how block size influences the performance of `blockproc`. In each of these cases, the total number of pixels in each block is approximately the same. Only the dimensions of the blocks are different.

Read an image file and convert it to a TIFF file.

```
I = imread('concordorthophoto.png','PNG');
imshow(I)
```

```
imwrite(I,'concordorthophoto.tif','TIFF');
```

Use `imfinfo` to determine whether `concordorthophoto.tif` is organized in strips or tiles. The `RowsPerStrip` field of the `info` struct indicates that this TIFF image is organized in strips with 34 rows per strip. Each strip spans the width of the image and is 34 pixels tall.

```
info = imfinfo('concordorthophoto.tif');
info.RowsPerStrip
```

```
ans = 34
```

Get the image size from the `Height` and `Width` fields of `info`. This image has size 2215-by-2956 pixels.

```
h = info.Height
```

```
h = 2215
```

```
w = info.Width
```

```
w = 2956
```

Case 1: Square Blocks

Process the image using square blocks of size 500-by-500 pixels. 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 34 rows per strip and 500 rows per block, `blockproc` accesses the disk 15 times for each block. The image is approximately 6 blocks wide (2956/500 = 5.912). `blockproc` 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.

```
blockSizeSquare = 500;
tic
im = blockproc('concordorthophoto.tif',[blockSizeSquare blockSizeSquare],@(s) s.data);
toc
```

```
Elapsed time is 1.131524 seconds.
```

Case 2: Column-Shaped Blocks

Process the image using blocks that span the full height of the image. Stripped TIFF files are organized in rows, so this block layout is exactly opposite the actual file layout on disk.

Select a block width such that the blocks have approximately the same number of pixels as the square block.

```
numCols = ceil(blockSizeSquare.^2 / h)
```

```
numCols = 113
```

The image is over 26 blocks wide (2956/`numCols` = 26.1593). Every strip must be read for every block, therefore `blockproc` reads the entire image from disk 26 times.

```
tic
im = blockproc('concordorthophoto.tif',[h numCols],@(s) s.data);
toc
```

```
Elapsed time is 0.356147 seconds.
```

Case 3: Row-Shaped Blocks

Process the image using blocks that span the full width of the image. This block layout aligns with the TIFF file layout on disk.

Select a block height such that the blocks have approximately the same number of pixels as the square block.

```
numRows = ceil(blockSizeSquare.^2 / w)
```

```
numRows = 85
```

Each block spans the width of the image, therefore `blockproc` reads each strip exactly once. The execution time is shortest when the block layout aligns with the TIFF image strips.

```
tic
im = blockproc('concordorthophoto.tif',[numRows w],@(s) s.data);
toc
```

```
Elapsed time is 0.188580 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 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-by-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 using the `multibandread` function. The LAN format stores the RGB data from the visible spectrum in bands 3, 2, and 1, respectively. You could create a truecolor image for further processing.

```
truecolor = multibandread('rio.lan', [512, 512, 7],...
    'uint8=>uint8', 128,'bil', 'ieee-le', {'Band','Direct',[3 2 1]});
```

For very large LAN files, however, reading and processing the entire image in memory using `multibandread` can be impractical, depending on your system capabilities. To avoid memory limitations, use the `blockproc` function. With `blockproc`, you can process images with a file-based workflow. You can read, process, and then write the results, one block at a time.

The `blockproc` function only supports reading and writing certain file formats, but it is extensible via the `ImageAdapter` class. To write an Image Adapter class for a particular file format, you must be able to:

- Query the size of the file on disk
- Read a rectangular block of data from the file

If you meet these two conditions, you can write an Image Adapter class for LAN files. You can parse the image header to query the file size, and you can modify the call to `multibandread` to read a particular block of data. You can encapsulate the code for these two objectives in an Image Adapter class structure, and then operate directly on large LAN files with the `blockproc` function. The

`LanAdapter` class is an Image Adapter class for LAN files, and is part of the Image Processing Toolbox software.

Examining the LanAdapter Class

This section describes the constructor, properties, and methods of the `LanAdapter` class. Studying the `LanAdapter` class helps prepare you for writing your own Image Adapter class. If you are new to object-oriented programming, see Developing Classes—Typical Workflow for general information on writing classes.

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.

```matlab
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');
        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.

```matlab
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)
    % 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 "Compute Statistics for Large Images" on page 17-28 example to see how the `blockproc` function works with the `LanAdapter` class.

See Also
ImageAdapter | blockproc | multibandread

More About
•   "Compute Statistics for Large Images" on page 17-28

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 17-20
"Using Column Processing with Distinct Block Operations" on page 17-21

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.
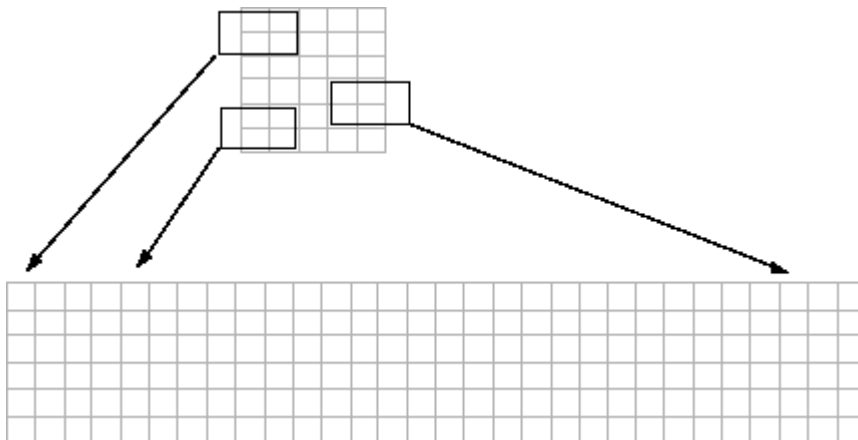
colfilt Creates a Temporary Matrix for Sliding Neighborhood

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 17-4.

```
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.
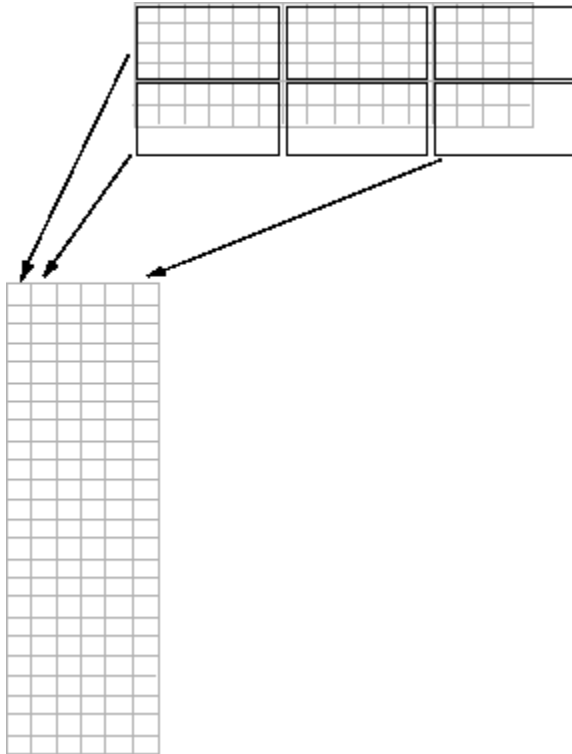
colfilt Creates a Temporary Matrix for Distinct Block Operation

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')
```

Original Image


```
imshow(normal_edges)
title('Conventional Edge Detection')
```

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.

```matlab
% 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);

imshow(block_edges)
title('Block Processing - Simplest Syntax')
```

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);

imshow(block_edges)
title('Block Processing - Block Borders')
```

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);

imshow(block_edges)
title('Block Processing - Borders & Fixed Threshold')
```

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

- "Compute Statistics for Large Images" on page 17-28

More About

- "Distinct Block Processing" on page 17-6
- "Block Size and Performance" on page 17-9
- "Create Function Handle"

Compute 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 "Enhance Multispectral Color Composite Images" on page 11-93 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: Create 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 "Perform Block Processing on Image Files in Unsupported Formats" on page 17-14.

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 results before enhancement.

```
imshow(truecolor)
title('Truecolor Composite (Not Enhanced)')
```

Truecolor Composite (Not Enhanced)

The resulting truecolor image is similar to that of `paris.lan` in the "Enhance Multispectral Color Composite Images" on page 11-93 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);

imshow(truecolor_enhanced)
title('Truecolor Composite with Blockwise Contrast Stretch')
```

Truecolor Composite with Blockwise Contrast Stretch
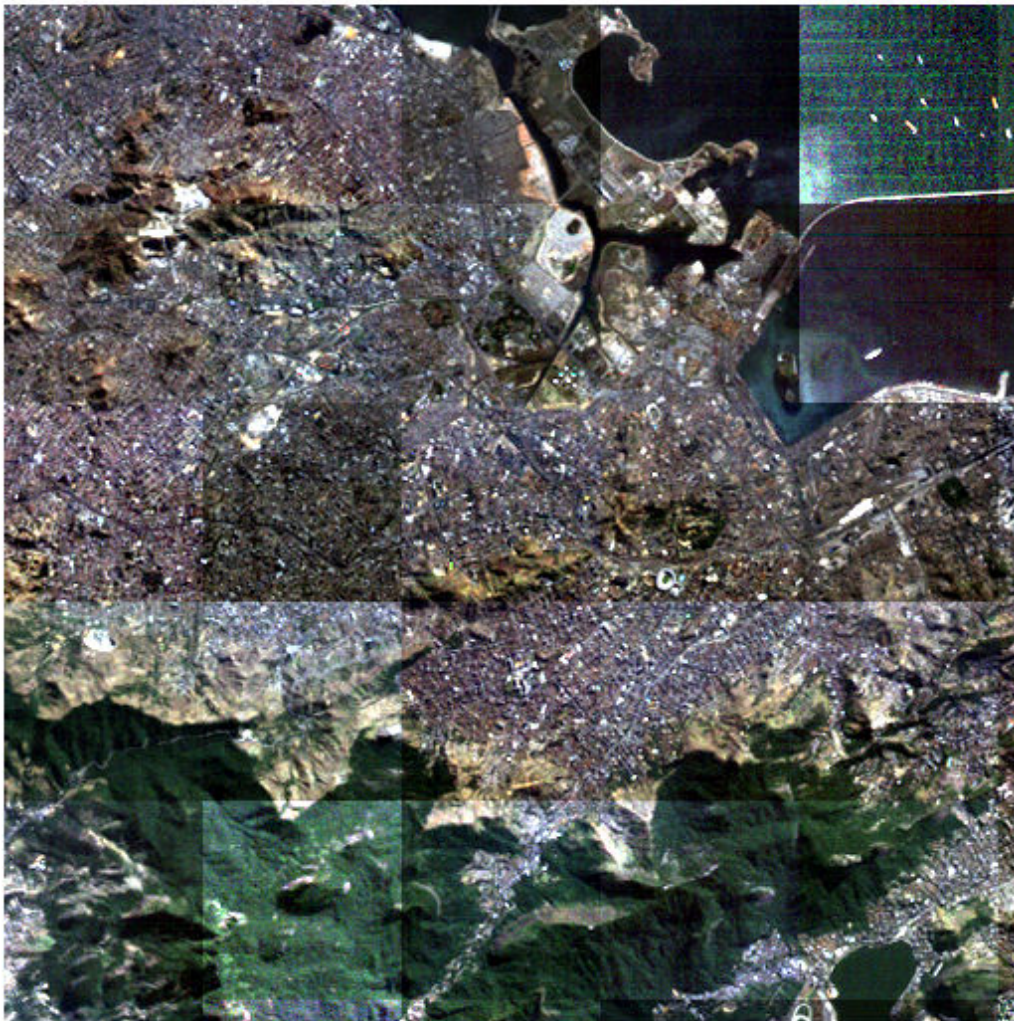
You can see immediately that the results are incorrect. The problem is that the `stretchlim` function computes the histogram on the input image and uses this information to compute the stretch limits. Since each block is adjusted in isolation from its neighbors, each block is computing different limits 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')
```
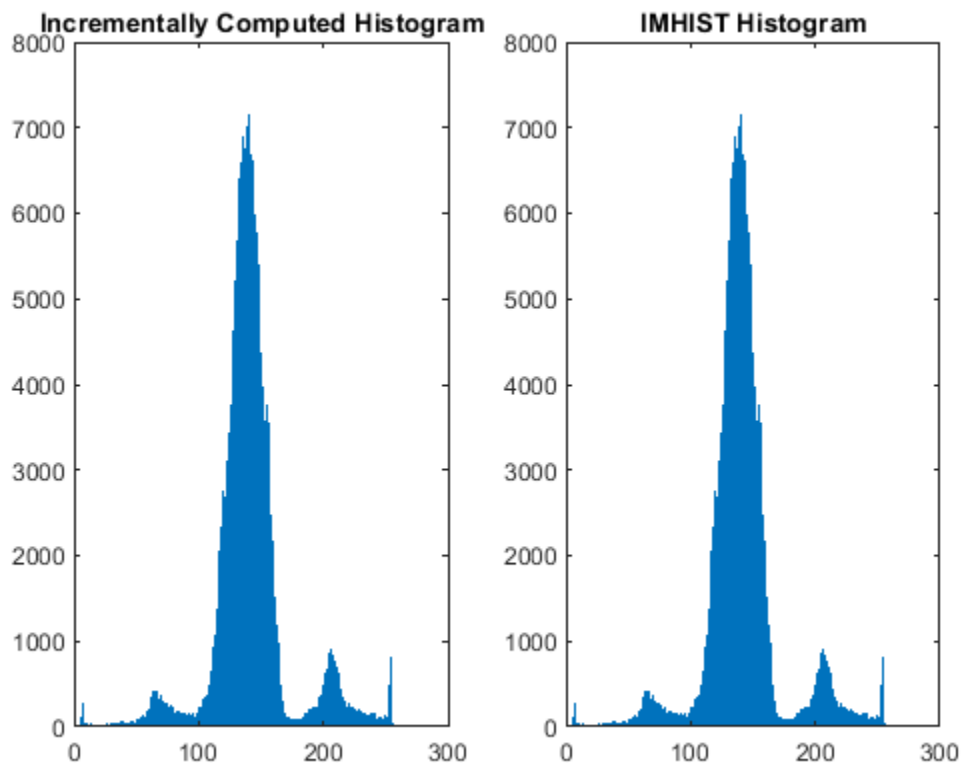

Step 4: Use the HistogramAccumulator Class with BLOCKPROC

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;
```

Set up `blockproc` function handle

```
addToHistFcn = @(block_struct) addToHistogram(hist_obj, block_struct.data);
```

Compute the histogram of the red channel. Notice that the `addToHistFcn` function handle does generate any output. Since the function handle passed 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 the results.

```
figure
stem(red_hist,'Marker','none')
title('Histogram of Red Band (Band 3)')
```
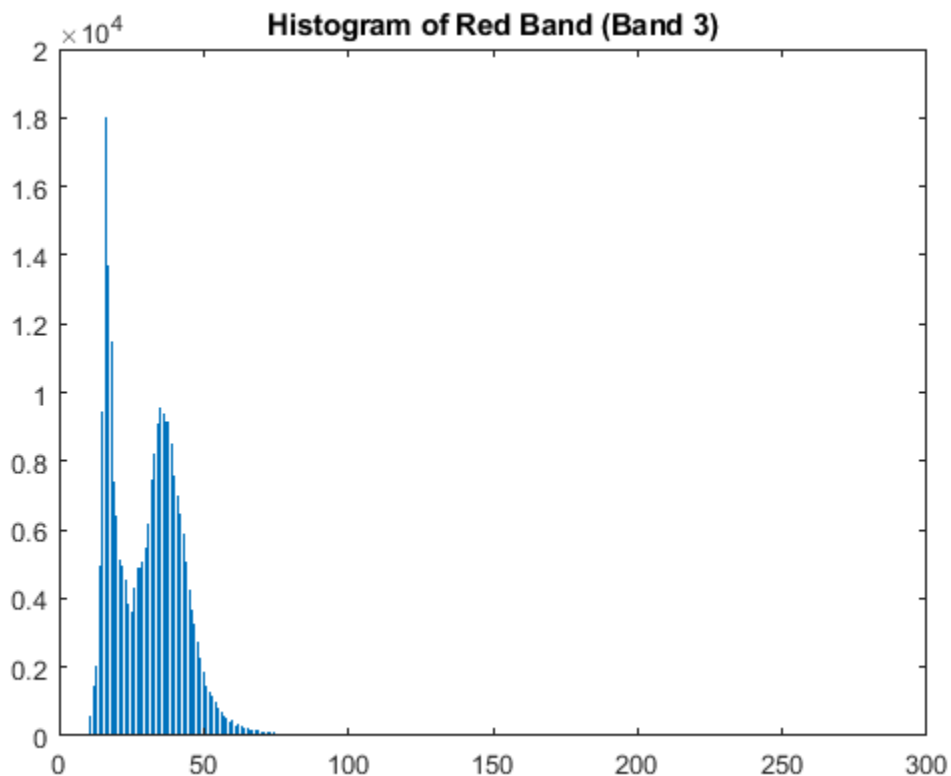

Step 5: Enhance the Truecolor Composite with a Contrast Stretch

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.

```
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;

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;
```

Compute the CDF of each histogram.

```
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 the range [0, 1].

```
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 the full RGB data.

```
input_adapter.SelectedBands = [3 2 1];
truecolor_enhanced = blockproc(input_adapter,[100 100],adjustFcn);
```

Display the result. 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.

```
imshow(truecolor_enhanced)
title('Truecolor Composite with Corrected Contrast Stretch')
```

Truecolor Composite with Corrected Contrast Stretch

See Also

Classes
ImageAdapter

Functions
blockproc | imadjust | imhist | stretchlim

Related Examples

- "Enhance Multispectral Color Composite Images" on page 11-93
- "Block Processing Large Images" on page 17-23

More About

- "Distinct Block Processing" on page 17-6
- "Perform Block Processing on Image Files in Unsupported Formats" on page 17-14
- "Create Function Handle"

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

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.

Remove Gaussian Noise 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 18-5.
- 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.
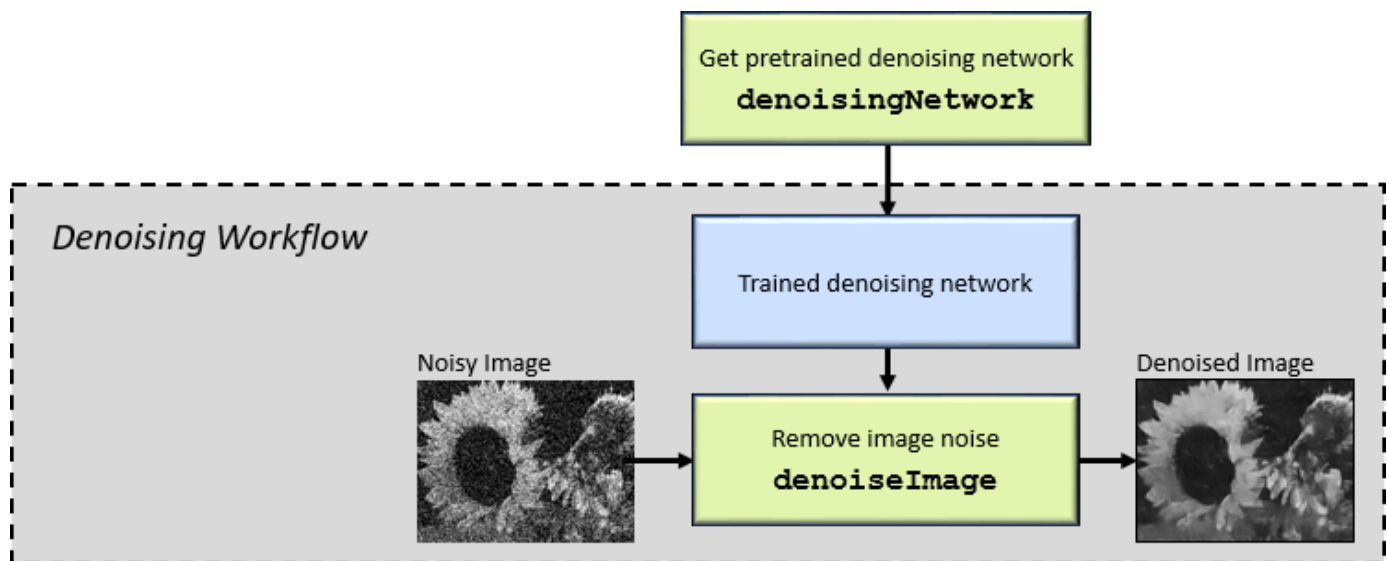
Train a Denoising Network Using Built-In Layers

You can train a network to detect a larger range of Gaussian noise standard deviations from grayscale images, starting with built-in 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. 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.
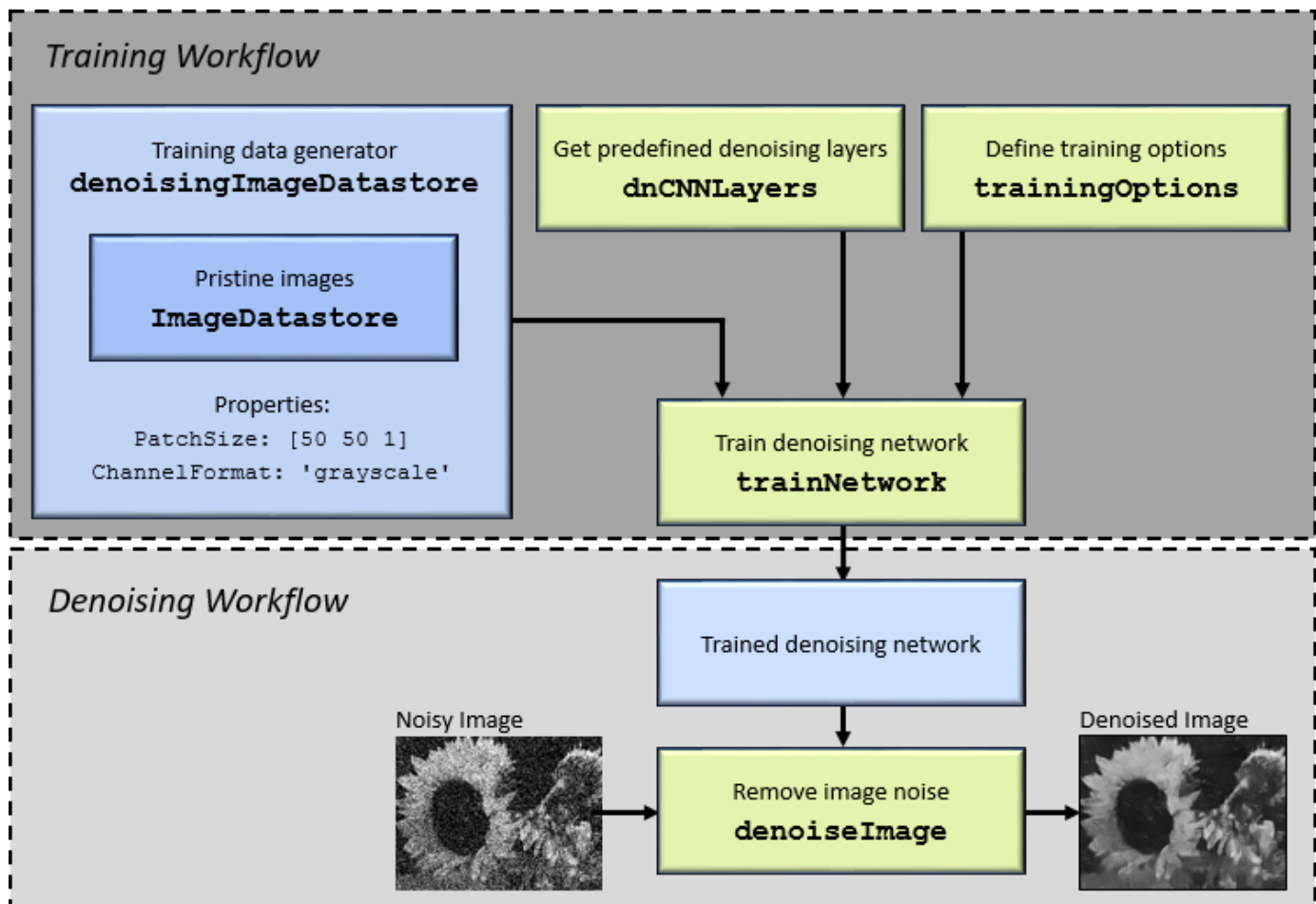
Train Fully Customized Denoising Neural Network

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 larger variety of noise, such as non-Gaussian noise distributions, in single-channel images. You can define the network architecture by using the layers returned by the `dnCNNLayers` function. To generate training images compatible with this network, use the `transform` and `combine` functions to batches of noisy images and the corresponding noise signal. For more information, see "Preprocess Images for Deep Learning" (Deep Learning Toolbox).

  After you train a denoising network using the DnCNN network architecture, you can use the `denoiseImage` function to remove image noise.

  > **Tip** The DnCNN network can also detect high-frequency image artifacts caused by other types of distortion. For example, you can train the DnCNN network to increase image resolution or remove JPEG compression artifacts. The "JPEG Image Deblocking Using Deep Learning" on page 18-25 example shows how to train a DnCNN network to remove JPEG compression artifacts

- 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 input images.

  After you train a denoising network using a custom network architecture, 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` | `combine` | `denoiseImage` | `denoisingImageDatastore` | `denoisingNetwork` | `dnCNNLayers` | `trainNetwork` | `trainingOptions` | `transform`

Related Examples
- "Remove Noise from Color Image Using Pretrained Neural Network" on page 18-5
- "JPEG Image Deblocking Using Deep Learning" on page 18-25
- "Prepare Datastore for Image-to-Image Regression" (Deep Learning Toolbox)

More About
- "Preprocess Images for Deep Learning" (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')
```

Split the noisy RGB image into its individual color channels.

```
[noisyR,noisyG,noisyB] = imsplit(noisyRGB);
```

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 18-2

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™.

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 one low-resolution image can yield several possible high-resolution images.
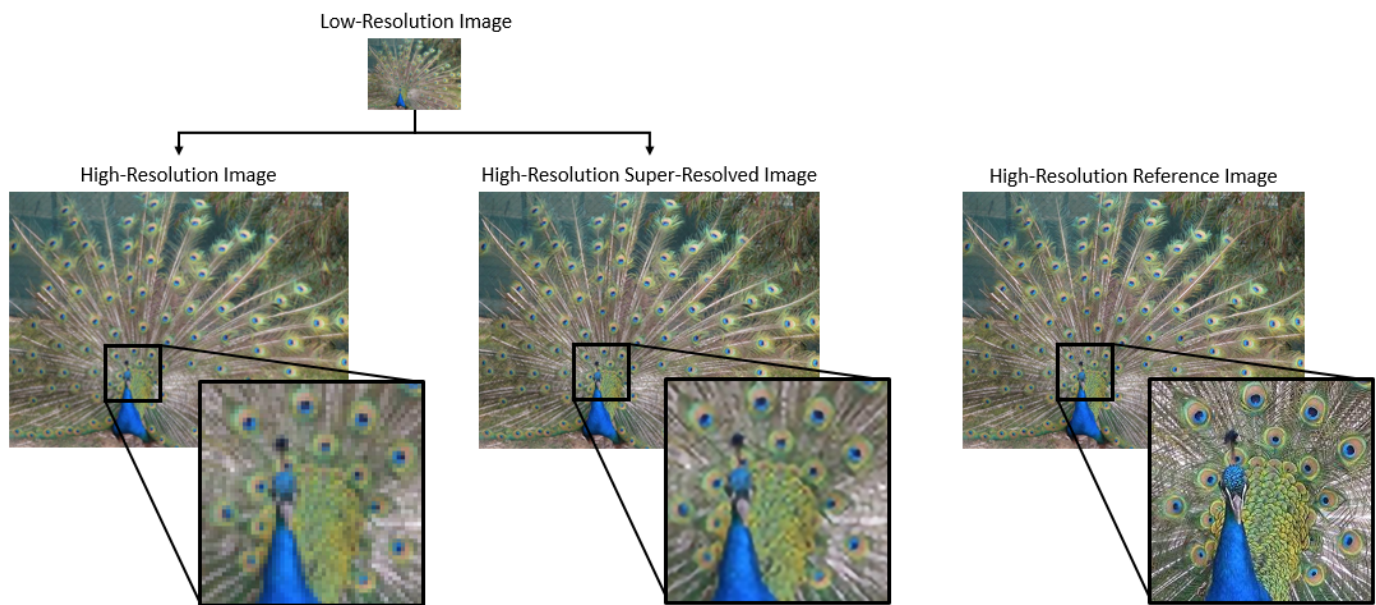
Several techniques, including deep learning algorithms, have been proposed to perform SISR. This example explores one deep learning algorithm for SISR, called very-deep super-resolution (VDSR) [1 on page 18-0   ].

The VDSR Network

VDSR is a convolutional neural network architecture designed to perform single image super-resolution [1 on page 18-0   ]. 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.
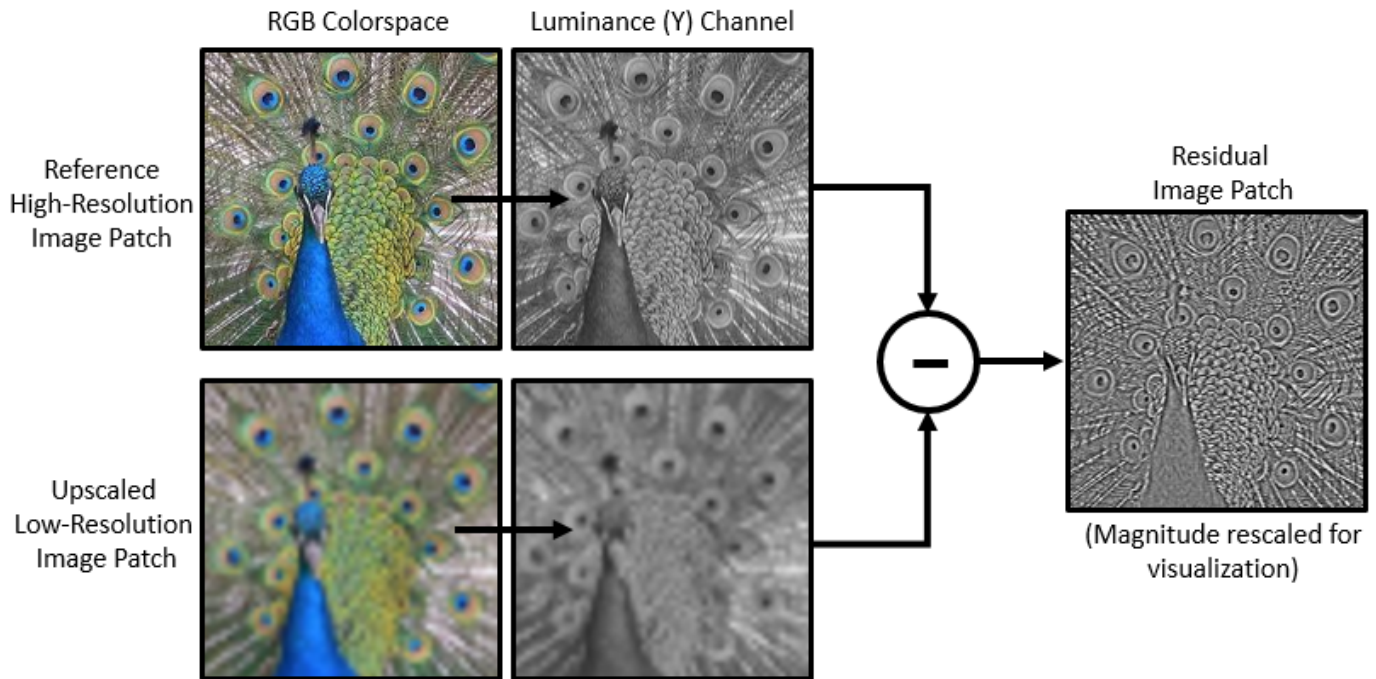
If Y_{highres} is the luminance of the high-resolution image and Y_{lowres} is the luminance a low-resolution image that has been upscaled using bicubic interpolation, then the input to the VDSR network is Y_{lowres} and the network learns to predict $Y_{\text{residual}} = Y_{\text{highres}} - Y_{\text{lowres}}$ 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 noninteger scale factors.

Download Training and Test Data

Download the IAPR TC-12 Benchmark, which consists of 20,000 still natural images [2 on page 18-0   ]. The data set includes photos of people, animals, cities, and more. The size of the data file is

~1.8 GB. If you do not want to download the training data set, 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 on page 18-0 section in this example.

Use the helper function, `downloadIAPRTC12Data`, to download the data. This function is attached to the example as a supporting file.

```
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'};
pristineImages = imageDatastore(trainImagesDir,'FileExtensions',exts);
```

List the number of training images.

```
numel(pristineImages.Files)
```

```
ans = 616
```

Prepare Training Data

To create a training data set, generate pairs of images consisting of upsampled images and the corresponding residual images.

The upsampled images are stored on disk as MAT files in the directory `upsampledDirName`. The computed residual images representing the network responses are stored on disk as MAT files in the directory `residualDirName`. The MAT files are stored as data type `double` for greater precision when training the network.

```
upsampledDirName = [trainImagesDir filesep 'upsampledImages'];
residualDirName = [trainImagesDir filesep 'residualImages'];
```

Use the helper function `createVDSRTrainingSet` to preprocess the training data. This function is attached to the example as a supporting file.

The helper function performs these operations for each pristine image in `trainImages`:

- Convert the image to the YCbCr color space
- Downsize the luminance (Y) channel by different scale factors to create sample low-resolution images, then resize the images to the original size using bicubic interpolation
- Calculate the difference between the pristine and resized images.
- Save the resized and residual images to disk.

```
scaleFactors = [2 3 4];
createVDSRTrainingSet(pristineImages,scaleFactors,upsampledDirName,residualDirName);
```

Define Preprocessing Pipeline for Training Set

In this example, the network inputs are low-resolution images that have been upsampled using bicubic interpolation. The desired network responses are the residual images. 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. This function is attached to the example as a supporting file.

```
upsampledImages = imageDatastore(upsampledDirName,'FileExtensions','.mat','ReadFcn',@matRead);
residualImages = imageDatastore(residualDirName,'FileExtensions','.mat','ReadFcn',@matRead);
```

Create an `imageDataAugmenter` (Deep Learning Toolbox) 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 a `randomPatchExtractionDatastore` that performs randomized patch extraction from the upsampled and residual image datastores. Patch extraction is the process of extracting a large set of small image patches, or tiles, from a single larger image. This type of data augmentation is frequently used in image-to-image regression problems, where many network architectures can be trained on very small input image sizes. This means that a large number of patches can be extracted from each full-sized image in the original training set, which greatly increases the size of the training set.

```
patchSize = [41 41];
patchesPerImage = 64;
dsTrain = randomPatchExtractionDatastore(upsampledImages,residualImages,patchSize, ...
    "DataAugmentation",augmenter,"PatchesPerImage",patchesPerImage);
```

The resulting datastore, `dsTrain`, provides mini-batches of data to the network at each iteration of the epoch. Preview the result of reading from the datastore.

```
inputBatch = preview(dsTrain);
disp(inputBatch)
```

```
    InputImage        ResponseImage
  _____      _____

  {41×41 double}      {41×41 double}
  {41×41 double}      {41×41 double}
  {41×41 double}      {41×41 double}
  {41×41 double}      {41×41 double}
  {41×41 double}      {41×41 double}
  {41×41 double}      {41×41 double}
  {41×41 double}      {41×41 double}
  {41×41 double}      {41×41 double}
```

Set Up VDSR Layers

This example defines the VDSR network using 41 individual layers from Deep Learning Toolbox™, including:

- `imageInputLayer` (Deep Learning Toolbox) - Image input layer
- `convolution2dLayer` (Deep Learning Toolbox) - 2-D convolution layer for convolutional neural networks
- `reluLayer` (Deep Learning Toolbox) - Rectified linear unit (ReLU) layer

- `regressionLayer` (Deep Learning Toolbox) - 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 the field can see all the high-level features in the image. In this case, for a network with D convolutional layers, the receptive field is $(2D+1)$-by-$(2D+1)$.

VDSR has 20 convolutional layers so 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 on page 18-0   ] 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.

```
convLayer = convolution2dLayer(3,64,'Padding',1, ...
    'WeightsInitializer','he','BiasInitializer','zeros','Name','Conv1');
```

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. As before, a ReLU layer follows every convolutional layer.

```
middleLayers = [convLayer relLayer];
for layerNumber = 2:networkDepth-1
    convLayer = convolution2dLayer(3,64,'Padding',[1 1], ...
        'WeightsInitializer','he','BiasInitializer','zeros', ...
        'Name',['Conv' num2str(layerNumber)]);

    relLayer = reluLayer('Name',['ReLU' num2str(layerNumber)]);
    middleLayers = [middleLayers convLayer relLayer];
end
```

The penultimate layer is a convolutional layer with a single filter of size 3-by-3-by-64 that reconstructs the image.

```
convLayer = convolution2dLayer(3,1,'Padding',[1 1], ...
    'WeightsInitializer','he','BiasInitializer','zeros', ...
    'NumChannels',64,'Name',['Conv' num2str(networkDepth)]);
```

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 = [convLayer regressionLayer('Name','FinalRegressionLayer')];
```

Concatenate all the layers to form the VDSR network.

```
layers = [firstLayer middleLayers finalLayers];
```

Alternatively, you can use the `vdsrLayers` helper function to create VDSR layers. This function is attached to the example as a supporting file.

```
layers = vdsrLayers;
```

Specify Training Options

Train the network using stochastic gradient descent with momentum (SGDM) optimization. Specify the hyperparameter settings for SGDM by using the `trainingOptions` (Deep Learning Toolbox) function. The learning rate is initially `0.1` and decreased by a factor of 10 every 10 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 specifying `'GradientThreshold'` as `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;
miniBatchSize = 64;
options = trainingOptions('sgdm', ...
    'Momentum',0.9, ...
    'InitialLearnRate',initLearningRate, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',10, ...
    'LearnRateDropFactor',learningRateFactor, ...
    'L2Regularization',l2reg, ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThresholdMethod','l2norm', ...
    'GradientThreshold',0.01, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the Network

After configuring the training options and the random patch extraction datastore, train the VDSR network using the `trainNetwork` (Deep Learning Toolbox) 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(dsTrain,layers,options);
    save(['trainedVDSR-' modelDateTime '-Epoch-' num2str(maxEpochs*epochIntervals) '-ScaleFactors
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','yellowlily.jpg',
    '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')
```

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')
```

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` (Deep Learning Toolbox) 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')
```


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')
```

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. The VDSR image has clearer details and sharper edges than the high-resolution image created using bicubic interpolation.

```
montage({imcrop(Ibicubic,roi),imcrop(Ivdsr,roi)})
title('High-Resolution Results Using Bicubic Interpolation (Left) vs. VDSR (Right)');
```

High-Resolution Results Using Bicubic Interpolation (Left) vs. VDSR (Right)

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 PSNR 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 = 39.2346
```

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 = 0.9874
```

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 = 5.1721
```

```
vdsrNIQE = niqe(Ivdsr)
```

```
vdsrNIQE = 4.7611
```

Calculate the average PSNR and SSIM of the entire set of test images for the scale factors 2, 3, and 4. For simplicity, you can use the helper function, `superResolutionMetrics`, to compute the average metrics. This function is attached to the example as a supporting file.

```
scaleFactors = [2 3 4];
superResolutionMetrics(net,testImages,scaleFactors);

Results for Scale factor 2

Average PSNR for Bicubic = 31.809683
Average PSNR for VDSR = 31.921784
Average SSIM for Bicubic = 0.938194
Average SSIM for VDSR = 0.949404

Results for Scale factor 3

Average PSNR for Bicubic = 28.170441
Average PSNR for VDSR = 28.563952
Average SSIM for Bicubic = 0.884381
Average SSIM for VDSR = 0.895830

Results for Scale factor 4

Average PSNR for Bicubic = 27.010839
Average PSNR for VDSR = 27.837260
Average SSIM for Bicubic = 0.861604
Average SSIM for VDSR = 0.877132
```

VDSR has better metric scores than bicubic interpolation for each scale factor.

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
`combine` | `randomPatchExtractionDatastore` | `rgb2ycbcr` | `trainNetwork` | `trainingOptions` | `transform` | `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™).

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 on page 18-0   ] 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.
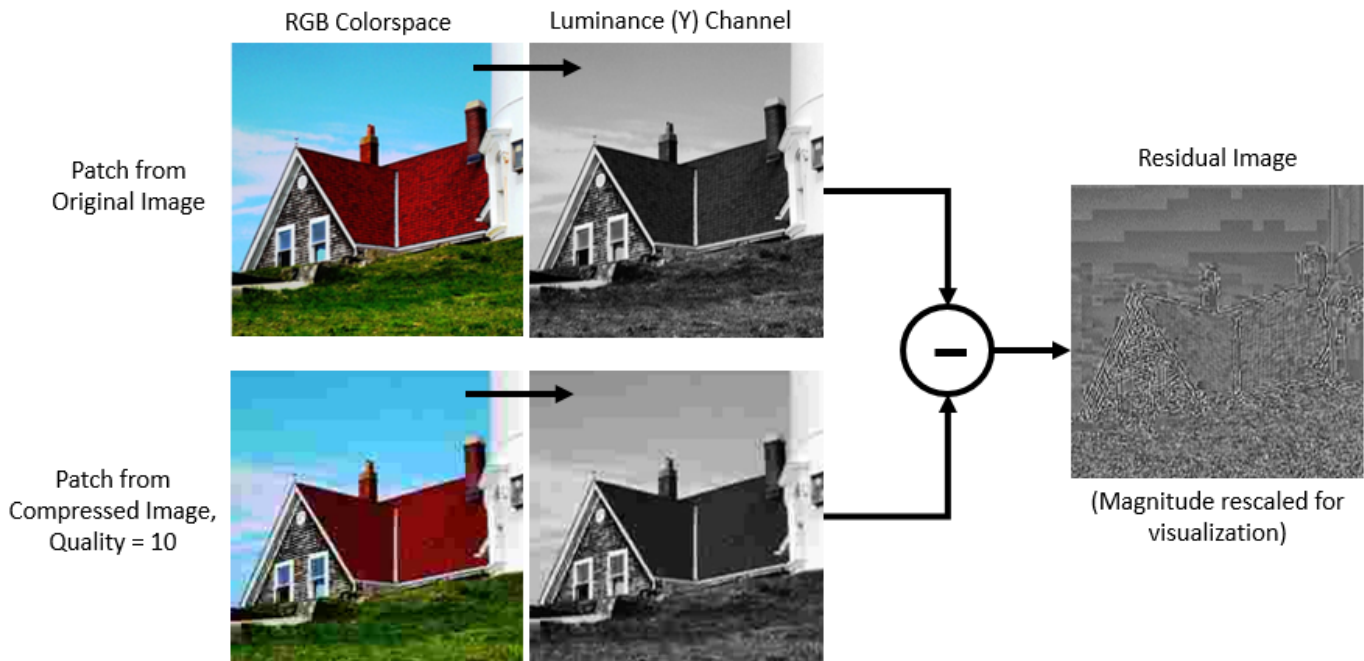
If Y_{Original} is the luminance of the pristine image and $Y_{\text{Compressed}}$ is the luminance of the image containing JPEG compression artifacts, then the input to the DnCNN network is $Y_{\text{Compressed}}$ and the network learns to predict $Y_{\text{Residual}} = Y_{\text{Compressed}} - Y_{\text{Original}}$ 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 the RGB color space.

Download Training Data

Download the IAPR TC-12 Benchmark, which consists of 20,000 still natural images [2 on page 18-0   ]. The data set includes photos of people, animals, cities, and more. The size of the data file is ~1.8 GB. If you do not want to download the training data set needed to 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 on page 18-0   section in this example.

Use the helper function, `downloadIAPRTC12Data`, to download the data. This function is attached to the example as a supporting file.

```
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'};
imdsPristine = imageDatastore(trainImagesDir,'FileExtensions',exts);
```

List the number of training images.

```
numel(imdsPristine.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.

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];
```

The compressed images are stored on disk as MAT files in the directory `compressedImagesDir`. The computed residual images are stored on disk as MAT files in the directory `residualImagesDir`. The MAT files are stored as data type `double` for greater precision when training the network.

```
compressedImagesDir = fullfile(imagesDir,'iaprtc12','JPEGDeblockingData','compressedImages');
residualImagesDir = fullfile(imagesDir,'iaprtc12','JPEGDeblockingData','residualImages');
```

Use the helper function `createJPEGDeblockingTrainingSet` to preprocess the training data. This function is attached to the example as a supporting file.

For each pristine training image, the helper function writes a copy of the image with quality factor 100 to use as a reference image and copies of the image with each quality factor to use as the network inputs. The function computes the luminance (Y) channel of the reference and compressed images in data type `double` for greater precision when calculating the residual images. The compressed images are stored on disk as .MAT files in the directory `compressedDirName`. The computed residual images are stored on disk as .MAT files in the directory `residualDirName`.

```
[compressedDirName,residualDirName] = createJPEGDeblockingTrainingSet(imdsPristine,JPEGQuality);
```

Create 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 compressed images. The desired network responses are the residual images. Create an image datastore called `imdsCompressed` from the collection of compressed image files. Create an image datastore called `imdsResidual` from the collection of

computed residual image files. Both datastores require a helper function, `matRead`, to read the image data from the image files. This function is attached to the example as a supporting file.

```matlab
imdsCompressed = imageDatastore(compressedDirName,'FileExtensions','.mat','ReadFcn',@matRead);
imdsResidual = imageDatastore(residualDirName,'FileExtensions','.mat','ReadFcn',@matRead);
```

Create an `imageDataAugmenter` (Deep Learning Toolbox) 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.

```matlab
augmenter = imageDataAugmenter( ...
    'RandRotation',@()randi([0,1],1)*90, ...
    'RandXReflection',true);
```

Create the `randomPatchExtractionDatastore` from the two image datastores. Specify a patch size of 50-by-50 pixels. Each image generates 128 random patches of size 50-by-50 pixels. Specify a mini-batch size of 128.

```matlab
patchSize = 50;
patchesPerImage = 128;
dsTrain = randomPatchExtractionDatastore(imdsCompressed,imdsResidual,patchSize, ...
    'PatchesPerImage',patchesPerImage, ...
    'DataAugmentation',augmenter);
dsTrain.MiniBatchSize = patchesPerImage;
```

The random patch extraction datastore `dsTrain` provides mini-batches of data to the network at iteration of the epoch. Preview the result of reading from the datastore.

```matlab
inputBatch = preview(dsTrain);
disp(inputBatch)
```

```
     InputImage        ResponseImage
    _____     _____

    {50×50 double}     {50×50 double}
    {50×50 double}     {50×50 double}
    {50×50 double}     {50×50 double}
    {50×50 double}     {50×50 double}
    {50×50 double}     {50×50 double}
    {50×50 double}     {50×50 double}
    {50×50 double}     {50×50 double}
    {50×50 double}     {50×50 double}
```

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.

```matlab
layers = dnCNNLayers
```

```
layers =
  1x59 Layer array with layers:

     1   'InputLayer'          Image Input          50x50x1 images
     2   'Conv1'               Convolution          64 3x3x1 convolutions with stride [1  1]
     3   'ReLU1'               ReLU                 ReLU
     4   'Conv2'               Convolution          64 3x3x64 convolutions with stride [1  1]
```

```
 5   'BNorm2'                Batch Normalization   Batch normalization with 64 channels
 6   'ReLU2'                 ReLU                  ReLU
 7   'Conv3'                 Convolution           64 3x3x64 convolutions with stride [1  1]
 8   'BNorm3'                Batch Normalization   Batch normalization with 64 channels
 9   'ReLU3'                 ReLU                  ReLU
10   'Conv4'                 Convolution           64 3x3x64 convolutions with stride [1  1]
11   'BNorm4'                Batch Normalization   Batch normalization with 64 channels
12   'ReLU4'                 ReLU                  ReLU
13   'Conv5'                 Convolution           64 3x3x64 convolutions with stride [1  1]
14   'BNorm5'                Batch Normalization   Batch normalization with 64 channels
15   'ReLU5'                 ReLU                  ReLU
16   'Conv6'                 Convolution           64 3x3x64 convolutions with stride [1  1]
17   'BNorm6'                Batch Normalization   Batch normalization with 64 channels
18   'ReLU6'                 ReLU                  ReLU
19   'Conv7'                 Convolution           64 3x3x64 convolutions with stride [1  1]
20   'BNorm7'                Batch Normalization   Batch normalization with 64 channels
21   'ReLU7'                 ReLU                  ReLU
22   'Conv8'                 Convolution           64 3x3x64 convolutions with stride [1  1]
23   'BNorm8'                Batch Normalization   Batch normalization with 64 channels
24   'ReLU8'                 ReLU                  ReLU
25   'Conv9'                 Convolution           64 3x3x64 convolutions with stride [1  1]
26   'BNorm9'                Batch Normalization   Batch normalization with 64 channels
27   'ReLU9'                 ReLU                  ReLU
28   'Conv10'                Convolution           64 3x3x64 convolutions with stride [1  1]
29   'BNorm10'               Batch Normalization   Batch normalization with 64 channels
30   'ReLU10'                ReLU                  ReLU
31   'Conv11'                Convolution           64 3x3x64 convolutions with stride [1  1]
32   'BNorm11'               Batch Normalization   Batch normalization with 64 channels
33   'ReLU11'                ReLU                  ReLU
34   'Conv12'                Convolution           64 3x3x64 convolutions with stride [1  1]
35   'BNorm12'               Batch Normalization   Batch normalization with 64 channels
36   'ReLU12'                ReLU                  ReLU
37   'Conv13'                Convolution           64 3x3x64 convolutions with stride [1  1]
38   'BNorm13'               Batch Normalization   Batch normalization with 64 channels
39   'ReLU13'                ReLU                  ReLU
40   'Conv14'                Convolution           64 3x3x64 convolutions with stride [1  1]
41   'BNorm14'               Batch Normalization   Batch normalization with 64 channels
42   'ReLU14'                ReLU                  ReLU
43   'Conv15'                Convolution           64 3x3x64 convolutions with stride [1  1]
44   'BNorm15'               Batch Normalization   Batch normalization with 64 channels
45   'ReLU15'                ReLU                  ReLU
46   'Conv16'                Convolution           64 3x3x64 convolutions with stride [1  1]
47   'BNorm16'               Batch Normalization   Batch normalization with 64 channels
48   'ReLU16'                ReLU                  ReLU
49   'Conv17'                Convolution           64 3x3x64 convolutions with stride [1  1]
50   'BNorm17'               Batch Normalization   Batch normalization with 64 channels
51   'ReLU17'                ReLU                  ReLU
52   'Conv18'                Convolution           64 3x3x64 convolutions with stride [1  1]
53   'BNorm18'               Batch Normalization   Batch normalization with 64 channels
54   'ReLU18'                ReLU                  ReLU
55   'Conv19'                Convolution           64 3x3x64 convolutions with stride [1  1]
56   'BNorm19'               Batch Normalization   Batch normalization with 64 channels
57   'ReLU19'                ReLU                  ReLU
58   'Conv20'                Convolution           1 3x3x64 convolutions with stride [1  1]
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 SGDM by using the `trainingOptions` (Deep Learning Toolbox) 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 `0.005`, and specify `'GradientThresholdMethod'` to use the absolute value of the gradients.

```
maxEpochs = 30;
initLearningRate = 0.1;
l2reg = 0.0001;
batchSize = 64;

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', ...
    'Verbose',false);
```

Train the Network

After configuring the training options and the random patch extraction datastore, train the DnCNN network using the `trainNetwork` (Deep Learning Toolbox) 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™ Titan X and can take even longer depending on your GPU hardware.

```
% Training runs when doTraining is true
doTraining = false;
if doTraining
    modelDateTime = datestr(now,'dd-mmm-yyyy-HH-MM-SS');
    [net,info] = trainNetwork(dsTrain,layers,options);
    save(['trainedJPEGDnCNN-' modelDateTime '-Epoch-' num2str(maxEpochs) '.mat'],'net','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','yellowlily.jpg',..
    '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);
```

18-33

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)')
```


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)')
```


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 Right)')
```

Patches from JPEG-Compressed Images with Quality Factor 50, 20 and 10 (Left to Right)

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)')
```

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. This function is attached to the example as a supporting file.

- 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 PSNR 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: 3.9478
I20: 4.5158    I20_predicted: 3.0685
I50: 2.8874    I50_predicted: 2.4106
NOTE: Smaller NIQE score signifies better perceptual quality
-------------------------------------------
BRISQUE Comparison
==================
I10: 52.372    I10_predicted: 38.9271
I20: 45.3772    I20_predicted: 30.8991
I50: 27.7093    I50_predicted: 24.3845
NOTE: Smaller BRISQUE score signifies better perceptual quality
```

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 | randomPatchExtractionDatastore | rgb2ycbcr | trainNetwork | trainingOptions | ycbcr2rgb

More About
- "Preprocess Images for Deep Learning" (Deep Learning Toolbox)
- "Datastores for Deep Learning" (Deep Learning Toolbox)
- "List of Deep Learning Layers" (Deep Learning Toolbox)

Image Processing Operator Approximation Using Deep Learning

This example shows how to approximate an image filtering operation using a multiscale context aggregation network (CAN).

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™.

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 on page 18-0   ] 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.

Noisy Input Image   Output from Bilateral Filtering   Output Using Operator Approximation Network
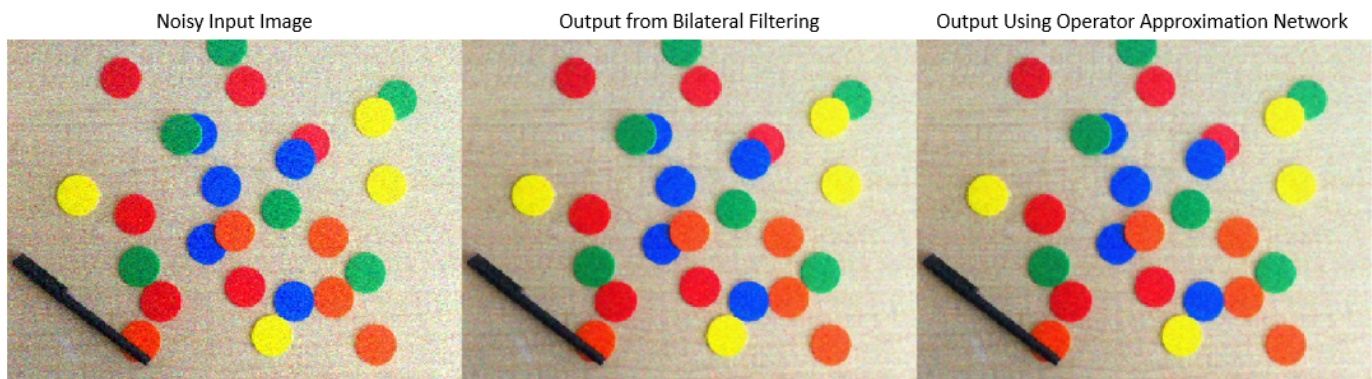
The Operator Approximation Network

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.
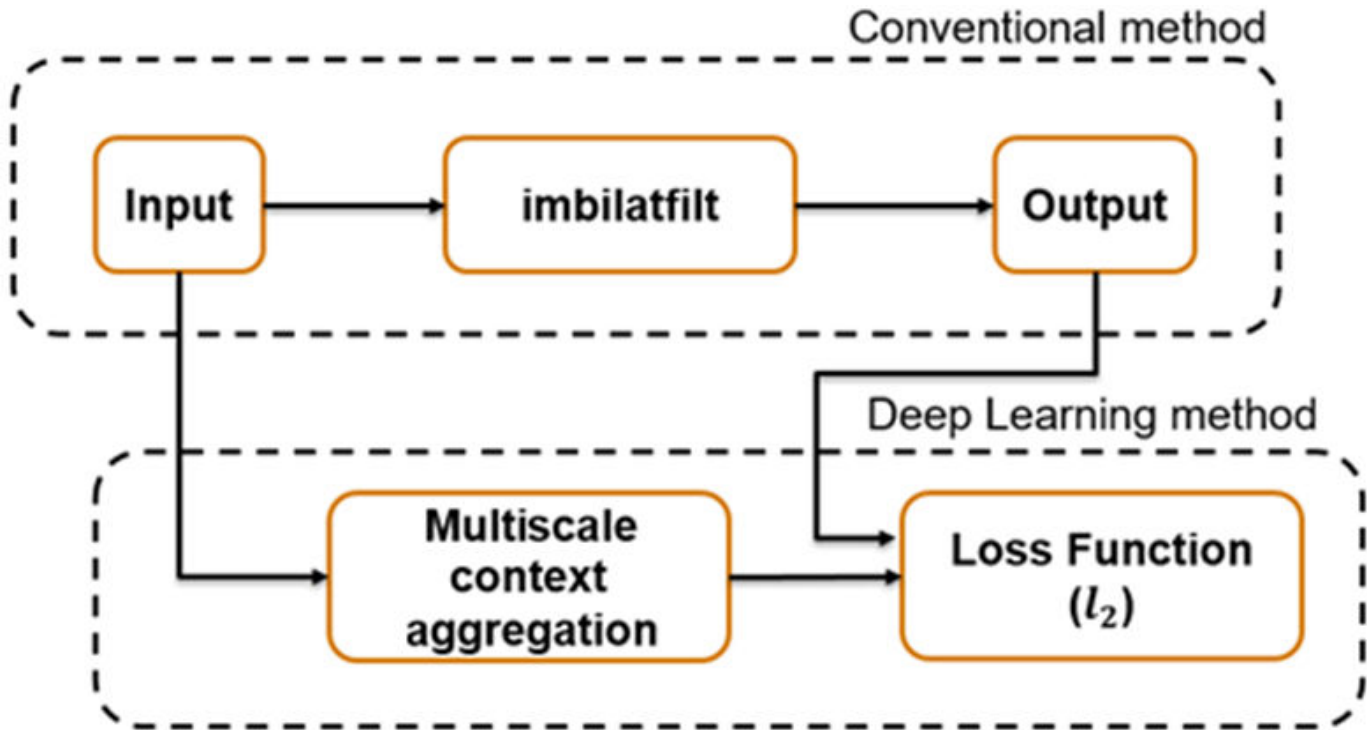
To help the network learn global image properties, the multiscale CAN architecture has a large receptive field. The first and last layers have the same size because the operator should not change the size of the image. Successive intermediate layers are dilated by exponentially increasing scale factors (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 on page 18-0]. The data set includes photos of people, animals, cities, and more. The size of the data file is ~1.8 GB. If you do not want to download the training data set needed to 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 on page 18-0 section in this example.

```
imagesDir = tempdir;
url_1 = 'http://www-i6.informatik.rwth-aachen.de/imageclef/resources/iaprtc12.tgz';
downloadIAPRTC12Data(url_1,imagesDir);
```

This example trains the network with small subset of the IAPRTC-12 Benchmark data.

```
trainImagesDir = fullfile(imagesDir,'iaprtc12','images','39');
exts = {'.jpg','.bmp','.png'};
pristineImages = imageDatastore(trainImagesDir,'FileExtensions',exts);
```

List the number of training images.

```
numel(pristineImages.Files)
```

```
ans = 916
```

Prepare Training Data

To create a training data set, read in pristine images and write out images that have been bilateral filtered. The filtered images are stored on disk in the directory specified by `preprocessDataDir`.

```
preprocessDataDir = [trainImagesDir filesep 'preprocessedDataset'];
```

Use the helper function `bilateralFilterDataset` to preprocess the training data. This function is attached to the example as a supporting file.

The helper function performs these operations for each pristine image in `inputImages`:

- Calculate the degree of smoothing for bilateral filtering. Smoothing the filtered image reduces image noise.
- Perform bilateral filtering using `imbilatfilt`.
- Save the filtered image to disk using `imwrite`.

```
bilateralFilterDataset(pristineImages,preprocessDataDir);
```

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 pristine images in `pristineImages`. The desired network responses are the processed images after bilateral filtering. Create an image datastore called `bilatFilteredImages` from the collection of bilateral filtered image files.

```
bilatFilteredImages = imageDatastore(preprocessDataDir,'FileExtensions',exts);
```

Create a `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];
dsTrain = randomPatchExtractionDatastore(pristineImages,bilatFilteredImages,patchSize, ....
    'PatchesPerImage',1);
dsTrain.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(dsTrain);
disp(inputBatch)
```

InputImage	ResponseImage
{256×256×3 uint8}	{256×256×3 uint8}

Set Up Multiscale CAN Layers

This example defines the multiscale CAN using layers from Deep Learning Toolbox™, including:

- `imageInputLayer` (Deep Learning Toolbox) - Image input layer
- `convolution2dLayer` (Deep Learning Toolbox) - 2D convolution layer for convolutional neural networks
- `batchNormalizationLayer` (Deep Learning Toolbox) - Batch normalization layer
- `leakyReluLayer` (Deep Learning Toolbox) - Leaky rectified linear unit layer
- `regressionLayer` (Deep Learning Toolbox) - Regression output layer for a neural network

Two custom scale layers are added to implement an adaptive batch normalization layer. 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 3-by-3. 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.

```
Wgts = zeros(3,3,3,numberOfFilters);
for ii = 1:3
    Wgts(2,2,ii,ii) = 1;
end
convolutionLayer = convolution2dLayer(3,numberOfFilters,'Padding',1, ...
    'Weights',Wgts,'Name','Conv1');
```

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,'DilationFactor',dilationFactor, ...
        'Weights',Wgts,'Name',['Conv' num2str(layerNumber)]);
    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 leakyrelLayer];
end
```

Do not apply a dilation factor to the second-to-last convolution layer.

```
conv2dLayer = convolution2dLayer(3,numberOfFilters, ...
    'Padding',1,'Weights',Wgts,'Name','Conv9');

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 1-by-1-by-32-by-3 that reconstructs the image.

```
Wgts = sqrt(2/(9*numberOfFilters))*randn(1,1,numberOfFilters,3);
conv2dLayer = convolution2dLayer(1,3,'NumChannels',numberOfFilters, ...
    'Weights',Wgts,'Name','Conv10');
```

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');
```

Plot the layer graph.

```
plot(lgraph)
```
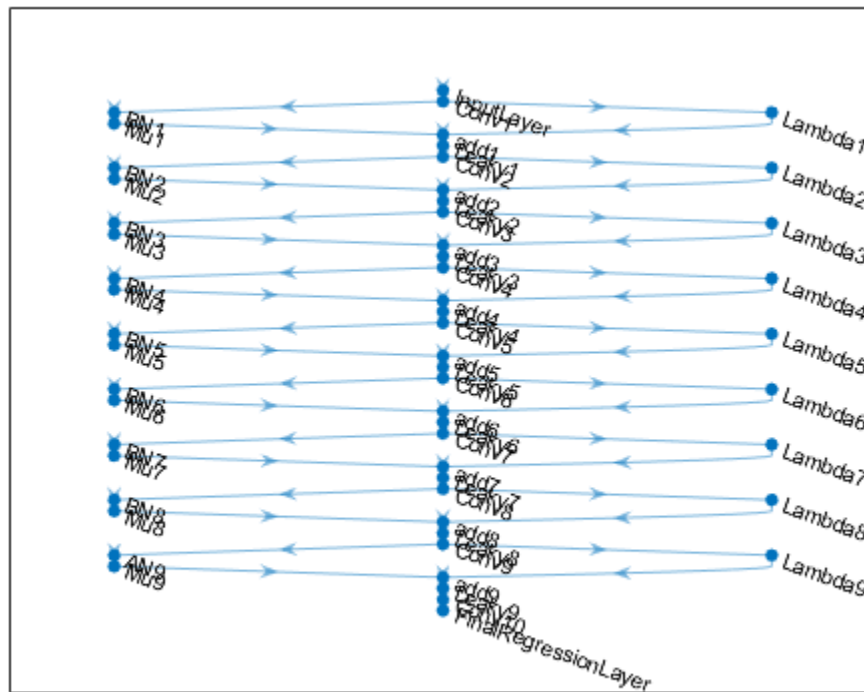
Specify Training Options

Train the network using the Adam optimizer. Specify the hyperparameter settings by using the `trainingOptions` (Deep Learning Toolbox) 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, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the Network

Now that the data source and training options are configured, train the multiscale CAN using the `trainNetwork` (Deep Learning Toolbox) `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(dsTrain,lgraph,options);
    save(['trainedOperatorLearning-' modelDateTime '-Epoch-' num2str(maxEpochs) '.mat'],'net');
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','yellowlily.jpg',..
    '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)
```
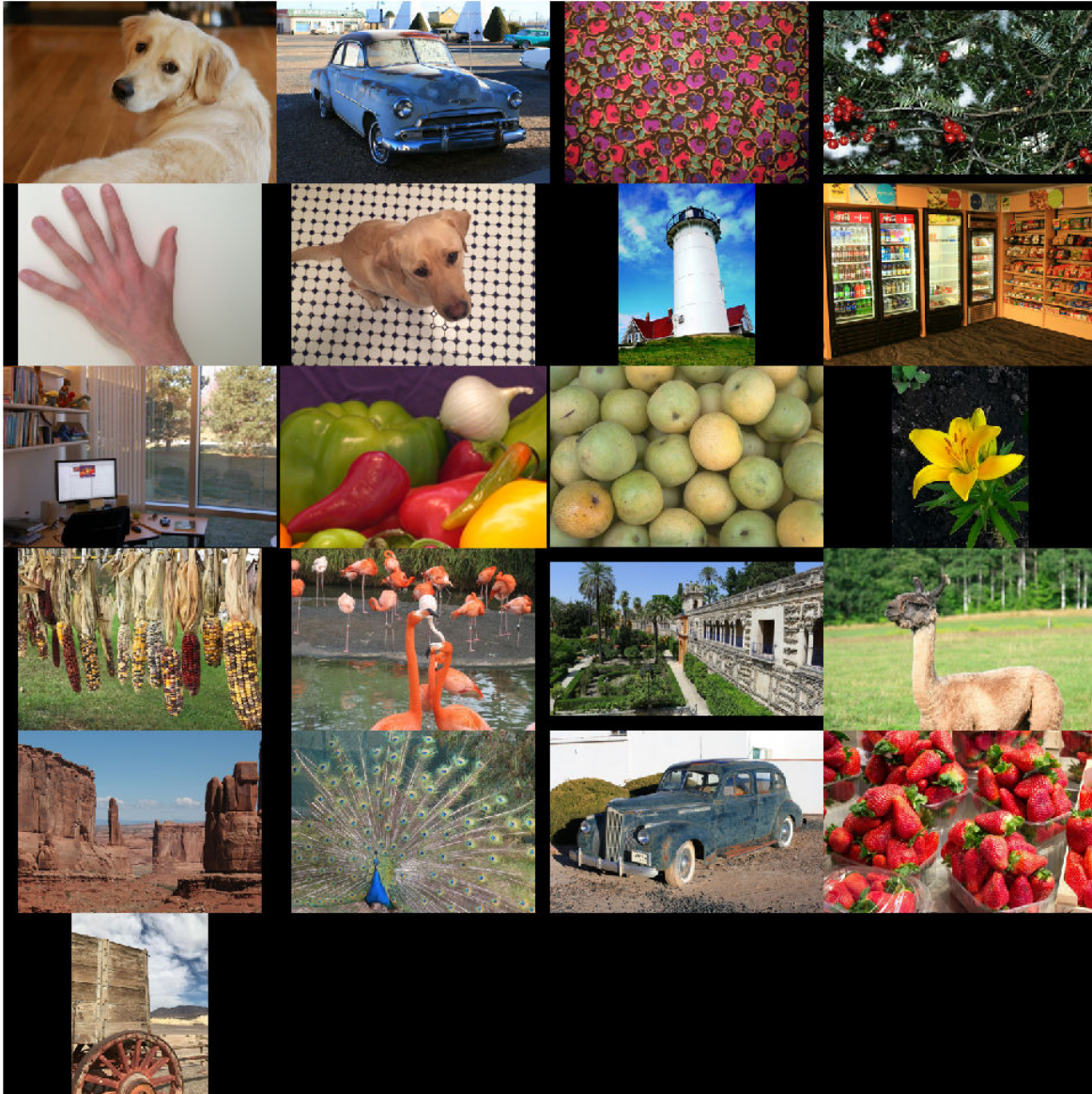
Select one of the images to use as the reference image for bilateral filtering. Convert the image to data type `uint8`.

```
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 256-by-256. If the test image is smaller than 256-by-256, 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')
```

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')
```

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')
```

Denoised Image Obtained Using Bilateral Filtering

Process Image Using Trained Network

Pass the normalized input image through the trained network and observe the `activations` (Deep Learning Toolbox) 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')
```

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');
```

Reference Image | Noisy Image | Bilateral-Filtered Image | CAN Prediction
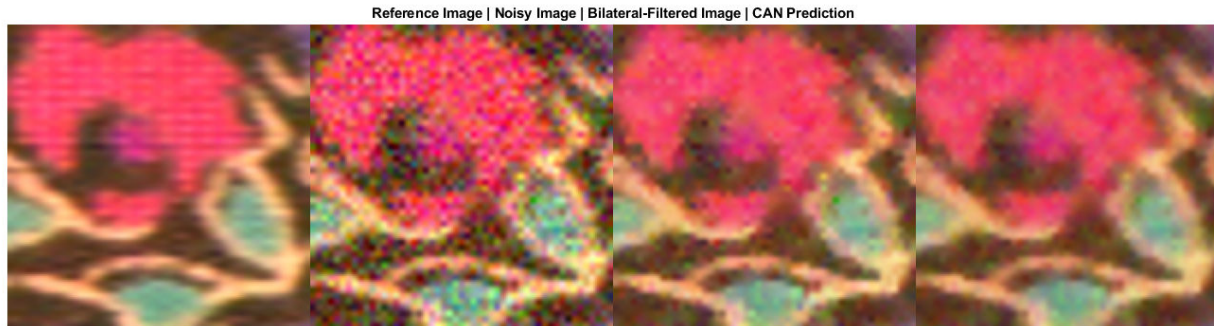
The CAN removes more noise than conventional bilateral filtering. Both techniques preserve edge sharpness.

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 PSNR 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 = 20.2857   25.

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 = 0.76251   0.915

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 Image = ', ...
    num2str([noisyNIQE bilatNIQE approxNIQE])])
```

NIQE score of: Noisy Image / Bilateral-Filtered Image / Operator Approximated Image = 12.1865

18-51

Compared to conventional bilateral filtering, the operator approximation produces better metric scores.

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)
- "Datastores for Deep Learning" (Deep Learning Toolbox)
- "List of Deep Learning Layers" (Deep Learning Toolbox)

Develop Raw Camera Processing Pipeline Using Deep Learning

This example shows how to convert raw camera data to an aesthetically pleasing color image using a U-Net.

The example shows how to train a RAW-to-RGB U-Net network and also provides a pretrained RAW-to-RGB 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 Parallel Computing Toolbox™.

Introduction

DSLRs and many modern phone cameras offer the ability to save data collected directly from the camera sensor as a RAW file. Each pixel of RAW data corresponds directly to the amount of light captured by a corresponding camera photosensor. The data depends on fixed characteristics of the camera hardware, such as the sensitivity to each photosensor to a particular range of wavelengths of the electromagnetic spectrum. The data also depends on camera acquisition settings such as exposure time, and factors of the scene such as the light source.

Demosaicing is the only required operation to convert single-channel RAW data to a three-channel RGB image. However, without additional image processing operations, the resulting RGB image has subjectively poor visual quality.

A traditional image processing pipeline performs a combination of additional operations including denoising, linearization, white-balancing, color correction, brightness adjustment, and contrast adjustment [1]. The challenge of designing a pipeline lies in refining algorithms to optimize the subjective appearance of the final RGB image regardless of variations in the scene and acquisition settings.
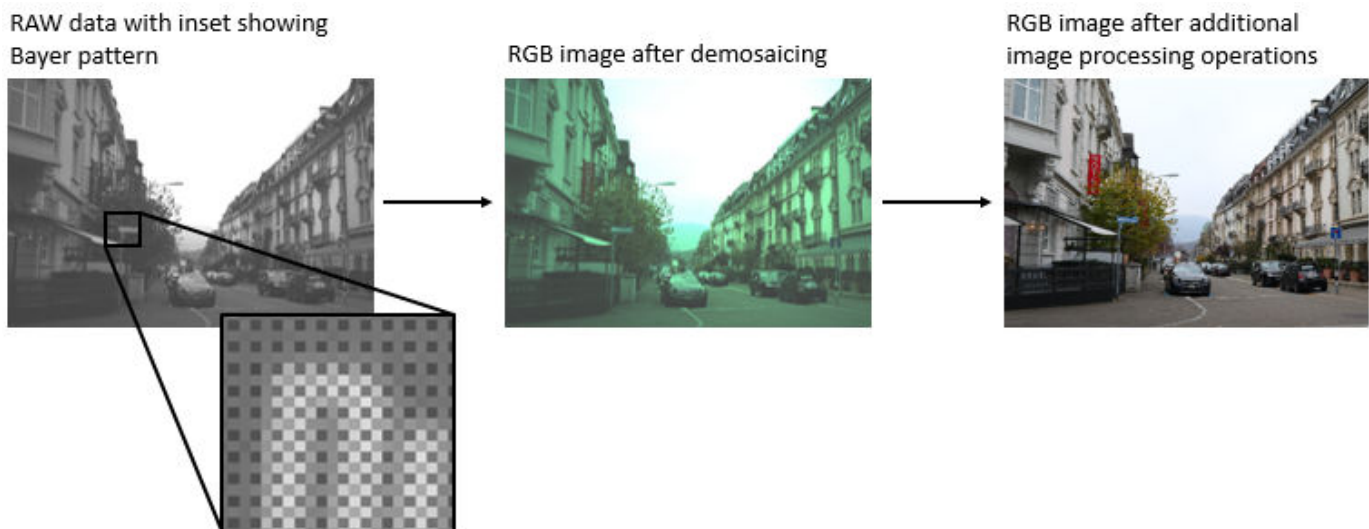
Deep learning techniques enable direct RAW to RGB conversion without the necessity of developing a traditional processing pipeline. For instance, one technique compensates for underexposure when converting RAW images to RGB [2]. This example shows how to convert RAW images from a lower end phone camera to RGB images that approximate the quality of a higher end DSLR camera [3].

Download Training, Validation, and Test Data

This example uses the Zurich RAW to RGB data set [3]. The size of the data set is 22 GB. The data set contains 48,043 spatially registered pairs of RAW and RGB training image patches of size 448-by-448. The data set contains two separate test sets. One test set consists of 1,204 spatially registered pairs of RAW and RGB image patches of size 448-by-448. The other test set consists of unregistered full-resolution RAW and RGB images.

Create a directory to store the data set.

```
imageDir = fullfile(tempdir,'ZurichRAWToRGB');
if ~exist(imageDir,'dir')
    mkdir(imageDir);
end
```

To download the data set, request access using the Zurich RAW to RGB dataset form. Extract the data into the directory specified by the `imageDir` variable. When extracted successfully, `imageDir` contains three directories named `full_resolution`, `test`, and `train`.

Create Datastores for Training, Validation, and Testing

RGB Image Patch Training Data

Create an `imageDatastore` that reads the target RGB training image patches acquired using a high-end Canon DSLR.

```
trainImageDir = fullfile(imageDir,'train');
dsTrainRGB = imageDatastore(fullfile(trainImageDir,'canon'),'ReadSize',16);
```

Preview an RGB training image patch.

```
groundTruthPatch = preview(dsTrainRGB);
imshow(groundTruthPatch)
```

RAW Image Patch Training Data

Create an `imageDatastore` that reads the input RAW training image patches acquired using a Huawei phone camera. The RAW images are captured with 10-bit precision and are represented as both 8-bit and 16-bit PNG files. The 8-bit files provide a compact representation of patches with data in the range [0, 255]. No scaling has been done on any of the RAW data.

```
dsTrainRAW = imageDatastore(fullfile(trainImageDir,'huawei_raw'),'ReadSize',16);
```

Preview an input RAW training image patch. The datastore reads this patch as an 8-bit `uint8` image because the sensor counts are in the range [0, 255]. To simulate the 10-bit dynamic range of the training data, divide the image intensity values by 4. If you zoom in on the image, then you can see the RGGB Bayer pattern.

```
inputPatch = preview(dsTrainRAW);
inputPatchRAW = inputPatch/4;
imshow(inputPatchRAW)
```
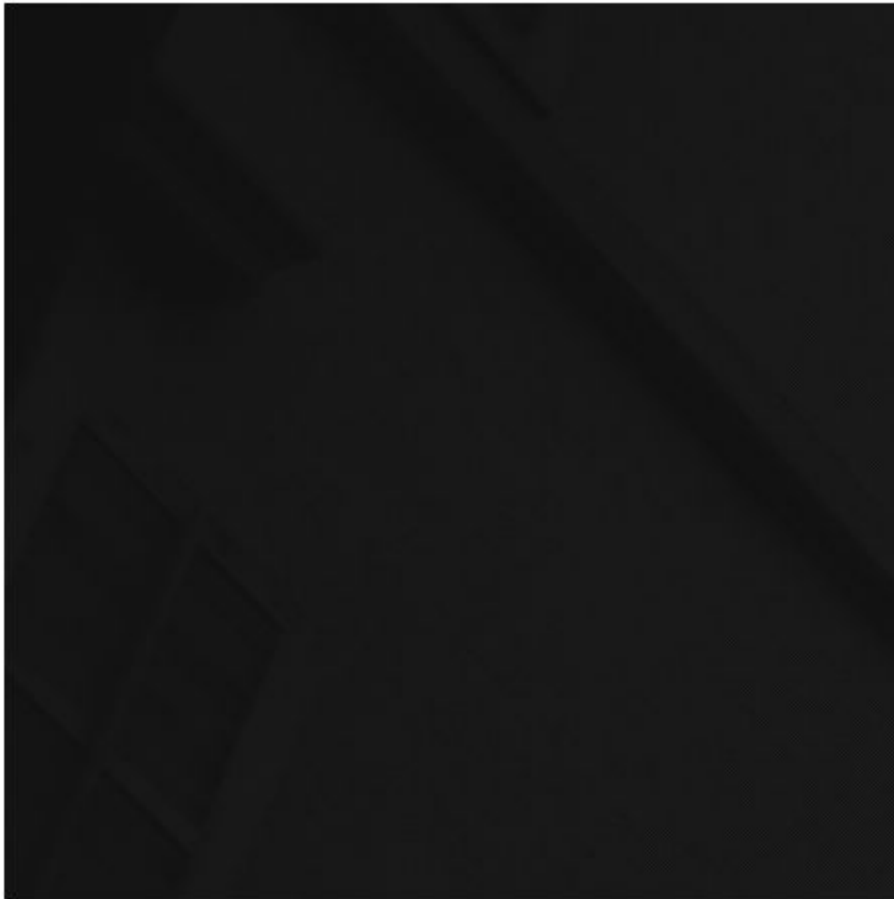
To simulate the minimal traditional processing pipeline, demosaic the RGGB Bayer pattern of the RAW data using the `demosaic` function. Display the processed image and brighten the display. Compared to the target RGB image, the minimally-processed RGB image is dark and has imbalanced colors and noticeable artifacts. A trained RAW-to-RGB network performs preprocessing operations so that the output RGB image resembles the target image.

```
inputPatchRGB = demosaic(inputPatch,'rggb');
imshow(rescale(inputPatchRGB))
```

Partition Test Images into Validation and Test Sets

The test data contains RAW and RGB image patches and full-sized images. This example partitions the test image patches into a validation set and test set. The example uses the full-sized test images for qualitative testing only. See Evaluate Trained Image Processing Pipeline on Full-Sized Images on page 18-0 .

Create image datastores that read the RAW and RGB test image patches.

```
testImageDir = fullfile(imageDir,'test');
dsTestRAW = imageDatastore(fullfile(testImageDir,'huawei_raw'),'ReadSize',16);
dsTestRGB = imageDatastore(fullfile(testImageDir,'canon'),'ReadSize',16);
```

Randomly split the test data into two sets for validation and training. The validation data set contains 200 images. The test set contains the remaining images.

```
numTestImages = dsTestRAW.numpartitions;
numValImages = 200;
```

```
testIdx = randperm(numTestImages);
validationIdx = testIdx(1:numValImages);
testIdx = testIdx(numValImages+1:numTestImages);

dsValRAW = subset(dsTestRAW,validationIdx);
dsValRGB = subset(dsTestRGB,validationIdx);

dsTestRAW = subset(dsTestRAW,testIdx);
dsTestRGB = subset(dsTestRGB,testIdx);
```

Preprocess and Augment Data

The sensor acquires color data in a repeating Bayer pattern that includes one red, two green, and one blue photosensor. Preprocess the data into a four-channel image expected of the network using the `transform` function. The `transform` function processes the data using the operations specified in the `preprocessRAWDataForRAWToRGB` helper function. The helper function is attached to the example as a supporting file.

The `preprocessRAWDataForRAWToRGB` helper function converts an *H*-by-*W*-by-1 RAW image to an *H*/2-by-*W*/2-by-4 multichannel image consisting of one red, two green, and one blue channel.

$$\begin{bmatrix} r_1 & g_5 & r_3 & g_7 \\ g_1 & b_1 & g_3 & b_3 \\ r_2 & g_6 & r_4 & g_8 \\ g_2 & b_2 & g_4 & b_4 \end{bmatrix} \longrightarrow \begin{bmatrix} b_1 & b_3 \\ b_2 & b_4 \end{bmatrix} \begin{bmatrix} g_5 & g_7 \\ g_6 & g_8 \end{bmatrix} \begin{bmatrix} r_1 & r_3 \\ r_2 & r_4 \end{bmatrix} \begin{bmatrix} g_1 & g_3 \\ g_2 & g_4 \end{bmatrix}$$

The function also casts the data to data type `single` scaled to the range [0, 1].

```
dsTrainRAW = transform(dsTrainRAW,@preprocessRAWDataForRAWToRGB);
dsValRAW = transform(dsValRAW,@preprocessRAWDataForRAWToRGB);
dsTestRAW = transform(dsTestRAW,@preprocessRAWDataForRAWToRGB);
```

The target RGB images are stored on disk as unsigned 8-bit data. To make the computation of metrics and the network design more convenient, preprocess the target RGB training images using the `transform` function and the `preprocessRGBDataForRAWToRGB` helper function. The helper function is attached to the example as a supporting file.

The `preprocessRGBDataForRAWToRGB` helper function casts images to data type `single` scaled to the range [0, 1].

```
dsTrainRGB = transform(dsTrainRGB,@preprocessRGBDataForRAWToRGB);
dsValRGB = transform(dsValRGB,@preprocessRGBDataForRAWToRGB);
```

Combine the input RAW and target RGB data for the training, validation, and test image sets by using the `combine` function.

```
dsTrain = combine(dsTrainRAW,dsTrainRGB);
dsVal = combine(dsValRAW,dsValRGB);
dsTest = combine(dsTestRAW,dsTestRGB);
```

Randomly augment the training data using the `transform` function and the `augmentDataForRAWToRGB` helper function. The helper function is attached to the example as a supporting file.

The `augmentDataForRAWToRGB` helper function randomly applies 90 degree rotation and horizontal reflection to pairs of input RAW and target RGB training images.

```
dsTrainAug = transform(dsTrain,@augmentDataForRAWToRGB);
```

Preview the augmented training data.

```
exampleAug = preview(dsTrainAug)
```

```
exampleAug=8×2 cell array
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
```

Display the network input and target image in a montage. The network input has four channels, so display the first channel rescaled to the range [0, 1]. The input RAW and target RGB images have identical augmentation.

```
exampleInput = exampleAug{1,1};
exampleOutput = exampleAug{1,2};
montage({rescale(exampleInput(:,:,1)),exampleOutput})
```
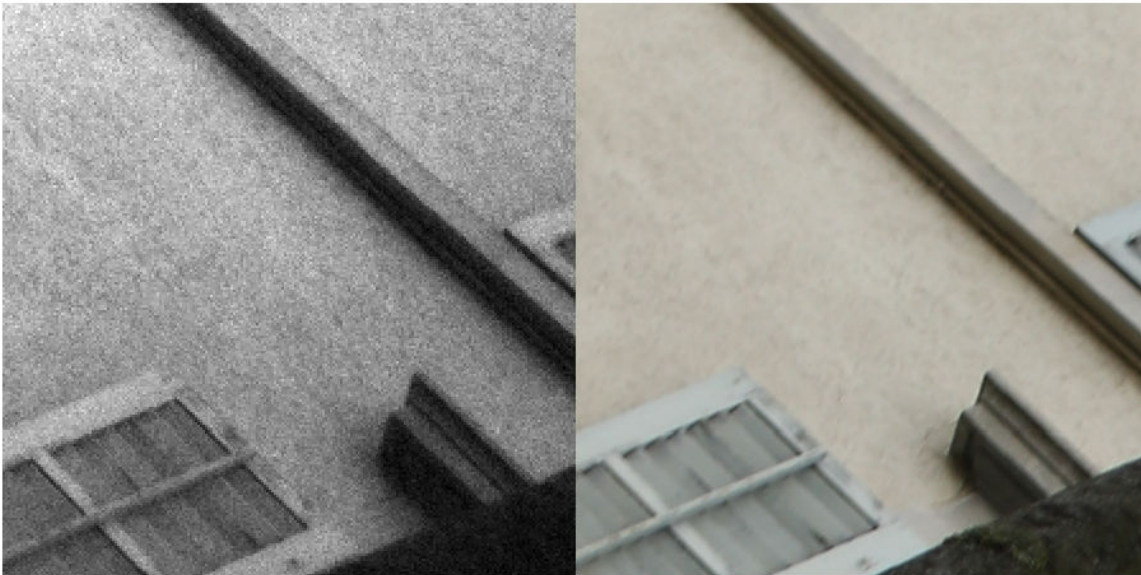

Batch Training and Validation Data During Training

This example uses a custom training loop. The `minibatchqueue` (Deep Learning Toolbox) object is useful for managing the mini-batching of observations in custom training loops. The

minibatchqueue object also casts data to a dlarray (Deep Learning Toolbox) object that enables auto differentiation in deep learning applications.

```
miniBatchSize = 12;
valBatchSize = 10;
trainingQueue = minibatchqueue(dsTrainAug,'MiniBatchSize',miniBatchSize,'PartialMiniBatch','disca
validationQueue = minibatchqueue(dsVal,'MiniBatchSize',valBatchSize,'MiniBatchFormat','SSCB');
```

The next (Deep Learning Toolbox) function of minibatchqueue yields the next mini-batch of data. Preview the outputs from one call to the next function. The outputs have data type dlarray. The data is already cast to gpuArray, on the GPU, and ready for training.

```
[inputRAW,targetRGB] = next(trainingQueue);
whos inputRAW
whos targetRGB
```

```
  Name             Size                    Bytes  Class      Attributes

  targetRGB      448x448x3x12            28901384  dlarray
```

Set Up 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. The name U-Net comes from the fact that the network can be drawn with a symmetric shape like the letter U.

This example uses a simple U-Net architecture with two modifications. First, the network replaces the final transposed convolution operation with a custom pixel shuffle upsampling (also known as a depth-to-space) operation. Second, the network uses a custom hyperbolic tangent activation layer as the final layer in the network.

Pixel Shuffle Upsampling

Convolution followed by pixel shuffle upsampling can define subpixel convolution for super resolution applications. Subpixel convolution prevents the checkboard artifacts that can arise from transposed convolution [6]. Because the model needs to map H/2-by-W/2-by-4 RAW inputs to W-by-H-by-3 RGB outputs, the final upsampling stage of the model can be thought of similarly to super resolution where the number of spatial samples grows from the input to the output.

The figure shows how pixel shuffling upsampling works for a 2-by-2-by-4 input. The first two dimensions are spatial dimensions and the 3rd dimension is a channel dimension. In general, pixel shuffle upsampling by a factor of S takes a H-by-W-by-C input and yields a S*H-by-S*W-by-$\frac{C}{S^2}$ output.

$$\begin{bmatrix} r_1 & r_3 \\ r_2 & r_4 \end{bmatrix} \begin{bmatrix} g_1 & g_3 \\ g_2 & g_4 \end{bmatrix} \begin{bmatrix} g_5 & g_7 \\ g_6 & g_8 \end{bmatrix} \begin{bmatrix} b_1 & b_3 \\ b_2 & b_4 \end{bmatrix} \longrightarrow \begin{bmatrix} r_1 & g_5 & r_3 & g_7 \\ g_1 & b_1 & g_3 & b_3 \\ r_2 & g_6 & r_4 & g_8 \\ g_2 & b_2 & g_4 & b_4 \end{bmatrix}$$
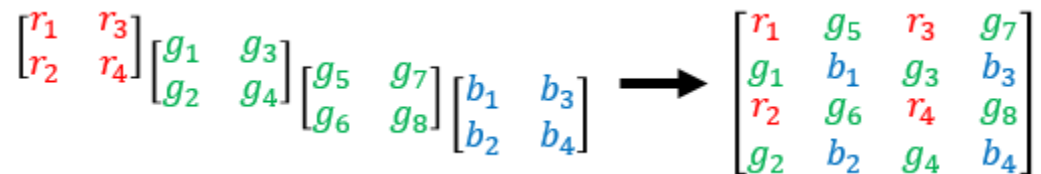
The pixel shuffle function grows the spatial dimensions of the output by mapping information from channel dimensions at a given spatial location into *S*-by-*S* spatial blocks in the output in which each channel contributes to a consistent spatial position relative to its neighbors during upsampling.

Scaled and Hyperbolic Tangent Activation

A hyperbolic tangent activation layer applies the `tanh` function on the layer inputs. This example uses a scaled and shfited version of the `tanh` function, which encourages but does not strictly enforce that the RGB network outputs are in the range [0, 1] [6].

$$f(x) = 0.58 * \tanh(x) + 0.5$$

Scaled and Shifted Hyperbolic Tangent
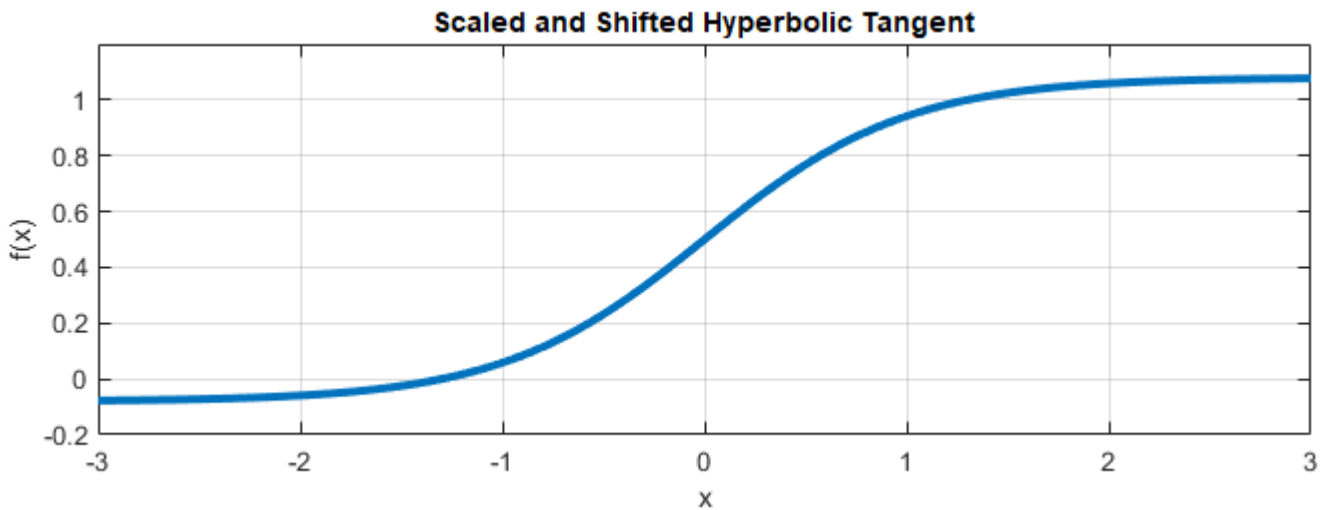
Calculate Training Set Statistics for Input Normalization

Use `tall` to compute per-channel mean reduction across the training data set. The input layer of the network performs mean centering of inputs during training and testing using the mean statistics.

```
dsIn = copy(dsTrainRAW);
dsIn.UnderlyingDatastore.ReadSize = 1;
t = tall(dsIn);
perChannelMean = gather(mean(t,[1 2]));
```

Create the U-Net

Create layers of the initial subnetwork, specifying the per-channel mean.

```
inputSize = [256 256 4];
initialLayer = imageInputLayer(inputSize,"Normalization","zerocenter","Mean",perChannelMean, ...
    "Name","ImageInputLayer");
```

Add layers of the first encoding subnetwork. The first encoder has 32 convolutional filters.

```
numEncoderStages = 4;
numFiltersFirstEncoder = 32;
encoderNamePrefix = "Encoder-Stage-";

encoderLayers = [
    convolution2dLayer([3 3],numFiltersFirstEncoder,"Padding","same","WeightsInitializer","narro
        "Name",strcat(encoderNamePrefix,"1-Conv-1"))
    leakyReluLayer(0.2,"Name",strcat(encoderNamePrefix,"1-ReLU-1"))
    convolution2dLayer([3 3],numFiltersFirstEncoder,"Padding","same","WeightsInitializer","narro
        "Name",strcat(encoderNamePrefix,"1-Conv-2"))
    leakyReluLayer(0.2,"Name",strcat(encoderNamePrefix,"1-ReLU-2"))
```

```
    maxPooling2dLayer([2 2],"Stride",[2 2], ...
        "Name",strcat(encoderNamePrefix,"1-MaxPool"));
    ];
```

Add layers of additional encoding subnetworks. These subnetworks add channel-wise instance normalization after each convolutional layer using a `groupNormalizationLayer`. Each encoder subnetwork has twice the number of filters as the previous encoder subnetwork.

```
cnIdx = 1;
for stage = 2:numEncoderStages

    numFilters = numFiltersFirstEncoder*2^(stage-1);
    layerNamePrefix = strcat(encoderNamePrefix,num2str(stage));

    encoderLayers = [
        encoderLayers
        convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal"
            "Name",strcat(layerNamePrefix,"-Conv-1"))
        groupNormalizationLayer("channel-wise","Name",strcat("cn",num2str(cnIdx)))
        leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-ReLU-1"))
        convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal"
            "Name",strcat(layerNamePrefix,"-Conv-2"))
        groupNormalizationLayer("channel-wise","Name",strcat("cn",num2str(cnIdx+1)))
        leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-ReLU-2"))
        maxPooling2dLayer([2 2],"Stride",[2 2],"Name",strcat(layerNamePrefix,"-MaxPool"))
        ];

    cnIdx = cnIdx + 2;
end
```

Add bridge layers. The bridge subnetwork has twice the number of filters as the final encoder subnetwork and first decoder subnetwork.

```
numFilters = numFiltersFirstEncoder*2^numEncoderStages;
bridgeLayers = [
    convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal", .
        "Name","Bridge-Conv-1")
    groupNormalizationLayer("channel-wise","Name","cn7")
    leakyReluLayer(0.2,"Name","Bridge-ReLU-1")
    convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal", .
        "Name","Bridge-Conv-2")
    groupNormalizationLayer("channel-wise","Name","cn8")
    leakyReluLayer(0.2,"Name","Bridge-ReLU-2")];
```

Add layers of the first three decoder subnetworks.

```
numDecoderStages = 4;
cnIdx = 9;
decoderNamePrefix = "Decoder-Stage-";

decoderLayers = [];
for stage = 1:numDecoderStages-1

    numFilters = numFiltersFirstEncoder*2^(numDecoderStages-stage);
    layerNamePrefix = strcat(decoderNamePrefix,num2str(stage));

    decoderLayers = [
        decoderLayers
```

```
        transposedConv2dLayer([3 3],numFilters,"Stride",[2 2],"Cropping","same","WeightsInitializ
            "Name",strcat(layerNamePrefix,"-UpConv"))
        leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-UpReLU"))
        depthConcatenationLayer(2,"Name",strcat(layerNamePrefix,"-DepthConcatenation"))
        convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal"
            "Name",strcat(layerNamePrefix,"-Conv-1"))
        groupNormalizationLayer("channel-wise","Name",strcat("cn",num2str(cnIdx)))
        leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-ReLU-1"))
        convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal"
            "Name",strcat(layerNamePrefix,"-Conv-2"))
        groupNormalizationLayer("channel-wise","Name",strcat("cn",num2str(cnIdx+1)))
        leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-ReLU-2"))
        ];

    cnIdx = cnIdx + 2;
end
```

Add layers of the last decoder subnetwork. This subnetwork excludes the channel-wise instance normalization performed by the other decoder subnetworks. Each decoder subnetwork has half the number of filters as the previous subnetwork.

```
numFilters = numFiltersFirstEncoder;
layerNamePrefix = strcat(decoderNamePrefix,num2str(stage+1));

decoderLayers = [
    decoderLayers
    transposedConv2dLayer([3 3],numFilters,"Stride",[2 2],"Cropping","same","WeightsInitializer"
        "Name",strcat(layerNamePrefix,"-UpConv"))
    leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-UpReLU"))
    depthConcatenationLayer(2,"Name",strcat(layerNamePrefix,"-DepthConcatenation"))
    convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal", .
        "Name",strcat(layerNamePrefix,"-Conv-1"))
    leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-ReLU-1"))
    convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal", .
        "Name",strcat(layerNamePrefix,"-Conv-2"))
    leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-ReLU-2"))];
```

Add the final layers of the U-Net. The pixel shuffle layer moves from the H/2-by-W/2-by-12 channel size of the activations from the final convolution to H-by-W-by 3 channel activations using pixel shuffle upsampling. The final layer encourages outputs to the desired range [0, 1] using a hyperbolic tangent function.

```
finalLayers = [
    convolution2dLayer([3 3],12,"Padding","same","WeightsInitializer","narrow-normal", ...
        "Name","Decoder-Stage-4-Conv-3")
    pixelShuffleLayer("pixelShuffle",2)
    tanhScaledAndShiftedLayer("tanhActivation")];

layers = [initialLayer;encoderLayers;bridgeLayers;decoderLayers;finalLayers];
lgraph = layerGraph(layers);
```

Connect layers of the encoding and decoding subnetworks.

```
lgraph = connectLayers(lgraph,"Encoder-Stage-1-ReLU-2","Decoder-Stage-4-DepthConcatenation/in2")
lgraph = connectLayers(lgraph,"Encoder-Stage-2-ReLU-2","Decoder-Stage-3-DepthConcatenation/in2")
lgraph = connectLayers(lgraph,"Encoder-Stage-3-ReLU-2","Decoder-Stage-2-DepthConcatenation/in2")
lgraph = connectLayers(lgraph,"Encoder-Stage-4-ReLU-2","Decoder-Stage-1-DepthConcatenation/in2")
net = dlnetwork(lgraph);
```

Visualize the network architecture using the Deep Network Designer (Deep Learning Toolbox) app.

```
deepNetworkDesigner(lgraph)
```

Load the Feature Extraction Network

This function modifies a pretrained VGG-16 deep neural network to extract image features at various layers. These multilayer features are used to compute content loss.

To get a pretrained VGG-16 network, install `vgg16` (Deep Learning Toolbox). If you do not have the required support package installed, then the software provides a download link.

```
vggNet = vgg16;
```

To make the VGG-16 network suitable for feature extraction, use the layers up to 'relu5_3'.

```
vggNet = vggNet.Layers(1:31);
vggNet = dlnetwork(layerGraph(vggNet));
```

Define Model Gradients and Loss Functions

The helper function `modelGradients` calculates the gradients and overall loss for batches of training data. This function is defined in Supporting Functions on page 18-0   section of this example.

The overall loss is a weighted sum of two losses: mean of absolute error (MAE) loss and content loss. The content loss is weighted such that the MAE loss and content loss contribute approximately equally to the overall loss:

$$lossOverall = lossMAE + weightFactor*lossContent$$

The MAE loss penalises the L^1 distance between samples of the network predictions and samples of the target image. L^1 is often a better choice than L^2 for image processing applications because it can help reduce blurring artifacts [4]. This loss is implemented using the `maeLoss` helper function defined in the Supporting Functions on page 18-0   section of this example.

The content loss helps the network learn both high-level structural content and low-level edge and color information. The loss function calculates a weighted sum of the mean square error (MSE) between predictions and targets for each activation layer. This loss is implemented using the `contentLoss` helper function defined in the Supporting Functions on page 18-0   section of this example.

Calculate Content Loss Weight Factor

The `modelGradients` helper function requires the content loss weight factor as an input argument. Calculate the weight factor for a sample batch of training data such that the MAE loss is equal to the weighted content loss.

Preview a batch of training data, which consists of pairs of RAW network inputs and RGB target outputs.

```
trainingBatch = preview(dsTrainAug);
networkInput = dlarray((trainingBatch{1,1}),'SSC');
targetOutput = dlarray((trainingBatch{1,2}),'SSC');
```

Predict the response of the untrained U-Net network using the `forward` (Deep Learning Toolbox) function.

```
predictedOutput = forward(net,networkInput);
```

Calculate the MAE and content losses between the predicted and target RGB images.

```
sampleMAELoss = maeLoss(predictedOutput,targetOutput);
sampleContentLoss = contentLoss(vggNet,predictedOutput,targetOutput);
```

Calculate the weight factor.

```
weightContent = sampleMAELoss/sampleContentLoss;
```

Specify Training Options

Define the training options that are used within the custom training loop to control aspects of Adam optimization. Train for 20 epochs.

```
learnRate = 5e-5;
numEpochs = 20;
```

Train Network

By default, the example downloads a pretrained version of the RAW-to-RGB network by using the helper function `downloadTrainedRAWToRGBNet`. The helper function is attached to the example as a supporting file. The pretrained network enables you to run the entire example without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the model in a custom training loop. For each iteration:

- Read the data for current mini-batch using the `next` (Deep Learning Toolbox) function.
- Evaluate the model gradients using the `dlfeval` (Deep Learning Toolbox) function and the `modelGradients` helper function.
- Update the network parameters using the `adamupdate` (Deep Learning Toolbox) function and the gradient information.
- Update the training progress plot for every iteration and display various computed losses.

A CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training (requires Parallel Computing Toolbox™). Training takes about 36 hours on an NVIDIA™ Titan RTX and can take even longer depending on your GPU hardware. If your GPU device has less memory, consider decreasing the mini-batch size in the Batch Training and Validation Data During Training on page 18-0 section of the example.

```
doTraining = false;
if doTraining

    % Create a directory to store checkpoints
    checkpointDir = fullfile(imageDir,'checkpoints',filesep);
    if ~exist(checkpointDir,'dir')
        mkdir(checkpointDir);
    end

    % Initialize training plot
    [hFig,batchLine,validationLine] = initializeTrainingPlotRAWPipeline;
```

```
% Initialize Adam solver state
[averageGrad,averageSqGrad] = deal([]);
iteration = 0;

start = tic;
for epoch = 1:numEpochs
    reset(trainingQueue);
    shuffle(trainingQueue);
    while hasdata(trainingQueue)
        [inputRAW,targetRGB] = next(trainingQueue);

        [grad,loss] = dlfeval(@modelGradients,net,vggNet,inputRAW,targetRGB,weightContent);

        iteration = iteration + 1;

        [net,averageGrad,averageSqGrad] = adamupdate(net, ...
            grad,averageGrad,averageSqGrad,iteration,learnRate);

        updateTrainingPlotRAWPipeline(batchLine,validationLine,iteration,loss,start,epoch,..
            validationQueue,valSetSize,valBatchSize,net,vggNet,weightContent);
    end
    % Save checkpoint of network state
    save(checkpointDir + "epoch" + epoch,'net');
end
% Save the final network state
save(checkpointDir + "trainedRAWToRGBNet.mat",'net');

else
    trainedRAWToRGBNet_url = 'https://ssd.mathworks.com/supportfiles/vision/data/trainedRAWToRGBI
    downloadTrainedRAWToRGBNet(trainedRAWToRGBNet_url,imageDir);
    load(fullfile(imageDir,'trainedRAWToRGBNet.mat'));
end
```

Calculate Image Quality Metrics

Reference-based quality metrics such as MSSIM or PSNR enable a quantitative measure of image quality. You can calculate the MSSIM and PSNR of the patched test images because they are spatially registered and the same size.

Iterate through the test set of patched images using a `minibatchqueue` (Deep Learning Toolbox) object.

```
patchTestSet = combine(dsTestRAW,dsTestRGB);
testPatchQueue = minibatchqueue(patchTestSet,'MiniBatchSize',16,'MiniBatchFormat','SSCB');
```

Iterate through the test set and calculate the MSSIM and PSNR for each test image using the `multissim` and `psnr` functions. Although the functions accept RGB images, the metrics are not well-defined for RGB images. Therefore, approximate the MSSIM and PSNR by calculating the metrics of the color channels separately. You can use the `calculateRAWToRGBQualityMetrics` helper function to calculate channel-wise metrics. This function is attached to the example as a supporting file.

```
totalMSSIM = 0;
totalPSNR = 0;
while hasdata(testPatchQueue)
    [inputRAW,targetRGB] = next(testPatchQueue);
```

```
    outputRGB = forward(net,inputRAW);
    [mssimOut,psnrOut] = calculateRAWToRGBQualityMetrics(outputRGB,targetRGB);
    totalMSSIM = totalMSSIM + mssimOut;
    totalPSNR = totalPSNR + psnrOut;
end
```

Calculate the mean MSSIM and mean PSNR over the test set. This result is consistent with the similar U-Net approach from [3] for mean MSSIM and competitive with the PyNet approach in [3] in mean PSNR. The differences in loss functions and use of pixel shuffle upsampling compared to [3] likely account for these differences.

```
numObservations = dsTestRGB.numpartitions;
meanMSSIM = totalMSSIM / numObservations
```

```
meanMSSIM = single
    0.8534
```

```
meanPSNR = totalPSNR / numObservations
```

```
meanPSNR = 21.2956
```

Evaluate Trained Image Processing Pipeline on Full-Sized Images

Because of sensor differences between the phone camera and DSLR used to acquire the full-resolution test images, the scenes are not registered and are not the same size. Reference-based comparison of the full-resolution images from the network and the DSLR ISP is difficult. However, a qualitative comparison of the images is useful because a goal of image processing is to create an aesthetically pleasing image.

Create an image datastore that contains full-sized RAW images acquired by a phone camera.

```
testImageDir = fullfile(imageDir,'test');
testImageDirRAW = "huawei_full_resolution";
dsTestFullRAW = imageDatastore(fullfile(testImageDir,testImageDirRAW));
```

Get the names of the image files in the full-sized RAW test set.

```
targetFilesToInclude = extractAfter(string(dsTestFullRAW.Files),fullfile(testImageDirRAW,filesep
targetFilesToInclude = extractBefore(targetFilesToInclude,".png");
```

Preprocess the RAW data by converting the data to the form expected by the network using the `transform` function. The `transform` function processes the data using the operations specified in the `preprocessRAWDataForRAWToRGB` helper function. The helper function is attached to the example as a supporting file.

```
dsTestFullRAW = transform(dsTestFullRAW,@preprocessRAWDataForRAWToRGB);
```

Create an image datastore that contains full-sized RGB test images captured from the high-end DSLR. The Zurich RAW to RGB data set contains more full-sized RGB images than RAW images, so include only the RGB images with a corresponding RAW image.

```
dsTestFullRGB = imageDatastore(fullfile(imageDir,'full_resolution','canon'));
dsTestFullRGB.Files = dsTestFullRGB.Files(contains(dsTestFullRGB.Files,targetFilesToInclude));
```

Read in the target RGB images. Get a sense of the overall output by looking at a montage view.

```
targetRGB = readall(dsTestFullRGB);
montage(targetRGB,"Size",[5 2],"Interpolation","bilinear")
```

Iterate through the test set of full-sized images using a `minibatchqueue` object. If you have a GPU device with sufficient memory to process full-resolution images, then you can run preduction on a GPU by specifying the output environment as `"gpu"`.

```
testQueue = minibatchqueue(dsTestFullRAW,"MiniBatchSize",1,"MiniBatchFormat","SSCB", ...
    "OutputEnvironment","cpu");
```

For each full-sized RAW test image, predict the output RGB image by calling `forward` (Deep Learning Toolbox) on the network.

```
outputSize = 2*size(preview(dsTestFullRAW),[1 2]);
outputImages = zeros([outputSize,3,dsTestFullRAW.numpartitions],'uint8');

idx = 1;
while hasdata(testQueue)
    inputRAW = next(testQueue);
    rgbOut = forward(net,inputRAW);
    rgbOut = gather(extractdata(rgbOut));
    outputImages(:,:,:,idx) = im2uint8(rgbOut);
    idx = idx+1;
end
```

Get a sense of the overall output by looking at a montage view. The network produces images that are aesthetically pleasing, with similar characteristics.

```
montage(outputImages,"Size",[5 2],"Interpolation","bilinear")
```

Compare one target RGB image with the corresponding image predicted by the network. The network produces colors which are more saturated than the target DSLR images. Although the colors from the simple U-Net architecture are not the same as the DSLR targets, the images are still qualitively pleasing in many cases.

```
imgIdx = 1;
imTarget = targetRGB{imgIdx};
imPredicted = outputImages(:,:,:,imgIdx);
```

```
figure
montage({imTarget,imPredicted},"Interpolation","bilinear")
```


To improve the performance of the RAW to RGB network, a network architecture would learn detailed localized spatial features using multiple scales from global features that describe color and contrast [3].

Supporting Functions

Model Gradients Function

The `modelGradients` helper function calculates the gradients and overall loss. The gradient information is returned as a table which includes the layer, parameter name and value for each learnable parameter in the model.

```
function [gradients,loss] = modelGradients(dlnet,vggNet,Xpatch,Target,weightContent)
    Y = forward(dlnet,Xpatch);
    lossMAE = maeLoss(Y,Target);
    lossContent = contentLoss(vggNet,Y,Target);
    loss = lossMAE + weightContent.*lossContent;
    gradients = dlgradient(loss,dlnet.Learnables);
end
```

Mean Absolute Error Loss Function

The helper function `maeLoss` computes the mean absolute error between network predictions, Y, and target images, T.

```
function loss = maeLoss(Y,T)
    loss = mean(abs(Y-T),'all');
end
```

Content Loss Function

The helper function `contentLoss` calculates a weighted sum of the MSE between network predictions, Y, and target images, T, for each activation layer. The `contentLoss` helper function calculates the MSE for each activation layer using the `mseLoss` helper function. Weights are selected such that the loss from each activation layers contributes roughly equally to the overall content loss.

```matlab
function loss = contentLoss(net,Y,T)

    layers = ["relu1_1","relu1_2","relu2_1","relu2_2","relu3_1","relu3_2","relu3_3","relu4_1"];
    [T1,T2,T3,T4,T5,T6,T7,T8] = forward(net,T,'Outputs',layers);
    [X1,X2,X3,X4,X5,X6,X7,X8] = forward(net,Y,'Outputs',layers);

    l1 = mseLoss(X1,T1);
    l2 = mseLoss(X2,T2);
    l3 = mseLoss(X3,T3);
    l4 = mseLoss(X4,T4);
    l5 = mseLoss(X5,T5);
    l6 = mseLoss(X6,T6);
    l7 = mseLoss(X7,T7);
    l8 = mseLoss(X8,T8);

    layerLosses = [l1 l2 l3 l4 l5 l6 l7 l8];
    weights = [1 0.0449 0.0107 0.0023 6.9445e-04 2.0787e-04 2.0118e-04 6.4759e-04];
    loss = sum(layerLosses.*weights);
end
```

Mean Square Error Loss Function

The helper function `mseLoss` computes the MSE between network predictions, Y, and target images, T.

```matlab
function loss = mseLoss(Y,T)
    loss = mean((Y-T).^2,'all');
end
```

References

1) Sumner, Rob. "Processing RAW Images in MATLAB". May 19, 2014. https://rcsumner.net/raw_guide/RAWguide.pdf

2) Chen, Chen, Qifeng Chen, Jia Xu, and Vladlen Koltun. "Learning to See in the Dark." *ArXiv:1805.01934 [Cs]*, May 4, 2018. http://arxiv.org/abs/1805.01934.

3) Ignatov, Andrey, Luc Van Gool, and Radu Timofte. "Replacing Mobile Camera ISP with a Single Deep Learning Model." *ArXiv:2002.05509 [Cs, Eess]*, February 13, 2020. http://arxiv.org/abs/2002.05509. Project Website.

4) Zhao, Hang, Orazio Gallo, Iuri Frosio, and Jan Kautz. "Loss Functions for Neural Networks for Image Processing." *ArXiv:1511.08861 [Cs]*, April 20, 2018. http://arxiv.org/abs/1511.08861.

5) Johnson, Justin, Alexandre Alahi, and Li Fei-Fei. "Perceptual Losses for Real-Time Style Transfer and Super-Resolution." *ArXiv:1603.08155 [Cs]*, March 26, 2016. http://arxiv.org/abs/1603.08155.

6) Shi, Wenzhe, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P. Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. "Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network." *ArXiv:1609.05158 [Cs, Stat]*, September 23, 2016. http://arxiv.org/abs/1609.05158.

See Also

combine | imageDatastore | trainNetwork | trainingOptions | transform

More About

- "Preprocess Images for Deep Learning" (Deep Learning Toolbox)
- "Datastores for Deep Learning" (Deep Learning Toolbox)
- "List of Deep Learning Layers" (Deep Learning Toolbox)
- "Define Custom Training Loops, Loss Functions, and Networks" (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.
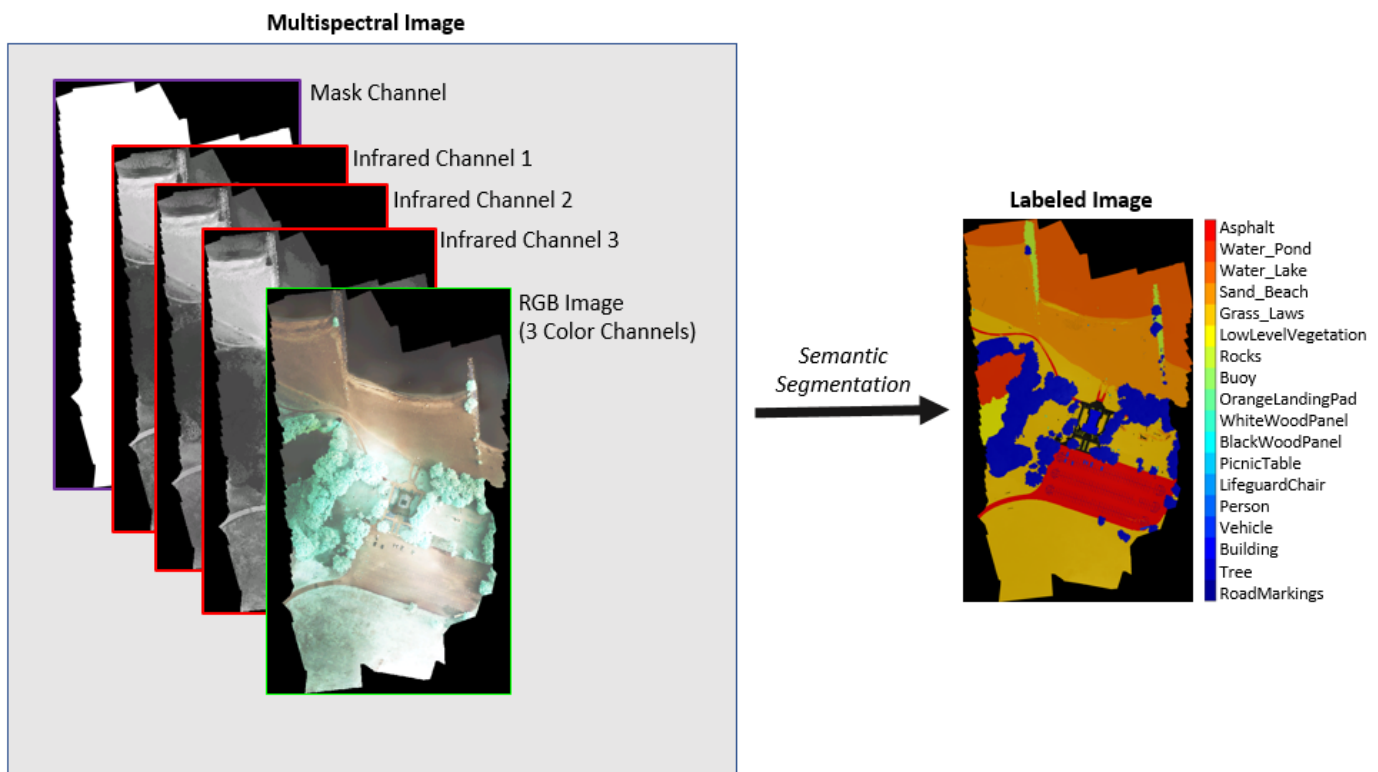
This example shows how to use deep-learning-based semantic segmentation techniques to calculate the percentage vegetation cover in a region from a set of multispectral images.

Download Data

This example uses a high-resolution multispectral data set to train the network [1 on page 18-0   ]. 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. The size of the data file is ~3.0 GB.

Download the MAT-file version of the data set using the `downloadHamlinBeachMSIData` helper function. This function is attached to the example as a supporting file.

```
imageDir = tempdir;
url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';
downloadHamlinBeachMSIData(url,imageDir);
```

In addition, download a pretrained version of U-Net for this dataset using the `downloadTrainedUnet` helper function. This function is attached to the example as a supporting file. 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.mat';
downloadTrainedUnet(trainedUnet_url,imageDir);
```

Inspect Training Data

Load the data set into the 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`. This function is attached to the example as a supporting file.

```
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
```

18-75

The RGB color channels are the 3rd, 2nd and 1st 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(:,:,[3 2 1])), ...
    histeq(val_data(:,:,[3 2 1])), ...
    histeq(test_data(:,:,[3 2 1]))}, ...
    'BorderSize',10,'BackgroundColor','white')
title('RGB Component of Training Image (Left), Validation Image (Center), and Test Image (Right)
```
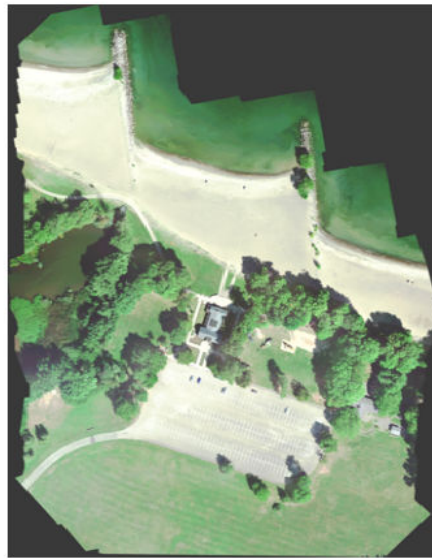

Display the last three histogram-equalized channels of the training data as a montage. These channels correspond to the near-infrared bands and highlight different components of the image based on their heat signatures. For example, the trees near the center of the second channel image show more detail than the trees in the other two channels.

```
figure
montage(...
    {histeq(train_data(:,:,4)), ...
    histeq(train_data(:,:,5)), ...
    histeq(train_data(:,:,6))}, ...
    'BorderSize',10,'BackgroundColor','white')
title('IR Channels 1 (Left), 2, (Center), and 3 (Right) of Training Image')
```
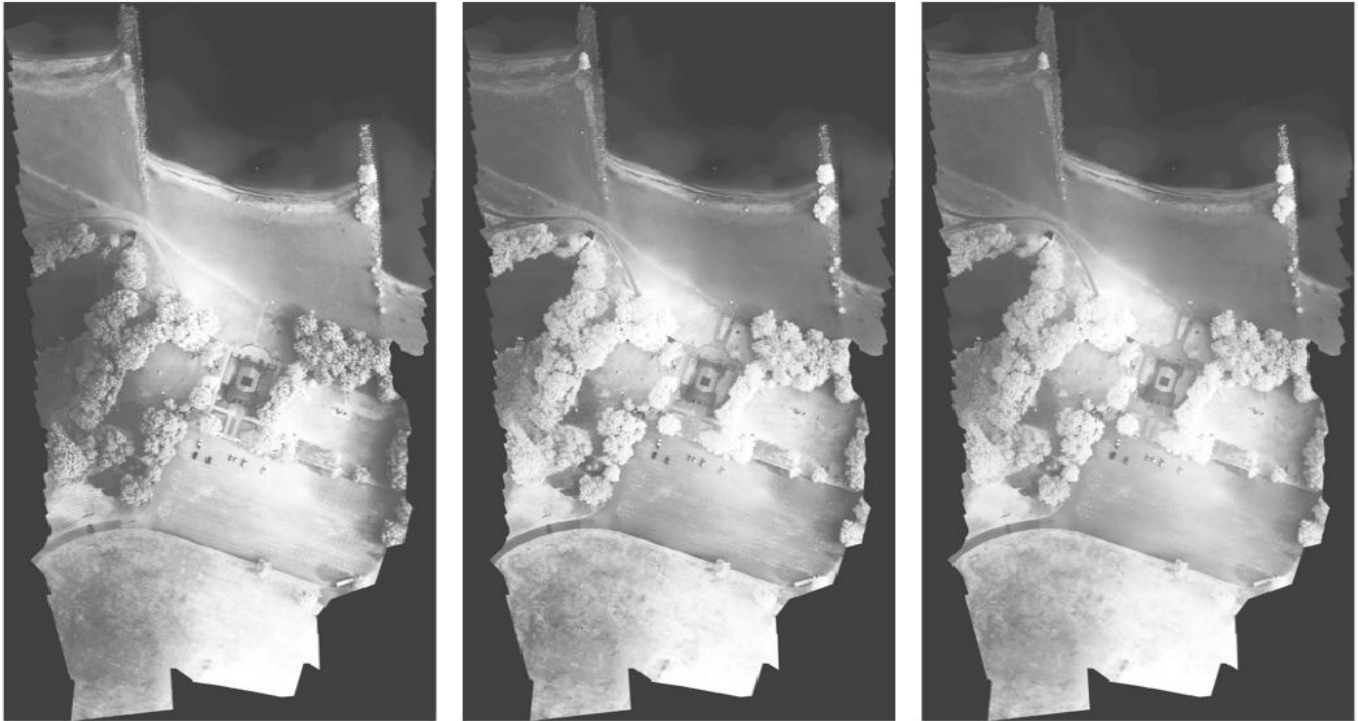
Channel 7 is a mask that indicates the valid segmentation region. Display the mask for the training, validation, and test images.

```
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',cmap);

figure
title('Training Labels')
imshow(B)
N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterpreter','no
colormap(cmap)
```
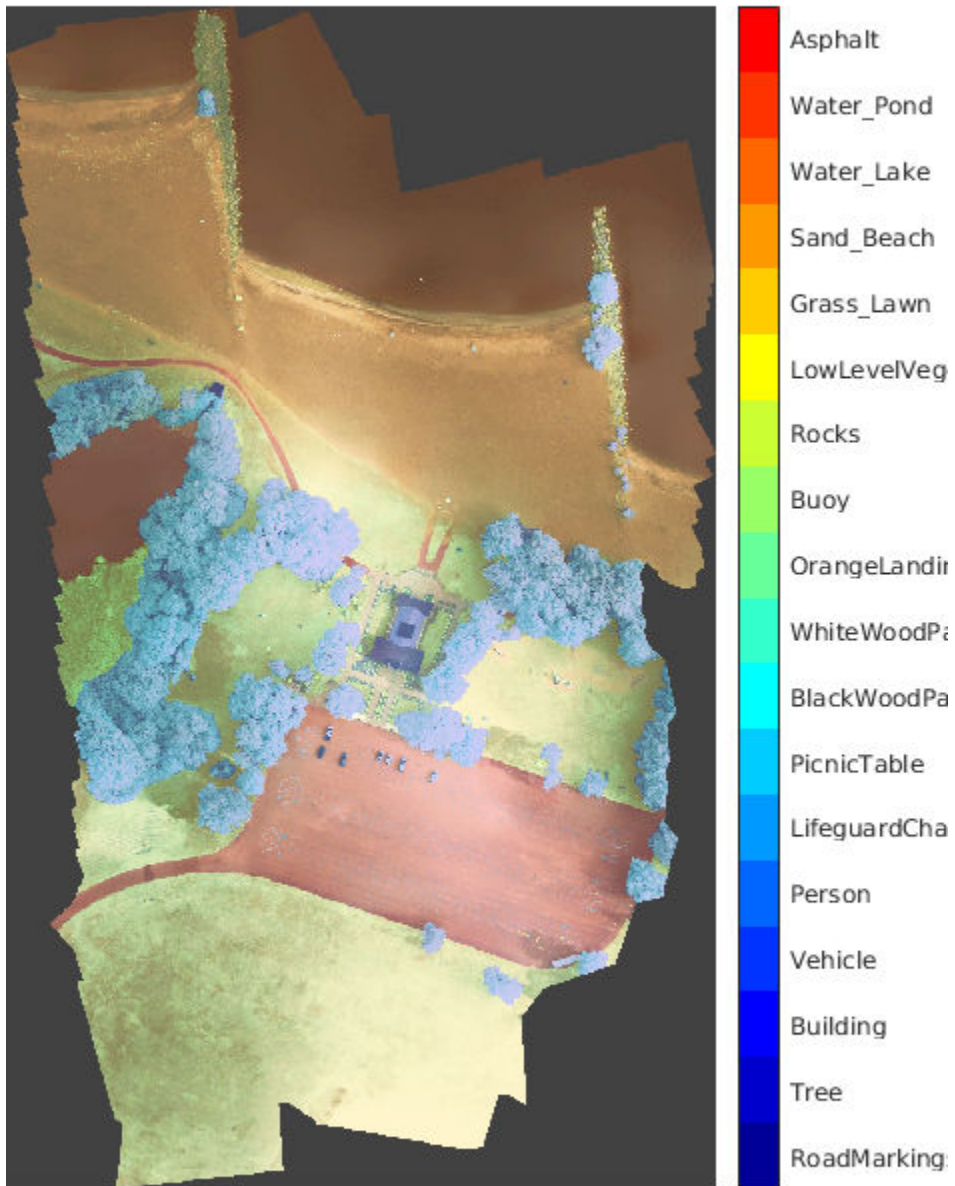
Save the training data as a MAT file and the training labels as a PNG file.

```
save('train_data.mat','train_data');
imwrite(train_labels,'train_labels.png');
```

Create Random Patch Extraction Datastore for Training

Use a random patch extraction datastore to feed the training data to the network. This datastore extracts multiple corresponding random patches from an image datastore and pixel label datastore that contain ground truth images and pixel label data. Patching is a common technique to prevent running out of memory for large images and to effectively increase the amount of available training data.

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. This function is attached to the example as a supporting file.

```
imds = imageDatastore('train_data.mat','FileExtensions','.mat','ReadFcn',@matReader);
```

Create a `pixelLabelDatastore` (Computer Vision Toolbox) to store the label patches containing the 18 labeled regions.

```
pixelLabelIds = 1:18;
pxds = pixelLabelDatastore('train_labels.png',classNames,pixelLabelIds);
```

Create a `randomPatchExtractionDatastore` from the image datastore and the pixel label datastore. Each mini-batch contains 16 patches of size 256-by-256 pixels. One thousand mini-batches are extracted at each iteration of the epoch.

```
dsTrain = randomPatchExtractionDatastore(imds,pxds,[256,256],'PatchesPerImage',16000);
```

The random patch extraction datastore `dsTrain` provides mini-batches of data to the network at each iteration of the epoch. Preview the datastore to explore the data.

```
inputBatch = preview(dsTrain);
disp(inputBatch)
```

```
        InputImage          ResponsePixelLabelImage
    _____      _____

    {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}
```

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 on page 18-0   ]. 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. This function is attached to the example as a supporting file.

```
inputTileSize = [256,256,6];
lgraph = createUnet(inputTileSize);
disp(lgraph.Layers)
```

```
  58x1 Layer array with layers:

     1   'ImageInputLayer'                         Image Input             256x256x6 images
     2   'Encoder-Section-1-Conv-1'                Convolution             64 3x3x6 convolut
     3   'Encoder-Section-1-ReLU-1'                ReLU                    ReLU
     4   'Encoder-Section-1-Conv-2'                Convolution             64 3x3x64 convolu
     5   'Encoder-Section-1-ReLU-2'                ReLU                    ReLU
     6   'Encoder-Section-1-MaxPool'               Max Pooling             2x2 max pooling w
     7   'Encoder-Section-2-Conv-1'                Convolution             128 3x3x64 convol
     8   'Encoder-Section-2-ReLU-1'                ReLU                    ReLU
     9   'Encoder-Section-2-Conv-2'                Convolution             128 3x3x128 convo
    10   'Encoder-Section-2-ReLU-2'                ReLU                    ReLU
    11   'Encoder-Section-2-MaxPool'               Max Pooling             2x2 max pooling w
    12   'Encoder-Section-3-Conv-1'                Convolution             256 3x3x128 convo
    13   'Encoder-Section-3-ReLU-1'                ReLU                    ReLU
    14   'Encoder-Section-3-Conv-2'                Convolution             256 3x3x256 convo
    15   'Encoder-Section-3-ReLU-2'                ReLU                    ReLU
    16   'Encoder-Section-3-MaxPool'               Max Pooling             2x2 max pooling w
    17   'Encoder-Section-4-Conv-1'                Convolution             512 3x3x256 convo
    18   'Encoder-Section-4-ReLU-1'                ReLU                    ReLU
    19   'Encoder-Section-4-Conv-2'                Convolution             512 3x3x512 convo
    20   'Encoder-Section-4-ReLU-2'                ReLU                    ReLU
    21   'Encoder-Section-4-DropOut'               Dropout                 50% dropout
    22   'Encoder-Section-4-MaxPool'               Max Pooling             2x2 max pooling w
    23   'Mid-Conv-1'                              Convolution             1024 3x3x512 conv
    24   'Mid-ReLU-1'                              ReLU                    ReLU
    25   'Mid-Conv-2'                              Convolution             1024 3x3x1024 conv
    26   'Mid-ReLU-2'                              ReLU                    ReLU
    27   'Mid-DropOut'                             Dropout                 50% dropout
    28   'Decoder-Section-1-UpConv'                Transposed Convolution  512 2x2x1024 trans
    29   'Decoder-Section-1-UpReLU'                ReLU                    ReLU
    30   'Decoder-Section-1-DepthConcatenation'    Depth concatenation     Depth concatenatio
    31   'Decoder-Section-1-Conv-1'                Convolution             512 3x3x1024 convo
    32   'Decoder-Section-1-ReLU-1'                ReLU                    ReLU
    33   'Decoder-Section-1-Conv-2'                Convolution             512 3x3x512 convo
    34   'Decoder-Section-1-ReLU-2'                ReLU                    ReLU
    35   'Decoder-Section-2-UpConv'                Transposed Convolution  256 2x2x512 transp
    36   'Decoder-Section-2-UpReLU'                ReLU                    ReLU
    37   'Decoder-Section-2-DepthConcatenation'    Depth concatenation     Depth concatenatio
    38   'Decoder-Section-2-Conv-1'                Convolution             256 3x3x512 convo
    39   'Decoder-Section-2-ReLU-1'                ReLU                    ReLU
    40   'Decoder-Section-2-Conv-2'                Convolution             256 3x3x256 convo
    41   'Decoder-Section-2-ReLU-2'                ReLU                    ReLU
    42   'Decoder-Section-3-UpConv'                Transposed Convolution  128 2x2x256 transp
    43   'Decoder-Section-3-UpReLU'                ReLU                    ReLU
    44   'Decoder-Section-3-DepthConcatenation'    Depth concatenation     Depth concatenatio
    45   'Decoder-Section-3-Conv-1'                Convolution             128 3x3x256 convo
    46   'Decoder-Section-3-ReLU-1'                ReLU                    ReLU
    47   'Decoder-Section-3-Conv-2'                Convolution             128 3x3x128 convo
```

```
48   'Decoder-Section-3-ReLU-2'             ReLU                      ReLU
49   'Decoder-Section-4-UpConv'            Transposed Convolution    64 2x2x128 transp
50   'Decoder-Section-4-UpReLU'            ReLU                      ReLU
51   'Decoder-Section-4-DepthConcatenation' Depth concatenation     Depth concatenati
52   'Decoder-Section-4-Conv-1'            Convolution               64 3x3x128 convolu
53   'Decoder-Section-4-ReLU-1'            ReLU                      ReLU
54   'Decoder-Section-4-Conv-2'            Convolution               64 3x3x64 convolut
55   'Decoder-Section-4-ReLU-2'            ReLU                      ReLU
56   'Final-ConvolutionLayer'             Convolution               18 1x1x64 convolut
57   'Softmax-Layer'                      Softmax                   softmax
58   'Segmentation-Layer'                 Pixel Classification Layer Cross-entropy loss
```

Select Training Options

Train the network using stochastic gradient descent with momentum (SGDM) optimization. Specify the hyperparameter settings for SGDM by using the `trainingOptions` (Deep Learning Toolbox) 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 specifying `'GradientThreshold'` as `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,...
    'LearnRateSchedule','piecewise',...
    'Shuffle','every-epoch',...
    'GradientThresholdMethod','l2norm',...
    'GradientThreshold',0.05, ...
    'Plots','training-progress', ...
    'VerboseFrequency',20);
```

Train the Network

After configuring the training options and the random patch extraction datastore, train the U-Net network by using the `trainNetwork` (Deep Learning Toolbox) 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™ 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,info] = trainNetwork(dsTrain,lgraph,options);
    save(['multispectralUnet-' modelDateTime '-Epoch-' num2str(maxEpochs) '.mat'],'net','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. This function is attached to the example as a supporting file. `segmentImage` performs segmentation on image patches using the `semanticseg` (Computer Vision Toolbox) 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,[])
title('Segmented Image')
```
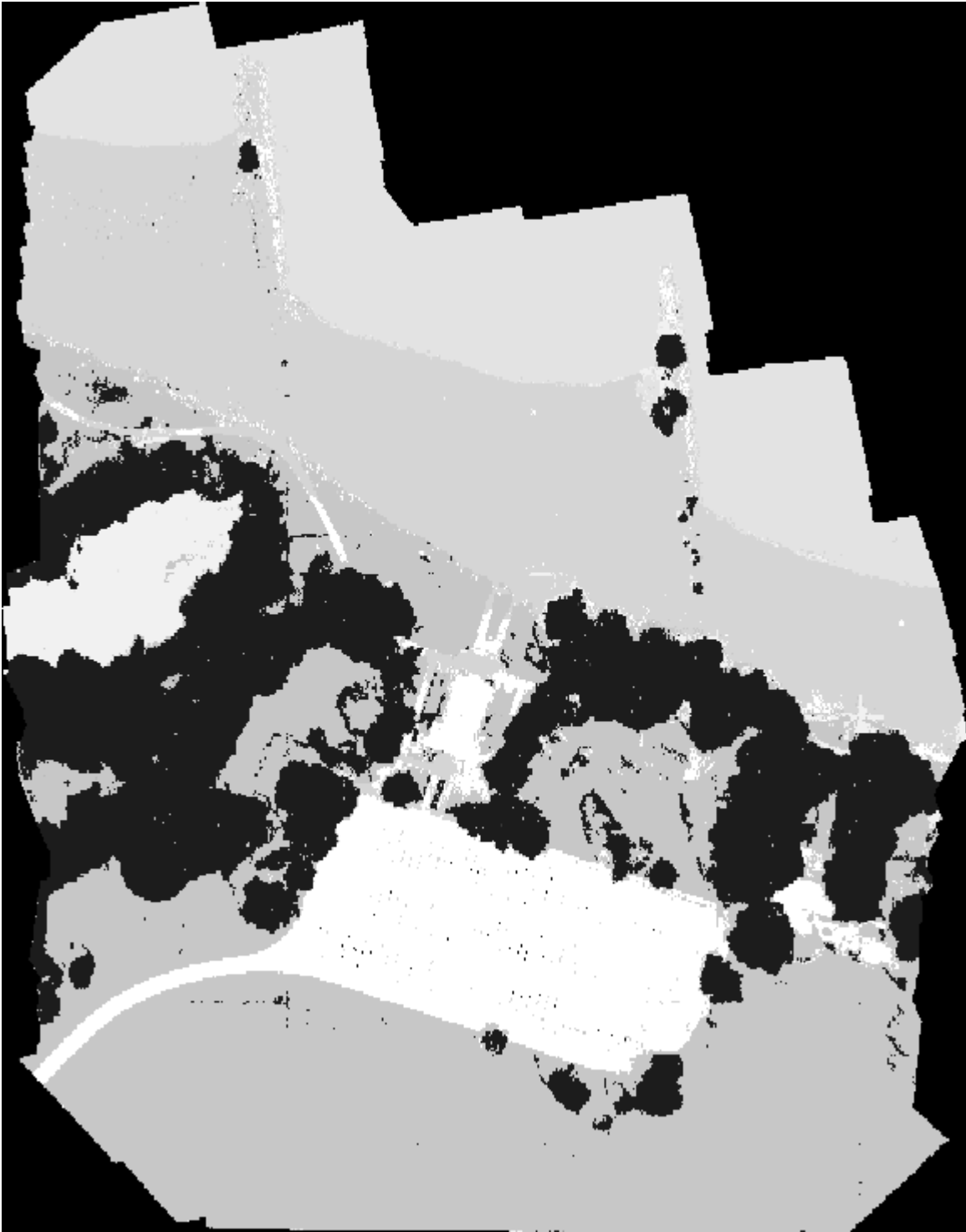
The output of semantic segmentation is noisy. Perform post image processing to remove noise and stray pixels. Use the `medfilt2` function to remove salt-and-pepper noise from the segmentation. Visualize the segmented image with the noise removed.

```
segmentedImage = medfilt2(segmentedImage,[7,7]);
imshow(segmentedImage,[]);
title('Segmented Image  with Noise Removed')
```
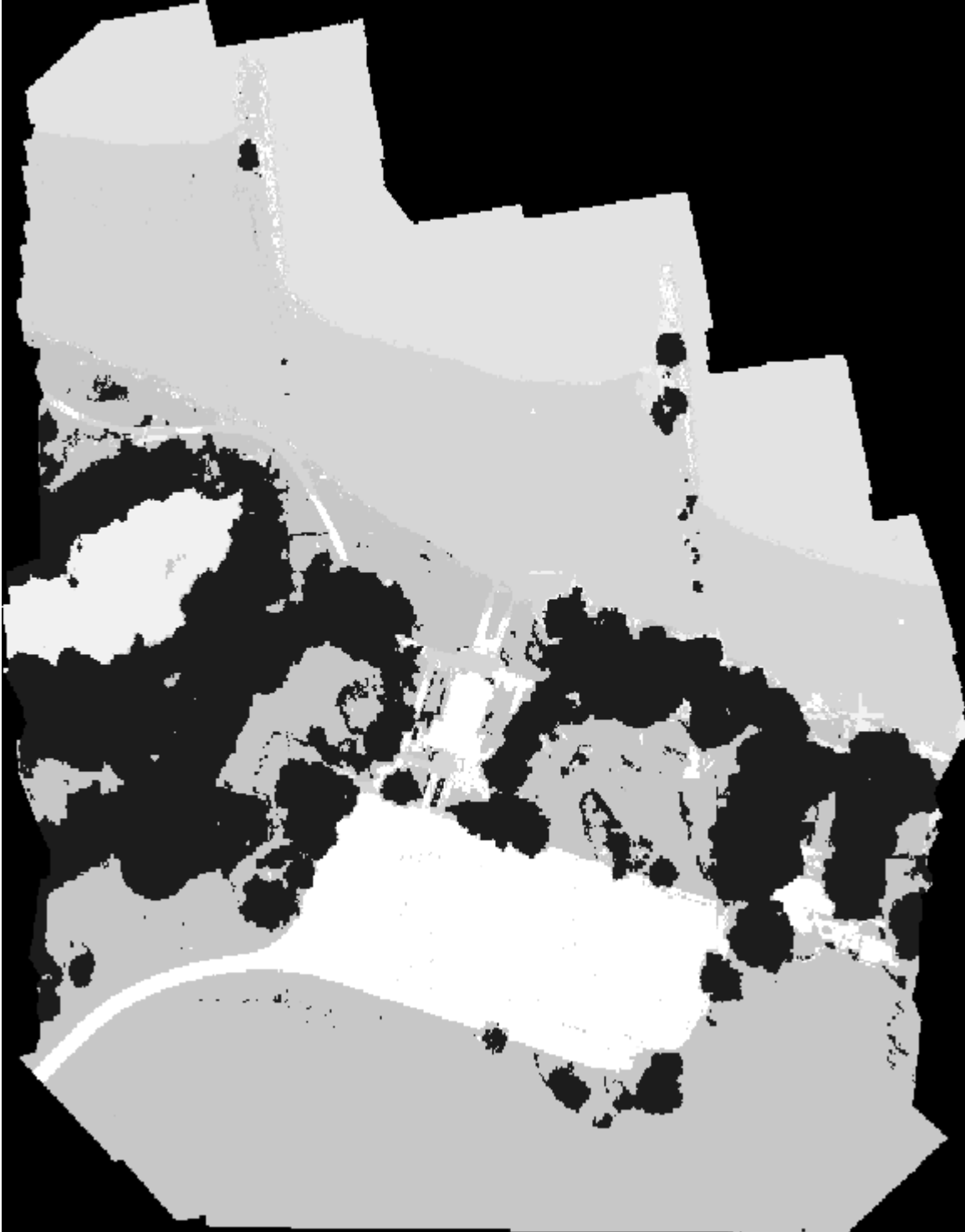

Overlay the segmented image on the histogram-equalized RGB validation image.

```
B = labeloverlay(histeq(val_data(:,:,[3 2 1])),segmentedImage,'Transparency',0.8,'Colormap',cmap)

figure
imshow(B)
title('Labeled Validation Image')
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterpreter','n
colormap(cmap)
```

Labeled Validation Image
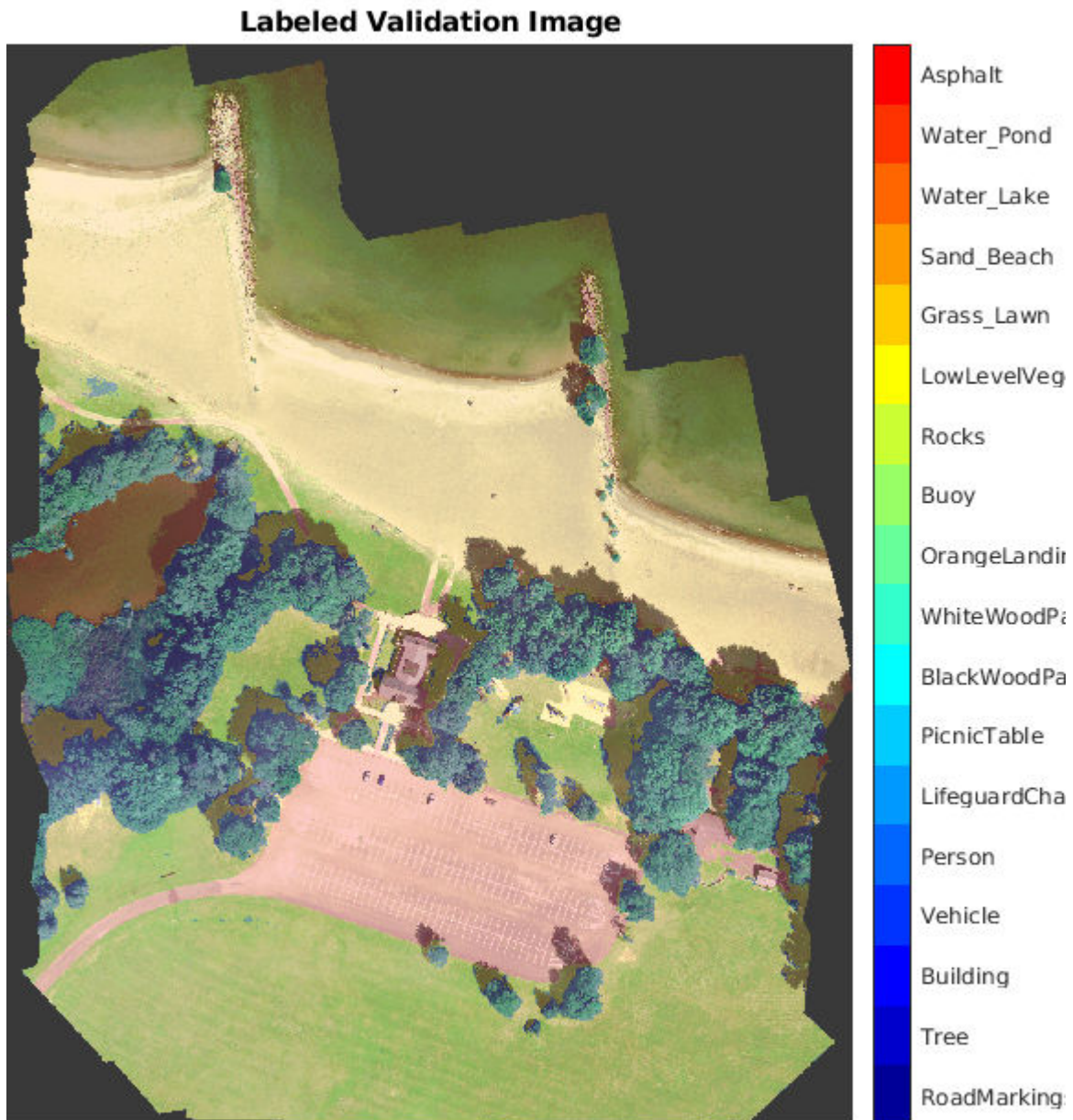
Save the segmented image and ground truth labels as PNG files. These will be used to compute accuracy metrics.

```
imwrite(segmentedImage,'results.png');
imwrite(val_labels,'gtruth.png');
```

Quantify Segmentation Accuracy

Create a `pixelLabelDatastore` (Computer Vision Toolbox) 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` (Computer Vision Toolbox) function.

```
ssm = evaluateSemanticSegmentation(pxdsResults,pxdsTruth,'Metrics','global-accuracy');
```

```
Evaluating semantic segmentation results
----------------------------------------
* Selected metrics: global accuracy.
* Processed 1 images.
* 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%.
```

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 | randomPatchExtractionDatastore | semanticseg | trainNetwork | trainingOptions | unetLayers

More About

- "Getting Started with Semantic Segmentation Using Deep Learning" (Computer Vision Toolbox)
- "Semantic Segmentation Using Deep Learning" (Computer Vision Toolbox)
- "Datastores for Deep Learning" (Deep Learning Toolbox)

External Websites

- https://github.com/rmkemker/RIT-18

3-D Brain Tumor Segmentation Using Deep Learning

This example shows how to train a 3-D U-Net neural network and perform semantic segmentation of brain tumors from 3-D medical images. The example shows how to train a 3-D U-Net network and also provides a pretrained network. Use of a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for 3-D semantic segmentation (requires Parallel Computing Toolbox™).

Introduction

Semantic segmentation involves labeling each pixel in an image or voxel of a 3-D volume with a class. This example illustrates the use of deep learning methods to perform binary semantic segmentation of brain tumors in magnetic resonance imaging (MRI) scans. In this binary segmentation, each pixel is labeled as tumor or background.

This example performs brain tumor segmentation using a 3-D U-Net architecture [1 on page 18-0   ]. U-Net is a fast, efficient and simple network that has become popular in the semantic segmentation domain.

One challenge of medical image segmentation is the amount of memory needed to store and process 3-D volumes. Training a network on the full input volume is impractical due to GPU resource constraints. This example solves the problem by training the network on image patches. The example uses an overlap-tile strategy to stitch test patches into a complete segmented test volume. The example avoids border artifacts by using the valid part of the convolution in the neural network [5 on page 18-0   ].

A second challenge of medical image segmentation is class imbalance in the data that hampers training when using conventional cross entropy loss. This example solves the problem by using a weighted multiclass Dice loss function [4 on page 18-0   ]. Weighting the classes helps to counter the influence of larger regions on the Dice score, making it easier for the network to learn how to segment smaller regions.

Download Training, Validation, and Test Data

This example uses the BraTS data set [2 on page 18-0   ]. The BraTS data set contains MRI scans of brain tumors, namely gliomas, which are the most common primary brain malignancies. The size of the data file is ~7 GB. If you do not want to download the BraTS data set, then go directly to the Download Pretrained Network and Sample Test Set on page 18-0   section in this example.

Create a directory to store the BraTS data set.

```
imageDir = fullfile(tempdir,'BraTS');
if ~exist(imageDir,'dir')
    mkdir(imageDir);
end
```

To download the BraTS data, go to the Medical Segmentation Decathlon website and click the "Download Data" link. Download the "Task01_BrainTumour.tar" file [3 on page 18-0   ]. Unzip the TAR file into the directory specified by the `imageDir` variable. When unzipped successfully, `imageDir` will contain a directory named `Task01_BrainTumour` that has three subdirectories: `imagesTr`, `imagesTs`, and `labelsTr`.

The data set contains 750 4-D volumes, each representing a stack of 3-D images. Each 4-D volume has size 240-by-240-by-155-by-4, where the first three dimensions correspond to height, width, and

depth of a 3-D volumetric image. The fourth dimension corresponds to different scan modalities. The data set is divided into 484 training volumes with voxel labels and 266 test volumes, The test volumes do not have labels so this example does not use the test data. Instead, the example splits the 484 training volumes into three independent sets that are used for training, validation, and testing.

Preprocess Training and Validation Data

To train the 3-D U-Net network more efficiently, preprocess the MRI data using the helper function `preprocessBraTSdataset`. This function is attached to the example as a supporting file.

The helper function performs these operations:

- Crop the data to a region containing primarily the brain and tumor. Cropping the data reduces the size of data while retaining the most critical part of each MRI volume and its corresponding labels.
- Normalize each modality of each volume independently by subtracting the mean and dividing by the standard deviation of the cropped brain region.
- Split the 484 training volumes into 400 training, 29 validation, and 55 test sets.

Preprocessing the data can take about 30 minutes to complete.

```
sourceDataLoc = [imageDir filesep 'Task01_BrainTumour'];
preprocessDataLoc = fullfile(tempdir,'BraTS','preprocessedDataset');
preprocessBraTSdataset(preprocessDataLoc,sourceDataLoc);
```

Create Random Patch Extraction Datastore for Training and Validation

Use a random patch extraction datastore to feed the training data to the network and to validate the training progress. This datastore extracts random patches from ground truth images and corresponding pixel label data. Patching is a common technique to prevent running out of memory when training with arbitrarily large volumes.

Create an `imageDatastore` to store the 3-D image data. 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, `matRead`. This function is attached to the example as a supporting file.

```
volReader = @(x) matRead(x);
volLoc = fullfile(preprocessDataLoc,'imagesTr');
volds = imageDatastore(volLoc, ...
    'FileExtensions','.mat','ReadFcn',volReader);
```

Create a `pixelLabelDatastore` (Computer Vision Toolbox) to store the labels.

```
lblLoc = fullfile(preprocessDataLoc,'labelsTr');
classNames = ["background","tumor"];
pixelLabelID = [0 1];
pxds = pixelLabelDatastore(lblLoc,classNames,pixelLabelID, ...
    'FileExtensions','.mat','ReadFcn',volReader);
```

Preview one image volume and label. Display the labeled volume using the `labelvolshow` function. Make the background fully transparent by setting the visibility of the background label (1) to 0.

```
volume = preview(volds);
label = preview(pxds);

viewPnl = uipanel(figure,'Title','Labeled Training Volume');
```

```
hPred = labelvolshow(label,volume(:,:,:,1),'Parent',viewPnl, ...
    'LabelColor',[0 0 0;1 0 0]);
hPred.LabelVisibility(1) = 0;
```


Labeled Training Volume
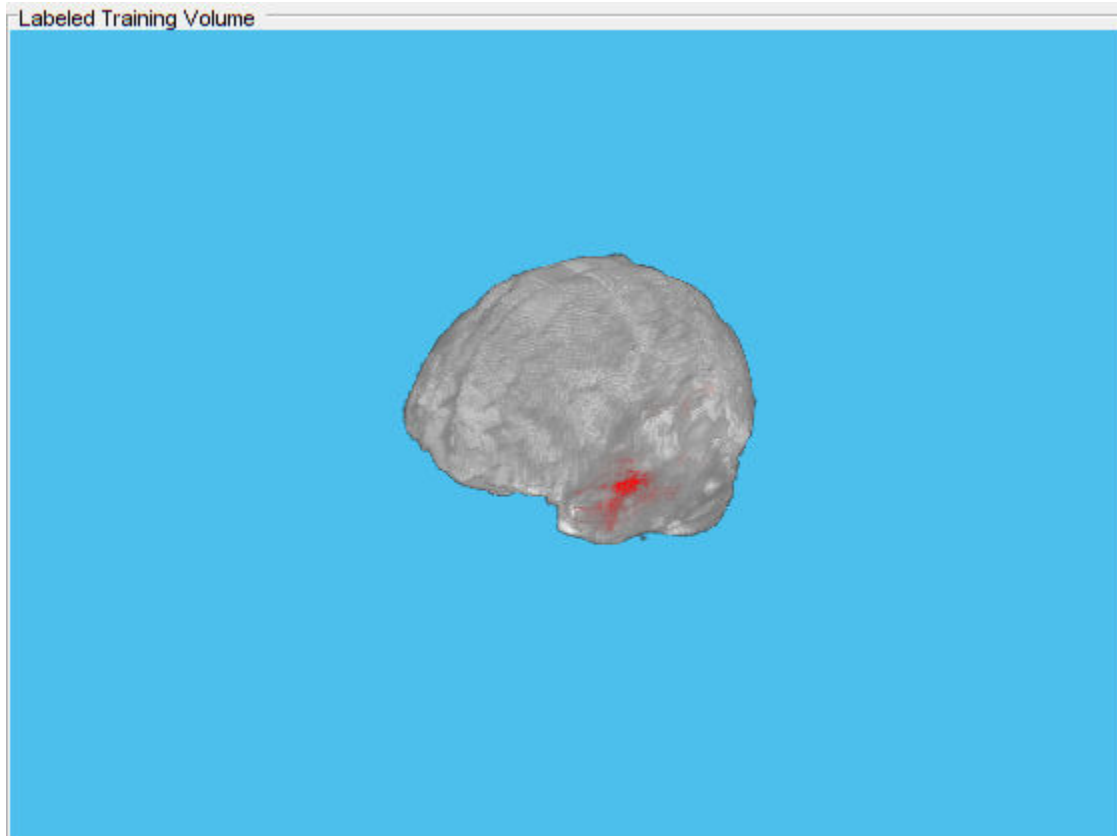
Create a `randomPatchExtractionDatastore` that contains the training image and pixel label data. Specify a patch size of 132-by-132-by-132 voxels. Specify `'PatchesPerImage'` to extract 16 randomly positioned patches from each pair of volumes and labels during training. Specify a mini-batch size of 8.

```
patchSize = [132 132 132];
patchPerImage = 16;
miniBatchSize = 8;
patchds = randomPatchExtractionDatastore(volds,pxds,patchSize, ...
    'PatchesPerImage',patchPerImage);
patchds.MiniBatchSize = miniBatchSize;
```

Follow the same steps to create a `randomPatchExtractionDatastore` that contains the validation image and pixel label data. You can use validation data to evaluate whether the network is continuously learning, underfitting, or overfitting as time progresses.

```
volLocVal = fullfile(preprocessDataLoc,'imagesVal');
voldsVal = imageDatastore(volLocVal, ...
    'FileExtensions','.mat','ReadFcn',volReader);

lblLocVal = fullfile(preprocessDataLoc,'labelsVal');
pxdsVal = pixelLabelDatastore(lblLocVal,classNames,pixelLabelID, ...
    'FileExtensions','.mat','ReadFcn',volReader);
```

```
dsVal = randomPatchExtractionDatastore(voldsVal,pxdsVal,patchSize, ...
    'PatchesPerImage',patchPerImage);
dsVal.MiniBatchSize = miniBatchSize;
```

Augment the training and validation data by using the `transform` function with custom preprocessing operations specified by the helper function `augmentAndCrop3dPatch`. This function is attached to the example as a supporting file.

The `augmentAndCrop3dPatch` function performs these operations:

1 Randomly rotate and reflect training data to make the training more robust. The function does not rotate or reflect validation data.

2 Crop response patches to the output size of the network, 44-by-44-by-44 voxels.

```
dataSource = 'Training';
dsTrain = transform(patchds,@(patchIn)augmentAndCrop3dPatch(patchIn,dataSource));

dataSource = 'Validation';
dsVal = transform(dsVal,@(patchIn)augmentAndCrop3dPatch(patchIn,dataSource));
```

Set Up 3-D U-Net Layers

This example uses the 3-D U-Net network [1 on page 18-0   ]. 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. A batch normalization layer is introduced before each ReLU layer. The name U-Net comes from the fact that the network can be drawn with a symmetric shape like the letter U.

Create a default 3-D U-Net network by using the `unetLayers` (Computer Vision Toolbox) function. Specify two class segmentation. Also specify valid convolution padding to avoid border artifacts when using the overlap-tile strategy for prediction of the test volumes.

```
inputPatchSize = [132 132 132 4];
numClasses = 2;
[lgraph,outPatchSize] = unet3dLayers(inputPatchSize,numClasses,'ConvolutionPadding','valid');
```

To better segment smaller tumor regions and reduce the influence of larger background regions, this example uses a `dicePixelClassificationLayer` (Computer Vision Toolbox). Replace the pixel classification layer with the Dice pixel classification layer.

```
outputLayer = dicePixelClassificationLayer('Name','Output');
lgraph = replaceLayer(lgraph,'Segmentation-Layer',outputLayer);
```

The data has already been normalized in the Preprocess Training and Validation Data on page 18-0 section of this example. Data normalization in the `image3dInputLayer` (Deep Learning Toolbox) is unnecessary, so replace the input layer with an input layer that does not have data normalization.

```
inputLayer = image3dInputLayer(inputPatchSize,'Normalization','none','Name','ImageInputLayer');
lgraph = replaceLayer(lgraph,'ImageInputLayer',inputLayer);
```

Alternatively, you can modify the 3-D U-Net network by using Deep Network Designer App from Deep Learning Toolbox™.

Plot the graph of the updated 3-D U-Net network.

```
analyzeNetwork(lgraph)
```

Specify Training Options

Train the network using the `adam` optimization solver. Specify the hyperparameter settings using the `trainingOptions` (Deep Learning Toolbox) function. The initial learning rate is set to 5e-4 and gradually decreases over the span of training. You can experiment with the `MiniBatchSize` property based on your GPU memory. To maximize GPU memory utilization, favor large input patches over a large batch size. Note that batch normalization layers are less effective for smaller values of `MiniBatchSize`. Tune the initial learning rate based on the `MiniBatchSize`.

```
options = trainingOptions('adam', ...
    'MaxEpochs',50, ...
    'InitialLearnRate',5e-4, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',5, ...
    'LearnRateDropFactor',0.95, ...
    'ValidationData',dsVal, ...
    'ValidationFrequency',400, ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'MiniBatchSize',miniBatchSize);
```

Download Pretrained Network and Sample Test Set

Optionally, download a pretrained version of 3-D U-Net and five sample test volumes and their corresponding labels from the BraTS data set [3 on page 18-0   ]. The pretrained model and sample data enable you to perform segmentation on test data without downloading the full data set or waiting for the network to train.

```
trained3DUnet_url = 'https://www.mathworks.com/supportfiles/vision/data/brainTumor3DUNetValid.ma
sampleData_url = 'https://www.mathworks.com/supportfiles/vision/data/sampleBraTSTestSetValid.tar

imageDir = fullfile(tempdir,'BraTS');
if ~exist(imageDir,'dir')
    mkdir(imageDir);
end

downloadTrained3DUnetSampleData(trained3DUnet_url,sampleData_url,imageDir);

Downloading pretrained 3-D U-Net for BraTS data set.
This will take several minutes to download...
Done.

Downloading sample BraTS test dataset.
This will take several minutes to download and unzip...
Done.
```

Train Network

After configuring the training options and the data source, train the 3-D U-Net network by using the `trainNetwork` (Deep Learning Toolbox) function. To train the network, set the `doTraining` variable 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` variable in the following code as `false`, then the example returns a pretrained 3-D U-Net network.

Note: Training takes about 30 hours on a multi-GPU system with 4 NVIDIA™ Titan Xp GPUs and can take even longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    modelDateTime = datestr(now,'dd-mmm-yyyy-HH-MM-SS');
    [net,info] = trainNetwork(dsTrain,lgraph,options);
    save(['trained3DUNetValid-' modelDateTime '-Epoch-' num2str(options.MaxEpochs) '.mat'],'net'
else
    inputPatchSize = [132 132 132 4];
    outPatchSize = [44 44 44 2];
    load(fullfile(imageDir,'trained3DUNet','brainTumor3DUNetValid.mat'));
end
```

You can now use the U-Net to semantically segment brain tumors.

Perform Segmentation of Test Data

A GPU is highly recommended for performing semantic segmentation of the image volumes (requires Parallel Computing Toolbox™).

Select the source of test data that contains ground truth volumes and labels for testing. If you keep the `useFullTestSet` variable in the following code as `false`, then the example uses five volumes for testing. If you set the `useFullTestSet` variable to `true`, then the example uses 55 test images selected from the full data set.

```
useFullTestSet = false;
if useFullTestSet
    volLocTest = fullfile(preprocessDataLoc,'imagesTest');
    lblLocTest = fullfile(preprocessDataLoc,'labelsTest');
else
    volLocTest = fullfile(imageDir,'sampleBraTSTestSetValid','imagesTest');
    lblLocTest = fullfile(imageDir,'sampleBraTSTestSetValid','labelsTest');
    classNames = ["background","tumor"];
    pixelLabelID = [0 1];
end
```

The `voldsTest` variable stores the ground truth test images. The `pxdsTest` variable stores the ground truth labels.

```
volReader = @(x) matRead(x);
voldsTest = imageDatastore(volLocTest, ...
    'FileExtensions','.mat','ReadFcn',volReader);
pxdsTest = pixelLabelDatastore(lblLocTest,classNames,pixelLabelID, ...
    'FileExtensions','.mat','ReadFcn',volReader);
```

Use the overlap-tile strategy to predict the labels for each test volume. Each test volume is padded to make the input size a multiple of the output size of the network and compensates for the effects of valid convolution. The overlap-tile algorithm selects overlapping patches, predicts the labels for each patch by using the `semanticseg` (Computer Vision Toolbox) function, and then recombines the patches.

```
id = 1;
while hasdata(voldsTest)
    disp(['Processing test volume ' num2str(id)]);

    tempGroundTruth = read(pxdsTest);
```

```
        groundTruthLabels{id} = tempGroundTruth{1};
        vol{id} = read(voldsTest);

        % Use reflection padding for the test image.
        % Avoid padding of different modalities.
        volSize = size(vol{id},(1:3));
        padSizePre  = (inputPatchSize(1:3)-outPatchSize(1:3))/2;
        padSizePost = (inputPatchSize(1:3)-outPatchSize(1:3))/2 + (outPatchSize(1:3)-mod(volSize,outF
        volPaddedPre = padarray(vol{id},padSizePre,'symmetric','pre');
        volPadded = padarray(volPaddedPre,padSizePost,'symmetric','post');
        [heightPad,widthPad,depthPad,~] = size(volPadded);
        [height,width,depth,~] = size(vol{id});

        tempSeg = categorical(zeros([height,width,depth],'uint8'),[0;1],classNames);

        % Overlap-tile strategy for segmentation of volumes.
        for k = 1:outPatchSize(3):depthPad-inputPatchSize(3)+1
            for j = 1:outPatchSize(2):widthPad-inputPatchSize(2)+1
                for i = 1:outPatchSize(1):heightPad-inputPatchSize(1)+1
                    patch = volPadded( i:i+inputPatchSize(1)-1,...
                        j:j+inputPatchSize(2)-1,...
                        k:k+inputPatchSize(3)-1,:);
                    patchSeg = semanticseg(patch,net);
                    tempSeg(i:i+outPatchSize(1)-1, ...
                        j:j+outPatchSize(2)-1, ...
                        k:k+outPatchSize(3)-1) = patchSeg;
                end
            end
        end

        % Crop out the extra padded region.
        tempSeg = tempSeg(1:height,1:width,1:depth);

        % Save the predicted volume result.
        predictedLabels{id} = tempSeg;
        id=id+1;
end
```

```
Processing test volume 1
Processing test volume 2
Processing test volume 3
Processing test volume 4
Processing test volume 5
```

Compare Ground Truth Against Network Prediction

Select one of the test images to evaluate the accuracy of the semantic segmentation. Extract the first modality from the 4-D volumetric data and store this 3-D volume in the variable vol3d.

```
volId = 1;
vol3d = vol{volId}(:,:,:,1);
```

Display in a montage the center slice of the ground truth and predicted labels along the depth direction.

```
zID = size(vol3d,3)/2;
zSliceGT = labeloverlay(vol3d(:,:,zID),groundTruthLabels{volId}(:,:,zID));
zSlicePred = labeloverlay(vol3d(:,:,zID),predictedLabels{volId}(:,:,zID));
```

```
figure
montage({zSliceGT,zSlicePred},'Size',[1 2],'BorderSize',5)
title('Labeled Ground Truth (Left) vs. Network Prediction (Right)')
```


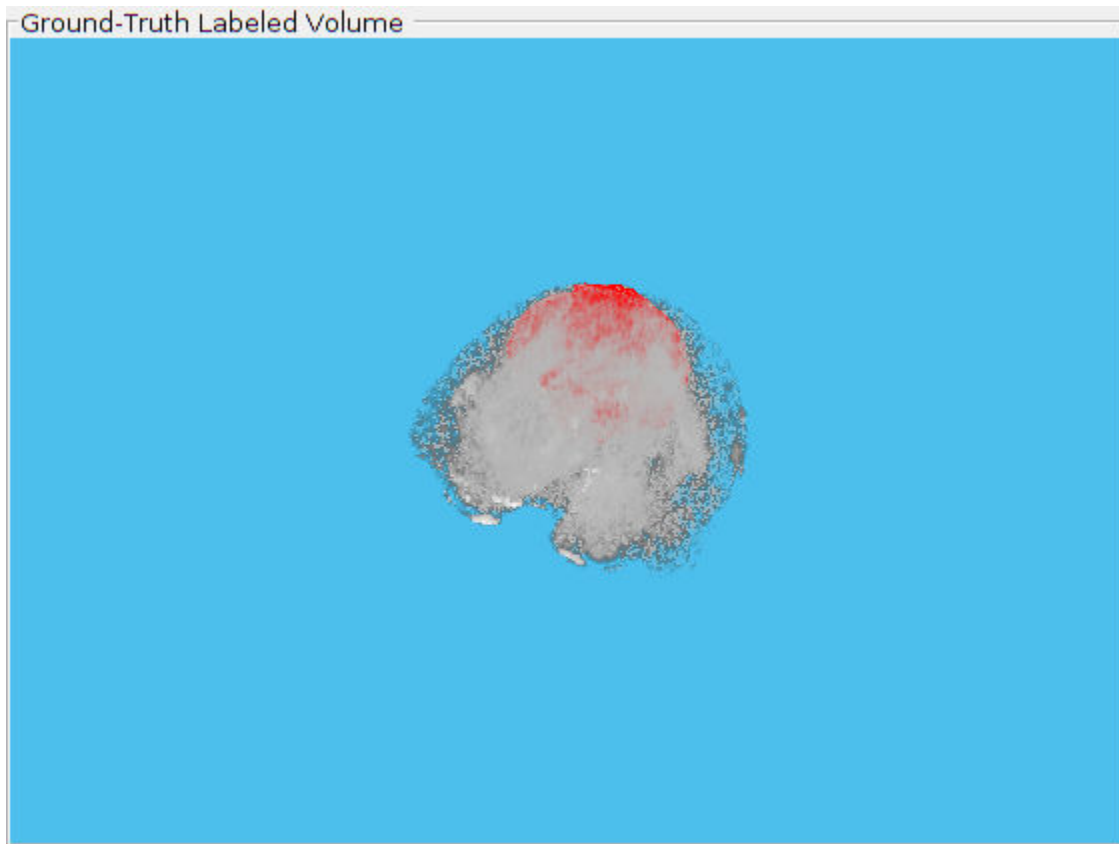Labeled Ground Truth (Left) vs. Network Prediction (Right)
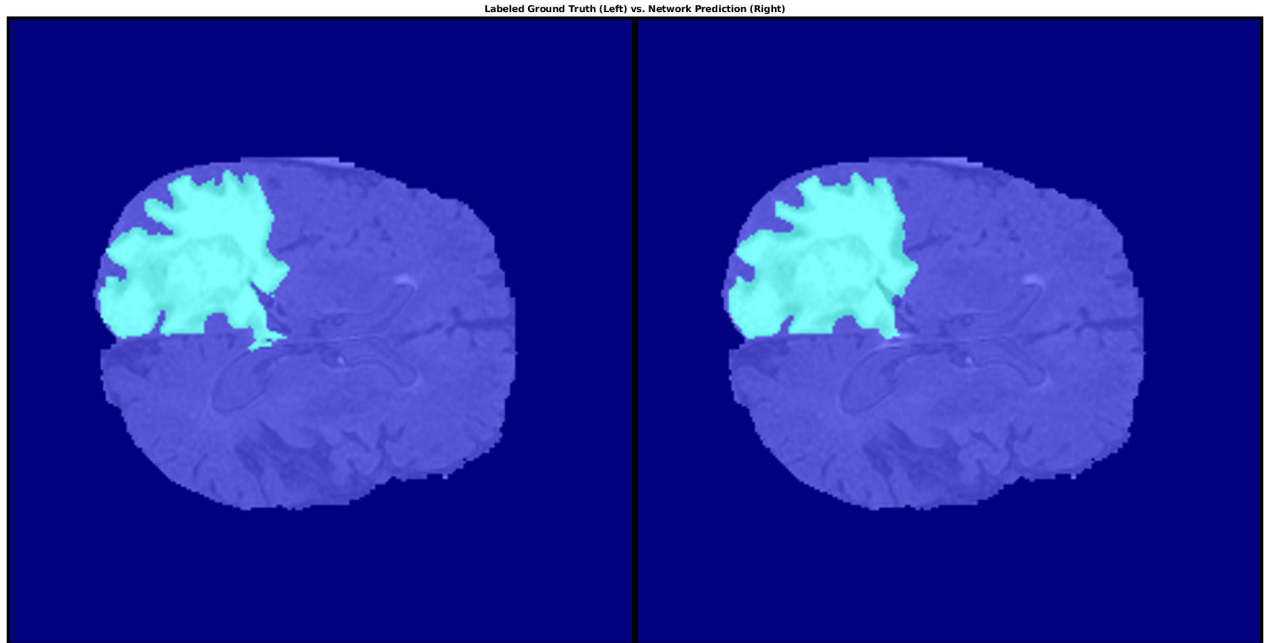
Display the ground-truth labeled volume using the `labelvolshow` function. Make the background fully transparent by setting the visibility of the background label (1) to 0. Because the tumor is inside the brain tissue, make some of the brain voxels transparent, so that the tumor is visible. To make some brain voxels transparent, specify the volume threshold as a number in the range [0, 1]. All normalized volume intensities below this threshold value are fully transparent. This example sets the volume threshold as less than 1 so that some brain pixels remain visible, to give context to the spatial location of the tumor inside the brain.

```
viewPnlTruth = uipanel(figure,'Title','Ground-Truth Labeled Volume');
hTruth = labelvolshow(groundTruthLabels{volId},vol3d,'Parent',viewPnlTruth, ...
    'LabelColor',[0 0 0;1 0 0],'VolumeThreshold',0.68);
hTruth.LabelVisibility(1) = 0;
```

For the same volume, display the predicted labels.

```
viewPnlPred = uipanel(figure,'Title','Predicted Labeled Volume');
hPred = labelvolshow(predictedLabels{volId},vol3d,'Parent',viewPnlPred, ...
    'LabelColor',[0 0 0;1 0 0],'VolumeThreshold',0.68);
```

```
hPred.LabelVisibility(1) = 0;
```
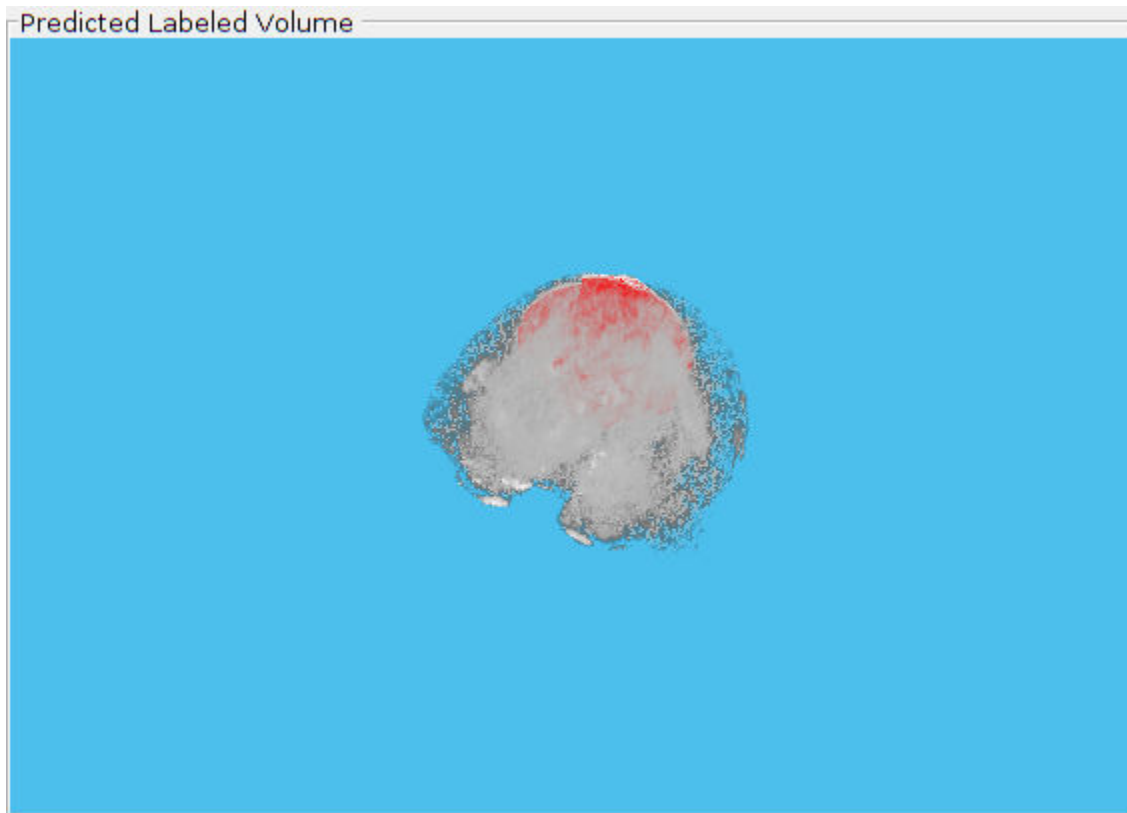
This image shows the result of displaying slices sequentially across the one of the volume. The labeled ground truth is on the left and the network prediction is on the right.
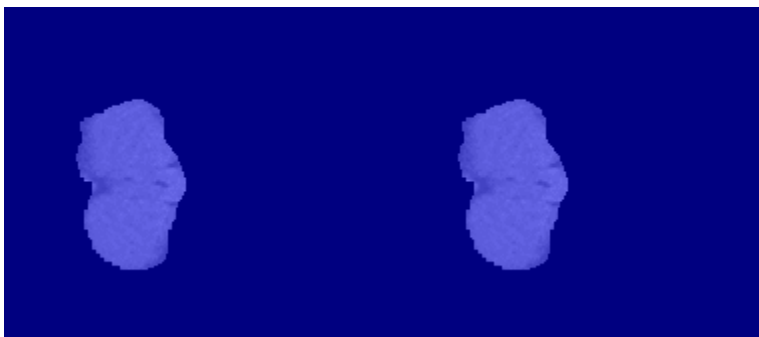
Quantify Segmentation Accuracy

Measure the segmentation accuracy using the `dice` function. This function computes the Dice similarity coefficient between the predicted and ground truth segmentations.

```
diceResult = zeros(length(voldsTest.Files),2);

for j = 1:length(vol)
    diceResult(j,:) = dice(groundTruthLabels{j},predictedLabels{j});
end
```

Calculate the average Dice score across the set of test volumes.

```
meanDiceBackground = mean(diceResult(:,1));
disp(['Average Dice score of background across ',num2str(j), ...
    ' test volumes = ',num2str(meanDiceBackground)])
```

```
Average Dice score of background across 5 test volumes = 0.9993
```

```
meanDiceTumor = mean(diceResult(:,2));
disp(['Average Dice score of tumor across ',num2str(j), ...
    ' test volumes = ',num2str(meanDiceTumor)])
```

```
Average Dice score of tumor across 5 test volumes = 0.9585
```

The figure shows a `boxplot` (Statistics and Machine Learning Toolbox) that visualizes statistics about the Dice scores across the set of five sample test volumes. The red lines in the plot show the median Dice value for the classes. The upper and lower bounds of the blue box indicate the 25th and 75th percentiles, respectively. Black whiskers extend to the most extreme data points not considered outliers.
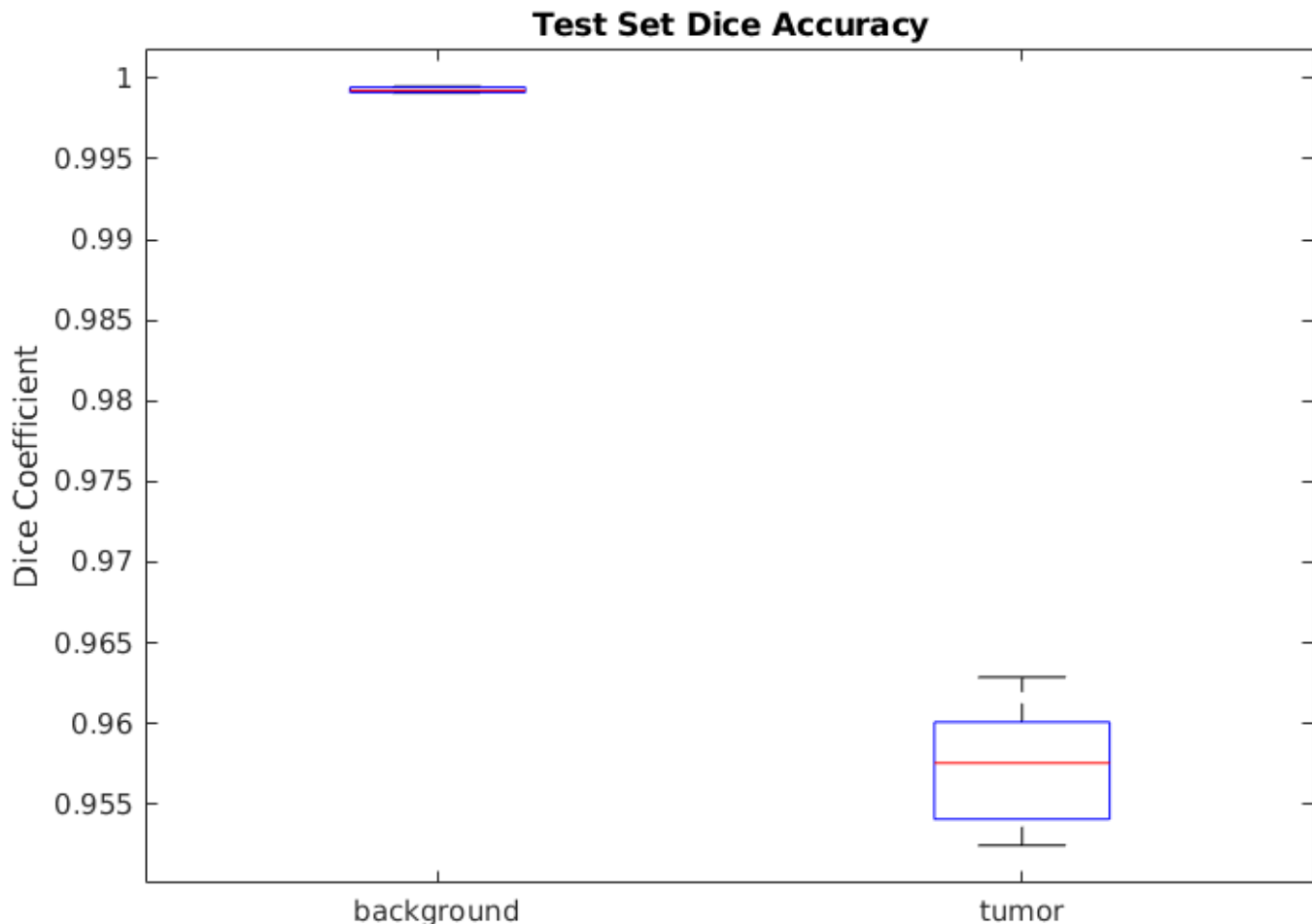
If you have Statistics and Machine Learning Toolbox™, then you can use the `boxplot` function to visualize statistics about the Dice scores across all your test volumes. To create a `boxplot`, set the `createBoxplot` variable in the following code to `true`.

```
createBoxplot = false;
if createBoxplot
```

```
    figure
    boxplot(diceResult)
    title('Test Set Dice Accuracy')
    xticklabels(classNames)
    ylabel('Dice Coefficient')
end
```

References

[1] Çiçek, Ö., A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger. "3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation." In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention - MICCAI 2016*. Athens, Greece, Oct. 2016, pp. 424-432.

[2] Isensee, F., P. Kickingereder, W. Wick, M. Bendszus, and K. H. Maier-Hein. "Brain Tumor Segmentation and Radiomics Survival Prediction: Contribution to the BRATS 2017 Challenge." In *Proceedings of BrainLes: International MICCAI Brainlesion Workshop*. Quebec City, Canada, Sept. 2017, pp. 287-297.

[3] "Brain Tumours". *Medical Segmentation Decathlon.* http://medicaldecathlon.com/

The BraTS dataset is provided by Medical Segmentation Decathlon under the CC-BY-SA 4.0 license. All warranties and representations are disclaimed; see the license for details. MathWorks® has modified the data set linked in the Download Pretrained Network and Sample Test Set on page 18-0   section of this example. The modified sample dataset has been cropped to a region containing primarily the brain and tumor and each channel has been normalized independently by subtracting the mean and dividing by the standard deviation of the cropped brain region.

[4] Sudre, C. H., W. Li, T. Vercauteren, S. Ourselin, and M. J. Cardoso. "Generalised Dice Overlap as a Deep Learning Loss Function for Highly Unbalanced Segmentations." *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support: Third International Workshop*. Quebec City, Canada, Sept. 2017, pp. 240-248.

[5] Ronneberger, O., P. Fischer, and T. Brox. "U-Net:Convolutional Networks for Biomedical Image Segmentation." In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015*. Munich, Germany, Oct. 2015, pp. 234-241. Available at arXiv:1505.04597.

See Also
dicePixelClassificationLayer | imageDatastore | pixelLabelDatastore | randomPatchExtractionDatastore | semanticseg | trainNetwork | trainingOptions | transform

More About
- "Preprocess Volumes for Deep Learning" (Deep Learning Toolbox)
- "Datastores for Deep Learning" (Deep Learning Toolbox)
- "List of Deep Learning Layers" (Deep Learning Toolbox)

External Websites
- 3-D Brain Tumor Segmentation Using Deep Learning Video Tutorial

Deep Learning Classification of Large Multiresolution Images

This example shows how to train an Inception-v3 deep neural network to classify multiresolution whole slide images (WSIs) that do not fit in memory.

The example shows how to train an Inception-v3 tumor classification network and also provides a pretrained Inception-v3 network. If you choose to train the Inception-v3 network, use of a CUDA capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended. Use of a GPU requires Parallel Computing Toolbox™.

Introduction

The only definitive way to diagnose breast cancer is by examining tissue samples collected from biopsy or surgery. The samples are commonly prepared with hematoxylin and eosin (H&E) staining to increase the contrast of structures in the tissue. Traditionally, pathologists examine the tissue on glass slides under a microscope to detect tumor tissue. Diagnosis takes time as pathologists must thoroughly inspect an entire slide at close magnification. Further, pathologists may not notice small tumors. Deep learning methods aim to automate the detection of tumor tissue, saving time and improving the detection rate of small tumors.

Deep learning methods for tumor classification rely on digital pathology, in which whole tissue slides are imaged and digitized. The resulting WSIs have extremely high resolution. WSIs are frequently stored in a multiresolution file to facilitate the display, navigation, and processing of the images.

Reading WSIs is a challenge because the images cannot be loaded as a whole into memory and therefore require out-of-core image processing techniques. Image Processing Toolbox™ `bigimage` objects can store and process this type of large multiresolution image. `bigimageDatastore` objects can prepare training patches from a `bigimage` to feed into a neural network.

This example shows how to train a deep learning network to classify tumors in very large multiresolution images using `bigimage` and `bigimageDatastore`. The example presents classification results as heatmaps that depict the probability that local tissue is tumorous. The localization of tumor regions enables medical pathologists to investigate specific regions and quickly identify tumors of any size in an image.

Download Training Data

This example uses WSIs from the Camelyon16 challenge [1 on page 18-0]. The data from this challenge contains a total of 400 WSIs of lymph nodes from two independent sources, separated into 270 training images and 130 test images. The WSIs are stored as TIF files in a stripped format with an 11-level pyramid structure.

`bigimage` objects warn when images contain stripped formats because this format is slow to process. To avoid displaying warnings in the command window, turn off the warning while the example runs and restore the warning state on completion.

```
warningState = warning('off','images:bigimage:singleStripTiff');
c = onCleanup(@()warning(warningState));
```

The training data set consists of 159 WSIs of normal lymph nodes and 111 WSIs of lymph nodes with tumor and healthy tissue. Usually, the tumor tissue is a small fraction of the healthy tissue. Ground truth coordinates of the lesion boundaries accompany the tumor images.

The size of each training file is approximately 2 GB. If you do not want to download the training data set or train the network, then go directly to the Download and Preprocess Testing Data on page 18-0   section in this example.

Create a directory to store the training data set.

```
trainingImageDir = fullfile(tempdir,'Camelyon16','training');
if ~exist(trainingImageDir,'dir')
    mkdir(trainingImageDir);
    mkdir(fullfile(trainingImageDir,'normal'));
    mkdir(fullfile(trainingImageDir,'tumor'));
    mkdir(fullfile(trainingImageDir,'lesion_annotations'));
end

trainNormalDataDir = fullfile(trainingImageDir,'normal');
trainTumorDataDir = fullfile(trainingImageDir,'tumor');
trainTumorAnnotationDir = fullfile(trainingImageDir,'lesion_annotations');
```

To download the training data, go to the Camelyon17 website and click the first "CAMELYON16 data set" link. Open the "training" directory, then follow these steps.

- Download the "lesion_annotations.zip" file. Extract the files to the directory specified by the `trainTumorAnnotationDir` variable.
- Open the "normal" directory. Download the images to the directory specified by the `trainNormalDataDir` variable.
- Open the "tumor" directory. Download the images to the directory specified by the `trainTumorDataDir` variable.

Specify the number of training images. Note that one of the training images of normal tissue, 'normal_144.tif', has metadata that cannot be read by the `bigimage` object. This example uses the remaining 158 training files.

```
numNormalFiles = 158;
numTumorFiles = 111;
```

Visualize Training Data

To get a better understanding of the training data, display one training image. Because loading the entire image into memory at the finest resolution is not possible, you cannot use traditional image display functions such as `imshow`. To display and process the image data, use a `bigimage` object.

Create a `bigimage` object from a tumor training image.

```
tumorFileName = fullfile(trainTumorDataDir,'tumor_001.tif');
tumorImage = bigimage(tumorFileName);
```

Inspect the dimensions of the `bigimage` at each resolution level. Level 1 has the most pixels and is the finest resolution level. Level 10 has the fewest pixels and is the coarsest resolution level. The aspect ratio is not consistent, which indicates that levels do not all span the same world area.

```
levelSizeInfo = table((1:length(tumorImage.LevelSizes))', ...
    tumorImage.LevelSizes(:,1), ...
    tumorImage.LevelSizes(:,2), ...
    tumorImage.LevelSizes(:,1)./tumorImage.LevelSizes(:,2), ...
    'VariableNames',["Resolution Level" "Image Width" "Image Height" "Aspect Ratio"])
```

```
levelSizeInfo=11×4 table
    Resolution Level     Image Width     Image Height     Aspect Ratio
    _____     _____     _____     _____

           1             2.2118e+05         97792           2.2618
           2             1.1059e+05         49152           2.25
           3                55296           24576           2.25
           4                27648           12288           2.25
           5                13824            6144           2.25
           6                 7168            3072           2.3333
           7                 3584            1536           2.3333
           8                 2048            1024           2
           9                 1024             512           2
          10                  512             512           1
          11                 1577            3629           0.43455
```

Display the `bigimage` at a coarse resolution level by using the `bigimageshow` function. Return a handle to the `bigimageshow` object. You can use the handle to adjust the display. The image contains a lot of empty white space. The tissue occupies only a small portion of the image.

```
h = bigimageshow(tumorImage,'ResolutionLevel',7);
```
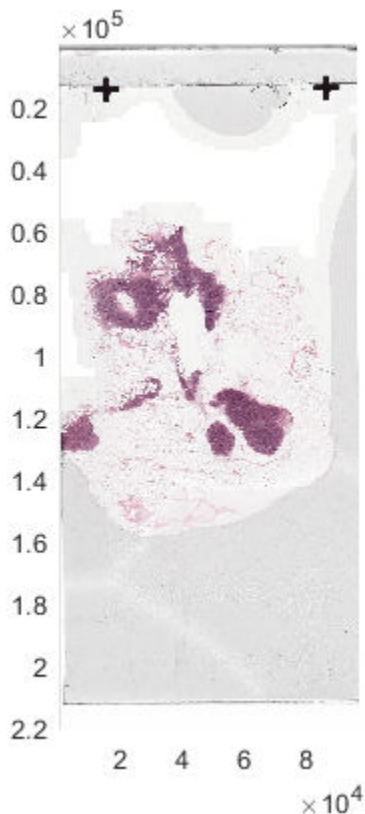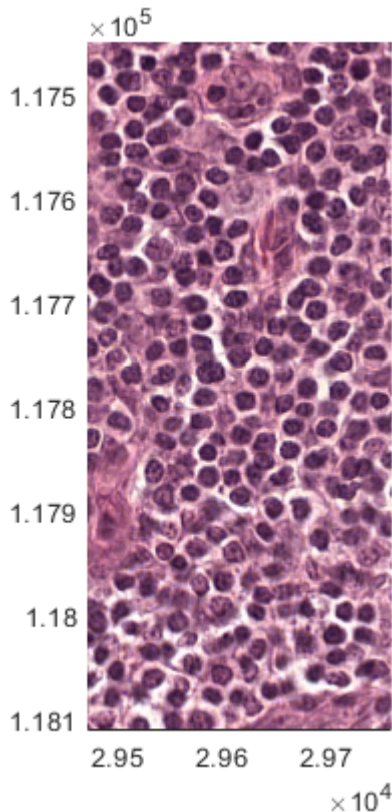

Zoom in on one part of the image by setting the horizontal and vertical spatial extents with respect to the finest resolution level. The image looks blurry because this resolution level is very coarse.

```
xlim([29471,29763]);
ylim([117450,118110]);
```

To see more details, change the resolution level to a finer level.

```
h.ResolutionLevel = 1;
```


Create Masks

You can reduce the amount of computation by processing only regions of interest (ROIs). Use a mask to define ROIs. A mask is a logical image in which `true` pixels represent the ROI.

To further reduce the amount of computation, create masks at a coarse resolution level that can be processed entirely in memory instead of on a block-by-block basis. If the spatial referencing of the coarse resolution level matches the spatial referencing of finer resolution levels, then locations at the coarse level correspond to locations in finer levels. In this case, you can use the coarse mask to select which blocks to process at the finer levels. For more information, see "Set Spatial Referencing for Big Images" on page 16-2 and "Process Big Images Efficiently Using Mask" on page 16-17.

Specify a resolution level to use for creating the mask. This example uses resolution level 7, which is coarse and fits in memory.

```
resolutionLevel = 7;
```

Create Masks for Normal Images

In normal images, the ROI consists of healthy tissue. The color of healthy tissue is distinct from the color of the background, so use color thresholding to segment the image and create an ROI. The L*a*b* color space provides the best color separation for segmentation. Convert the image to the L*a*b* color space, then threshold the a* channel to create the tissue mask.

You can use the helper function `createMaskForNormalTissue` to create masks using color thresholding. This helper function is attached to the example as a supporting file.

The helper function performs these operations on each training image of normal tissue:

- Create a `bigimage` object from the TIF image file.
- Set the spatial referencing of all resolution levels from the image metadata.
- Get the image at the coarse resolution level.
- Convert the coarse image to the L*a*b* color space, then extract the a* channel.
- Create a binary image by thresholding the image using Otsu's method, which minimizes the intraclass variance between the black and white pixels.
- Create a single-resolution `bigimage` object from the mask and set the spatial referencing of the mask to match the spatial referencing of the input image.
- Write the mask `bigimage` to memory. Only the `bigimage` object is in memory. The individual image blocks corresponding to the logical mask image are in a temporary directory. Writing to a directory preserves the custom spatial referencing, which ensures that the normal images and their corresponding mask images have the same spatial referencing.

```
trainNormalMaskDir = fullfile(trainNormalDataDir,['normal_mask_level' num2str(resolutionLevel)])
createMaskForNormalTissue(trainNormalDataDir,trainNormalMaskDir,resolutionLevel)
```

Now that both normal images and masks are on disk, create `bigimage` objects to manage the data by using the helper function `createBigImageAndMaskArrays`. This function creates an array of `bigimage` objects from the normal images and a corresponding array of `bigimage` objects from the normal mask images. The helper function is attached to the example as a supporting file.

```
[bigNormalImages,bigNormalMasks] = createBigImageAndMaskArrays(trainNormalDataDir,trainNormalMask
```

Select a sample normal image and mask. Confirm that the spatial world extents of the mask match the extents of the image at the finest resolution level. The spatial world extents are specified by the `XWorldLimits` and `YWorldLimits` properties.

```
idx = 2;
bigNormalImages(idx).SpatialReferencing(1)

ans =
  imref2d with properties:

           XWorldLimits: [0 97792]
           YWorldLimits: [0 221184]
              ImageSize: [221184 97792]
     PixelExtentInWorldX: 1
     PixelExtentInWorldY: 1
     ImageExtentInWorldX: 97792
     ImageExtentInWorldY: 221184
        XIntrinsicLimits: [0.5000 9.7793e+04]
        YIntrinsicLimits: [0.5000 2.2118e+05]


bigNormalMasks(idx).SpatialReferencing(1)

ans =
  imref2d with properties:

           XWorldLimits: [0 97792]
```

```
        YWorldLimits: [0 221184]
           ImageSize: [3456 1527]
   PixelExtentInWorldX: 64.0419
   PixelExtentInWorldY: 64
   ImageExtentInWorldX: 97792
   ImageExtentInWorldY: 221184
      XIntrinsicLimits: [0.5000 1.5275e+03]
      YIntrinsicLimits: [0.5000 3.4565e+03]
```

Verify that the mask contains the correct ROIs and spatial referencing. Display the sample image by using the `bigimageshow` function. Get the axes containing the display.

```
figure
hBigNormal = bigimageshow(bigNormalImages(idx));
hNormalAxes = hBigNormal.Parent;
```

Create a new axes on top of the displayed `bigimage`. In the new axes, display the corresponding mask image with partial transparency. The mask highlights the regions containing normal tissue.

```
hMaskAxes = axes;
hBigMask = bigimageshow(bigNormalMasks(idx),'Parent',hMaskAxes, ...
    'Interpolation','nearest','AlphaData',0.5);
hMaskAxes.Visible = 'off';
```
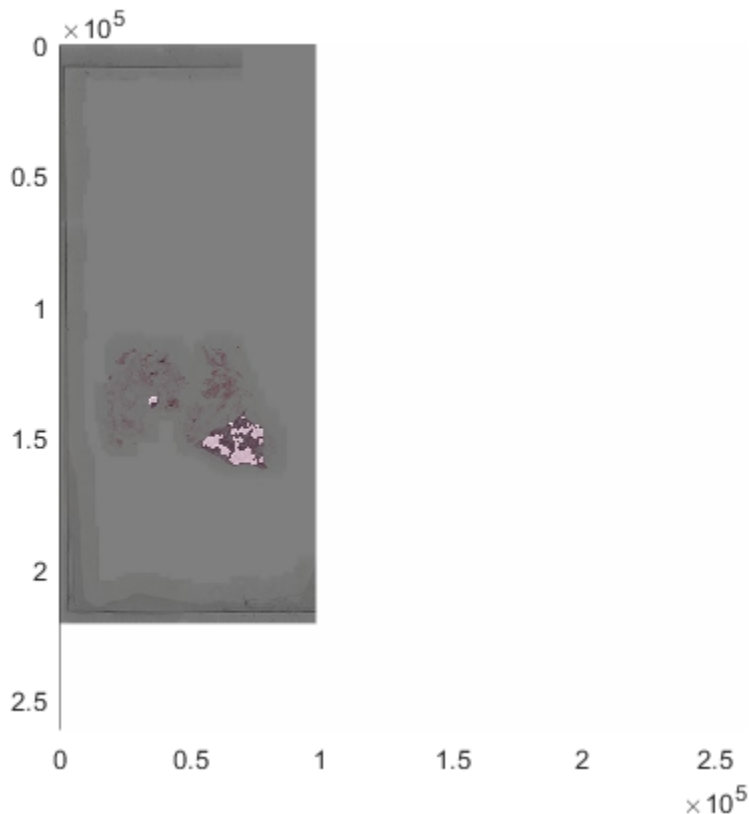
Link the axes of the image with the axes of the mask. When you zoom and pan, both axes are updated identically.
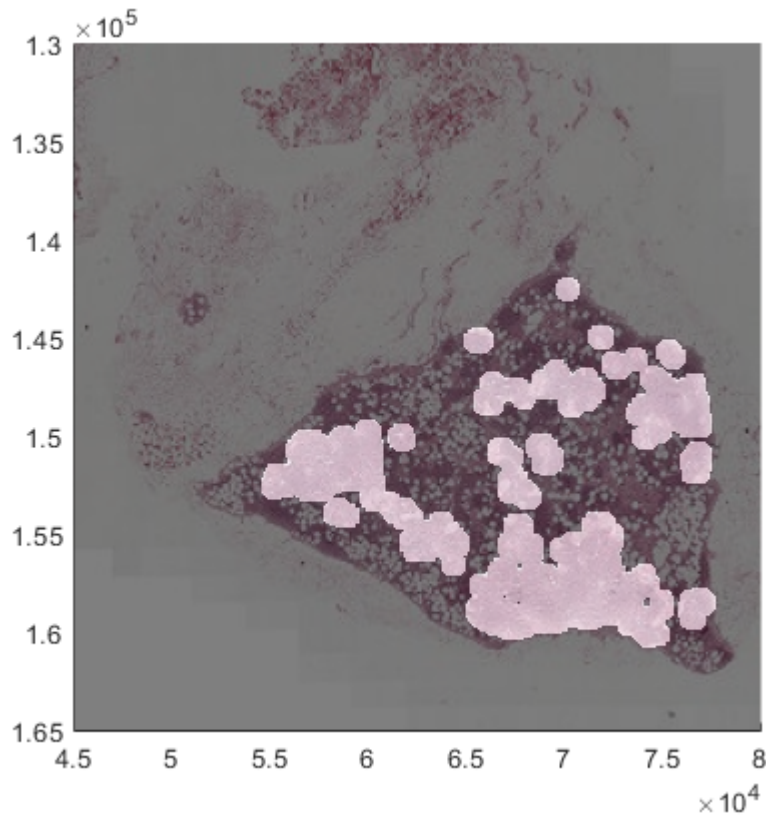
```
linkaxes([hNormalAxes,hMaskAxes]);
```

Zoom in on one part of the image by setting the horizontal and vertical spatial extents. The mask overlaps the normal tissue correctly.

```
xlim([45000 80000]);
ylim([130000 165000]);
```


Create Masks for Tumor Images

In tumor images, the ROI consists of tumor tissue. The color of tumor tissue is similar to the color of healthy tissue, so you cannot use color segmentation techniques. Instead, create ROIs by using the ground truth coordinates of the lesion boundaries that accompany the tumor images.

You can use the helper function `createMaskForTumorTissue` to create a mask using ROIs. This helper function is attached to the example as a supporting file.

The helper function performs these operations on each training image of tumor tissue:

* Create a `bigimage` object from the TIF image file.
* Set the spatial referencing from the image metadata.
* Read the corresponding lesion annotations in XML files and convert the annotations to polygons (`Polygon` objects).
* For each image block, use the polygon data to create a mask for the corresponding block. Images with tumor regions can contain some normal regions within them. Use the annotations of normal tissue to exclude those regions.
* Create an output logical mask `bigimage` object at a coarser resolution level. Write the mask image on a block-by-block basis by using the `setBlock` function.

- Write the mask `bigimage` object to a directory in memory. Only the `bigimage` object is in memory. The individual image blocks corresponding to the logical mask image are in a temporary directory. Writing to a directory preserves the custom spatial referencing, which ensures that the tumor images and their corresponding mask images have the same spatial referencing.

```
trainTumorMaskDir = fullfile(trainTumorDataDir,['tumor_mask_level' num2str(resolutionLevel)]);
createMaskForTumorTissue(trainTumorDataDir,trainTumorAnnotationDir, ...
    trainTumorMaskDir,resolutionLevel);
```

Now that both tumor images and masks are on disk, create `bigimage` objects to manage the data by using the helper function `createBigImageAndMaskArrays`. This function creates an array of `bigimage` objects from the tumor images and a corresponding array of `bigimage` objects from the tumor mask images. The helper function is attached to the example as a supporting file.

```
[bigTumorImages,bigTumorMasks] = createBigImageAndMaskArrays(trainTumorDataDir,trainTumorMaskDir
```

Select a sample tumor image and mask. Confirm that the spatial world extents of the mask match the extents of the image at the finest resolution level. The spatial world extents are specified by the `XWorldLimits` and `YWorldLimits` properties.

```
idx = 5;
bigTumorImages(idx).SpatialReferencing(1)

ans =
  imref2d with properties:

           XWorldLimits: [0 97792]
           YWorldLimits: [0 219648]
              ImageSize: [219648 97792]
      PixelExtentInWorldX: 1
      PixelExtentInWorldY: 1
      ImageExtentInWorldX: 97792
      ImageExtentInWorldY: 219648
         XIntrinsicLimits: [0.5000 9.7793e+04]
         YIntrinsicLimits: [0.5000 2.1965e+05]


bigTumorMasks(idx).SpatialReferencing(1)

ans =
  imref2d with properties:

           XWorldLimits: [0 97792]
           YWorldLimits: [0 219648]
              ImageSize: [3432 1527]
      PixelExtentInWorldX: 64.0419
      PixelExtentInWorldY: 64
      ImageExtentInWorldX: 97792
      ImageExtentInWorldY: 219648
         XIntrinsicLimits: [0.5000 1.5275e+03]
         YIntrinsicLimits: [0.5000 3.4325e+03]
```

Verify that the mask contains the correct ROIs and spatial referencing. Display the sample image by using the `bigimageshow` function. Get the axes containing the display.

```
figure
hBigTumor = bigimageshow(bigTumorImages(idx));
hTumorAxes = hBigTumor.Parent;
```

Create a new axes on top of the displayed big image. In the new axes, display the corresponding mask image with partial transparency. The mask highlights the regions containing normal tissue.

```
hMaskAxes = axes;
hBigMask = bigimageshow(bigTumorMasks(idx),'Parent',hMaskAxes, ...
    'Interpolation','nearest','AlphaData',0.5);
hMaskAxes.Visible = 'off';
```
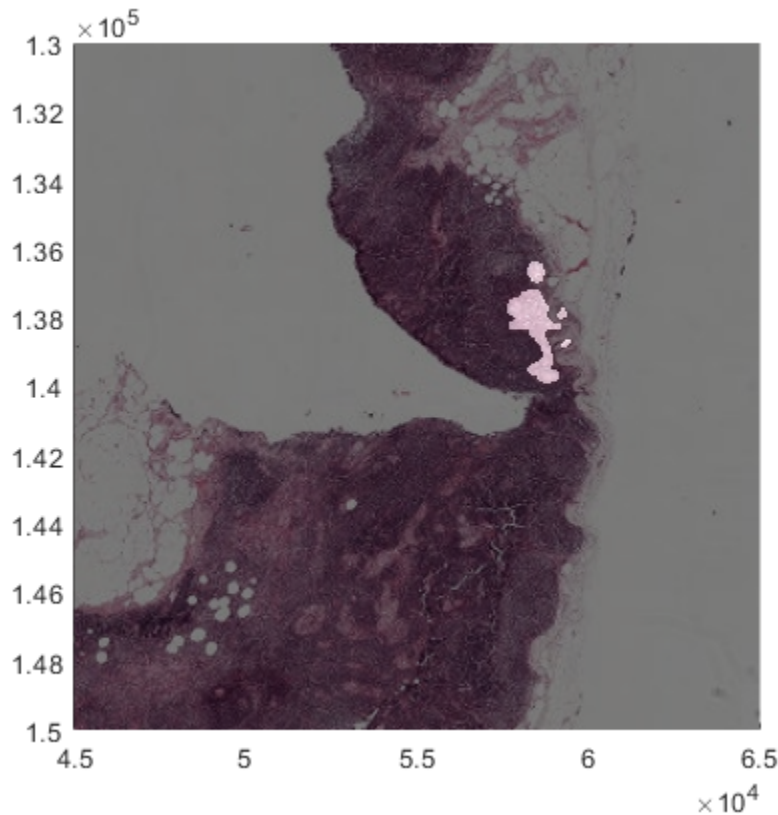
Link the axes of the image with the axes of the mask. When you zoom and pan, both axes are updated identically.

```
linkaxes([hTumorAxes,hMaskAxes]);
```
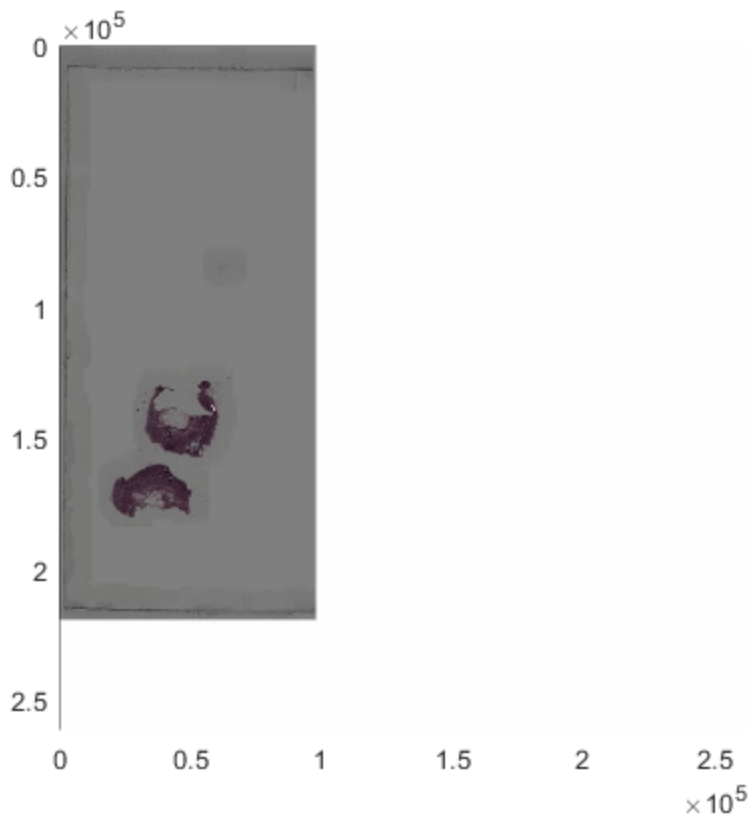

Zoom in on one part of the image by setting the horizontal and vertical spatial extents. The mask overlaps the tumor tissue correctly.

```
xlim([45000 65000]);
ylim([130000 150000]);
```

Create Big Image Datastores for Training and Validation

To extract patches of training data from the `bigimage` objects, use a `bigimageDatastore`. This datastore reads patches of `bigimage` data at a single resolution level.

Color imbalance and class imbalance in the raw training patches can potentially bias the network. Color imbalance results from nonuniform color staining of the tissue. Class imbalance results from an unequal amount of tumor and normal tissue in the data. To correct these imbalances, you can preprocess and augment the datastore.

This example shows how to create a `bigimageDatastore` that extracts tumor and normal patches for training the network. The example also shows how to preprocess and augment datastores to avoid biasing the network.

Select Location of Normal Tissue Patches to Read

Randomly split the normal images and corresponding masks into two sets. The validation set contains two randomly selected images and corresponding masks. The training set contains the remaining images and masks.

```
normalValidationIdx = randi(numNormalFiles,[1 2]);
normalTrainIdx = setdiff(1:numNormalFiles,normalValidationIdx);
```

The patch size is small compared to the size of features in the image. By default, a `bigimageDatastore` extracts patches with no overlap and no gap, which generates a huge quantity of training patches. You can reduce the amount of training data by specifying a subset of patches. Specify the coordinates of patches using the `selectBlockLocations` function. Add a gap between

the sampled training patches using the `BlockOffsets` name-value argument. Specify an offset that is larger than the patch size. Increase the inclusion threshold from the default value of 0.5 so that the network trains on relatively homogenous patches.

```
patchSize = [299,299,3];
normalStrideFactor = 10;
blsNormalData = selectBlockLocations(bigNormalImages(normalTrainIdx), ...
    "BlockSize",patchSize(1:2),"BlockOffsets",patchSize(1:2)*normalStrideFactor, ...
    "Masks",bigNormalMasks(normalTrainIdx),"InclusionThreshold",0.75,"ExcludeIncompleteBlocks",tr
```

Select the location of validation patches to read. Because there are fewer validation images, you do not need to add a gap between patches.

```
blsNormalDataValidation = selectBlockLocations(bigNormalImages(normalValidationIdx), ...
    "BlockSize",patchSize(1:2), ...
    "Masks",bigNormalMasks(normalValidationIdx),"InclusionThreshold",0.75,"ExcludeIncompleteBlock
```

Create Datastores for Normal Images

Create datastores `dsNormalData` and `dsNormalDataValidation` that read image patches from normal images at the finest resolution level for training and validation, respectively. Specify the coordinates of patches using the `BlockLocationSet` name-value pair argument.

```
dsNormalData = bigimageDatastore(bigNormalImages(normalTrainIdx), ...
    "BlockLocationSet",blsNormalData);
dsNormalDataValidation = bigimageDatastore(bigNormalImages(normalValidationIdx), ...
    "BlockLocationSet",blsNormalDataValidation);
```

Preview patches from the datastore containing normal training images.

```
imagesToPreview = zeros([patchSize 10],'uint8');
for n = 1:10
    im = read(dsNormalData);
    imagesToPreview(:,:,:,n) = im{1};
end
figure
montage(imagesToPreview,'Size',[2 5],'BorderSize',10,'BackgroundColor','k');
title("Training Patches of Normal Tissue")
```


Training Patches of Normal Tissue
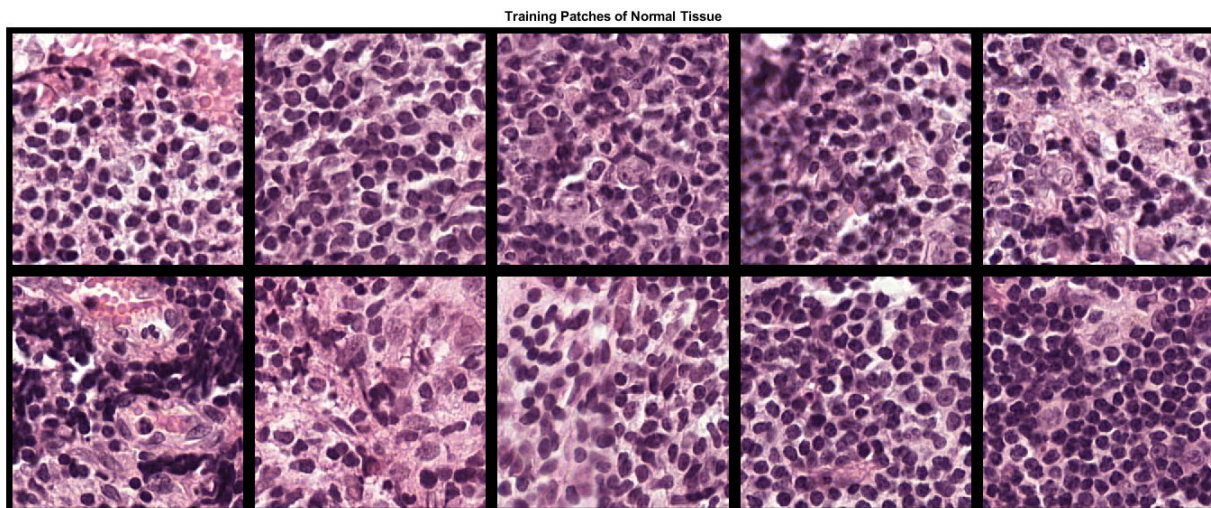
Select Location of Tumor Tissue Patches to Read

Randomly split the tumor images and corresponding masks into two sets. The validation set contains two randomly selected images and corresponding masks. The training set contains the remaining images and masks.

```
tumorValidationIdx = randi(numTumorFiles,[1 2]);
tumorTrainIdx = setdiff(1:numTumorFiles,tumorValidationIdx);
```

Specify the coordinates of patches to read using the `selectBlockLocations` function. Tumor tissue is more sparse than normal tissue, so increase the sampling density by specifying a smaller block offset than for normal tissue. Note that if you want to train using fewer training images, then you might need to increase the size of the training set by decreasing the block offset even further.

```
tumorStrideFactor = 4;
blsTumorData = selectBlockLocations(bigTumorImages(tumorTrainIdx), ...
    "BlockSize",patchSize(1:2),"BlockOffsets",patchSize(1:2)*tumorStrideFactor, ...
    "Masks",bigTumorMasks(tumorTrainIdx),"InclusionThreshold",0.75,"ExcludeIncompleteBlocks",true
```

Select the location of validation patches to read. Because there are fewer validation images, you do not need to add a gap between patches.

```
blsTumorDataValidation = selectBlockLocations(bigTumorImages(tumorValidationIdx), ...
    "BlockSize",patchSize(1:2), ...
    "Masks",bigTumorMasks(tumorValidationIdx),"InclusionThreshold",0.75,"ExcludeIncompleteBlocks"
```

Create Datastores for Tumor Images

Create a `bigimageDatastore` from the training tumor images and masks. The datastores `dsTumorData` and `dsTumorDataValidation` read image patches from tumor images at the finest resolution level for training and validation, respectively.

```
dsTumorData = bigimageDatastore(bigTumorImages(tumorTrainIdx), ...
    "BlockLocationSet",blsTumorData);
dsTumorDataValidation = bigimageDatastore(bigTumorImages(tumorValidationIdx), ...
    "BlockLocationSet",blsTumorDataValidation);
```

Preview patches from the datastore containing tumor training images.

```
imagesToPreview = zeros([patchSize 10],'uint8');
for n = 1:10
    im = read(dsTumorData);
    imagesToPreview(:,:,:,n) = im{1};
end
montage(imagesToPreview,'Size',[2 5],'BorderSize',10,'BackgroundColor','k');
title("Training Patches of Tumor Tissue")
```

Training Patches of Tumor Tissue
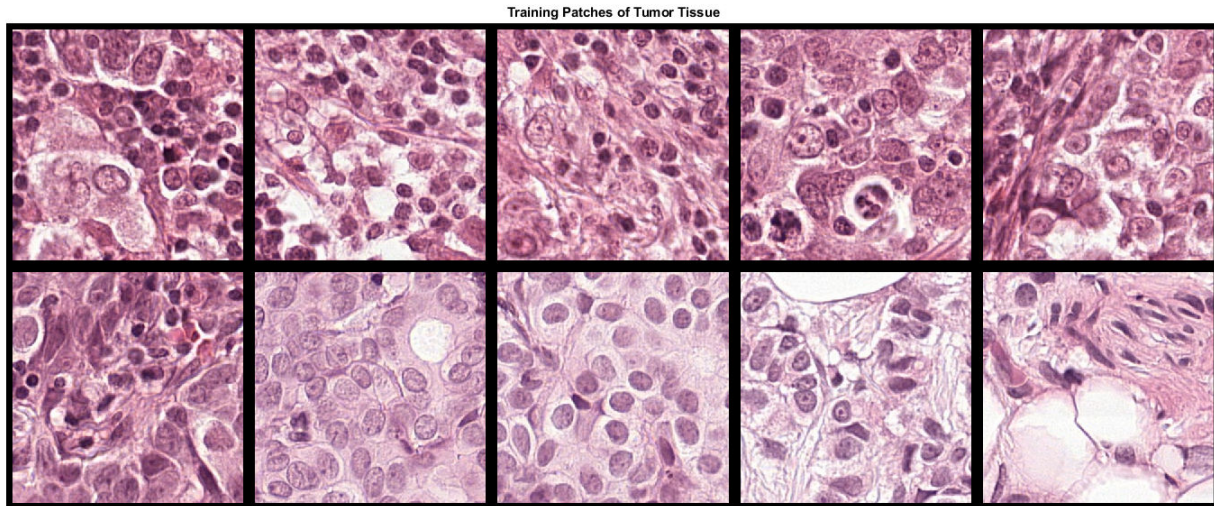
Normalize Colors and Augment Training Data

The training images have different color distributions because the data set came from different sources and color staining the tissue does not result in identically stained images. Additional preprocessing is necessary to avoid biasing the network.

To prevent color variability, this example preprocesses the data with standard stain normalization techniques. Apply stain normalization and augmentation by using the `transform` function with custom preprocessing operations specified by the helper function `augmentAndLabel`. This function is attached to the example as a supporting file.

The `augmentAndLabel` function performs these operations:

- Normalize staining by using the `normalizeStaining.m` function [4 on page 18-0   ]. Stain normalization is performed using Macenko's method, which separates H&E color channels by color deconvolution using a fixed matrix and then recreates the normalized images with individual corrected mixing. The function returns the normalized image as well as the H&E images.
- Add color jitter by using the `jitterColorHSV` function. Color jitter varies the color of each patch by perturbing the image contrast, hue, saturation, and brightness. Color jitter is performed in the HSV color space to avoid unwanted color artifacts in the RGB image.
- Apply random combinations of 90 degree rotations and vertical and horizontal reflection. Randomized affine transformations make the network agnostic to the orientation of the input image data.
- Label the patch as `'normal'` or `'tumor'`.

Each image patch generates five augmented and labeled patches: the stain-normalized patch, the stain-normalized patch with color jitter, the stain-normalized patch with color jitter and random affine transformation, the hematoxylin image with random affine transformation, and the eosin image with random affine transformation.

Create datastores that transform the normal training and validation images and label the generated patches as `'normal'`.

```
dsLabelledNormalData = transform(dsNormalData, ...
    @(x,info)augmentAndLabel(x,info,'normal'),'IncludeInfo',true);
```

```
dsLabelledNormalDataValidation = transform(dsNormalDataValidation, ...
    @(x,info)augmentAndLabel(x,info,'normal'),'IncludeInfo',true);
```

Create datastores that transform the tumor training and validation images and label the generated patches as `'tumor'`.

```
dsLabelledTumorData = transform(dsTumorData, ...
    @(x,info)augmentAndLabel(x,info,'tumor'),'IncludeInfo',true);
dsLabelledTumorDataValidation = transform(dsTumorDataValidation, ...
    @(x,info)augmentAndLabel(x,info,'tumor'),'IncludeInfo',true);
```

Balance Tumor and Normal Classes

The amount of cancer tissue in the tumor images is very small compared to the amount of normal tissue. Additional preprocessing is necessary to avoid training the network on class-imbalanced data containing a large amount of normal tissue and a very small amount of tumor tissue.

To prevent class imbalance, this example defines a custom datastore called a `randomSamplingDatastore` that randomly selects normal and tumor training patches in a balanced way. The script to define this custom datastore is attached to the example as a supporting file. For more information, see "Develop Custom Datastore".

Create the custom `randomSamplingDatastore` from the normal and tumor training datastores. The random sampling datastore `dsTrain` provides mini-batches of training data to the network at each iteration of the epoch.

```
dsTrain = randomSamplingDatastore(dsLabelledTumorData,dsLabelledNormalData);
```

To limit the number of patches used during validation, this example defines a custom datastore called a `validationDatastore` that returns five validation patches from each class. The script to define this custom datastore is attached to the example as a supporting file.

Create the custom `validationDatastore` from the normal and tumor validation datastores.

```
numValidationPatchesPerClass = 5;
dsValidation = validationDatastore(dsLabelledTumorDataValidation, ...
    dsLabelledNormalDataValidation,numValidationPatchesPerClass);
```

Set Up Inception-v3 Network Layers

This example uses the Inception-v3 network, a convolutional neural network that is trained on more than a million images from the ImageNet database [3 on page 18-0   ]. The network is 48 layers deep and can classify images into 1,000 object categories, such as keyboard, mouse, pencil, and many animals. The network expects an image input size of 299-by-299 with 3 channels.

The `inceptionv3` (Deep Learning Toolbox) function returns a pretrained Inception-v3 network. Inception-v3 requires the Deep Learning Toolbox™ Model for Inception-v3 Network support package. If this support package is not installed, then the function provides a download link.

```
net = inceptionv3;
```

Replace Final Layers

The convolutional layers of the network extract image features that the last learnable layer and the final classification layer use to classify the input image. These two layers contain information on how to combine the features that the network extracts into class probabilities, a loss value, and predicted labels. To retrain a pretrained network to classify new images, replace these two layers with new

layers adapted to the new data set. For more information, see "Train Deep Learning Network to Classify New Images" (Deep Learning Toolbox).

Extract the layer graph from the trained network.

```
lgraph = layerGraph(net);
```

Find the names of the two layers to replace by using the supporting function findLayersToReplace. This function is attached to the example as a supporting file. In Inception-v3, these two layers are named 'predictions' and 'ClassificationLayer_predictions'.

```
[learnableLayer,classLayer] = findLayersToReplace(lgraph)
```

```
learnableLayer =
  FullyConnectedLayer with properties:

         Name: 'predictions'

   Hyperparameters
     InputSize: 2048
    OutputSize: 1000

   Learnable Parameters
       Weights: [1000×2048 single]
          Bias: [1000×1 single]

  Show all properties


classLayer =
  ClassificationOutputLayer with properties:

           Name: 'ClassificationLayer_predictions'
        Classes: [1000×1 categorical]
     OutputSize: 1000

   Hyperparameters
    LossFunction: 'crossentropyex'
```

The goal of this example is to perform binary segmentation between two classes, tumor and nontumor regions. Create a new fully connected layer for two classes. Replace the original final fully connected layer with the new layer.

```
numClasses = 2;
newLearnableLayer = fullyConnectedLayer(numClasses,'Name','predictions');
lgraph = replaceLayer(lgraph,learnableLayer.Name,newLearnableLayer);
```

Create a new classification layer for two classes. Replace the original final classification layer with the new layer.

```
newClassLayer = classificationLayer('Name','ClassificationLayer_predictions');
lgraph = replaceLayer(lgraph,classLayer.Name,newClassLayer);
```

Specify Training Options

Train the network using the rmsprop optimization solver. This solver automatically adjusts the learning rate and momentum for faster convergence. Specify other hyperparameter settings by using

the `trainingOptions` (Deep Learning Toolbox) function. Reduce `MaxEpochs` to a small number because the large amount of training data enables the network to reach convergence sooner. If you have Parallel Computing Toolbox™ and the hardware resources available for multi-GPU or parallel training, then accelerate training by specifying the `ExecutionEnvironment` name-value pair argument as `'multi-gpu'` or `'parallel'`, respectively.

```
checkpointsDir = fullfile(trainingImageDir,'checkpoints');
if ~exist(checkpointsDir,'dir')
    mkdir(checkpointsDir);
end
```

```
options = trainingOptions('rmsprop', ...
    'InitialLearnRate',1e-5, ...
    'SquaredGradientDecayFactor',0.99, ...
    'MaxEpochs',3, ...
    'MiniBatchSize',32, ...
    'Plots','training-progress', ...
    'CheckpointPath',checkpointsDir, ...
    'ValidationData',dsValidation, ...
    'ExecutionEnvironment','auto', ...
    'Shuffle','every-epoch');
```

Download and Preprocess Test Data

The Camelyon16 test data set consists of 130 WSIs. These images have both normal and tumor tissue. This example uses two test images from the Camelyon16 test data. The size of each file is approximately 2 GB.

Create a directory to store the test data.

```
testingImageDir = fullfile(tempdir,'Camelyon16','testing');
if ~exist(testingImageDir,'dir')
    mkdir(testingImageDir);
    mkdir(fullfile(testingImageDir,'images'));
    mkdir(fullfile(testingImageDir,'lesion_annotations'));
end
```

```
testDataDir = fullfile(testingImageDir,'images');
testTumorAnnotationDir = fullfile(testingImageDir,'lesion_annotations');
```

To download the test data, go to the Camelyon17 website and click the first "CAMELYON16 data set" link. Open the "testing" directory, then follow these steps.

- Download the "lesion_annotations.zip" file. Extract all files to the directory specified by the `testTumorAnnotationDir` variable.

- Open the "images" directory. Download the first two files, "test_001.tif" and "test_002.tif". Move the files to the directory specified by the `testDataDir` variable.

Specify the number of test images.

```
numTestFiles = 2;
```

Create Masks for Test Images

The test images contain a mix of normal and tumor images. To reduce the amount of computation during classification, define the ROIs by creating masks.

Specify a resolution level to use for creating the mask. This example uses resolution level 7, which is coarse and fits in memory.

```
resolutionLevel = 7;
```

Create masks for regions containing tissue. You can use the helper function `createMaskForNormalTissue` to create masks using color thresholding. This helper function is attached to the example as a supporting file. For more information about this helper function, see Create Masks for Normal Images on page 18-0 .

```
testTissueMaskDir = fullfile(testDataDir,['test_tissuemask_level' num2str(resolutionLevel)]);
createMaskForNormalTissue(testDataDir,testTissueMaskDir,resolutionLevel);
```

Create masks for images that contain tumor tissue. Skip images that do not contain tumor tissue. You can use the helper function `createMaskForTumorTissue` to create masks using ROI objects. This helper function is attached to the example as a supporting file. For more information about this helper function, see Create Masks for Tumor Images on page 18-0 .

```
testTumorMaskDir = fullfile(testDataDir,['test_tumormask_level' num2str(resolutionLevel)]);
createMaskForTumorTissue(testDataDir,testTumorAnnotationDir,testTumorMaskDir,resolutionLevel);
```

Train Network

After configuring the training options and the data source, train the Inception-v3 network by using the `trainNetwork` function. By default, this example downloads a pretrained version of the Inception-v3 network for this data set by using the helper function `downloadTrainedCamelyonNet`. The helper function is attached to the example as a supporting file. The pretrained network enables you to run the entire example without waiting for training to complete.

Note: Training takes 20 hours on an NVIDIA™ Titan X and can take even longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    trainedNet = trainNetwork(dsTrain,lgraph,options);
    save(['trainedCamelyonNet-' datestr(now, 'dd-mmm-yyyy-HH-MM-SS') '.mat'],'trainedNet');
else
    trainedCamelyonNet_url = 'https://www.mathworks.com/supportfiles/vision/data/trainedCamelyonN
    netDir = fullfile(tempdir,'Camelyon16');
    downloadTrainedCamelyonNet(trainedCamelyonNet_url,netDir);
    load(fullfile(netDir,'trainedCamelyonNet.mat'));
end
```

```
Downloading pretrained Inception-v3 network for Cameylon16 data.
This can take several minutes to download...
Done.
```

Classify Test Data and Create Heatmaps

A GPU is highly recommended for classification of the test images (requires Parallel Computing Toolbox™).

Each test image has two masks, one indicating normal tissue and one indicating tumor tissue. Create `bigimage` objects to manage the test data and masks by using the helper function `createBigImageAndMaskArrays`. The helper function is attached to the example as a supporting file.

```
[bigTestImages,bigTestTissueMasks] = createBigImageAndMaskArrays(testDataDir,testTissueMaskDir);
[~,bigTestTumorMasks] = createBigImageAndMaskArrays(testDataDir,testTumorMaskDir);
```

Use the trained Inception-v3 network to identify tumor patches in the test images, `bigTestImages`. Classify the test images on a block-by-block basis by using the `apply` function with a custom processing pipeline specified by the helper function `createHeatMap`. This helper function is attached to the example as a supporting file. To reduce the amount of computation required, specify the tissue mask `bigTestTissueMask` so that the `apply` function processes only patches that contain tissue. If you have Parallel Computing Toolbox™ and a GPU, then you can evaluate blocks in parallel by specifying the `'UseParallel'` name-value pair argument as `true`.

The `createHeatMap` function performs these operations on each tissue block:

- Calculate the tumor probability score by using the `predict` (Deep Learning Toolbox) function.
- Create a heatmap image patch with pixel values equal to the tumor probability score.

The `apply` function stitches the heatmap of each block into a single heatmap for the test image. The heatmap shows where the network detects regions containing tumors.

To visualize the heatmap, overlay the heatmap on the original image and set the transparency `'AlphaData'` property as the tissue mask. The overlay shows how well the tumor is localized in the image. Regions with a high probability of being tumors are displayed with red pixels. Regions with a low probability of being tumors are displayed as blue pixels.

```
patchSize = [299,299,3];

for idx = 1:numTestFiles
    bigTestHeatMaps(idx) = apply(bigTestImages(idx),1,@(x)createHeatMap(x,trainedNet), ...
        'Mask',bigTestTissueMasks(idx),'InclusionThreshold',0, ...
        'BlockSize',patchSize(1:2),'UseParallel',false);

    figure
    hBigTest = bigimageshow(bigTestImages(idx));
    hTestAxes = hBigTest.Parent;
    hTestAxes.Visible = 'off';

    hMaskAxes = axes;
    hBigMask = bigimageshow(bigTestHeatMaps(idx),'Parent',hMaskAxes, ...
        "Interpolation","nearest","AlphaData",bigTestTissueMasks(idx));
    colormap(jet(255));
    hMaskAxes.Visible = 'off';

    linkaxes([hTestAxes,hMaskAxes]);
    title(['Tumor Heatmap of Test Image ',num2str(idx)])
end
```

Tumor Heatmap of Test Image 1
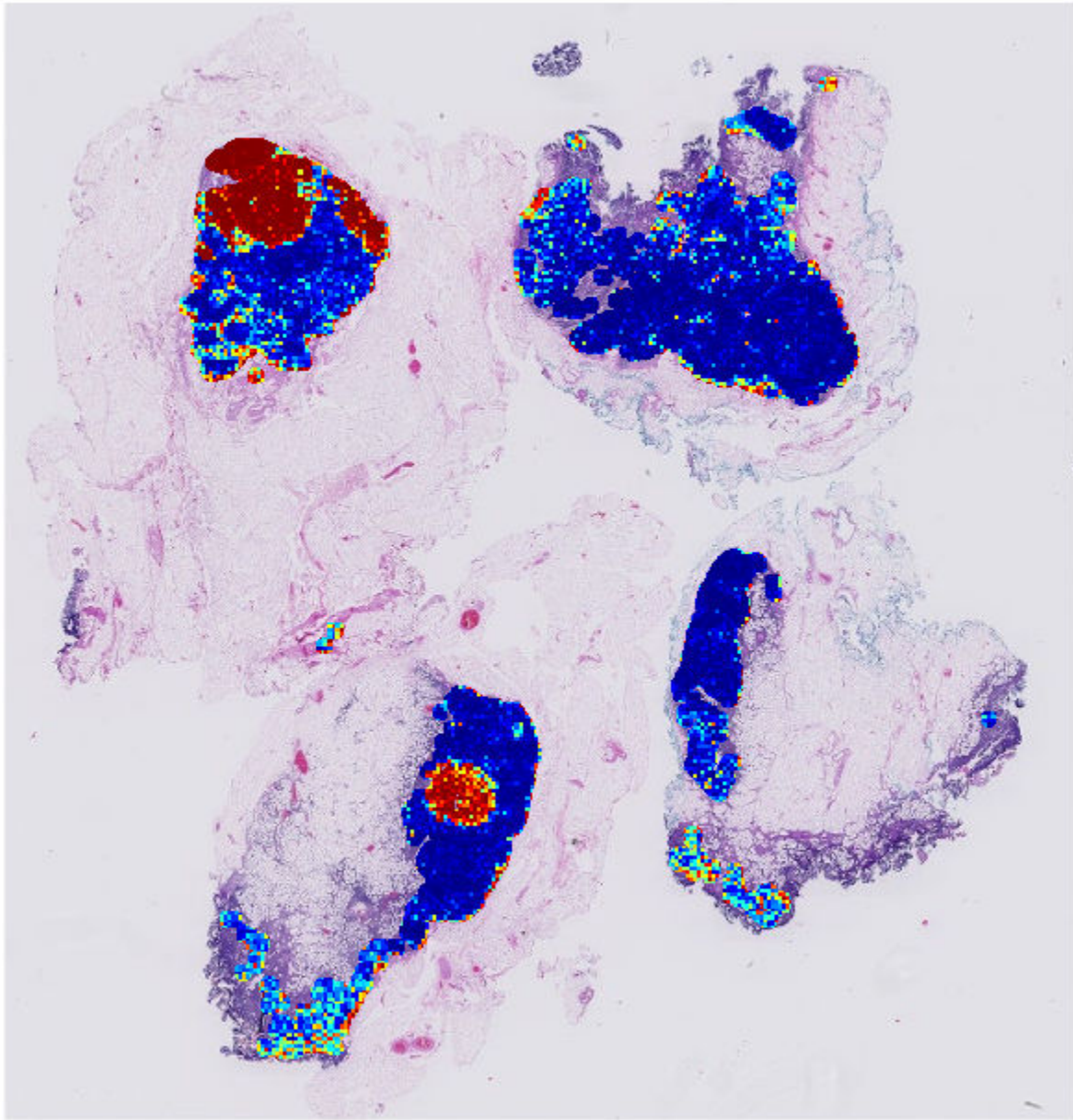
Tumor Heatmap of Test Image 2
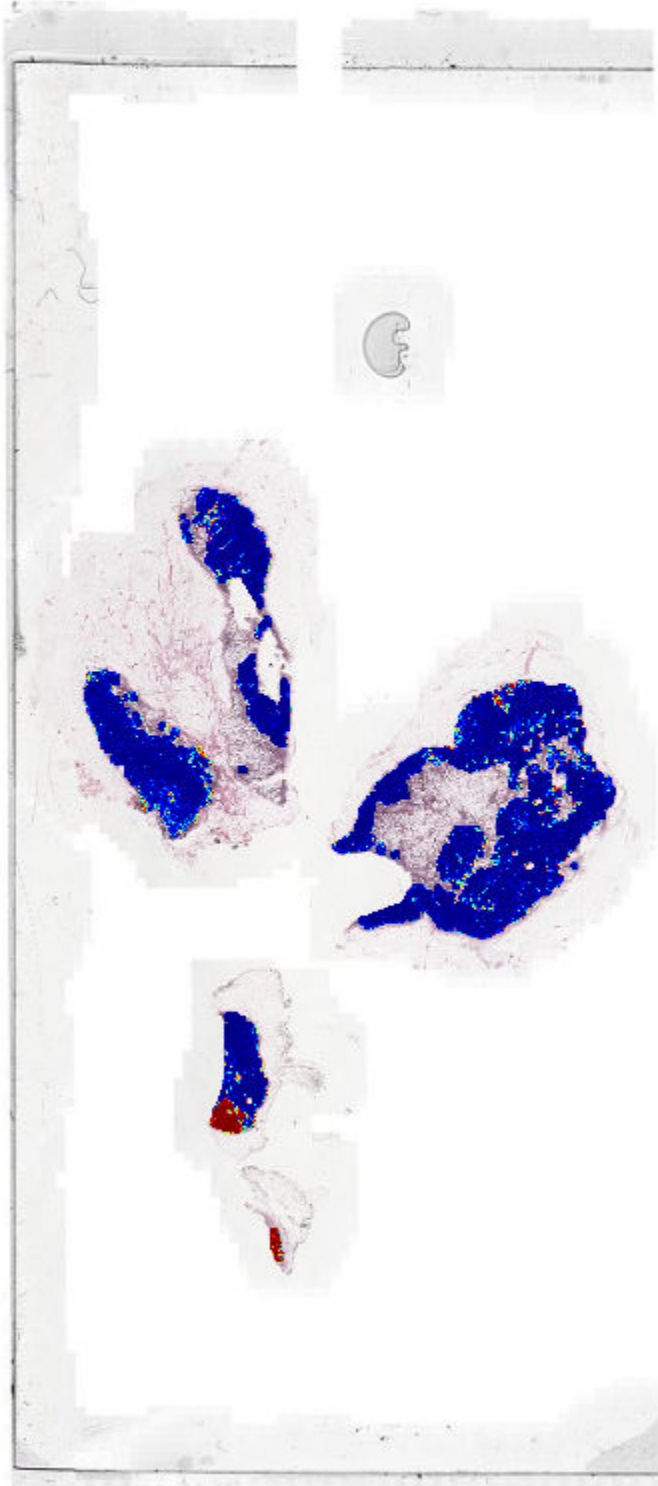
References

[1] Ehteshami B. B., et al. "Diagnostic Assessment of Deep Learning Algorithms for Detection of Lymph Node Metastases in Women With Breast Cancer." *Journal of the American Medical Association*. Vol. 318, No. 22, 2017, pp. 2199–2210. doi:10.1001/jama.2017.14585

[2] Szegedy, C., V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. "Rethinking the Inception Architecture for Computer Vision." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2818–2826. Las Vegas, NV: IEEE, 2016.

[3] ImageNet. http://www.image-net.org.

[4] Macenko, M., et al. "A Method for Normalizing Histology Slides for Quantitative Analysis." In *2009 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, 1107–1110. Boston, MA: IEEE, 2009.

[5] xml2struct https://www.mathworks.com/matlabcentral/fileexchange/28518-xml2struct.

See Also
bigimage | bigimageDatastore | bigimageshow | trainNetwork | trainingOptions | transform

More About
- "Datastores for Deep Learning" (Deep Learning Toolbox)
- "List of Deep Learning Layers" (Deep Learning Toolbox)

Neural Style Transfer Using Deep Learning

This example shows how to apply the stylistic appearance of one image to the scene content of a second image using a pretrained VGG-19 network [1] on page 18-0   . This example uses the distinctive Van Gogh painting "Starry Night" as the style image and a photograph of a lighthouse as the content image.

Load Data

Load the style image and content image.

```
styleImage = im2double(imread('starryNight.jpg'));
contentImage = imread('lighthouse.png');
```

Display the style image and content image as a montage.

```
imshow(imtile({styleImage,contentImage},'BackgroundColor','w'));
```


Load Feature Extraction Network

In this example, you use a modified pretrained VGG-19 deep neural network to extract the features of the content and style image at various layers. These multilayer features are used to compute respective content and style losses. The network generates the stylized transfer image using the combined loss.

To get a pretrained VGG-19 network, install `vgg19` (Deep Learning Toolbox). If you do not have the required support packages installed, then the software provides a download link.

```
net = vgg19;
```

To make the VGG-19 network suitable for feature extraction, remove all of the fully connected layers from the network.

```
lastFeatureLayerIdx = 38;
layers = net.Layers;
layers = layers(1:lastFeatureLayerIdx);
```

The max pooling layers of the VGG-19 network cause a fading effect. To decrease the fading effect and increase the gradient flow, replace all max pooling layers with average pooling layers [1] on page 18-0   .

```
for l = 1:lastFeatureLayerIdx
    layer = layers(l);
    if isa(layer,'nnet.cnn.layer.MaxPooling2DLayer')
        layers(l) = averagePooling2dLayer(layer.PoolSize,'Stride',layer.Stride,'Name',layer.Name
    end
end
```

Create a layer graph with the modified layers.

```
lgraph = layerGraph(layers);
```

Visualize the feature extraction network in a plot.

```
plot(lgraph)
title('Feature Extraction Network')
```
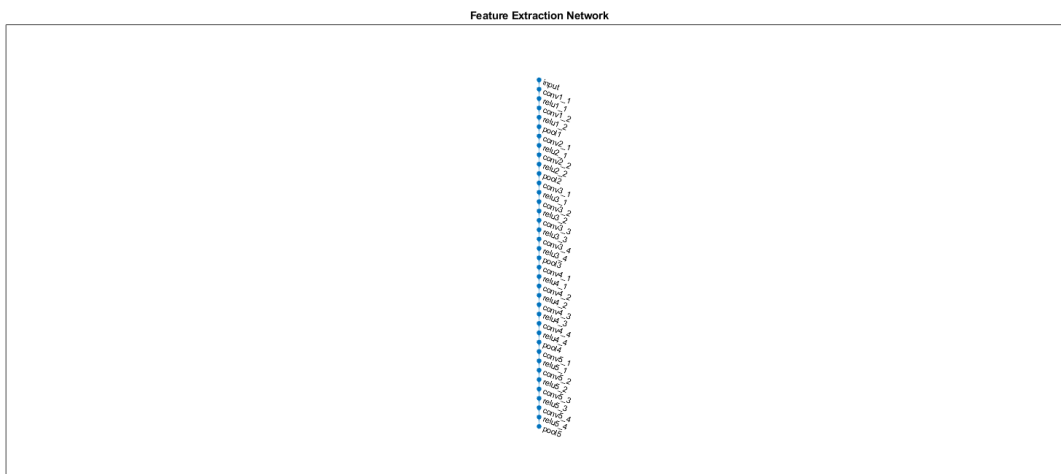

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnet = dlnetwork(lgraph);
```

Preprocess Data

Resize the style image and content image to a smaller size for faster processing.

```
imageSize = [384,512];
styleImg = imresize(styleImage,imageSize);
contentImg = imresize(contentImage,imageSize);
```

The pretrained VGG-19 network performs classification on a channel-wise mean subtracted image. Get the channel-wise mean from the image input layer, which is the first layer in the network.

```
imgInputLayer = lgraph.Layers(1);
meanVggNet = imgInputLayer.Mean(1,1,:);
```

The values of the channel-wise mean are appropriate for images of floating point data type with pixel values in the range [0, 255]. Convert the style image and content image to data type `single` with range [0, 255]. Then, subtract the channel-wise mean from the style image and content image.

```
styleImg = rescale(single(styleImg),0,255) - meanVggNet;
contentImg = rescale(single(contentImg),0,255) - meanVggNet;
```

Initialize Transfer Image

The transfer image is the output image as a result of style transfer. You can intitalize the transfer image with a style image, content image, or any random image. Initialization with a style image or content image biases the style transfer process and produces a transfer image more similar to the input image. In contrast, initialization with white noise removes the bias but takes longer to converge on the stylized image. For better stylization and faster convergence, this example initializes the output transfer image as a weighted combination of the content image and a white noise image.

```
noiseRatio = 0.7;
randImage = randi([-20,20],[imageSize 3]);
transferImage = noiseRatio.*randImage + (1-noiseRatio).*contentImg;
```

Define Loss Functions and Style Transfer Parameters

Content Loss

The objective of content loss is to make the features of the transfer image match the features of the content image. The content loss is computed as the mean squared difference between content image features and transfer image features for each content feature layer [1] on page 18-0 . $\widehat{Y}$ is the predicted feature map for the transfer image and Y is the predicted feature map for the content image. W_c^l is the content layer weight for the lth layer. H, W, C are the height, width, and channels of the feature maps, respectively.

$$L_{content} = \sum_l W_c^l \times \frac{1}{HWC} \sum_{i,j} (\widehat{Y}_{i,j}^l - Y_{i,j}^l)^2$$

Specify the content feature extraction layer names. The features extracted from these layers are used to compute the content loss. In the VGG-19 network, training is more effective using features from deeper layers rather than features from shallow layers. Therefore, specify the content feature extraction layer as the fourth convolutional layer.

```
styleTransferOptions.contentFeatureLayerNames = {'conv4_2'};
```

Specify the weights of the content feature extraction layers.

```
styleTransferOptions.contentFeatureLayerWeights = 1;
```

Style Loss

The objective of style loss is to make the texture of the transfer image match the texture of the style image. The style representation of an image is represented as a Gram matrix. Therefore, the style loss is computed as the mean squared difference between the Gram matrix of the style image and the Gram matrix of the transfer image [1] on page 18-0 . Z and $\widehat{Z}$ are the predicted feature maps for the style and transfer image, respectively. G_Z and $G_{\widehat{Z}}$ are Gram matrices for style features and transfer features, respectively. W_s^l is the style layer weight for the lth style layer.

$$G_{\widehat{Z}} = \sum_{i,j} \widehat{Z}_{i,j} \times \widehat{Z}_{j,i}$$

$$G_{Z} = \sum_{i,j} Z_{i,j} \times Z_{j,i}$$

$$L_{style} = \sum_{l} W_s^l \times \frac{1}{(2HWC)^2} \sum (G_{\widehat{Z}}^l - G_Z^l)^2$$

Specify the names of the style feature extraction layers. The features extracted from these layers are used to compute style loss.

```
styleTransferOptions.styleFeatureLayerNames = {'conv1_1','conv2_1','conv3_1','conv4_1','conv5_1'}
```

Specify the weights of the style feature extraction layers. Specify small weights for simple style images and increase the weights for complex style images.

```
styleTransferOptions.styleFeatureLayerWeights = [0.5,1.0,1.5,3.0,4.0];
```

Total Loss

The total loss is a weighted combination of content loss and style loss. α and β are weight factors for content loss and style loss, respectively.

$$L_{total} = \alpha \times L_{content} + \beta \times L_{style}$$

Specify the weight factors `alpha` and `beta` for content loss and style loss. The ratio of `alpha` to `beta` should be around 1e-3 or 1e-4 [1] on page 18-0   .

```
styleTransferOptions.alpha = 1;
styleTransferOptions.beta = 1e3;
```

Specify Training Options

Train for 2500 iterations.

```
numIterations = 2500;
```

Specify options for Adam optimization. Set the learning rate to 2 for faster convergence. You can experiment with the learning rate by observing your output image and losses. Initialize the trailing average gradient and trailing average gradient-square decay rates with `[]`.

```
learningRate = 2;
trailingAvg = [];
trailingAvgSq = [];
```

Train Network

Convert the style image, content image, and transfer image to `dlarray` (Deep Learning Toolbox) objects with underlying type `single` and dimension labels `'SSC'`.

```
dlStyle = dlarray(styleImg,'SSC');
dlContent = dlarray(contentImg,'SSC');
dlTransfer = dlarray(transferImage,'SSC');
```

Train on a GPU if one is available. Use of a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher. For GPU training, convert the data into a `gpuArray`.

```
if canUseGPU
    dlContent = gpuArray(dlContent);
    dlStyle = gpuArray(dlStyle);
    dlTransfer = gpuArray(dlTransfer);
end
```

Extract the content features from the content image.

```
numContentFeatureLayers = numel(styleTransferOptions.contentFeatureLayerNames);
contentFeatures = cell(1,numContentFeatureLayers);
[contentFeatures{:}] = forward(dlnet,dlContent,'Outputs',styleTransferOptions.contentFeatureLayer
```

Extract the style features from the style image.

```
numStyleFeatureLayers = numel(styleTransferOptions.styleFeatureLayerNames);
styleFeatures = cell(1,numStyleFeatureLayers);
[styleFeatures{:}] = forward(dlnet,dlStyle,'Outputs',styleTransferOptions.styleFeatureLayerNames)
```

Train the model using a custom training loop. For each iteration:

- Calculate the content loss and style loss using the features of the content image, style image, and transfer image. To calculate the loss and gradients, use the helper function `imageGradients` (defined in the Supporting Functions on page 18-0   section of this example).
- Update the transfer image using the `adamupdate` (Deep Learning Toolbox) function.
- Select the best style transfer image as the final output image.

```
figure

minimumLoss = inf;

for iteration = 1:numIterations
    % Evaluate the transfer image gradients and state using dlfeval and the
    % imageGradients function listed at the end of the example.
    [grad,losses] = dlfeval(@imageGradients,dlnet,dlTransfer,contentFeatures,styleFeatures,style
    [dlTransfer,trailingAvg,trailingAvgSq] = adamupdate(dlTransfer,grad,trailingAvg,trailingAvgS

    if losses.totalLoss < minimumLoss
        minimumLoss = losses.totalLoss;
        dlOutput = dlTransfer;
    end

    % Display the transfer image on the first iteration and after every 50
    % iterations. The postprocessing steps are described in the "Postprocess
    % Transfer Image for Display" section of this example.
    if mod(iteration,50) == 0 || (iteration == 1)

        transferImage = gather(extractdata(dlTransfer));
        transferImage = transferImage + meanVggNet;
        transferImage = uint8(transferImage);
        transferImage = imresize(transferImage,size(contentImage,[1 2]));

        image(transferImage)
        title(['Transfer Image After Iteration ',num2str(iteration)])
```

```
        axis off image
        drawnow
    end
end
```


Transfer Image After Iteration 2500

Postprocess Transfer Image for Display

Get the updated transfer image.

```
transferImage = gather(extractdata(dlOutput));
```

Add the network-trained mean to the transfer image.

```
transferImage = transferImage + meanVggNet;
```

Some pixel values can exceed the original range [0, 255] of the content and style image. You can clip the values to the range [0, 255] by converting the data type to `uint8`.

```
transferImage = uint8(transferImage);
```

Resize the transfer image to the original size of the content image.

```
transferImage = imresize(transferImage,size(contentImage,[1 2]));
```

Display the content image, transfer image, and style image in a montage.

```
imshow(imtile({contentImage,transferImage,styleImage}, ...
    'GridSize',[1 3],'BackgroundColor','w'));
```

Supporting Functions

Compute Image Loss and Gradients

The `imageGradients` helper function returns the loss and gradients using features of the content image, style image, and transfer image.

```
function [gradients,losses] = imageGradients(dlnet,dlTransfer,contentFeatures,styleFeatures,param

    % Initialize transfer image feature containers.
    numContentFeatureLayers = numel(params.contentFeatureLayerNames);
    numStyleFeatureLayers = numel(params.styleFeatureLayerNames);

    transferContentFeatures = cell(1,numContentFeatureLayers);
    transferStyleFeatures = cell(1,numStyleFeatureLayers);

    % Extract content features of transfer image.
    [transferContentFeatures{:}] = forward(dlnet,dlTransfer,'Outputs',params.contentFeatureLayerN

    % Extract style features of transfer image.
    [transferStyleFeatures{:}] = forward(dlnet,dlTransfer,'Outputs',params.styleFeatureLayerNames

    % Compute content loss.
    cLoss = contentLoss(transferContentFeatures,contentFeatures,params.contentFeatureLayerWeights

    % Compute style loss.
    sLoss = styleLoss(transferStyleFeatures,styleFeatures,params.styleFeatureLayerWeights);

    % Compute final loss as weighted combination of content and style loss.
    loss = (params.alpha * cLoss) + (params.beta * sLoss);

    % Calculate gradient with respect to transfer image.
    gradients = dlgradient(loss,dlTransfer);

    % Extract various losses.
```

```
    losses.totalLoss = gather(extractdata(loss));
    losses.contentLoss = gather(extractdata(cLoss));
    losses.styleLoss = gather(extractdata(sLoss));

end
```

Compute Content Loss

The `contentLoss` helper function computes the weighted mean squared difference between the content image features and the transfer image features.

```
function loss = contentLoss(transferContentFeatures,contentFeatures,contentWeights)

    loss = 0;
    for i=1:numel(contentFeatures)
        temp = 0.5 .* mean((transferContentFeatures{1,i} - contentFeatures{1,i}).^2,'all');
        loss = loss + (contentWeights(i)*temp);
    end
end
```

Compute Style Loss

The `styleLoss` helper function computes the weighted mean squared difference between the Gram matrix of the style image features and the Gram matrix of the transfer image features.

```
function loss = styleLoss(transferStyleFeatures,styleFeatures,styleWeights)

    loss = 0;
    for i=1:numel(styleFeatures)

        tsf = transferStyleFeatures{1,i};
        sf = styleFeatures{1,i};
        [h,w,c] = size(sf);

        gramStyle = computeGramMatrix(sf);
        gramTransfer = computeGramMatrix(tsf);
        sLoss = mean((gramTransfer - gramStyle).^2,'all') / ((h*w*c)^2);

        loss = loss + (styleWeights(i)*sLoss);
    end
end
```

Compute Gram Matrix

The `computeGramMatrix` helper function is used by the `styleLoss` helper function to compute the Gram matrix of a feature map.

```
function gramMatrix = computeGramMatrix(featureMap)
    [H,W,C] = size(featureMap);
    reshapedFeatures = reshape(featureMap,H*W,C);
    gramMatrix = reshapedFeatures' * reshapedFeatures;
end
```

References

[1] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. "A Neural Algorithm of Artistic Style." Preprint, submitted September 2, 2015. https://arxiv.org/abs/1508.06576

See Also

dlarray | trainNetwork | trainingOptions | vgg19

More About

- "Define Custom Training Loops, Loss Functions, and Networks" (Deep Learning Toolbox)
- "Specify Training Options in Custom Training Loop" (Deep Learning Toolbox)
- "Train Network Using Custom Training Loop" (Deep Learning Toolbox)
- "List of Functions with dlarray Support" (Deep Learning Toolbox)
- "List of Deep Learning Layers" (Deep Learning Toolbox)

Hyperspectral Image Processing

This topic describes functions that enable hyperspectral image analysis and provides examples for spectral classification and anomaly detection using endmembers and abundance maps.

Getting Started with Hyperspectral Image Processing

Hyperspectral imaging measures the spatial and spectral characteristics of an object by imaging it at different wavelengths. The wavelength range extends beyond the visible spectrum and covers ultraviolet (UV) to long wave infrared (LWIR) wavelengths. The most popular are the visible, near-infrared and mid-infrared wavelength bands. An hyperspectral imaging sensor acquires several number of images with narrow and contiguous wavelengths within a specified spectral range. Each of these contain more subtle and detailed information.

Hyperspectral image processing involves representing, analyzing, and interpreting information contained in the hyperspectral images.
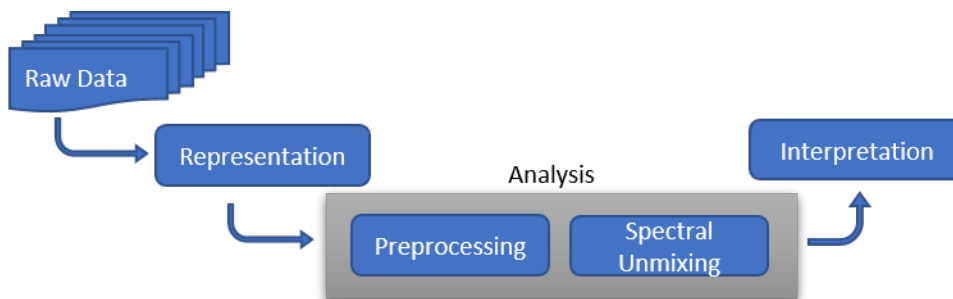
Representing Hyperspectral Data

The values measured by a hyperspectral imaging sensor are stored to a binary data file by using band sequential (BSQ), band-interleaved-by-pixel (BIP), or band-interleaved-by-line (BIL) encoding formats. The data file is associated to a header file that contains ancillary information (metadata) like sensor parameters, acquisition settings, spatial dimensions, spectral wavelengths, and encoding formats that are required for proper representation of the values in the data file.

For hyperspectral image processing, the values read from the data file are arranged into a three dimensional (3-D) array ($M \times N \times C$) where M and N are the spatial dimensions of the acquired data, C is the spectral dimension specifying the number of spectral wavelengths used during acquisition. Thus, you can consider the 3-D array as a set of two dimensional (2-D) monochromatic images captured at varying wavelengths. This set is known as the hyperspectral data cube or data cube.

The `hypercube` function constructs the data cube by reading the data file and the metadata information in the associated header file. The `hypercube` function creates a `hypercube` object and stores the data cube, spectral wavelengths, and the metadata to its properties. You can use the `hypercube` object as input to all other functions in the Image Processing Toolbox Hyperspectral Imaging Library.

RGB image

Hyperspectral data cube

Color representation

False-colored image

CIR image
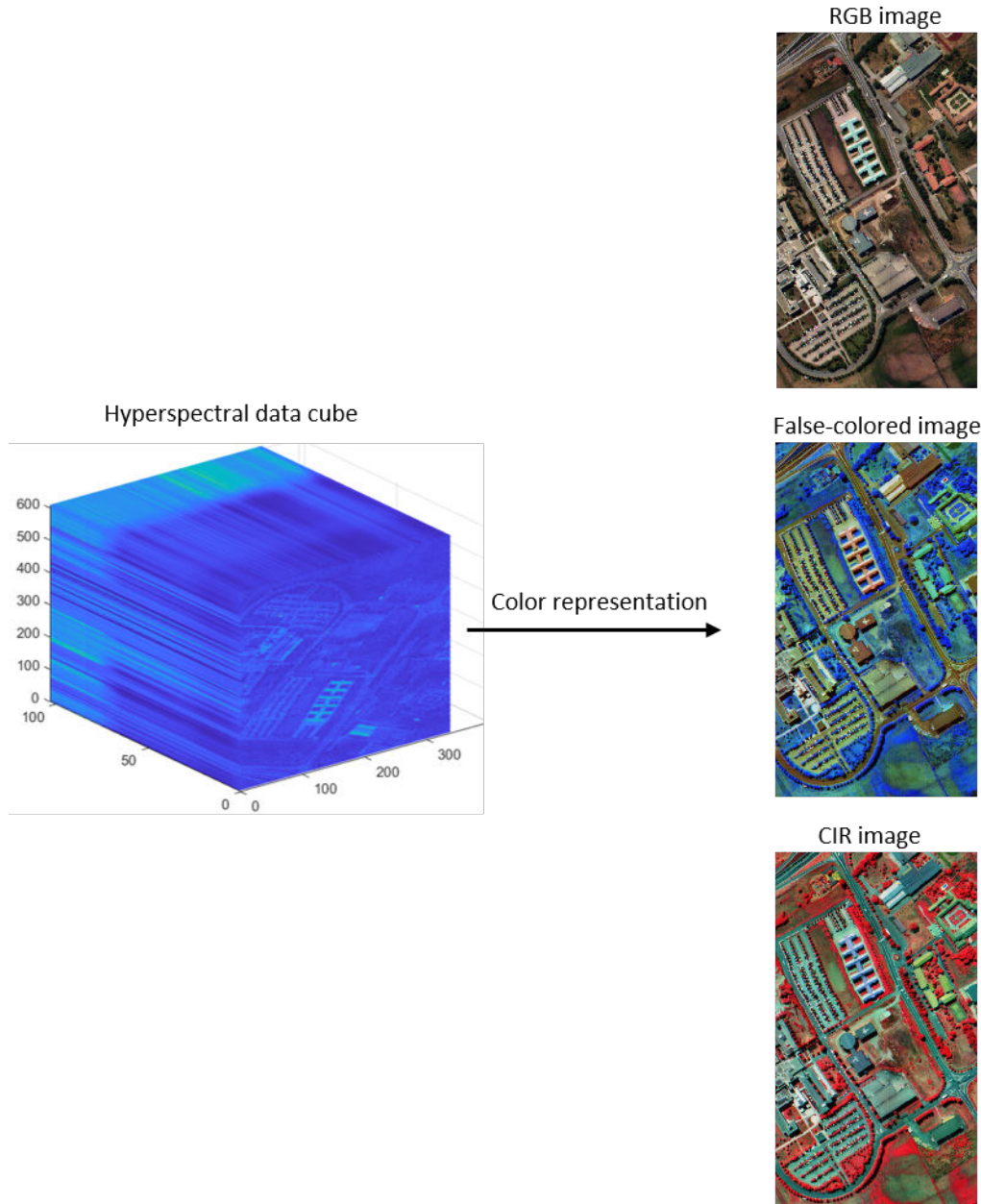
Color Representation of Data Cube: To visualize and understand the object being imaged, it is useful to represent the data cube as a 2-D image by using color schemes. The color representation of the data cube allows to visually inspect the data and supports decision making. You can use the `colorize` function to compute the Red-Green-Blue (RGB), false-color, and color-infrared (CIR) representation of the data cube.

- The RGB color scheme uses the red, green, and blue spectral band responses to generate the 2-D image of the hyperspectral data cube. The RGB color scheme brings the natural appearance but results in significant loss of subtle information.

- The false-color scheme uses a combination of any number of bands other than the visible red, green, and blue spectral bands. Use false-color representation to visualize the spectral responses

of bands outside the visible spectrum. The false-color scheme efficiently captures distinct information across all spectral bands of hyperspectral data.

- The CIR color scheme uses spectral bands in the NIR range. The CIR representation of hyperspectral data cube is particularly useful in displaying and analyzing vegetation areas of the data cube.

Preprocessing

The spatial and the spectral characteristics of the hyperspectral data are characterized by the pixels. Each pixel is a vector of values that specifies the intensities at a location (x,y) in z different bands. The vector is known as the pixel spectra and it defines the spectral signature of the pixel located at (x,y). The pixel spectra are important features in hyperspectral data analysis.
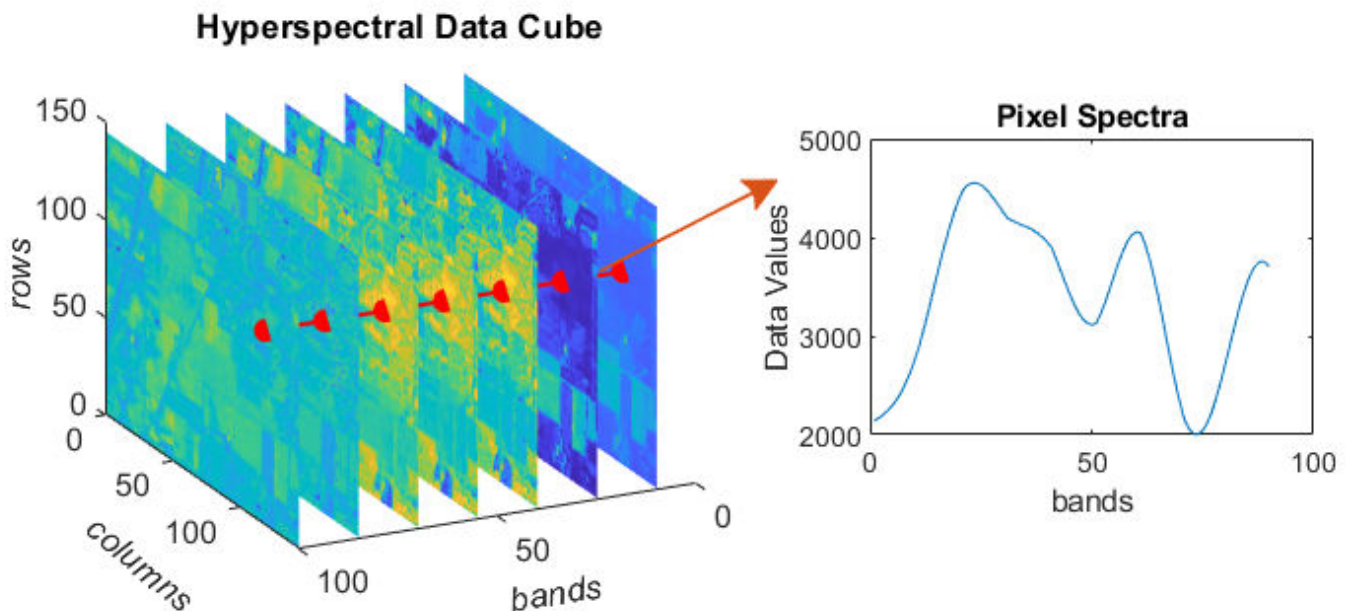
The pixel values can be uncalibrated digital numbers (DNs) or calibrated radiance and reflectance values. In case of remote sensing application, the important preprocessing step is to calibrate DNs by using radiometric and atmospheric correction methods. This process improves interpretation of the pixel spectra and provides better results when you analyse multiple data sets, as in a classification problem.

The other preprocessing step that is important in all hyperspectral imaging applications is the *dimensionality reduction*. The large number of bands in the hyperspectral data increases the computational complexity in processing the data cube. The contiguous nature of the band images exhibit redundant information across bands. The neighboring bands in a hyperspectral image have high correlation and results in spectral redundancy. You can remove the redundant bands by decorrelating the band images. Some of the popular approaches for reducing the spectral dimensionality of a data cube includes band selection and orthogonal transforms.

- The **band selection** approach uses orthogonal space projections to find the spectrally distinct and most informative bands in the data cube. Use `selectBands` and `removeBands` functions for finding the most informative bands and removing specified bands respectively.

- **Orthogonal transforms** such as principal component analysis (PCA) and maximum noise fraction (MNF) decorrelates the band information and finds the principal component bands. PCA transforms the data to a lower dimensional space and finds principal components with their directions along the maximum variances of the input bands. The principal components are ordered in decreasing order of amount of total variance explained. On the other hand, MNF computes the principal components that maximizes the signal-noise-ratio rather than the variance. MNF transform is particularly efficient in deriving principal components from noisy band images. The principal component bands are spectrally distinct bands with low inter-band correlation.

  The `hyperpca` and `hypermnf` functions reduces the spectral dimensionality of the data cube by using PCA and MNF transform respectively. The pixel spectra derived from the reduced data cube are used for hyperspectral data analysis.

Spectral Unmixing

In a hyperspectral image, the intensity values recorded at each pixel specifies the spectral characteristics of the region that the pixel belong to. The region can be a homogeneous surface or heterogeneous surface. The pixels that belong to the homogeneous surface are known as the pure pixels. These pure pixels constitute the endmembers of the hyperspectral data.

The heterogeneous surfaces are a combination of two or more distinct homogeneous surfaces. The pixels belonging to the heterogeneous surfaces are known as the mixed pixels. The spectral signature of a mixed pixel is a combination of two or more endmembers signatures. This spatial heterogeneity is mainly due to the low spatial resolution of the hyperspectral sensors.
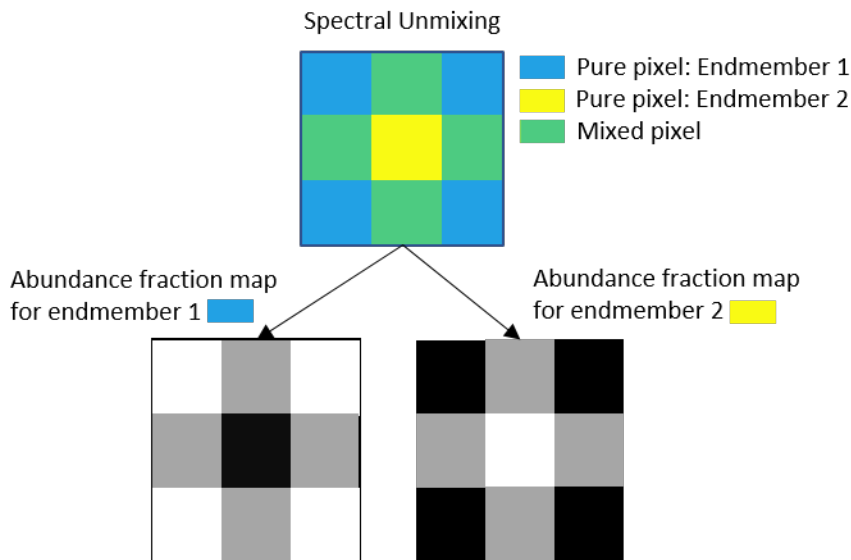
Spectral unmixing is the process for decomposing the spectral signatures of the mixed pixels into its constituent endmembers. The spectral unmixing process involves two steps:

1 **Endmember extraction**: The spectra of the endmembers are the prominent features and can be used for efficient spectral unmixing, segmentation, and classification of hyperspectral images. Convex geometry based approaches such as pixel-purity index (PPI), fast iterative pixel purity index (FIPPI), and N-FINDR are some of the efficient approaches for endmember extraction.

- Use `ppi` function to estimate the endmembers by using the PPI approach. The `ppi` method projects the pixel spectra to an orthogonal space and identifies extrema pixels in the projected space as endmembers. This is a non-iterative approach and the results depend on the unit random vectors used for orthogonal projection. For better results, the method requires a large number of random unit vectors for projection and hence, is computationally expensive.

- Use `fippi` function to estimate the endmembers by using FIPPI approach. The `fippi` method is an iterative approach and it uses automatic target generation process to estimate the initial set of unit vectors for orthogonal projection. The algorithm converges faster and identifies unique endmembers.

- Use `nfindr` function to estimate the endmembers by using the N-FINDR method. N-FINDR is an iterative approach that constructs a simplex by using the pixel spectra. The method assumes that the volume of a simplex formed by the endmembers is larger than any other volume defined by other combination of pixels. Hence, the set of pixel signatures for which the volume of the simplex is high, are chosen as endmembers.

2 **Abundance map estimation**: Given the endmember signatures, it is useful to estimate the fractional amount of each endmember present at each pixel. The abundance maps are generated for each endmember and they represent the distribution of endmember spectra in the image. You can label each pixel to belong to an endmember spectra by comparing all the abundance map values obtained for that pixel.

   Use the `estimateAbundanceLS` function to estimate the abundance maps for each endmember spectra

Spectral Matching

Spectral matching is the important step in interpreting the pixel spectra. Spectral matching is performed to identify the class of the endmember material by comparing its spectra with one or more reference spectra. The reference data are pure spectral signatures of materials that are available as spectral libraries.

Use the `readEcostressSig` function to read the reference spectra files from ECOSTRESS spectral library. Then, you can compute the similarity between the files in the ECOSTRESS library spectra and an endmember spectra by using `spectralMatch` function. You can also use `sam` and `sid` functions to match two spectral signatures of same length.

Applications

Classification and target detection are one among the important applications of hyperspectral image processing. You can segment and classify each pixel in a hyperspectral image through unmixing and spectral matching. For an example, see "Hyperspectral Image Analysis Using Maximum Abundance Classification" on page 19-16 and "Classify Hyperspectral Image Using Library Signatures and SAM" on page 19-23.

Similarly, you can perform target detection by matching the spectral signatures. For an example, see "Target Detection Using Spectral Signature Matching" on page 19-36.

In addition, hyperspectral image processing is widely used for anomaly detection and vegetation analysis.

- Use the `anomalyRX` function to detect anomalies in a hyperspectral image.

- Use the `ndvi` function to measure vegetation cover and analyse vegetation health.

See Also

Apps
Hyperspectral Viewer

Functions
anomalyRX | estimateAbundanceLS | hypercube | ndvi | ppi | spectralMatch

Related Examples

- "Classify Hyperspectral Image Using Library Signatures and SAM" on page 19-23
- "Hyperspectral Image Analysis Using Maximum Abundance Classification" on page 19-16
- "Target Detection Using Spectral Signature Matching" on page 19-36
- "Measure Vegetation Cover in Hyperspectral Data Using NDVI Image"
- "Detect Anomalous Pixels in Hyperspectral Data Using RX Detector"

Explore Hyperspectral Data in the Hyperspectral Viewer

This example shows how to explore hyperspectral data using the **Hyperspectral Viewer** app. Using the capabilities of the app, you can view the individual bands of a hyperspectral data set as grayscale images. You can also view color composite representations of the data set as RGB, color infrared (CIR), and false-color images. In addition to exploring these visual representations of the spatial dimensions of the data, you can create plots of individual points or small regions of the data along the spectral dimension. These plots, called *spectral profiles,* can identify elements in the hyperspectral data.

Load Hyperspectral Data into Workspace

For this example, load an aerial hyperspectral data set of an area called Jasper Ridge, captured via the airborne visible/infrared imaging spectrometer (AVIRIS). The data set contains areas of water, land, road, and vegetation. Load the hyperspectral data set into a `hypercube` object in the MATLAB® workspace.

```
hcube = hypercube('jasperRidge2_R198.img');
```

This command creates a `hypercube` object in the workspace called `hcube`. The `hcube` object contains a 100-by-100-by-198 cube of hyperspectral data.

View Hyperspectral Data in Hyperspectral Viewer

Open the **Hyperspectral Viewer** app. First, click the **Apps** tab on the MATLAB toolstrip. Then, in the **Image Processing and Computer Vision** section, click the **Hyperspectral Viewer** button.

With the app open, load the hyperspectral data into the app. On the app toolstrip, click **Import** and select **Hypercube Object**. In the Import from Workspace dialog box, select the `hypercube` object you loaded into the workspace, `hcube`. (You can also specify a data set when you open the app using the command: `hyperspectralViewer(hcube)`.)

The app displays several views of the Jasper Ridge hyperspectral data. The **Bands** pane displays the bands of the hyperspectral data as a stack of grayscale images. A second pane includes color composite representations of the hyperspectral data, displaying the **False Color** tab by default. The **Plots** pane displays a histogram of the band currently displayed in the **Bands** pane and a plot of the spectral dimension of the data by wavelength or by band. (You can rearrange these panes by clicking and dragging them inside the app. To return to the standard pane arrangement, click **Default Layout** on the app toolstrip.)
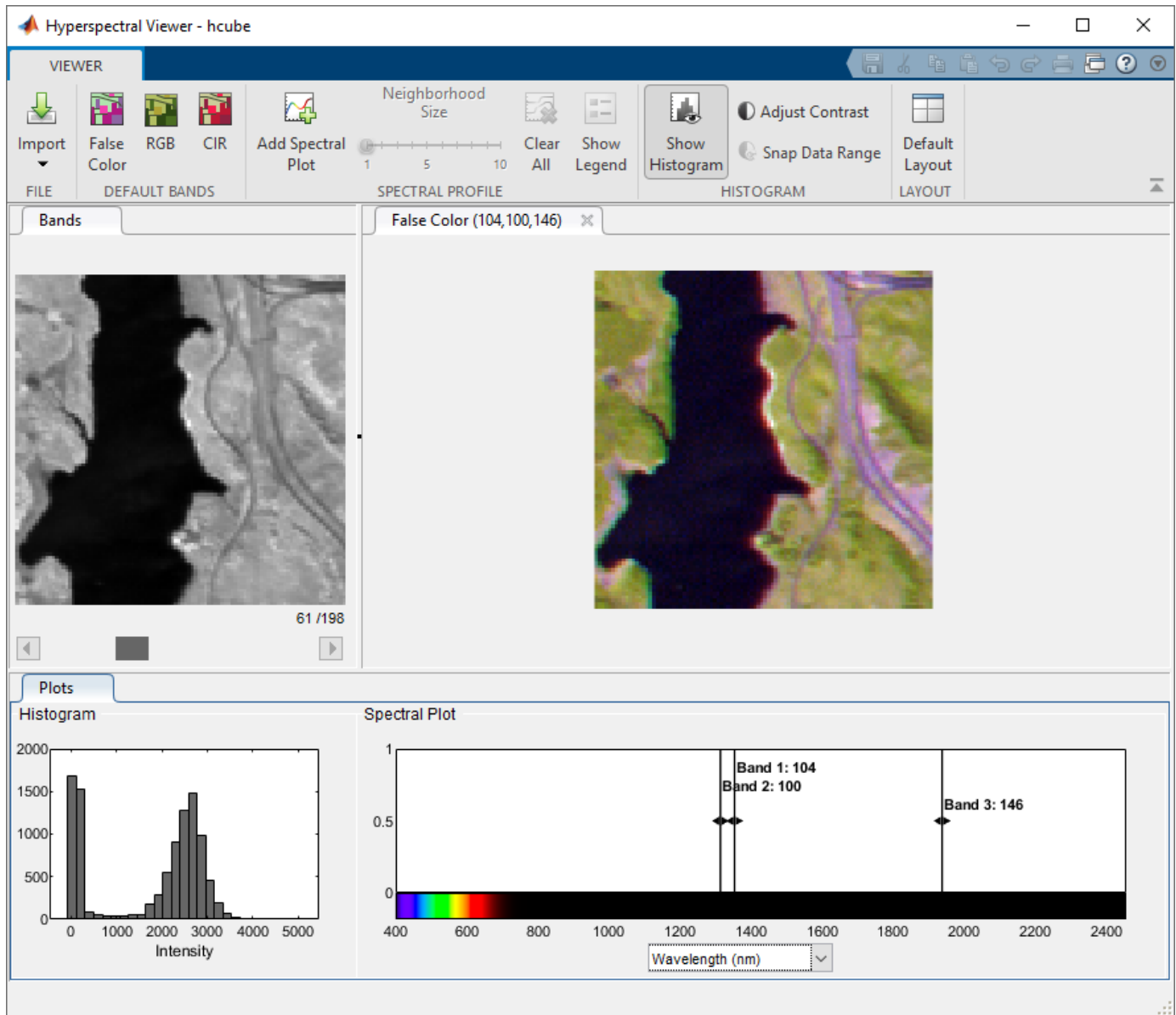
Explore the Spectral Bands

Explore the spectral bands of the Jasper Ridge data set as a stack of grayscale images in the **Bands** pane. Use the slider at the bottom of the pane to navigate through the images. Because each band isolates a specific range of wavelengths, aspects of the scene might be clearer in some bands than others.
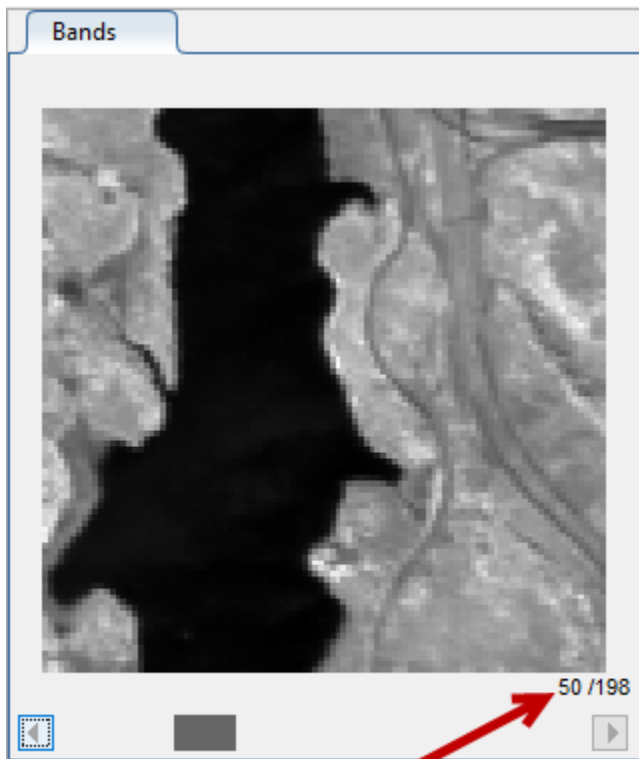
Band displayed

To get a closer look at a band, click **Zoom In** or **Zoom Out** in the axes toolbar that appears when you point the cursor over the image.

To improve the contrast of a band image, click **Adjust Contrast** on the app toolstrip. When you do, the app overlays a contrast adjustment window on the histogram of the image, displayed in the **Plot** pane. To adjust the contrast, move the window over the histogram or resize the window by clicking and dragging the handles. The app adjusts the contrast using a technique called *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. To return to the default view, click **Snap Data Range**. To remove the contrast adjustment window from the histogram, click **Adjust Contrast**.

Contrast adjustment window
with blue resizing handles
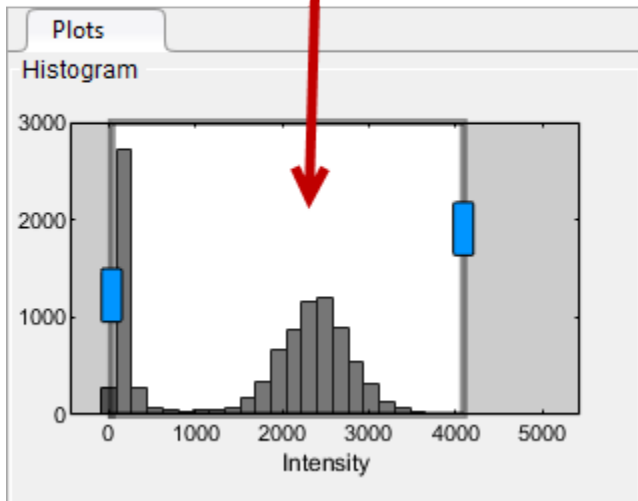
Explore Color Representations of Hyperspectral Data

Explore the Jasper Ridge hyperspectral data as a color composite image. To create these color images, the Hyperspectral Viewer automatically chooses three of the bands in the hyperspectral dataset to use for the red, green, and blue channels of a color image. The choice of which bands the app uses depends on the type of color representation. The app supports three types of color composite renditions: False Color, RGB, and Color Infrared (CIR). It can be usefult to view all of the color composite images because each one uses different bands and can highlight different spectral details, thus increasing the interpretability of the data.

By default, the app displays a false-color representation of the data. False-color composites visualize wavelengths that the human eye cannot see. The tab of the pane identifies the type of the color image, **False Color**, and the bands that the app used to form it, **(104,100,146)**, in red-green-blue order. The app indicates in the **Spectral Plot** which bands were used. To change these band selections, click and drag the handle of the band indicator in the **Spectral Plot**. If you choose a different band, the app updates the text in the tab with the new bands and adds the word "Custom," such as, **False Color-Custom**.

Bands used to create image

To select other bands, click and drag handle.
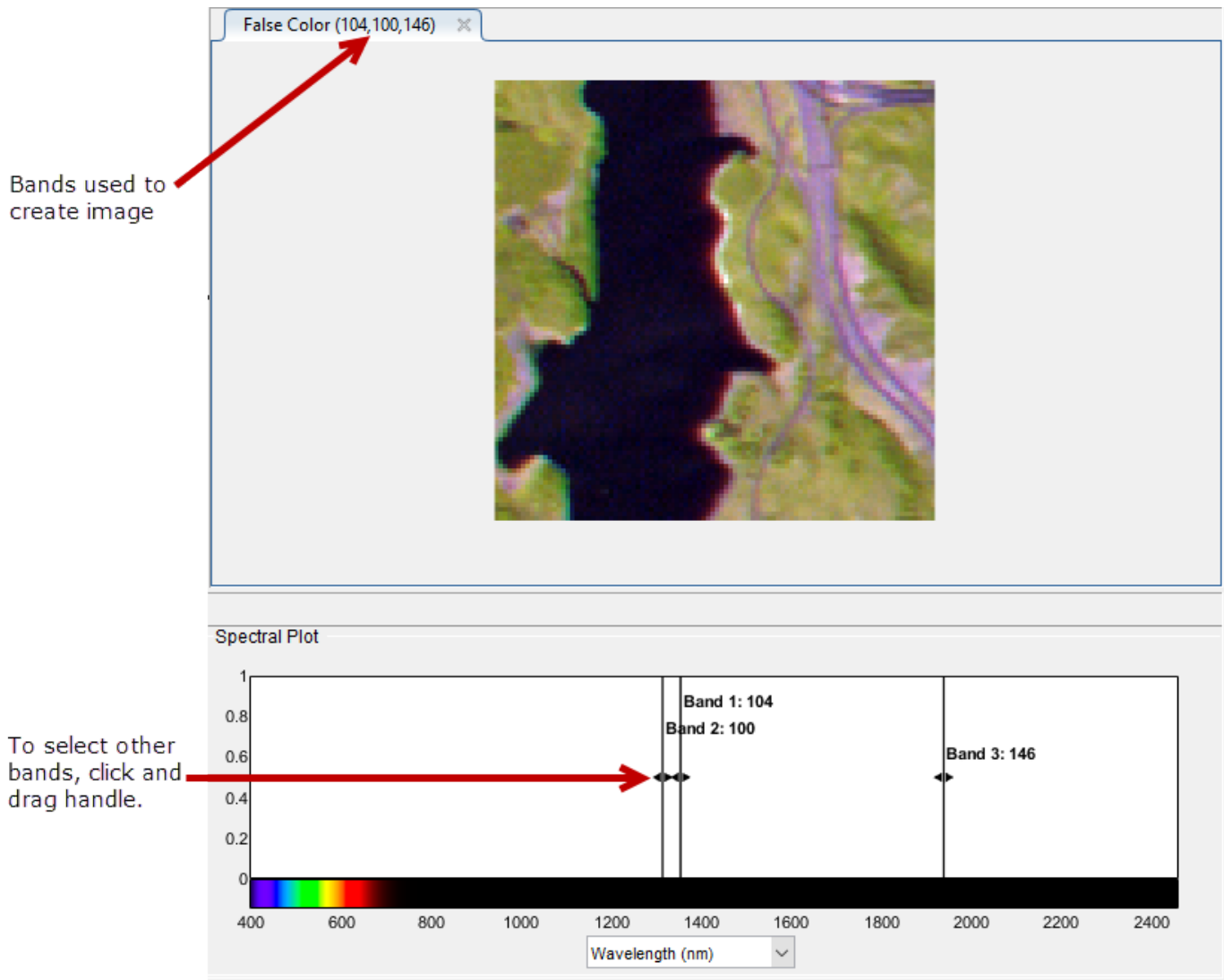
To create the **RGB** color composite image, the app chooses bands in the visible part of the electromagnetic spectrum. The resulting composite image resembles what the human eye would observe naturally. For example, vegetation appears green and water is bue. While RGB composites can appear natural to our eyes, it can be difficult to distinguish subtle differences in features. Natural color images can be low in contrast.

Bands used to create image.

False Color (104,100,146)   RGB (24,14,7)

Spectral Plot

Bands selected are all in the visible part of the spectrum.

Band 1: 24
Band 2: 14
Band 3: 7
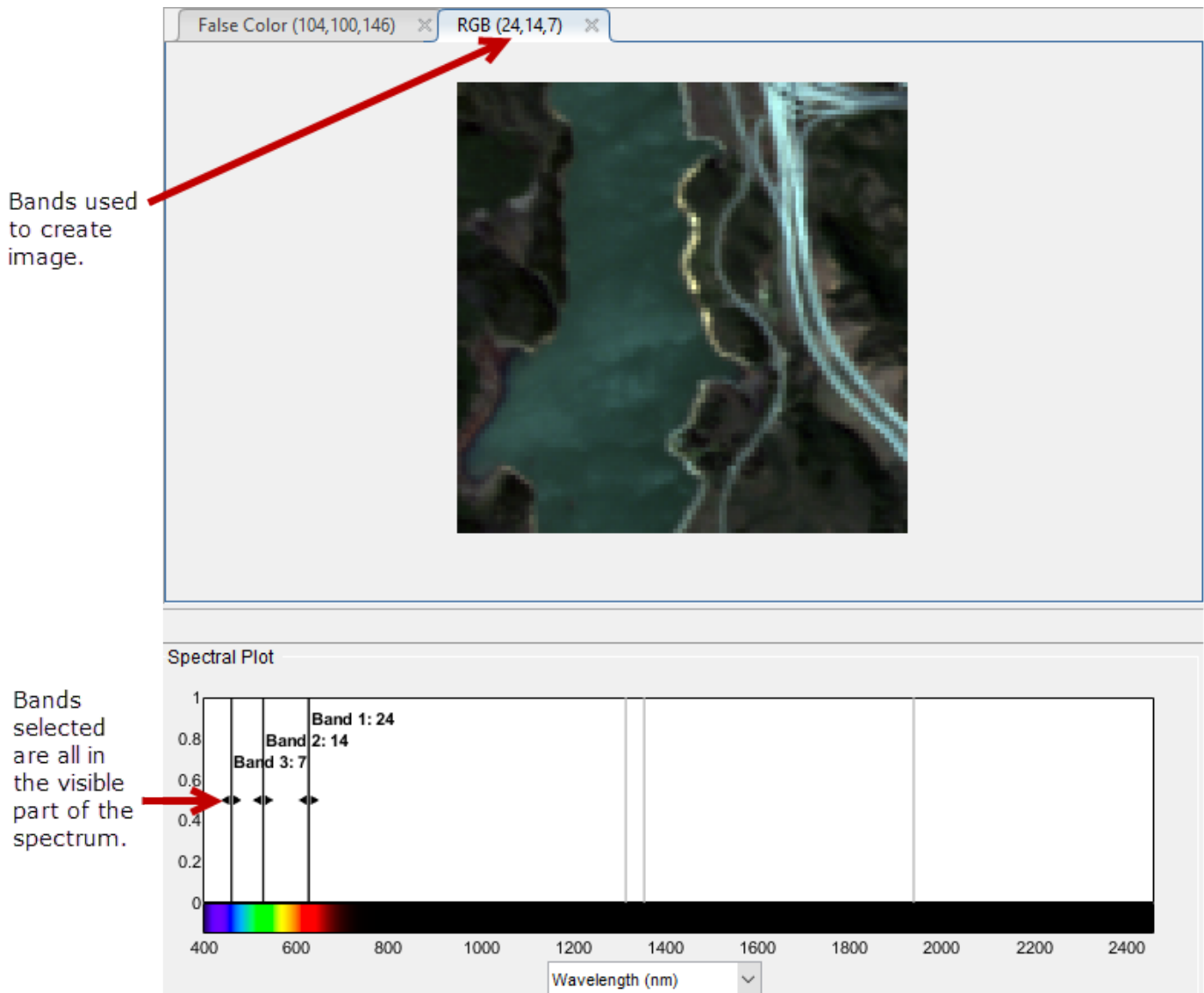
Wavelength (nm)

To create the **CIR** color composite image, the app chooses red, green, and near-infrared wavelengths. Near infrared wavelengths are slightly longer than red, and they are outside of the range visible to the human eye.

Bands used
to create
the image.

To select
other bands,
click and
drag handle.
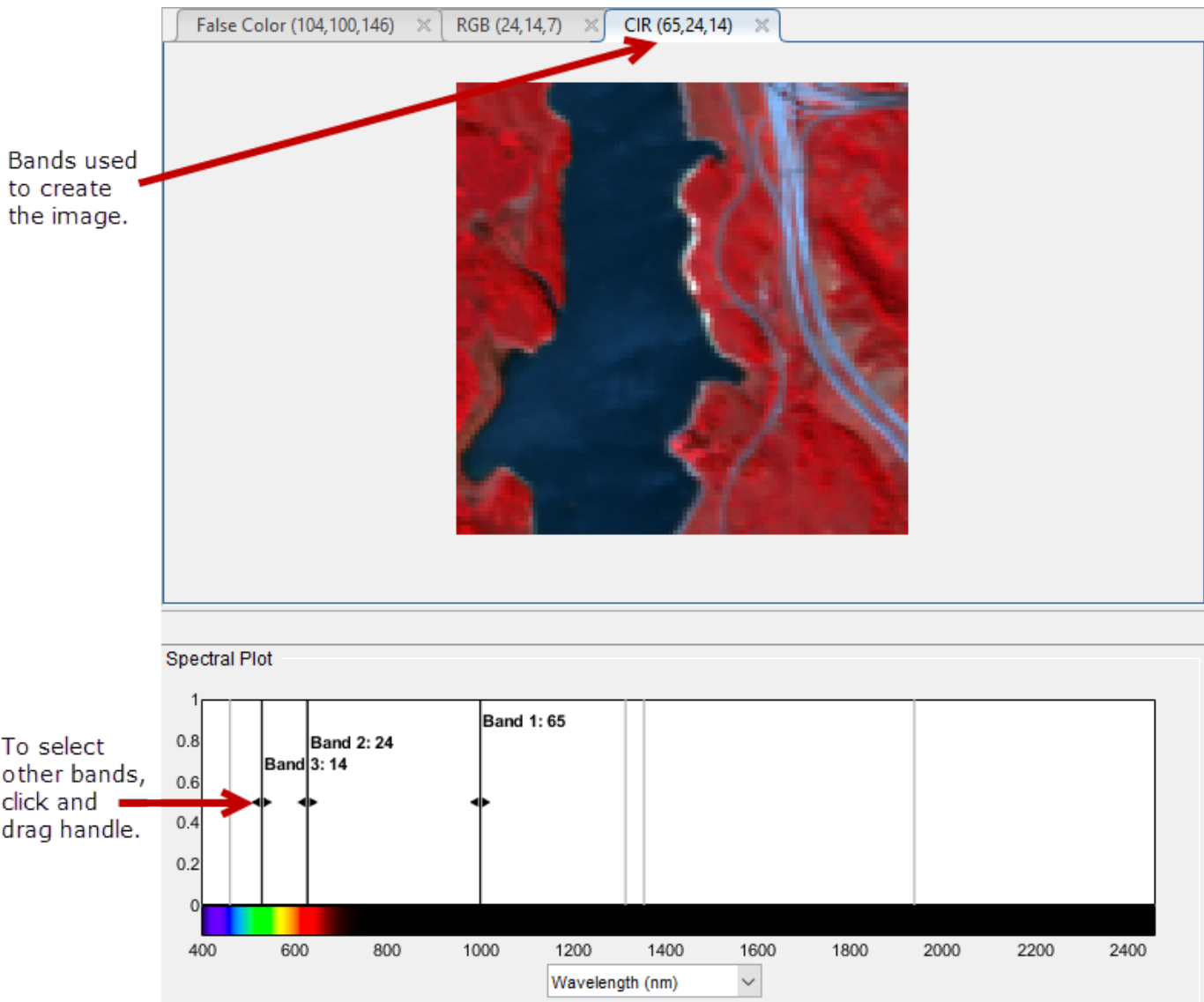
Create Spectral Profile Plots of Pixels and Regions

After exploring the grayscale and color visualizations of the hyperspectral data, you can plot points or small regions of the data along the spectral dimension to create spectral profiles. You can plot a single pixel or a region up to 10-by-10 pixels square. Use the **Neighborhood Size** slider to specify the region size. When you select a region, the app uses the mean of all the pixels in the region when plotting the data. Plotting a region, rather than an individual pixel, can smooth out spectral profiles.

To create a spectral plot, click **Add Spectral Plot** on the app toolstrip, move the cursor over a visualization in the app, and click to select the point or region. You can make your selections on any of the visualizations provided by the app. Your choice of which visualization to use can depend on which one provides the best view of the particular feature of the data you are interested in. When you make a selection, the app puts a point icon at that position on all of the visualizations. To select additional points, click **Add Spectral Plot** and repeat the process. To delete a point, right-click the point and choose **Delete Point** from the pop-up menu. To delete all of the points you have selected, click **Clear All** on the app toolstrip.

For example, the following figure shows four points selected in each visualization, each point representing a particular type of data: water, vegetation, road, and land.

Band Grayscale   False Color   RGB   CIR
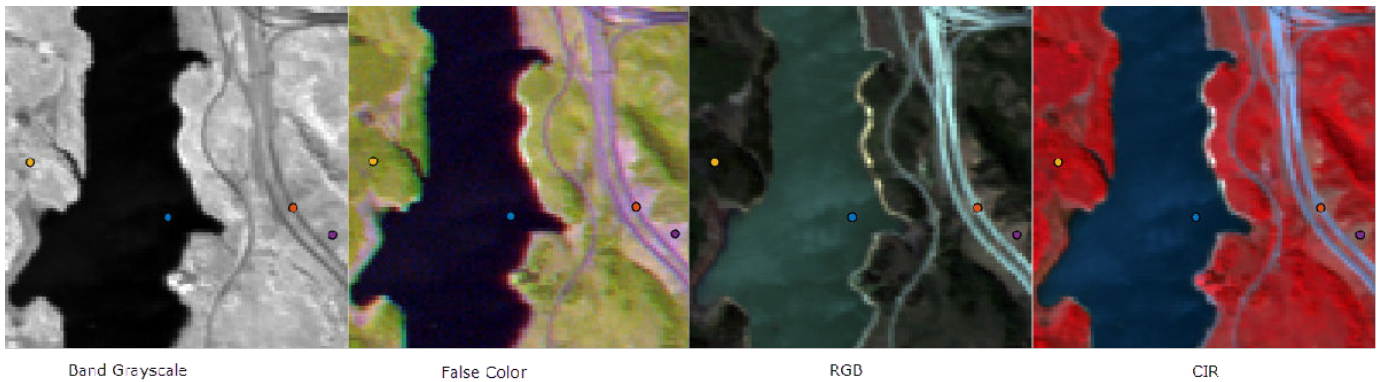
As you select each point, the app plots the data on the **Spectral Plot**, using a different color to identify each plot. By default, the **Spectral Plot** also includes a legend identifying the plot for each point. To toggle off inclusion of the legend, click **Show Legend**.
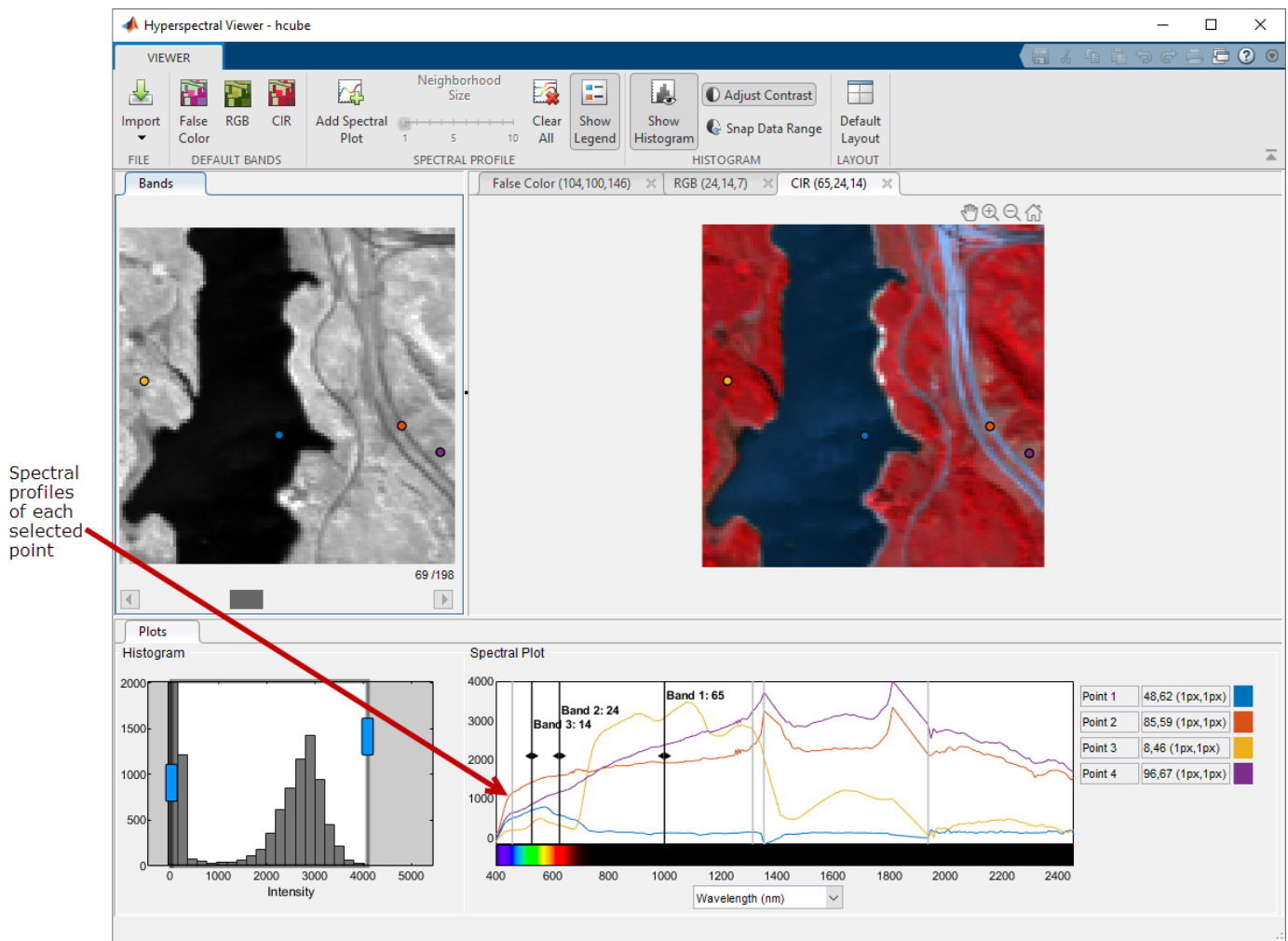
Hyperspectral Image Analysis Using Maximum Abundance Classification

This example shows how to identify different regions in a hyperspectral image by performing maximum abundance classification (MAC). An abundance map characterizes the distribution of an endmember across a hyperspectral image. Each pixel in the image is either a *pure pixel* or a *mixed pixel*. The set of abundance values obtained for each pixel represents the percentage of each endmembers present in that pixel. In this example, you will classify the pixels in a hyperspectral image by finding the maximum abundance value for each pixel and assigning it to the associated endmember class.

This example uses a data sample from the Pavia University dataset as test data. The test data contains nine endmembers that represent these ground truth classes: Asphalt, Meadows, Gravel, Trees, Painted metal sheets, Bare soil, Bitumen, Self blocking bricks, and Shadows.

Load and Visualize Data

Load the `.mat` file containing the test data into the workspace. The `.mat` file contains an array `paviaU`, representing the hyperspectral data cube and a matrix `signatures`, representing the nine endmember signatures taken from the hyperspectral data. The data cube has 103 spectral bands with wavelengths ranging from 430 nm to 860 nm. The geometric resolution is 1.3 meters and the spatial resolution of each band image is 610-by-340.

```
load('paviaU.mat');
image = paviaU;
sig = signatures;
```

Compute the central wavelength for each spectral band by evenly spacing the wavelength range across the number of spectral bands.

```
wavelengthRange = [430 860];
numBands = 103;
wavelength = linspace(wavelengthRange(1),wavelengthRange(2),numBands);
```

Create a `hypercube` object using the hyperspectral data cube and the central wavelengths. Then estimate an RGB image from the hyperspectral data. Set the `ContrastStretching` parameter value to `true` in order to improve the contrast of the RGB output. Visualize the RGB image.

```
hcube = hypercube(image,wavelength);
rgbImg = colorize(hcube,'Method','RGB','ContrastStretching',true);
figure
imshow(rgbImg)
```

The test data contains the endmember signatures of nine ground truth classes. Each column of `sig` contain the endmember signature of a ground truth class. Create a table that lists the class name for each endmember and the corresponding column of `sig`.

```
num = 1:size(sig,2);
endmemberCol = num2str(num');
```

```
classNames = {'Asphalt';'Meadows';'Gravel';'Trees';'Painted metal sheets';'Bare soil';...
              'Bitumen';'Self blocking bricks';'Shadows'};
table(endmemberCol,classNames,'VariableName',{'Column of sig';'Endmember Class Name'})
```

```
ans=9×2 table
    Column of sig        Endmember Class Name
    _____        _____

          1              {'Asphalt'              }
          2              {'Meadows'              }
          3              {'Gravel'               }
          4              {'Trees'                }
          5              {'Painted metal sheets'}
          6              {'Bare soil'            }
          7              {'Bitumen'              }
          8              {'Self blocking bricks'}
          9              {'Shadows'              }
```

Plot the endmember signatures.

```
figure
plot(sig)
xlabel('Band Number')
ylabel('Data Values')
ylim([400 2700])
title('Endmember Signatures')
legend(classNames,'Location','NorthWest')
```

Endmember Signatures
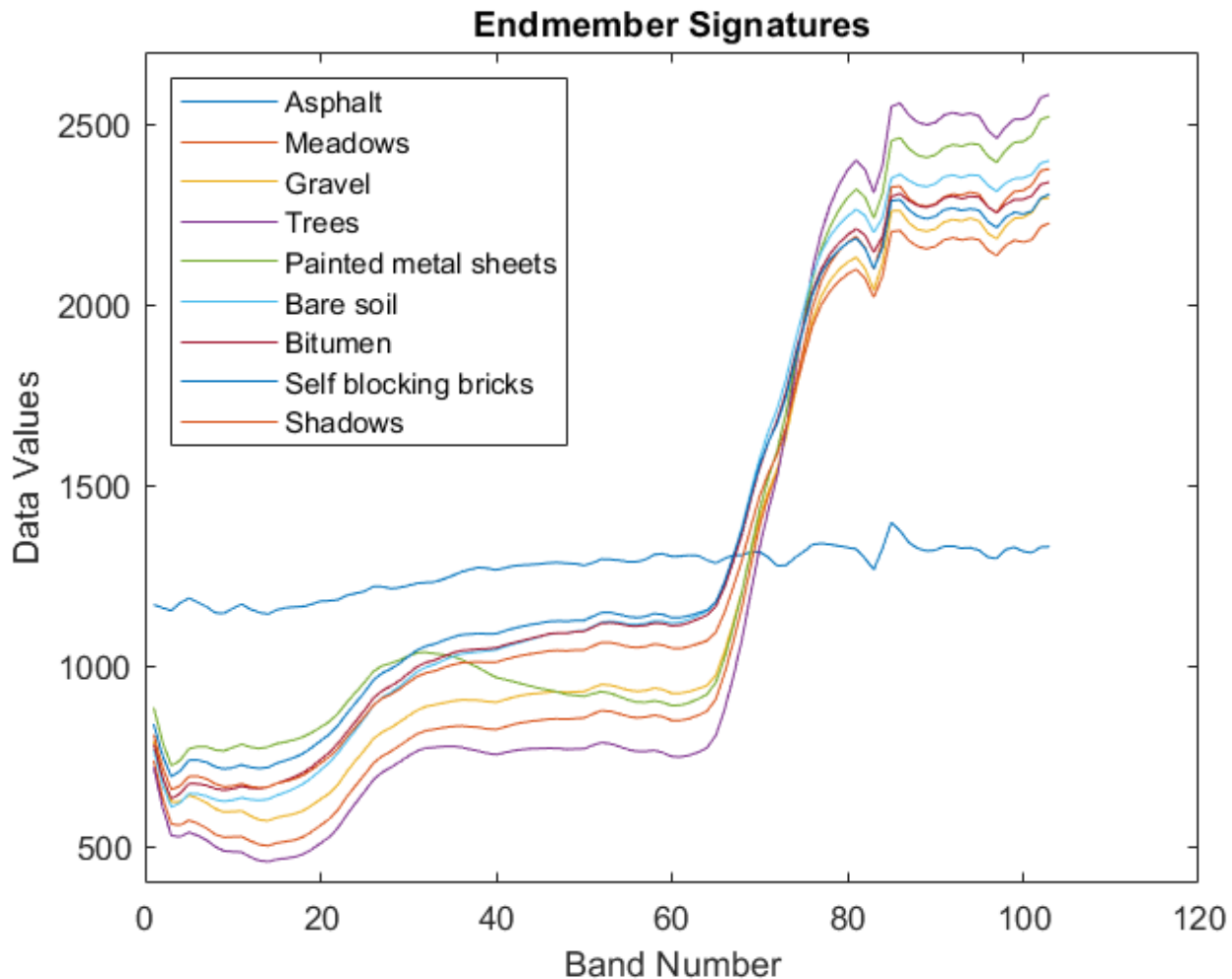
Estimate Abundance Maps

Create abundance maps the endmembers by using the `estimateAbundanceLS` function and select the method as full constrained least squares (FCLS). The function outputs the abundance maps as a 3-D array with the spatial dimensions as the input data. Each channel is the abundance map of the endmember from the corresponding column of signatures. In this example, the spatial dimension of the input data is 610-by-340 and number of endmembers is 9. So, the size of the output abundance map is 610-by-340-by-9.

```
abundanceMap = estimateAbundanceLS(hcube,sig,'Method','fcls');
```

Display the abundance maps.

```
fig = figure('Position',[0 0 1100 900]);
n = ceil(sqrt(size(abundanceMap,3)));
for cnt = 1:size(abundanceMap,3)
    subplot(n,n,cnt)
    imagesc(abundanceMap(:,:,cnt))
    title(['Abundance of ' classNames{cnt}])
    hold on
```

```
end
hold off
```
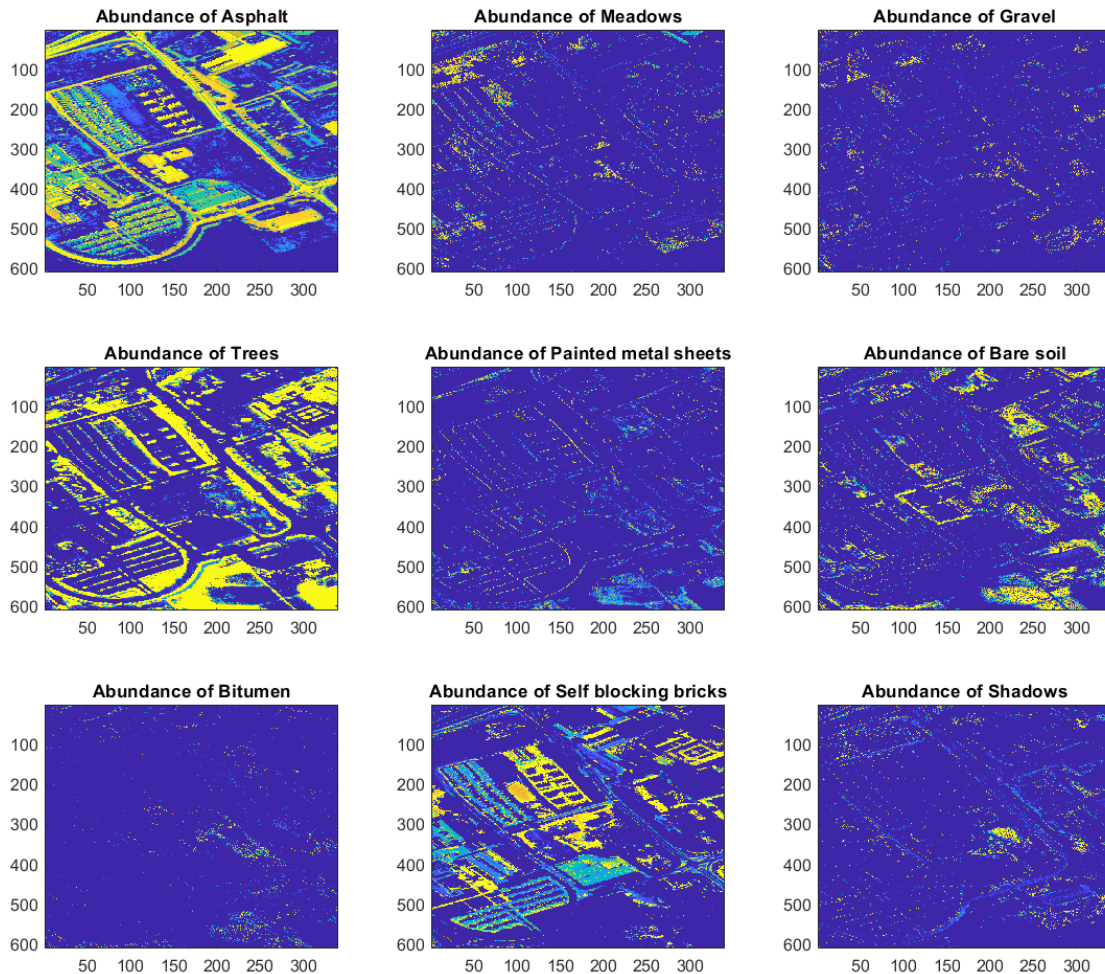

Perform Maximum Abundance Classification

Find the channel number of the largest abundance value for each pixel. The channel number returned for each pixel corresponds to the column in `sig` that contains the endmember signature associated with the maximum abundance value of that pixel. Display a color coded image of the pixels classified by maximum abundance value.

```
[~,matchIdx] = max(abundanceMap,[],3);
figure
imagesc(matchIdx)
colormap(jet(numel(classNames)))
colorbar('TickLabels',classNames)
```
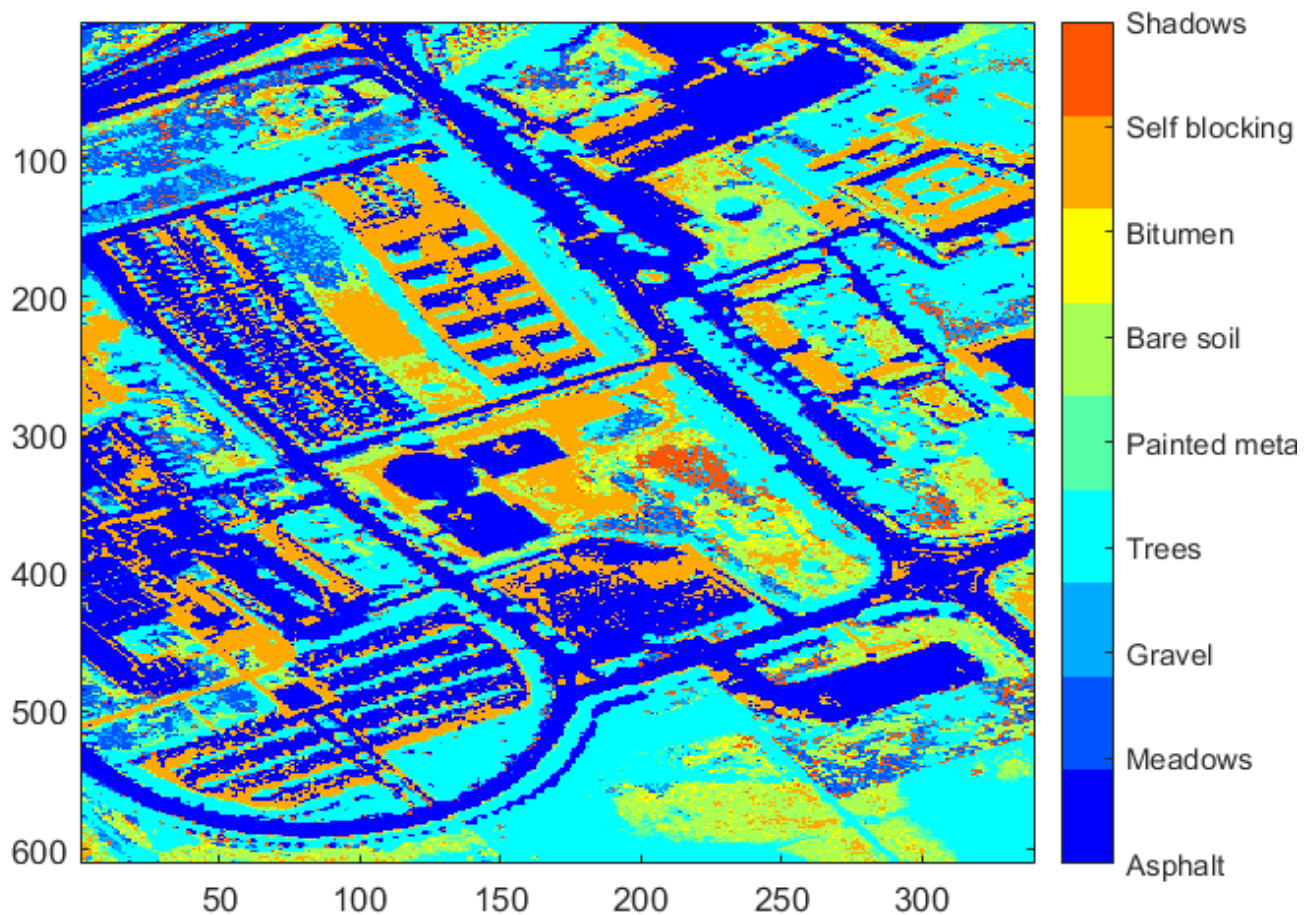
Segment the classified regions and overlay each of them on the RGB image estimated from the hyperspectral data cube.

```
segmentImg = zeros(size(matchIdx));
overlayImg = zeros(size(abundanceMap,1),size(abundanceMap,2),3,size(abundanceMap,3));
for i = 1:size(abundanceMap,3)
    segmentImg(matchIdx==i) = 1;
    overlayImg(:,:,:,i) = imoverlay(rgbImg,segmentImg);
    segmentImg = zeros(size(matchIdx));
end
```

Display the classified and the overlaid hyperspectral image regions along with their class names. From the images, you can see that the asphalt, trees, bare soil, and brick regions have been accurately classified.

```
figure('Position',[0 0 1100 900]);
n = ceil(sqrt(size(abundanceMap,3)));
for cnt = 1:size(abundanceMap,3)
    subplot(n,n,cnt);
    imagesc(uint8(overlayImg(:,:,:,cnt)));
    title(['Regions Classified as ' classNames{cnt}])
```

```
        hold on
end
hold off
```


Regions Classified as Asphalt

Regions Classified as Meadows

Regions Classified as Gravel

Regions Classified as Trees

Regions Classified as Painted metal sheets

Regions Classified as Bare soil

Regions Classified as Bitumen

Regions Classified as Self blocking bricks

Regions Classified as Shadows
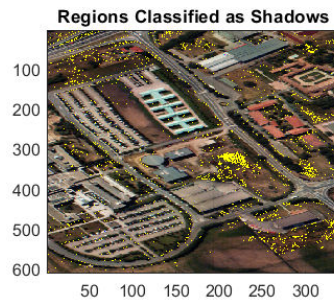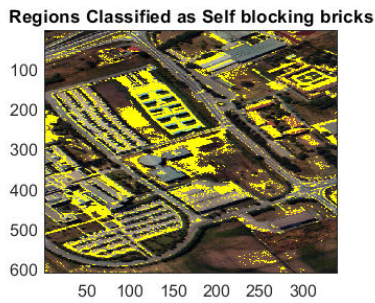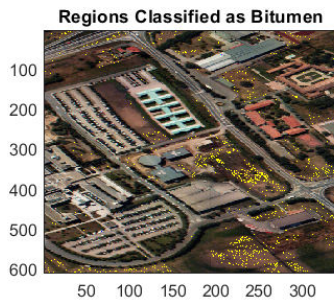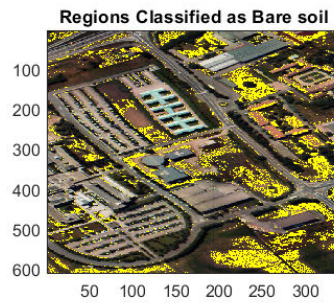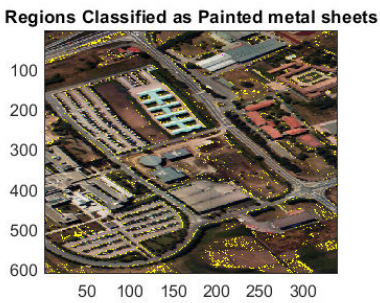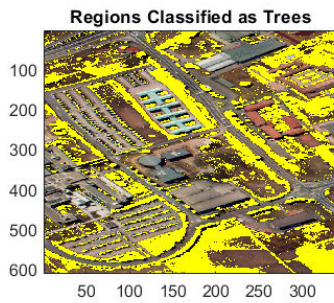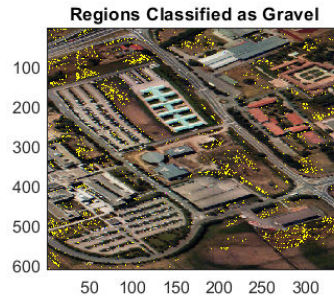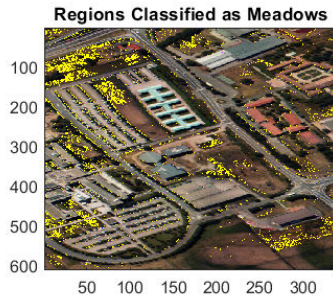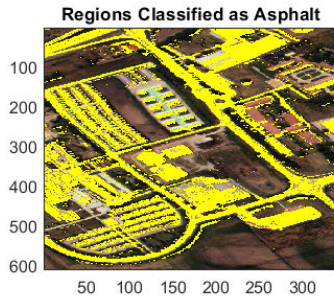
Classify Hyperspectral Image Using Library Signatures and SAM

This example shows how to classify pixels in a hyperspectral image by using the spectral angle mapper (SAM) classification algorithm. This algorithm classifies each pixel in the test data by computing the spectral match score between the spectra of a pixel and the pure spectral signatures read from the ECOSTRESS spectral library. This example uses a data sample from the Jasper Ridge dataset as the test data. The test data contains four endmembers latent, consisting of roads, soil, water, and trees. In this example, you will:

1 Generate a score map for different regions present in the test data by computing the SAM spectral match score between the spectra of each test pixel and the pure spectra. The pure spectra are from the ECOSTRESS spectral library.

2 Classify the regions by using minimum score criteria, and assign a class label for each pixel in the test data.

Read Test Data

Read test data from the Jasper Ridge dataset by using the `hypercube` function. The function returns a `hypercube` object, which stores the hyperspectral data cube and the corrsponding wavelength and metadata information read from the test data. The test data has 198 spectral bands and their wavelengths range from 399.4 nm to 2457 nm. The spectral resolution is up to 9.9 nm and the spatial resolution of each band image is 100-by-100.

```
hcube = hypercube('jasperRidge2_R198.img')

hcube =
  hypercube with properties:

      DataCube: [100×100×198 int16]
    Wavelength: [198×1 double]
      Metadata: [1×1 struct]
```

Estimate an RGB image from the data cube. Apply contrast stretching to enhance the contrast of the output RGB image.

```
rgbImg = colorize(hcube,'Method','rgb','ContrastStretching',true);
```

Display the RGB image of the test data.

```
figure
imagesc(rgbImg);
axis image off
title('RGB Image of Data Cube')
```

RGB Image of Data Cube

Read Signatures from ECOSTRESS Spectral Library

The ECOSTRESS spectral ibrary consists of pure spectral signatures for individual surface materials. If the spectra of a pixel matches a signature from the ECOSTRESS library, the pixel consists entirely of that single surface material. The library is a compilation of over 3400 spectral signatures for both natural and manmade materials. Since you know the the endmembers latent in the test data, choose the ECOSTRESS spectral library files related to those four endmembers.

Read spectral files related to water, vegetation, soil, and concrete from the ECOSTRESS spectral library. Use the spectral signatures of these types:

- Manmade to classify roads and highway structures
- Soil to classify sand, silt, and clay regions
- Vegetation to classify tree regions
- Water to classify water regions

```
fileroot = matlabshared.supportpkg.getSupportPackageRoot();
addpath(fullfile(fileroot,'toolbox','images','supportpackages','hyperspectral',...
    'hyperdata','ECOSTRESSSpectraFiles'));
```

```matlab
filenames = ["water.seawater.none.liquid.tir.seafoam.jhu.becknic.spectrum.txt",...
    "vegetation.tree.eucalyptus.maculata.vswir.jpl087.jpl.asd.spectrum.txt",...
    "soil.utisol.hapludult.none.all.87p707.jhu.becknic.spectrum.txt",...
    "soil.mollisol.cryoboroll.none.all.85p4663.jhu.becknic.spectrum.txt",...
    "manmade.concrete.pavingconcrete.solid.all.0092uuu_cnc.jhu.becknic.spectrum.txt"];
lib = readEcostressSig(filenames)
```

lib=1×5 struct array with fields:
    Name
    Type
    Class
    SubClass
    ParticleSize
    Genus
    Species
    SampleNo
    Owner
    WavelengthRange
    Origin
    CollectionDate
    Description
    Measurement
    FirstColumn
    SecondColumn
    WavelengthUnit
    DataUnit
    FirstXValue
    LastXValue
    NumberOfXValues
    AdditionalInformation
    Wavelength
    Reflectance
      ⋮

Extract the class names from the library structure.

```matlab
classNames = [lib.Class];
```

Plot the pure spectral signatures read from the ECOSTRESS spectral library.

```matlab
figure
hold on
for idx = 1:numel(lib)
    plot(lib(idx).Wavelength,lib(idx).Reflectance,'LineWidth',2)
end
axis tight
box on
title('Pure Spectral Signatures from ECOSTRESS Library')
xlabel('Wavelength (\mum)')
ylabel('Reflectance (%)')
legend(classNames,'Location','northeast')
title(legend,'Class Names')
hold off
```

Pure Spectral Signatures from ECOSTRESS Library
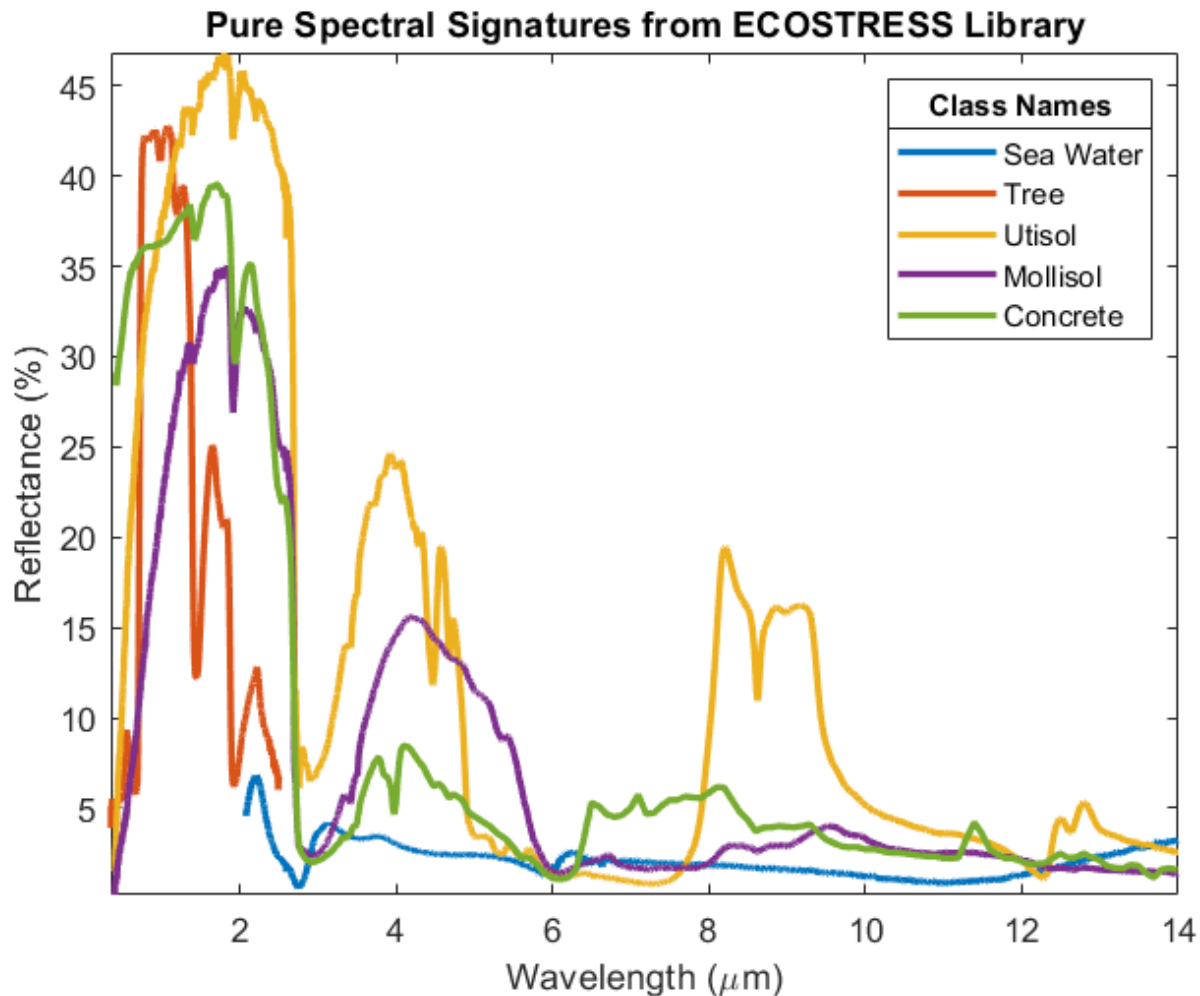
Compute Score Map for Pixels in Test Data

Find the spectral match score between each pixel spectra and the library signatures by using the `spectralMatch` function. By default, the `spectralMatch` function computes the degree of similarity between two spectra by using the SAM classification algorithm. The function returns an array with the same spatial dimensions as the hyperspectral data cube and channels equal to the number of library signatures specified. Each channel contains the score map for a single library signature. In this example, there are five ECOSTRESS spectral library files specified for comparison, and each band of the hyperspectral data cube has spatial dimensions of 100-by-100 pixels. The size of the output array of score maps thus is 100-by-100-by-5.

```
scoreMap = spectralMatch(lib,hcube);
```

Display the score maps.

```
figure
montage(scoreMap,'Size',[1 numel(lib)],'BorderSize',10)
title('Score Map Obtained for Each Pure Spectra','FontSize',14)
colormap(jet);
colorbar
```

Score Map Obtained for Each Pure Spectra
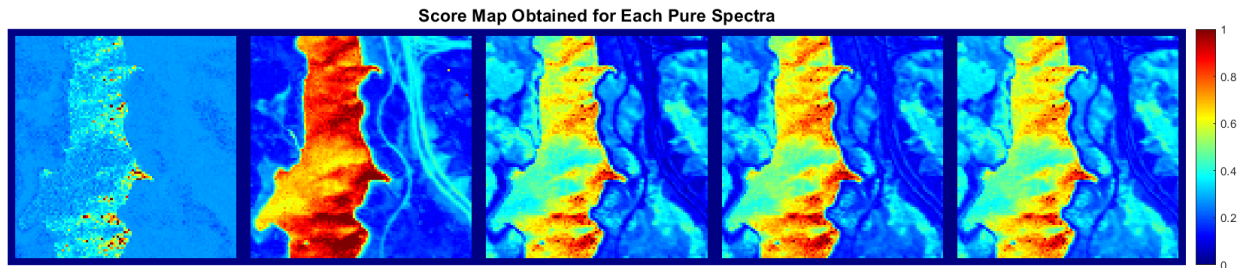
Classify Pixels Using Minimum Score Criteria

Lower SAM values indicate higher spectral similarity. Use the minimum score citeria to classify the test pixels by finding the best match for each pixel among the library signatures. The result is a pixel-wise classification map in which the value of each pixel is the index of library signature file in `lib` for which that pixel exhibits the lowest SAM value. For example, if the value of a pixel in the classification map is 1, the pixel exhibits high similarity to the first library signature in `lib`.

```
[~,classMap] = min(scoreMap,[],3);
```

Create a class table that maps the classification map values to the ECOSTRESS library signatures used for spectral matching.

```
classTable = table((min(classMap(:)):max(classMap(:)))',classNames',...
            'VariableNames',{'Classification map value','Matching library signature'})
```

classTable=*5×2 table*

Classification map value	Matching library signature
1	"Sea Water"
2	"Tree"
3	"Utisol"
4	"Mollisol"
5	"Concrete"

Display the RGB image of the hyperspectral data and the classification results. Visual inspection shows that spectral matching classifies each pixel effectively.

```
fig = figure('Position',[0 0 900 500]);
axes1 = axes('Parent',fig,'Position',[0.04 0.11 0.4 0.82]);
imagesc(rgbImg,'Parent',axes1);
axis off
title('RGB Image of Data Cube')
axes2 = axes('Parent',fig,'Position',[0.47 0.11 0.45 0.82]);
imagesc(classMap,'Parent',axes2)
axis off
colormap(jet(numel(lib)))
title('Pixel-wise Classification Map')
ticks = linspace(1.4,4.8,numel(lib));
colorbar('Ticks',ticks,'TickLabels',classNames)
```
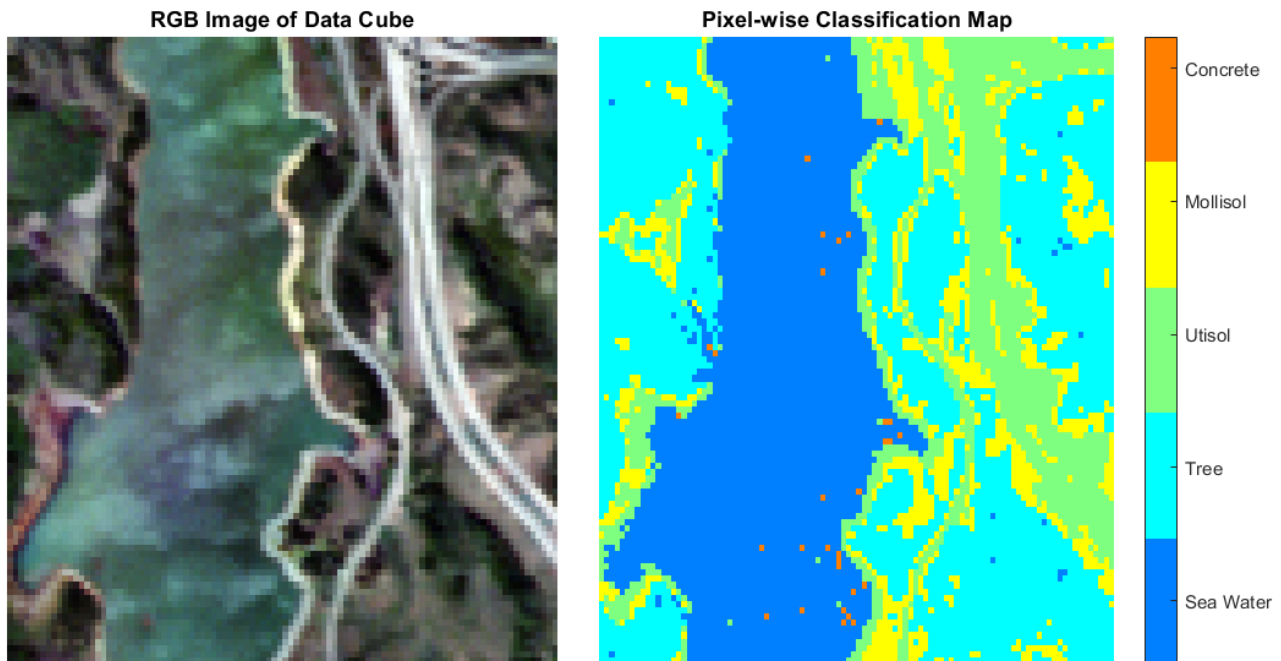
RGB Image of Data Cube

Pixel-wise Classification Map

References

[1] Kruse, F.A., A.B. Lefkoff, J.W. Boardman, K.B. Heidebrecht, A.T. Shapiro, P.J. Barloon, and A.F.H. Goetz. "The Spectral Image Processing System (SIPS)—Interactive Visualization and Analysis of Imaging Spectrometer Data." *Remote Sensing of Environment* 44, no. 2–3 (May 1993): 145–63. https://doi.org/10.1016/0034-4257(93)90013-N.

Endmember Material Identification Using Spectral Library

This example shows how to identify the classes of endmember materials present in a hyperspectral image. The endmembers are pure spectral signature that signifies the reflectance characteristics of pixels belonging to a single surface material. The existing endmember extraction or identification algorithms extracts or identifies the pure pixels in a hyperspectral image. However, these techniques do not identify the material name or class to which the endmember spectra belong to. In this example, you will extract the endmember signatures and then, classify or identify the class of an endmember material in the hyperspectral image by using spectral matching.

This example uses 1) the spectral signatures in the ECOSTRESS spectral library as the reference spectra and 2) a data sample from the Jasper Ridge dataset as the test data, for endmember material identification.

Read Reference Data from ECOSTRESS Spectral Library

Add the full file path containing the ECOSTRESS library files and specify the names of the files to be read from the library.

```
fileroot = matlabshared.supportpkg.getSupportPackageRoot();
addpath(fullfile(fileroot,'toolbox','images','supportpackages','hyperspectral','hyperdata',''ECOS'
filenames = ["water.seawater.none.liquid.tir.seafoam.jhu.becknic.spectrum.txt",...
             "water.tapwater.none.liquid.all.tapwater.jhu.becknic.spectrum.txt",...
             "water.ice.none.solid.all.ice_dat_.jhu.becknic.spectrum.txt",...
             "vegetation.tree.eucalyptus.maculata.vswir.jpl087.jpl.asd.spectrum.txt",...
             "soil.utisol.hapludult.none.all.87p707.jhu.becknic.spectrum.txt",...
             "soil.mollisol.cryoboroll.none.all.85p4663.jhu.becknic.spectrum.txt",...
             "manmade.road.tar.solid.all.0099uuutar.jhu.becknic.spectrum.txt",...
             "manmade.concrete.pavingconcrete.solid.all.0092uuu_cnc.jhu.becknic.spectrum.txt"];
lib = readEcostressSig(filenames);
```

Display the `lib` data and inspect its values. The data is a struct variable specifying the class, subclass, wavelength, and reflectance related information.

```
lib
```

```
lib=1×8 struct array with fields:
    Name
    Type
    Class
    SubClass
    ParticleSize
    Genus
    Species
    SampleNo
    Owner
    WavelengthRange
    Origin
    CollectionDate
    Description
    Measurement
    FirstColumn
    SecondColumn
    WavelengthUnit
    DataUnit
    FirstXValue
    LastXValue
```

```
        NumberOfXValues
        AdditionalInformation
        Wavelength
        Reflectance
            ⋮
```

Plot the spectral signatures read from the ECOSTRESS spectral library.

```
figure
hold on
for idx = 1:numel(lib)
    plot(lib(idx).Wavelength,lib(idx).Reflectance,'LineWidth',2);
end
axis tight
box on
xlabel('Wavelength (\mum)');
ylabel('Reflectance (%)');
classNames = {lib.Class};
legend(classNames,'Location','northeast')
title('Reference Spectra from ECOSTRESS Library');
hold off
```

Reference Spectra from ECOSTRESS Library
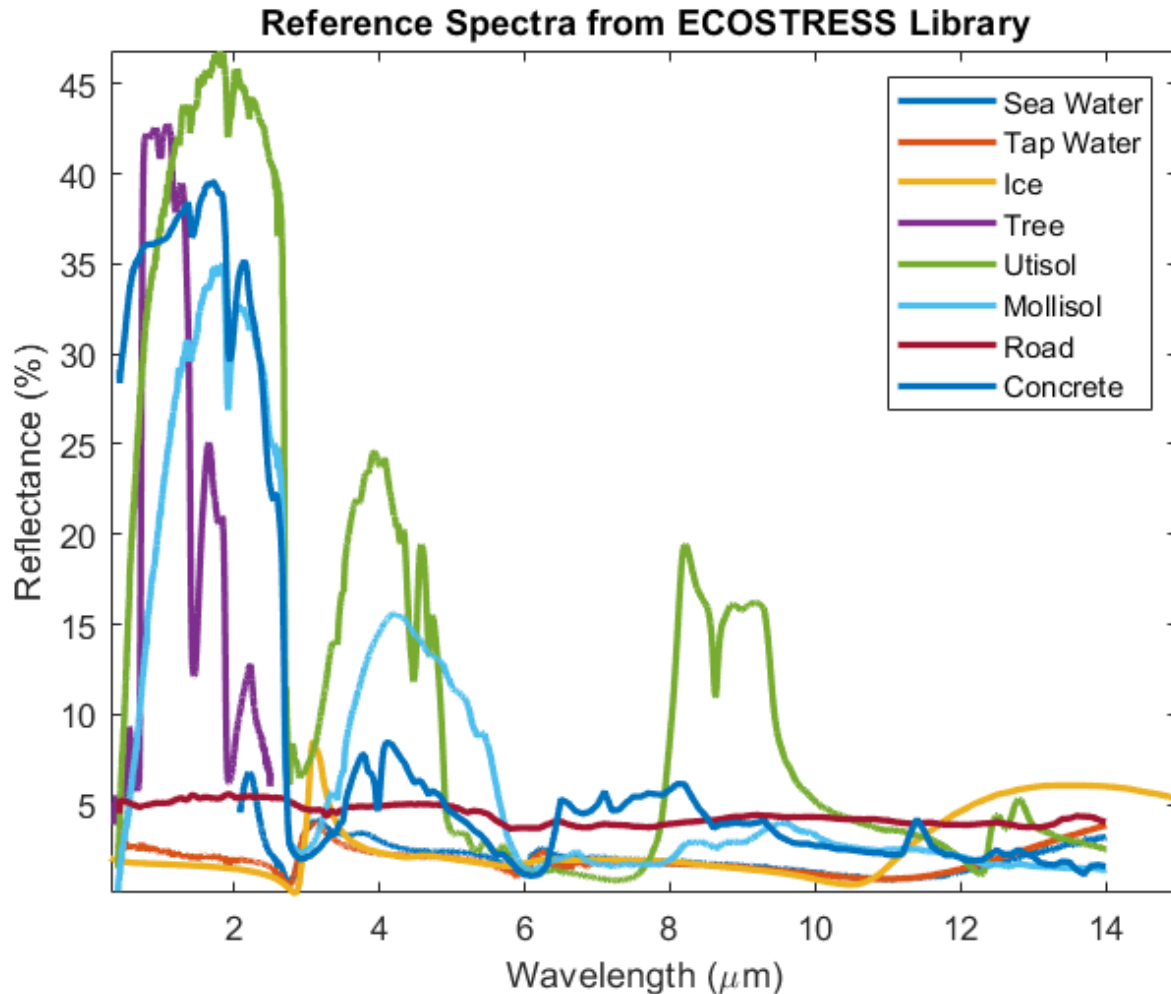
Read Test Data

Read a test data from Jasper Ridge dataset by using the `hypercube` function. The function returns a `hypercube` object that stores the data cube and the metadata information read from the test data. The test data has 198 spectral bands and their wavelengths range from 399.4 nm to 2457 nm. The spectral resolution is up to 9.9 nm and the spatial resolution of each band image is 100-by-100. The test data contains four endmembers latent that includes road, soil, water, and trees.

```
hcube = hypercube('jasperRidge2_R198.hdr');
```

Extract Endmember Spectra

To compute the total number of spectrally distinct endmembers present in the test data, use the `countEndmembersHFC` function. This function finds the number of endmembers by using the Harsanyi–Farrand–Chang (HFC) method. Set the probability of false alarm (PFA) to a low value in order to avoid false detections.

```
numEndmembers = countEndmembersHFC(hcube,'PFA',10^-27);
```

Extract the endmembers of the test data by using the N-FINDR method.

```
endMembers = nfindr(hcube,numEndmembers);
```

Read the wavelength values from the `hypercube` object `hcube`. Plot the extracted endmember signatures. The test data comprises of 4 endmember materials and the class names of these materials can be identified through spectral matching.

```
figure
plot(hcube.Wavelength,endMembers,'LineWidth',2)
axis tight
xlabel('Wavelength (nm)')
ylabel('Data Values')
title('Endmembers Extracted using N-FINDR')
num = 1:numEndmembers;
legendName = strcat('Endmember',{' '},num2str(num'));
legend(legendName)
```
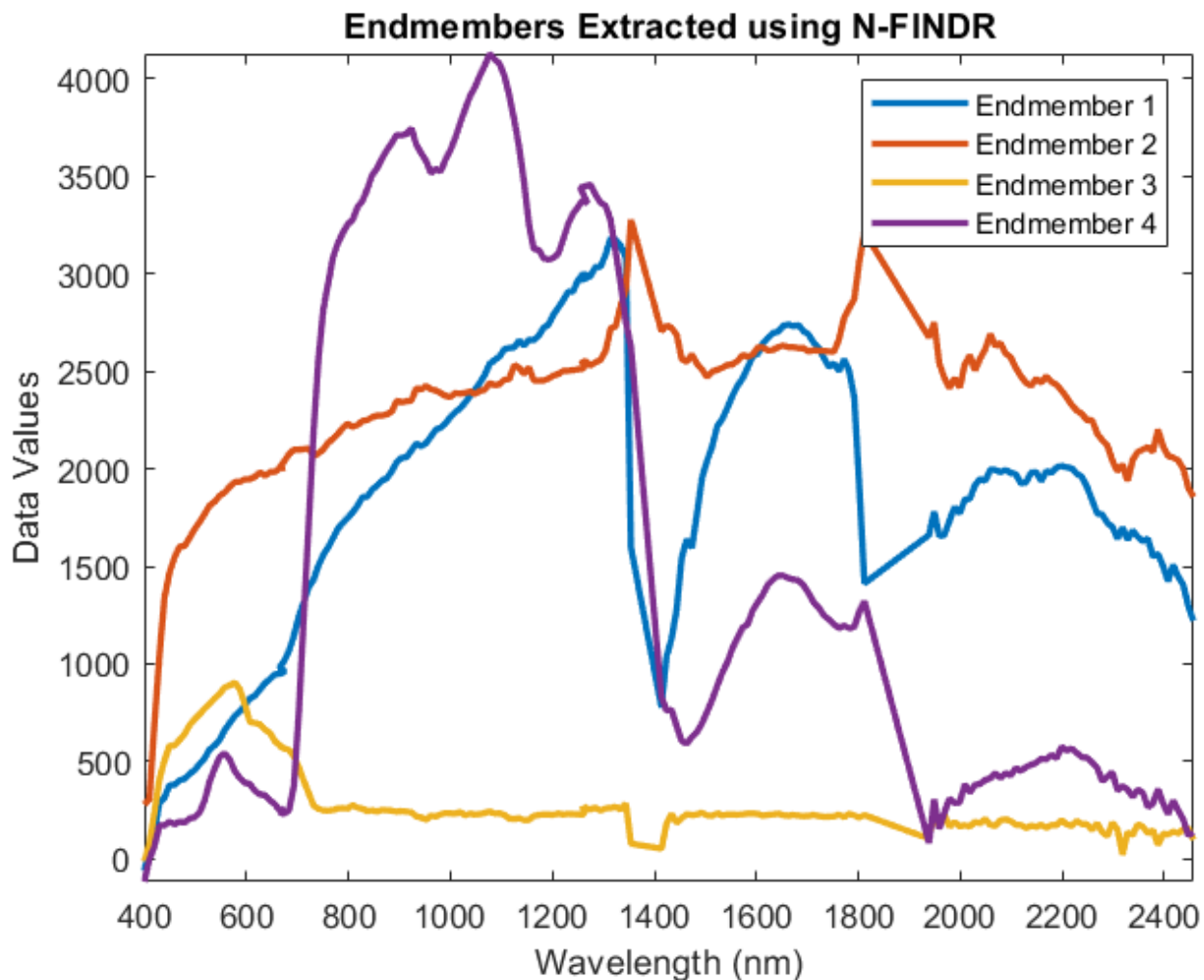

Identify Endmember Material

To identify the name of an endmember material, use the `spectralMatch` function. The function computes the spectral similarity between the library files and an endmember spectra to be classified. Select spectral information divergence (SID) method for computing the matching score. Typically, a

low value of SID score means better matching between the test and the reference spectra. Then, the test spectra is classified to belong to the class of the best matching reference spectra.

For example, to identify the class of the third and fourth endmember material, find the spectral similarity between the library signatures and the respective endmember spectra. The index of the minimum SID score value specifies the class name in the spectral library. The third endmember spectra is identified as Sea Water and the fourth endmember spectra is identified as Tree.

```matlab
wavelength = hcube.Wavelength;
detection = cell(1,1);
cnt = 1;
queryEndmember = [3 4];
for num = 1:numel(queryEndmember)
    spectra = endMembers(:,queryEndmember(num));
    scoreValues = spectralMatch(lib,spectra,wavelength,'Method','sid');
    [~, matchIdx] = min(scoreValues);
    detection{cnt} = lib(matchIdx).Class;
    disp(strcat('Endmember spectra ',{' '},num2str(queryEndmember(num)),' is identified as ',{'
    cnt=cnt+1;
end
```

```
Endmember spectra 3 is identified as Sea Water
Endmember spectra 4 is identified as Tree
```

Segment Endmember Regions in Test Data

To visually inspect the identification results, localise and segment the image regions specific to the endmember materials in the test data. Use the `sid` function to compute pixel-wise spectral similairty between the pixel spectra and the extracted endmember spectra. Then, perform thresholding to segment the desired endmember regions in the test data and generate the segmented image. Select the value for threshold as 15 to select the best matching pixels.

For visualization, generate the RGB version of the test data by using the `colorize` function and then, overlay the segmented image onto the test image.

```matlab
threshold = 15;
rgbImg = colorize(hcube,'method','rgb','ContrastStretching',true);
overlayImg = rgbImg;
labelColor = {'Blue','Green'};
segmentedImg = cell(size(hcube.DataCube,1),size(hcube.DataCube,2),numel(queryEndmember));
for num = 1:numel(queryEndmember)
    scoreMap = sid(hcube,endMembers(:,queryEndmember(num)));
    segmentedImg{num} = scoreMap <= threshold;
    overlayImg = imoverlay(overlayImg,segmentedImg{num},labelColor{num});
end
```

Display Results

Visually inspect the identification results by displaying the segmented images and the overlayed image that highlights the Sea Water and Tree endmember regions in the test data.

```matlab
figure('Position',[0 0 900 400])
plotdim = [0.02 0.2 0.3 0.7;0.35 0.2 0.3 0.7];
for num = 1:numel(queryEndmember)
    subplot('Position',plotdim(num,:))
    imagesc(segmentedImg{num})
    title(strcat('Segmented Endmember region :',{' '},detection{num}));
```

```
        colormap([0 0 0;1 1 1])
        axis off
    end
```


Segmented Endmember region : Sea Water **Segmented Endmember region : Tree**

```
figure('Position',[0 0 900 400])
subplot('Position',[0 0.2 0.3 0.7])
imagesc(rgbImg)
title('RGB Transformation of Test Data');
axis off
subplot('Position',[0.35 0.2 0.3 0.7])
imagesc(overlayImg)
title('Overlay Segmented Regions')
hold on
dim = [0.66 0.6 0.3 0.3];
annotation('textbox',dim,'String','Sea Water','Color',[1 1 1],'BackgroundColor',[0 0 1],'FitBoxTo
dim = [0.66 0.5 0.3 0.3];
annotation('textbox',dim,'String','Tree','BackgroundColor',[0 1 0],'FitBoxToText','on');
hold off
axis off
```
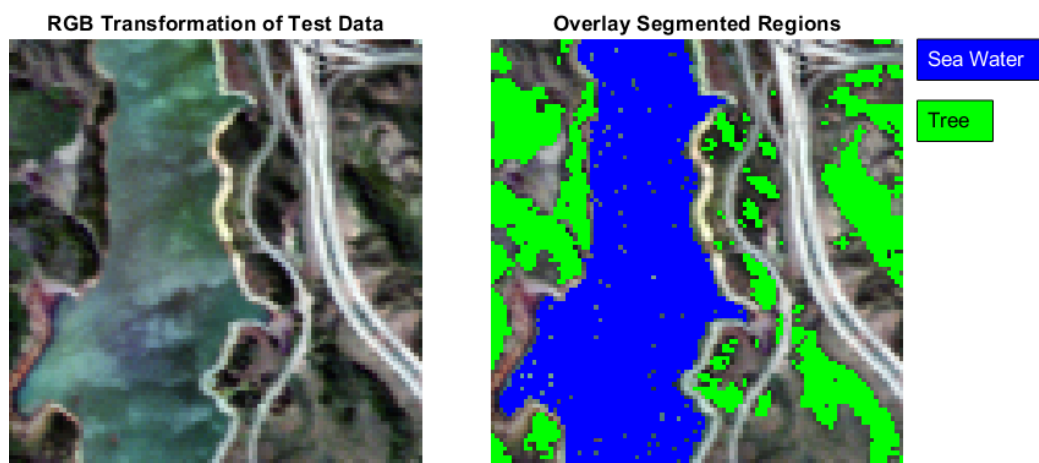
RGB Transformation of Test Data **Overlay Segmented Regions**

Sea Water

Tree

References

[1] Kruse, F.A., A.B. Lefkoff, J.W. Boardman, K.B. Heidebrecht, A.T. Shapiro, P.J. Barloon, and A.F.H. Goetz. "The Spectral Image Processing System (SIPS)—Interactive Visualization and Analysis of Imaging Spectrometer Data." *Remote Sensing of Environment* 44, no. 2–3 (May 1993): 145–63. https://doi.org/10.1016/0034-4257(93)90013-N.

Target Detection Using Spectral Signature Matching

This example shows how to detect a known target in the hyperspectral image by using spectral matching method. The pure spectral signature of the known target material is used to detect and locate the target in a hyperspectral image. In this example, you will use the spectral angle mapper (SAM) spectral matching method to detect man-made roofing materials (known target) in a hyperspectral image. The pure spectral signature of the roofing material is read from the ECOSTRESS spectral lbrary and is used as the reference spectra for spectral matching. The spectral signatures of all the pixels in the data cube are compared with the reference spectra and the best matching pixel spectra are classified as belonging to the target material.

This example uses the data sample taken from the Pavia Universtity dataset as the test data. The dataset contains endmember signatures for 9 groundtruth classes and each signature is a vector of length 103. The ground truth classes include Asphalt, Meadows, Gravel, Trees, Painted metal sheets, Bare soil, Bitumen, Self blocking bricks, and Shadows. Of these classes, the painted metal sheets typically belongs to the roofing materials type and it is the desired target to be located.

Read Test Data

Read the test data from Pavia University dataset by using the `hypercube` function. The function returns a `hypercube` object that stores the data cube and the metadata information read from the test data. The test data has 103 spectral bands and their wavelengths range from 430 nm to 860 nm. The geometric resolution is 1.3 meters and the spatial resolution of each band image is 610-by-340.

```
hcube = hypercube('paviaU.hdr');
```

Estimate an RGB color image from the data cube by using the `colorize` function. Set the `ContrastStretching` parameter value to `true` in order to improve the contrast of RGB color image. Display the RGB image.

```
rgbImg = colorize(hcube,'Method','rgb','ContrastStretching',true);
figure
imshow(rgbImg)
title('RGB Image')
```

RGB Image

Read Reference Spectra

Read the spectral information corresponding to a roofing material from the ECOSTRESS spectral library by using the `readEcostressSig` function. Add the full file path containing the ECOSTRESS spectral file and read the spectral signature for roofing material from the specified location.

```
fileroot = matlabshared.supportpkg.getSupportPackageRoot();
addpath(fullfile(fileroot,'toolbox','images','supportpackages','hyperspectral','hyperdata',''ECOS'
lib = readEcostressSig("manmade.roofingmaterial.metal.solid.all.0692uuucop.jhu.becknic.spectrum.
```

Inspect the properties of the referece spectra read from the ECOSTRESS library. The output
structure `lib` stores the metadata and the data values read from the ECOSTRESS library.

```
lib
```

```
lib = struct with fields:
                       Name: "Copper Metal"
                       Type: "manmade"
                      Class: "Roofing Material"
                   SubClass: "Metal"
               ParticleSize: "Solid"
                      Genus: [0×0 string]
                    Species: [0×0 string]
                   SampleNo: "0692UUUCOP"
                      Owner: "National Photographic Interpretation Center"
            WavelengthRange: "All"
                     Origin: "Spectra obtained from the Noncoventional Exploitation FactorsData Sys
             CollectionDate: "N/A"
                Description: "Extremely weathered bare copper metal from government building roof
                Measurement: "Directional (10 Degree) Hemispherical Reflectance"
                FirstColumn: "X"
               SecondColumn: "Y"
             WavelengthUnit: "micrometer"
                   DataUnit: "Reflectance (percent)"
                FirstXValue: "0.3000"
                 LastXValue: "12.5000"
            NumberOfXValues: "536"
      AdditionalInformation: "none"
                 Wavelength: [536×1 double]
                 Reflectance: [536×1 double]
```

Read the wavelength and the reflectance values stored in `lib`. The wavelength and the reflectance
pair comprises the reference spectra or the reference spectral signature.

```
wavelength = lib.Wavelength;
reflectance = lib.Reflectance;
```

Plot the reference spectra read from the ECOSTRESS library.

```
plot(wavelength,reflectance,'LineWidth',2)
axis tight
xlabel('Wavelength (\mum)')
ylabel('Reflectance (%)')
title('Reference Spectra')
```
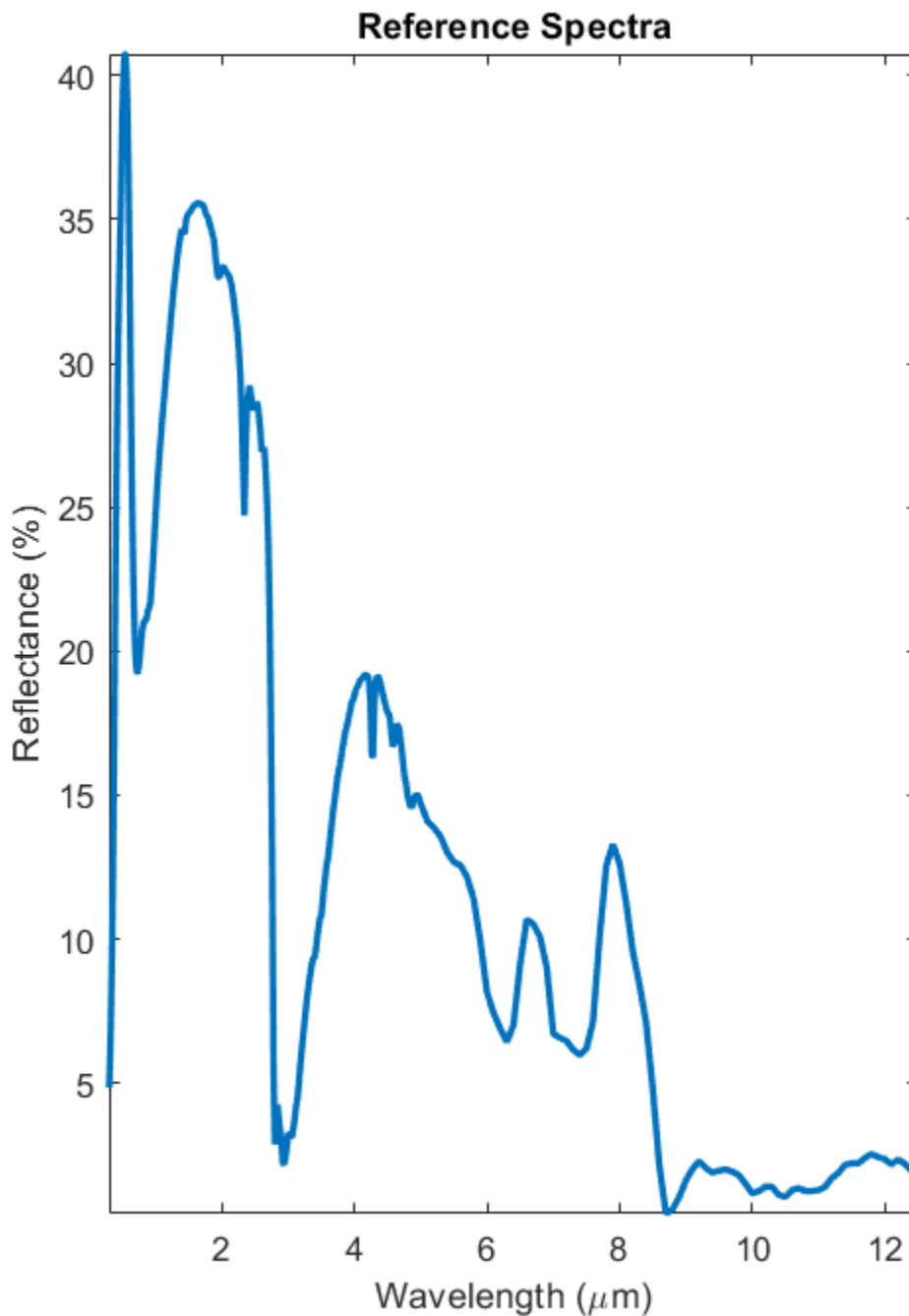
Reference Spectra

Perform Spectral Matching

Find the spectral similarity between the reference spectra and the data cube by using the `spectralMatch` function. By default, the function uses the spectral angle mapper (SAM) method for finding the spectral match. The output is a score map that signifies the matching between each pixel spectra and the reference spectra. Thus, the score map is a matrix of spatial dimension same as that

of the test data. In this case, the size of the score map is 610-by-340. SAM is insensitive to gain factors and hence, can be used to match pixel spectra that inherently have an unknown gain factor due to topographic illumination effects.

```
scoreMap = spectralMatch(lib,hcube);
```

Display the score map.

```
figure('Position',[0 0 500 600])
imagesc(scoreMap)
colormap parula
colorbar
title('Score Map')
```
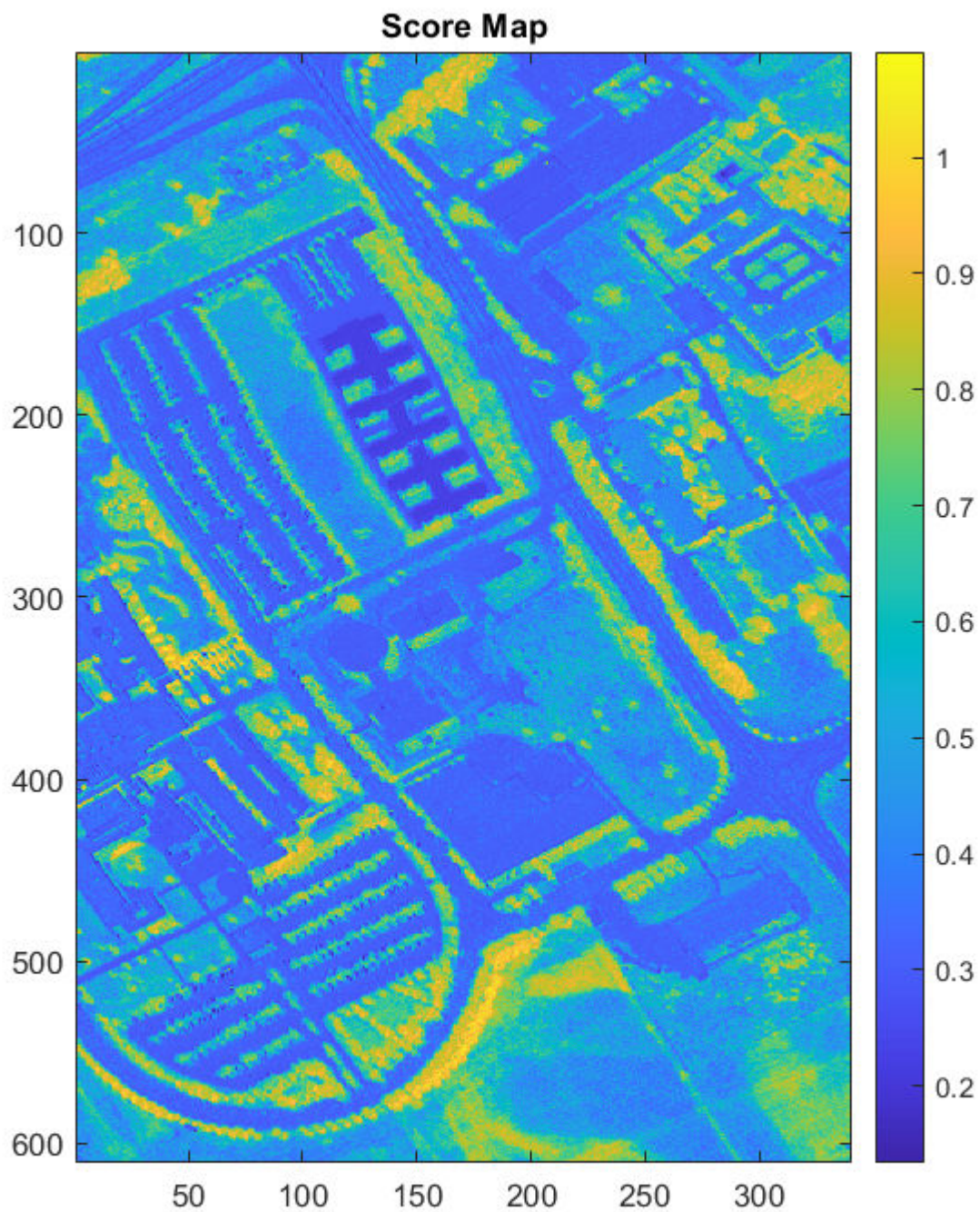
Classify and Detect Target

Typical values for the SAM score lies in the range [0, 3.142] and the measurement unit is radians. Lower value of SAM score represents better matching between the pixel spectra and the reference

spectra. Use thresholding method to spatially localize the target region in the input data. To determine the threshold, inspect the histogram of the score map. The minimum SAM score value with prominent number of occurrences can be used to select the threshold for detecting the target region.

```
figure
imhist(scoreMap);
title('Histogram Plot of Score Map');
xlabel('Score Map Values')
ylabel('Number of occurrences');
```
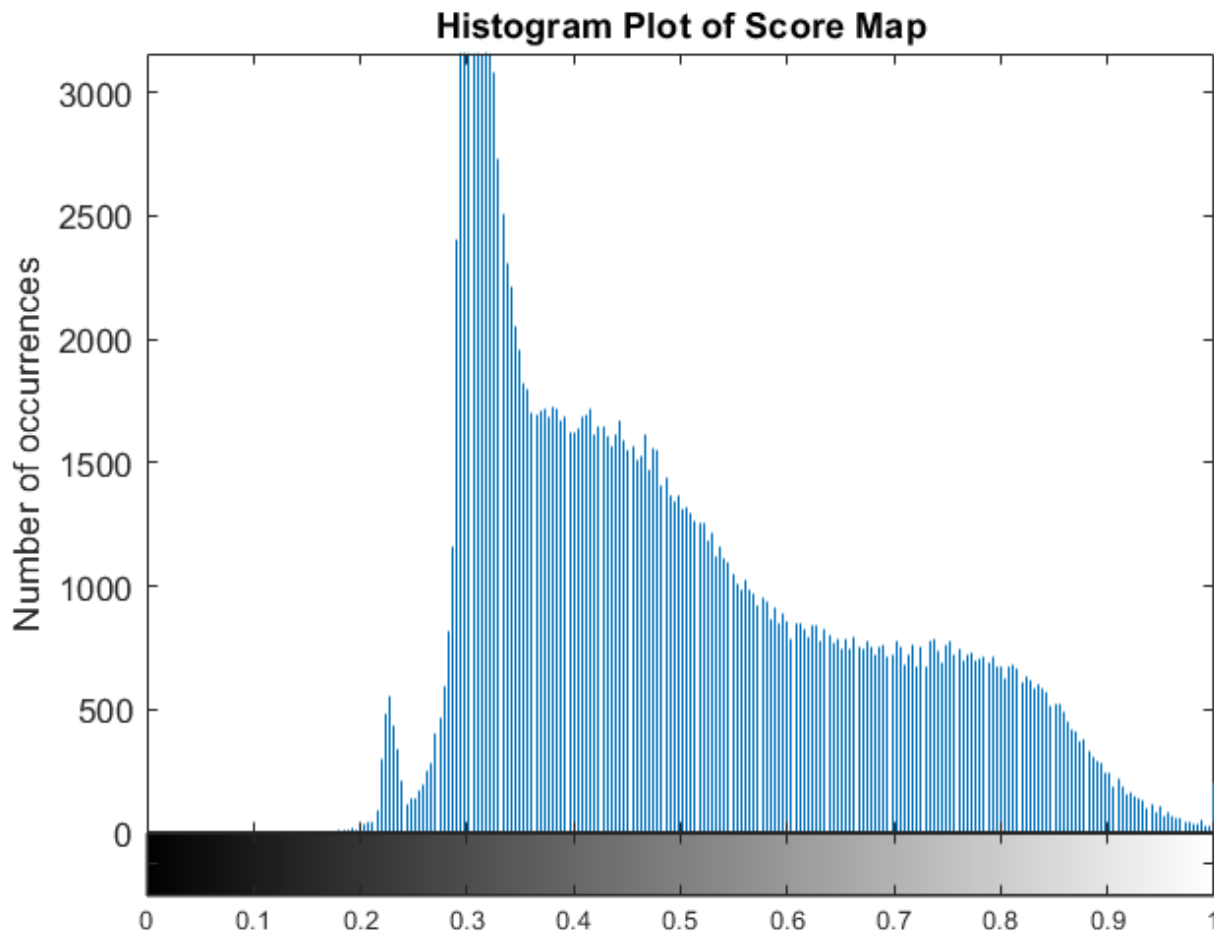

From the histogram plot, you can infer the minimum score value with prominant number of occurrence as approximately 0.22. Accordingly, you can set a value around the local maxima as the threshold. For this example, you can select the threshold for detecting the target as 0.25. The pixel values that are less than the maximum threshold are classified as the target region.

```
maxthreshold = 0.25;
```

Perform thresholding to detect the target region with maximum spectral similarity. Overlay the thresholded image on the RGB image of the hyperspectral data.

```
thresholdedImg = scoreMap <= maxthreshold;
overlaidImg = imoverlay(rgbImg,thresholdedImg,'green');
```

Display the results.

```
fig = figure('Position',[0 0 900 500]);
axes1 = axes('Parent',fig,'Position',[0.04 0.11 0.4 0.82]);
imagesc(thresholdedImg,'Parent',axes1);
colormap([0 0 0;1 1 1]);
title('Detected Target Region')
axis off
axes2 = axes('Parent',fig,'Position',[0.47 0.11 0.4 0.82]);
imagesc(overlaidImg,'Parent',axes2)
axis off
title('Overlaid Detection Results')
```


Validate Detection Results

You can validate the obtained target detection results by using the ground truth data taken from Pavia University dataset.

Load .mat file containing the ground truth data . To validate the result quantitatively, compute the mean squared error between the ground truth and the output. The error value is less if the obtained results are close to the ground truth.

```
load('paviauRoofingGT.mat');
err = immse(im2double(paviauRoofingGT), im2double(thresholdedImg));
fprintf('\n The mean squared error is %0.4f\n', err)
```

```
 The mean squared error is 0.0040
```

```
fig = figure('Position',[0 0 900 500]);
axes1 = axes('Parent',fig,'Position',[0.04 0.11 0.4 0.82]);
imagesc(thresholdedImg,'Parent',axes1);
colormap([0 0 0;1 1 1]);
title('Result Obtained')
axis off
axes2 = axes('Parent',fig,'Position',[0.47 0.11 0.4 0.82]);
imagesc(paviauRoofingGT,'Parent',axes2)
colormap([0 0 0;1 1 1]);
axis off
title('Ground Truth')
```
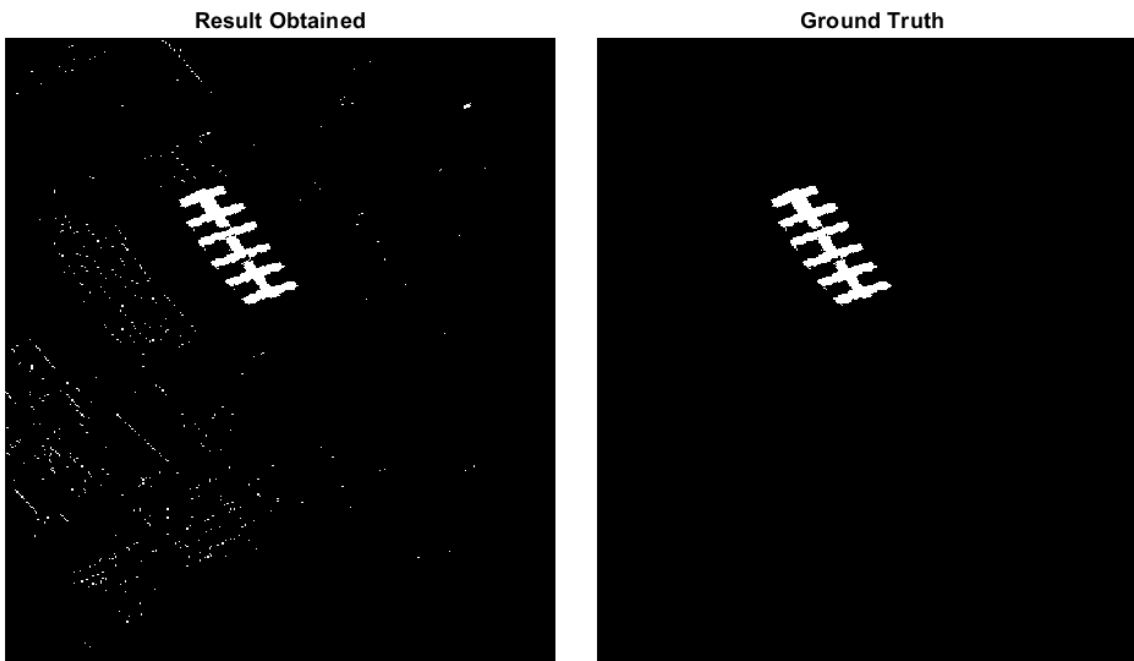

References

[1] Kruse, F.A., A.B. Lefkoff, J.W. Boardman, K.B. Heidebrecht, A.T. Shapiro, P.J. Barloon, and A.F.H. Goetz. "The Spectral Image Processing System (SIPS)—Interactive Visualization and Analysis of Imaging Spectrometer Data." *Remote Sensing of Environment* 44, no. 2–3 (May 1993): 145–63. https://doi.org/10.1016/0034-4257(93)90013-N.

[2] Chein-I Chang. "An Information-Theoretic Approach to Spectral Variability, Similarity, and Discrimination for Hyperspectral Image Analysis." *IEEE Transactions on Information Theory* 46, no. 5 (August 2000): 1927–32. https://doi.org/10.1109/18.857802.

Code Generation for Image Processing Toolbox Functions

Code Generation for Image Processing

Certain Image Processing Toolbox functions have been enabled to generate C code (requires 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 at the end of the function signature. This directive instructs the MATLAB code analyzer to diagnose issues that would prohibit successful code generation.

- Open the **MATLAB Coder** app, create a project, and add your file to the project. In the app, 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 a complete list of Image Processing Toolbox functions that support code generation, see Functions Supporting Code Generation. For an example of using code generation, see "Generate Code for Object Detection" on page 20-8.

Code Generation Using a Shared Library

Image Processing Toolbox functions can generate either standalone C code or code that depends on a precompiled, platform-specific shared library.

- 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 uses 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 diagram illustrates the difference between generating C code and generating code that uses a shared library.
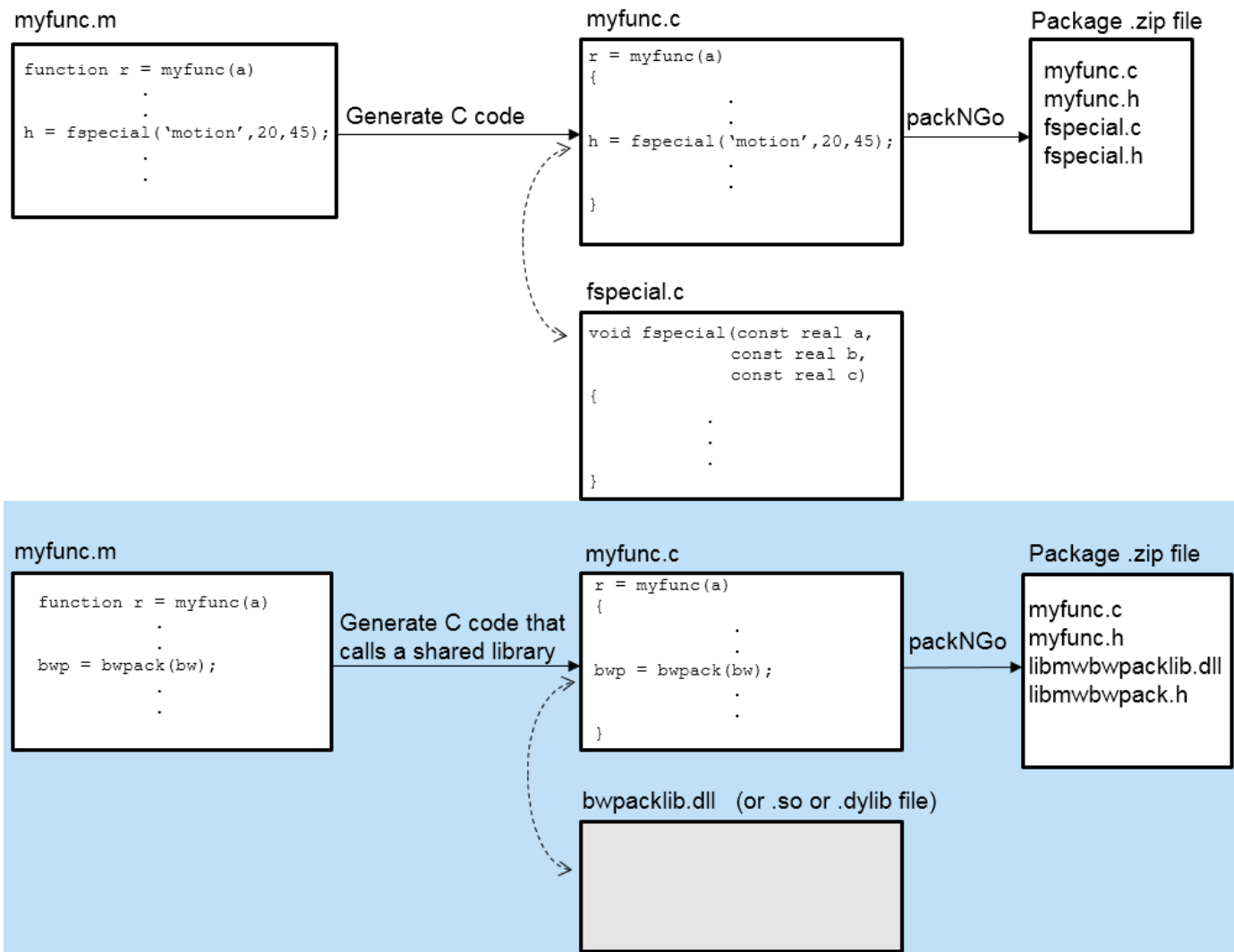
See Also

MATLAB Coder | codegen

Related Examples

- "Generate Code for Object Detection" on page 20-8

More About

- "Code Generation Workflow" (MATLAB Coder)
- "Generate C Code by Using the MATLAB Coder App" (MATLAB Coder)
- Functions Supporting Code Generation

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 20-2.
- 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*	Adaptive image threshold using local first-order statistics
affine2d*	2-D affine geometric transformation
affine3d*	3-D affine geometric transformation
boundarymask*	Find region boundaries of segmentation
bwareaopen*	Remove small objects from binary image
bwboundaries*	Trace region boundaries in binary image
bwconncomp*	Find connected components in binary image
bwdist*	Distance transform of binary image
bweuler*	Euler number of binary image
bwlabel*	Label connected components in 2-D binary image
bwlookup*	Nonlinear filtering using lookup tables
bwmorph*	Morphological operations on binary images
bwpack*	Pack binary image
bwperim*	Find perimeter of objects in binary image
bwselect*	Select objects in binary image
bwtraceboundary*	Trace object in binary image
bwunpack*	Unpack binary image

conndef*	Create connectivity array
demosaic*	Convert Bayer pattern encoded image to truecolor image
edge*	Find edges in intensity image
fft2*	2-D fast Fourier transform
fftshift	Shift zero-frequency component to center of spectrum
fitgeotrans*	Fit geometric transformation to control point pairs
freqspace	Frequency spacing for frequency response
fspecial*	Create predefined 2-D filter
fspecial3*	Create predefined 3-D filter
getrangefromclass*	Default display range of image based on its class
grayconnected*	Select contiguous image region with similar gray values
histeq*	Enhance contrast using histogram equalization
hough*	Hough transform
houghlines*	Extract line segments based on Hough transform
houghpeaks*	Identify peaks in Hough transform
hsv2rgb	Convert HSV colors to RGB
ifft2*	2-D inverse fast Fourier transform
ifftshift	Inverse zero-frequency shift
im2double	Convert image to double precision
im2int16*	Convert image to 16-bit signed integers
im2single*	Convert image to single precision
im2uint16*	Convert image to 16-bit unsigned integers
im2uint8*	Convert image to 8-bit unsigned integers
imabsdiff*	Absolute difference of two images
imadjust*	Adjust image intensity values or colormap
imbilatfilt*	Bilateral filtering of images with Gaussian kernels
imbinarize*	Binarize 2-D grayscale image or 3-D volume by thresholding
imbothat*	Bottom-hat filtering
imboxfilt*	2-D box filtering of images
imclearborder*	Suppress light structures connected to image border
imclose*	Morphologically close image
imcomplement*	Complement image
imcrop*	Crop image
imdilate*	Dilate image
imerode*	Erode image
imextendedmax*	Extended-maxima transform
imextendedmin*	Extended-minima transform

imfill*	Fill image regions and holes
imfilter*	N-D filtering of multidimensional images
imfindcircles*	Find circles using circular Hough transform
imgaborfilt*	Apply Gabor filter or set of filters to 2-D image
imgaussfilt*	2-D Gaussian filtering of images
imgradient3*	Find gradient magnitude and direction of 3-D image
imgradientxyz*	Find directional gradients of 3-D image
imhist*	Histogram of image data
imhmax*	H-maxima transform
imhmin*	H-minima transform
imlincomb*	Linear combination of images
immse*	Mean-squared error
imopen*	Morphologically open image
imoverlay*	Burn binary mask into 2-D image
impyramid*	Image pyramid reduction and expansion
imquantize*	Quantize image using specified quantization levels and output values
imread*	Read image from graphics file
imreconstruct*	Morphological reconstruction
imref2d*	Reference 2-D image to world coordinates
imref3d*	Reference 3-D image to world coordinates
imregcorr	Estimate geometric transformation that aligns two 2-D images using phase correlation
imregionalmax*	Regional maxima
imregionalmin*	Regional minima
imresize*	Resize image
imrotate*	Rotate image
imsplit	Split multichannel image into its individual channels
imtophat*	Top-hat filtering
imtranslate*	Translate image
imwarp*	Apply geometric transformation to image
ind2rgb	Convert indexed image to RGB image
inpaintCoherent*	Restore specific image regions using coherence transport based image inpainting
integralBoxFilter*	2-D box filtering of integral images
integralImage	Calculate 2-D integral image
intlut*	Convert integer values using lookup table
iptcheckconn*	Check validity of connectivity argument
iptcheckmap*	Check validity of colormap
lab2rgb*	Convert CIE 1976 L*a*b* to RGB

label2idx*	Convert label matrix to cell array of linear indices
label2rgb*	Convert label matrix into RGB image
mean2*	Average or mean of matrix elements
medfilt2*	2-D median filtering
multithresh*	Multilevel image thresholds using Otsu's method
offsetstrel*	Morphological offset structuring element
ordfilt2*	2-D order-statistic filtering
otsuthresh*	Global histogram threshold using Otsu's method
padarray*	Pad array
projective2d*	2-D projective geometric transformation
psnr*	Peak Signal-to-Noise Ratio (PSNR)
regionprops*	Measure properties of image regions
rgb2gray*	Convert RGB image or colormap to grayscale
rgb2hsv	Convert RGB colors to HSV
rgb2lab*	Convert RGB to CIE 1976 L*a*b*
rgb2lightness*	Convert RGB color values to lightness values
rgb2ycbcr*	Convert RGB color values to YCbCr color space
strel*	Morphological structuring element
stretchlim*	Find limits to contrast stretch image
superpixels*	2-D superpixel oversegmentation of images
watershed*	Watershed transform
ycbcr2rgb*	Convert YCbCr color values to RGB color space

See Also
MATLAB Coder | codegen

More About
- "Code Generation for Image Processing" on page 20-2
- "MATLAB Coder"

Generate Code for Object Detection

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 and prepare your MATLAB code for code generation.

This example also demonstrates how to solve issues that you might encounter in your MATLAB code that prevent code generation. 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.

For more information about generating code, see the MATLAB Coder documentation.

Set Up Compiler

Specify which C/C++ compiler you want to use with MATLAB Coder to generate code by using the `mex` function with the `-setup` option.

```
mex -setup

MEX configured to use 'MinGW64 Compiler (C)' for C language compilation.

To choose a different language, select one from the following:
 mex -setup C++
 mex -setup FORTRAN
```

Define Entry-Point Function

The entry-point function is a MATLAB function used as the source code for code generation. First, prototype the image processing workflow without support for code generation. This example defines a function called `detectCells.m` that performs cell detection using segmentation and morphological techniques. This function is attached to the example as a supporting file.

Test the example code with a sample image, `cell.tif`.

```
I = imread('cell.tif');
Iseg = detectCells(I);
```


Binary Gradient Mask

Dilated Gradient Mask
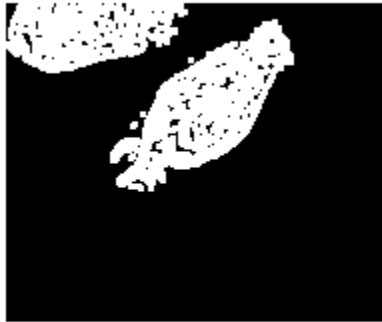
Binary Image with Filled Holes

**Cleared Border Image**
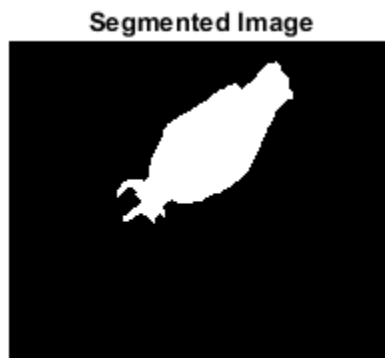
Segmented Image

Confirm the accuracy of the segmentation by overlaying the segmented image on the original image.

```
imshow(labeloverlay(I,Iseg))
```
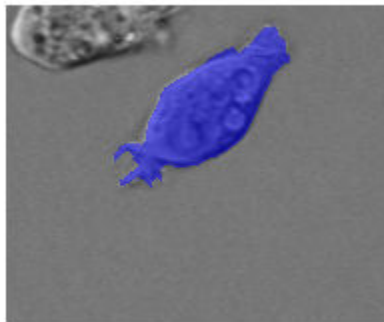

Because you modify this code for code generation, it is good to work with a copy of the code. This example includes a copy of the helper function `detectCells.m` named `detectCellsCodeGen.m`. The version of the function used for code generation includes the MATLAB Coder compilation directive `%#codegen` at the end of the function signature. This directive instructs the MATLAB code analyzer to diagnose issues that would prohibit successful code generation.

Open the MATLAB Coder app by using the `coder` function. (Alternatively, in MATLAB, select the Apps tab, navigate to Code Generation and click the MATLAB Coder app.)
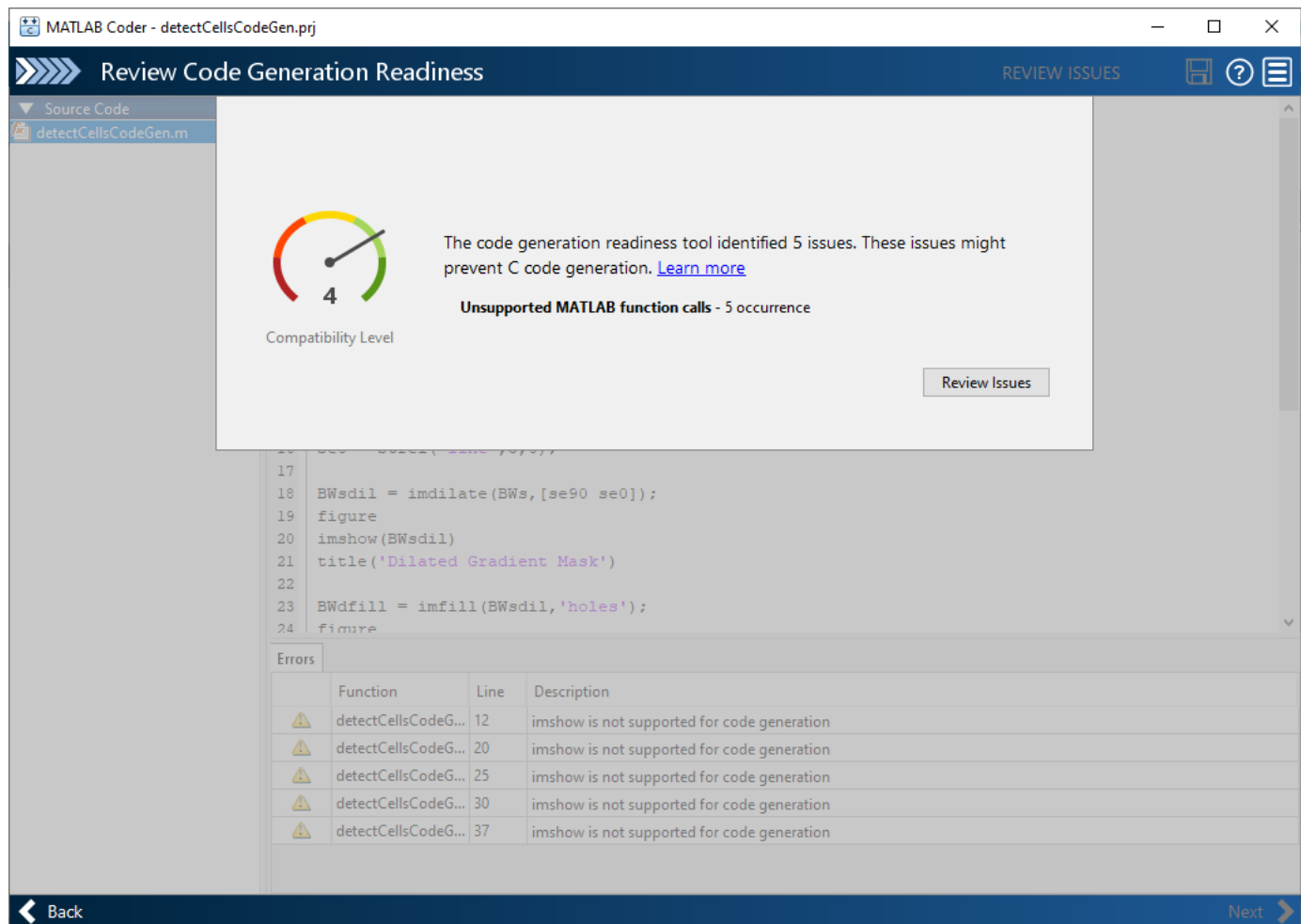
```
coder
```

Specify the name of your entry-point function, `detectCellsCodeGen`, and press **Enter**.
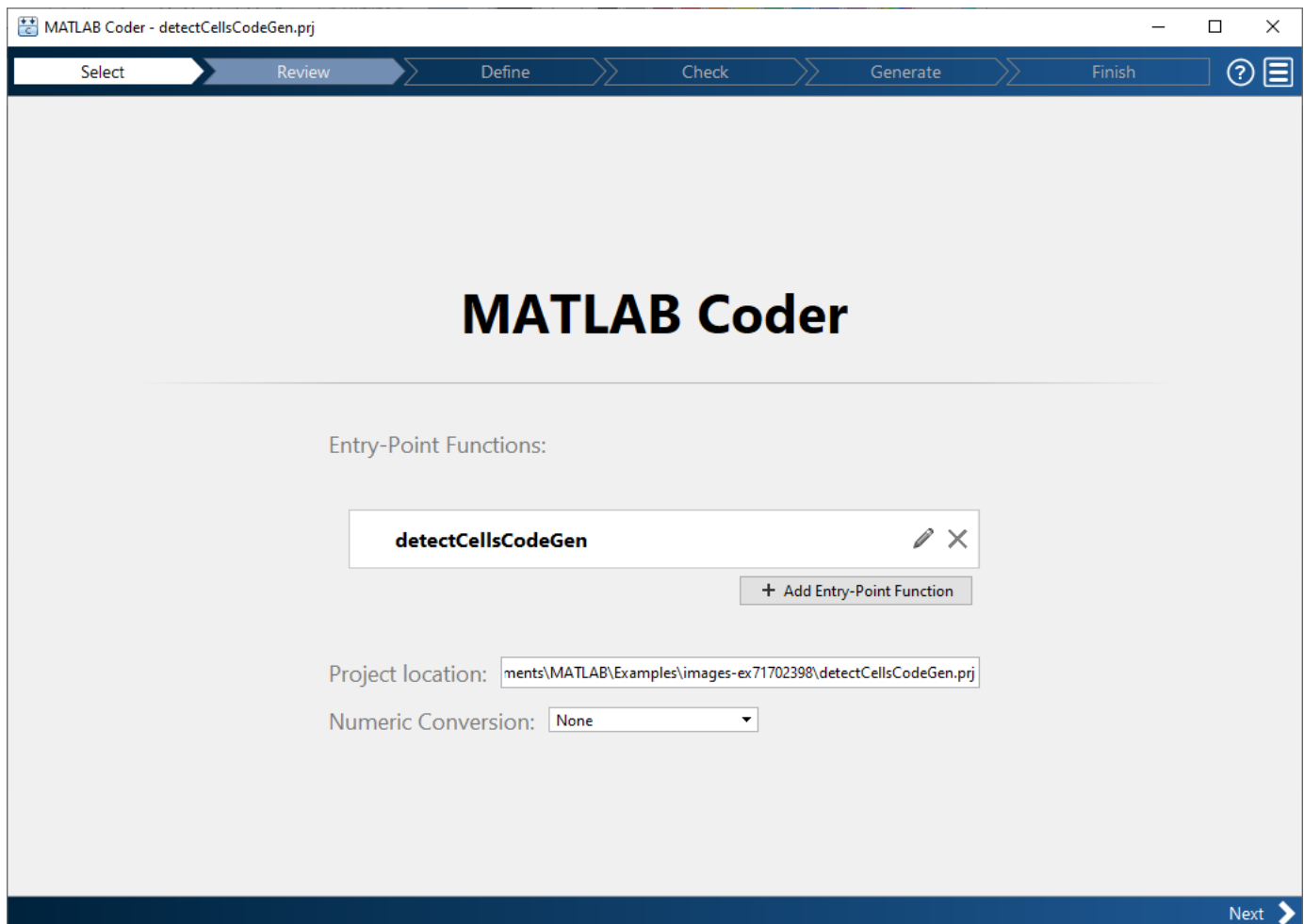
Determine Code Readiness for Code Generation

Click **Next**. 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.

Correct Readiness Issues

Address the readiness issues. Remove the calls to `imshow` and related display code from your example. The display statements are not necessary for the segmentation operation. You can edit the example code directly in 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 Size and Data Type of Function Inputs

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 in the MATLAB prompt and press **Autodefine Input Types**.

```
I = imread('cell.tif');
Iseg = detectCellsCodeGen(I);
```

Binary Gradient Mask
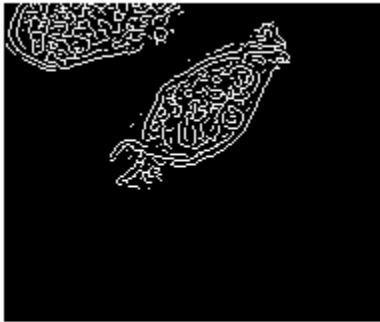
Dilated Gradient Mask

Binary Image with Filled Holes

Cleared Border Image

Segmented Image

For more information about defining inputs, see the MATLAB Coder documentation. After MATLAB Coder returns with the input type definition, click **Next**.
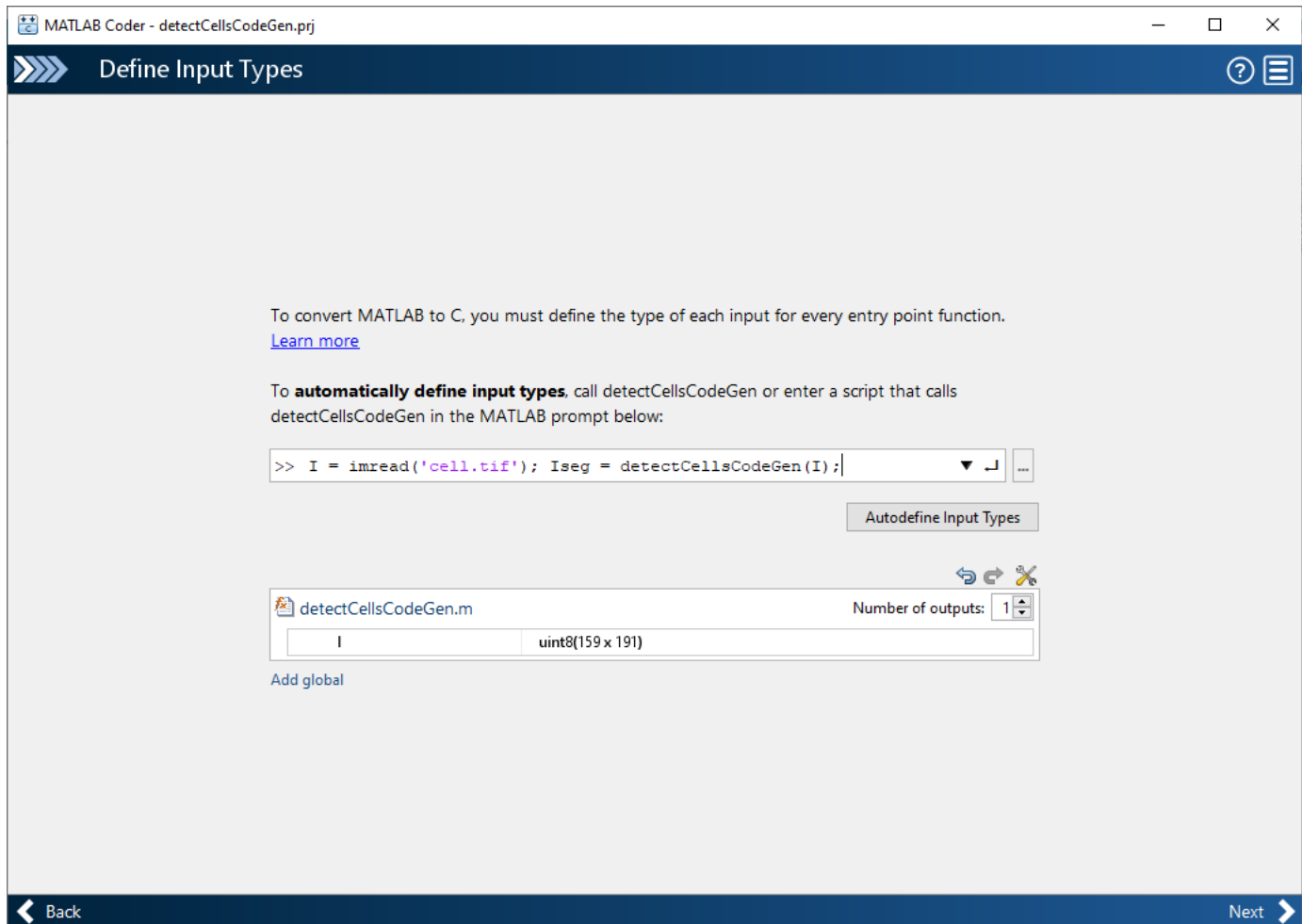
Check for and Resolve Run-Time Issues

Even though you performed 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**.
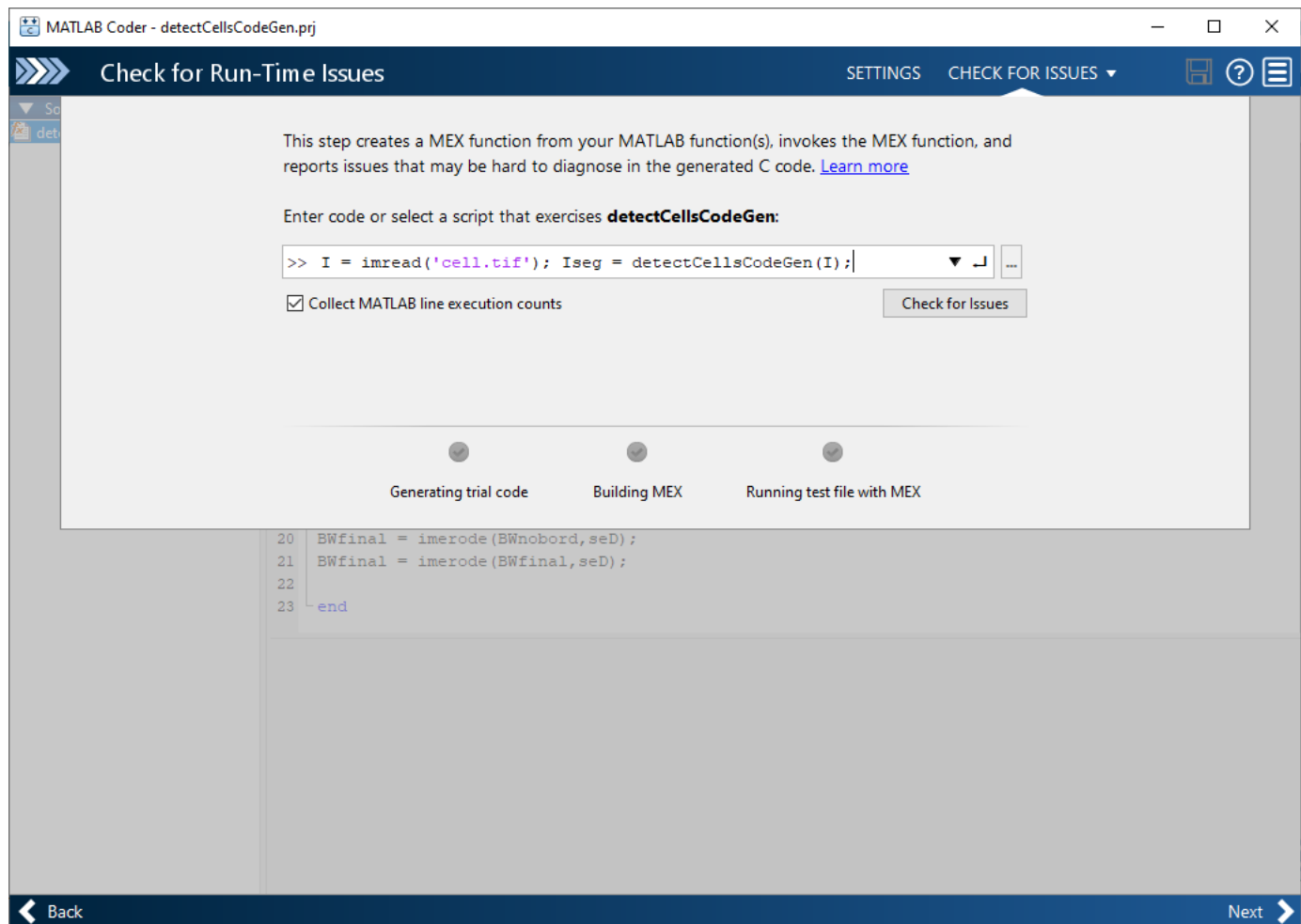
This example contains a build issue: it passes an array of `strel` objects to `imdilate` and arrays of objects are not supported for code generation.

Address the build issues identified. For this example, modify the call to `imdilate` to avoid passing an array of strel objects. Replace the single call to `imdilate` with two separate calls to `imdilate` where you pass one strel object with each call.

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.
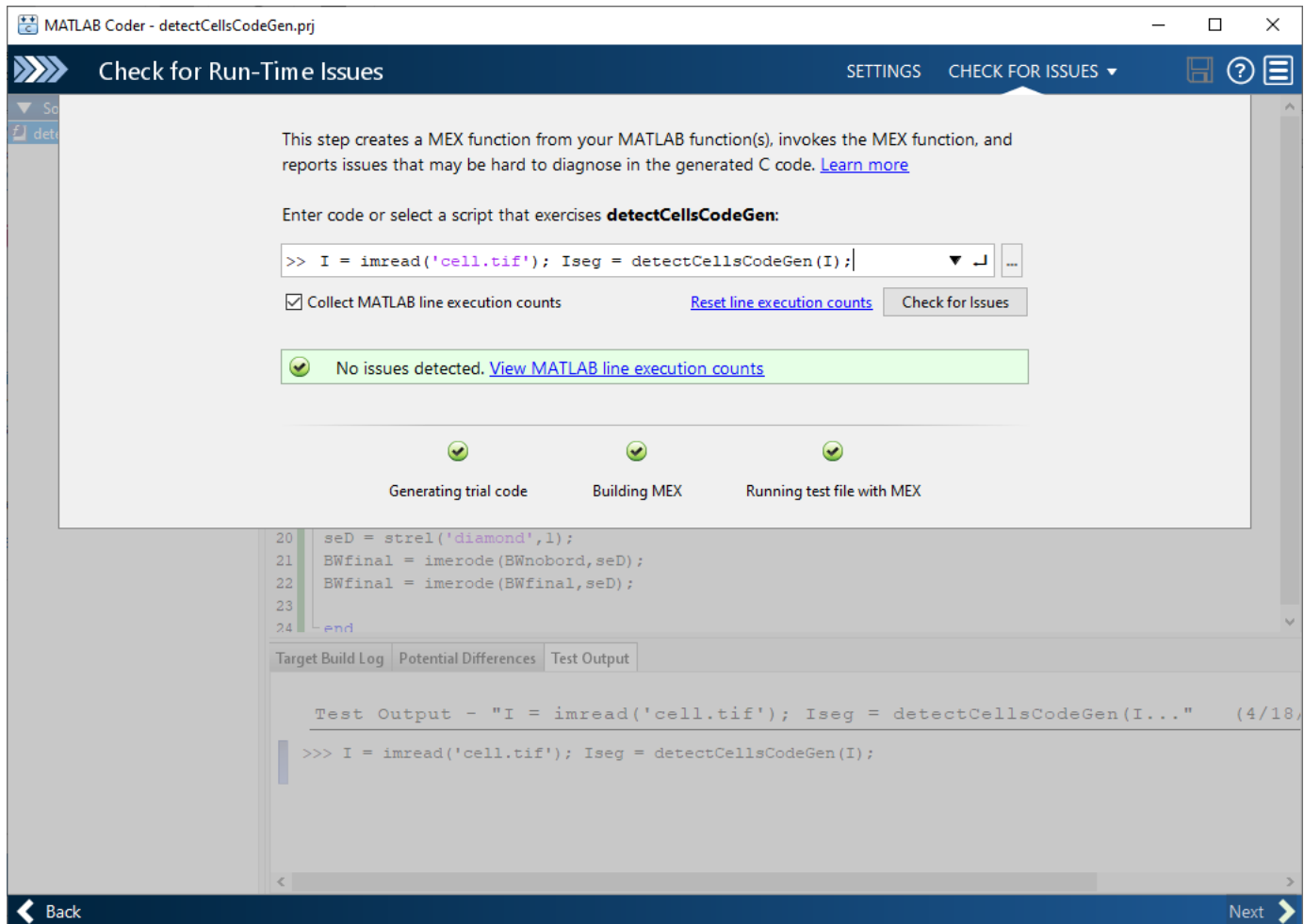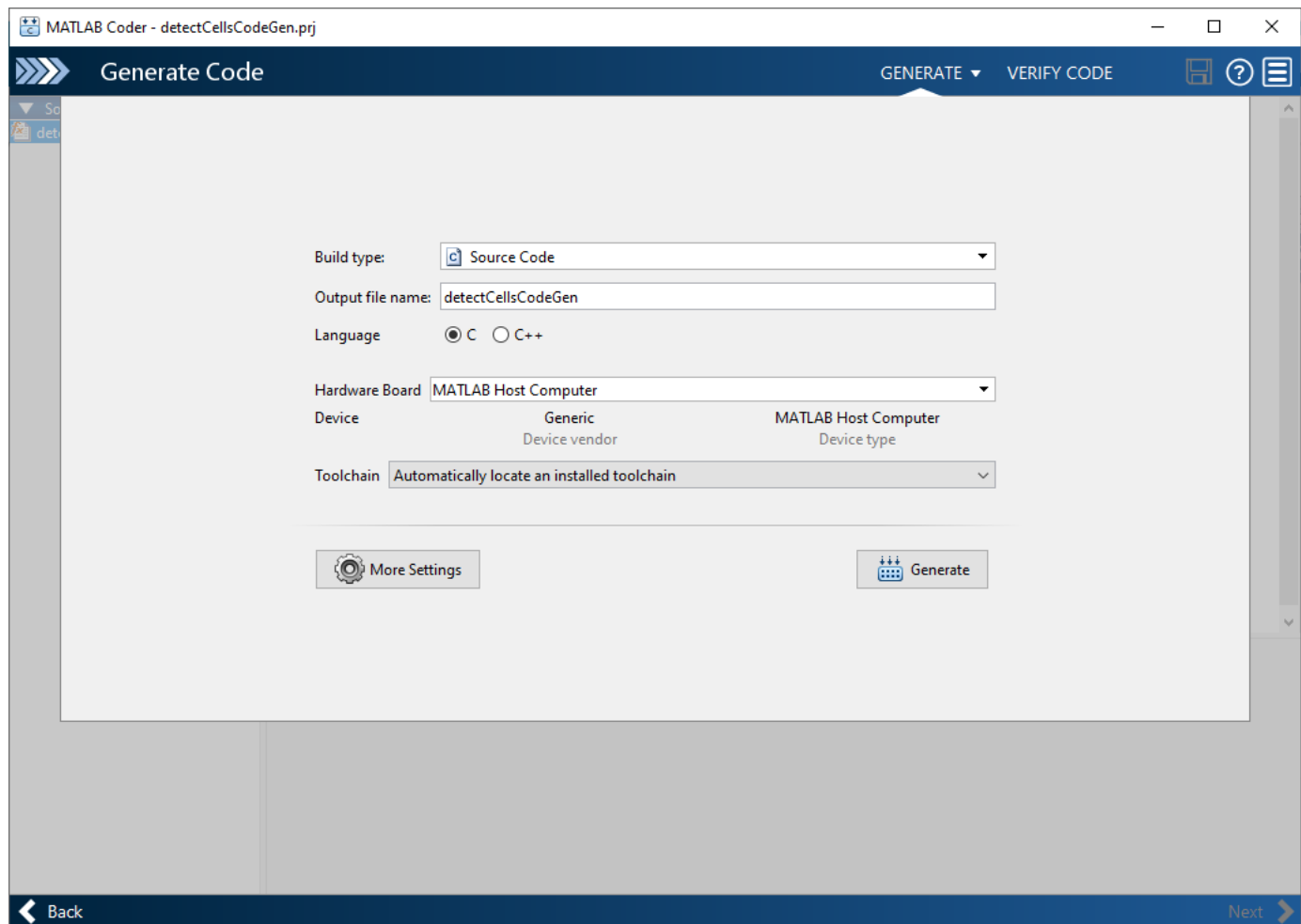
Generate Code

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.

This example uses the default options. The build type is Source Code and the language is **C**. For device options, specify a generic device from a device vendor and a MATLAB Host Computer for the 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.

Click **Generate**.

MATLAB Coder displays the generated code.

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.

See Also
MATLAB Coder | codegen

More About
- "Code Generation for Image Processing" on page 20-2
- "Code Generation Workflow" (MATLAB Coder)
- "Generate C Code by Using the MATLAB Coder App" (MATLAB Coder)
- "Input Type Specification for Code Generation" (MATLAB Coder)
- Functions Supporting Code Generation

GPU Computing with Image Processing Toolbox Functions

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 a supported 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. For a list of all the toolbox functions that have been GPU-enabled, see Functions Supporting GPU Computing.
- 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.

If you call a function with GPU support using at least one `gpuArray` input argument, then the function runs automatically on a GPU and generates a `gpuArray` as the result. You can mix inputs using both `gpuArray` and MATLAB arrays in the same function call. In this case, the function automatically transfers the MATLAB arrays to the GPU for execution.

When working with a GPU, note the following:

- Performance improvements can depend on the GPU device.
- There can 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 GPU" on page 21-3
- "Perform Pixel-Based Operations on GPU" on page 21-8

More About
- "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox)
- Functions Supporting GPU Computing

Perform Thresholding and Morphological Operations on GPU

This example shows how to perform image processing operations on a GPU. The example uses filtering to highlight the watery areas in an aerial photograph.

Read an image into the workspace.

```
imCPU = imread('concordaerial.png');
```

Move the image to the GPU by creating a `gpuArray` (Parallel Computing Toolbox) object.

```
imGPUoriginal = gpuArray(imCPU);
```

As a preprocessing step, change the RGB image to a grayscale image. `rgb2gray` performs the conversion operation on a GPU because the input argument is a `gpuArray`.

```
imGPUgray = rgb2gray(imGPUoriginal);
```

View the image in the Image Viewer app and inspect the pixel values to find the value of watery areas. To use Image Viewer, you must bring the image data back onto the CPU by using the `gather` (Parallel Computing Toolbox) function. 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 are dark and have pixel values less than 70.
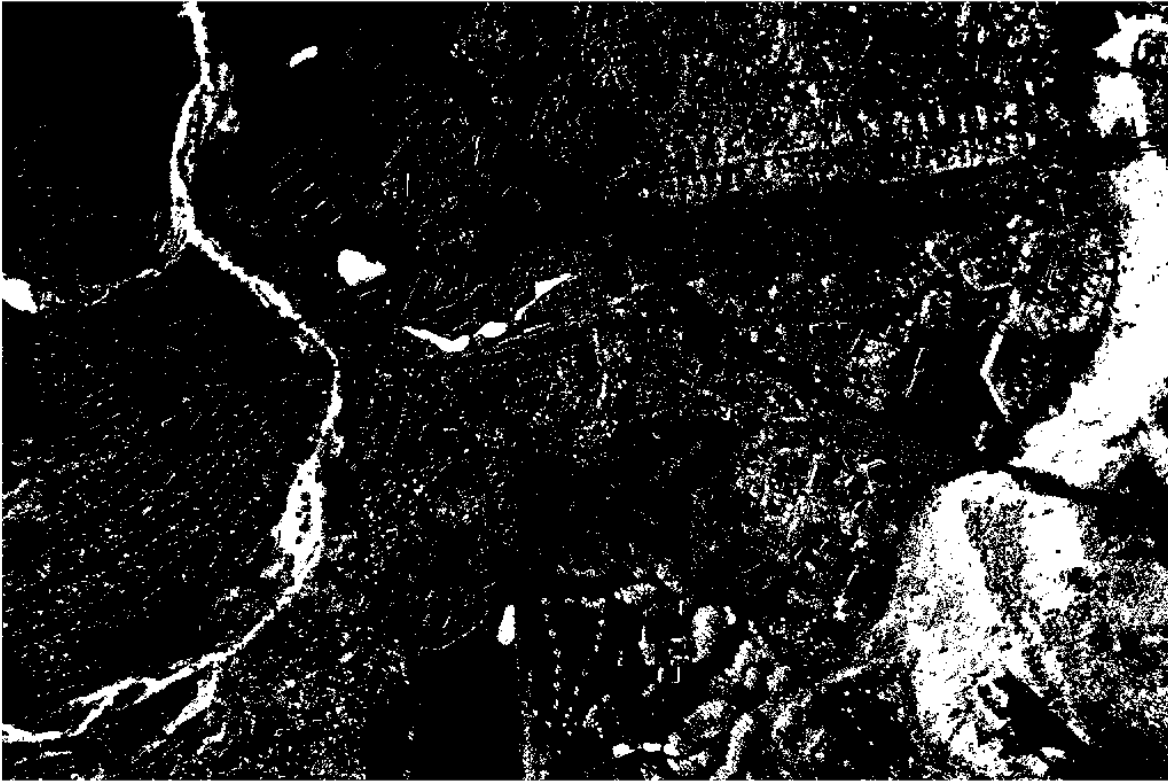
```
imtool(gather(imGPUgray));
```
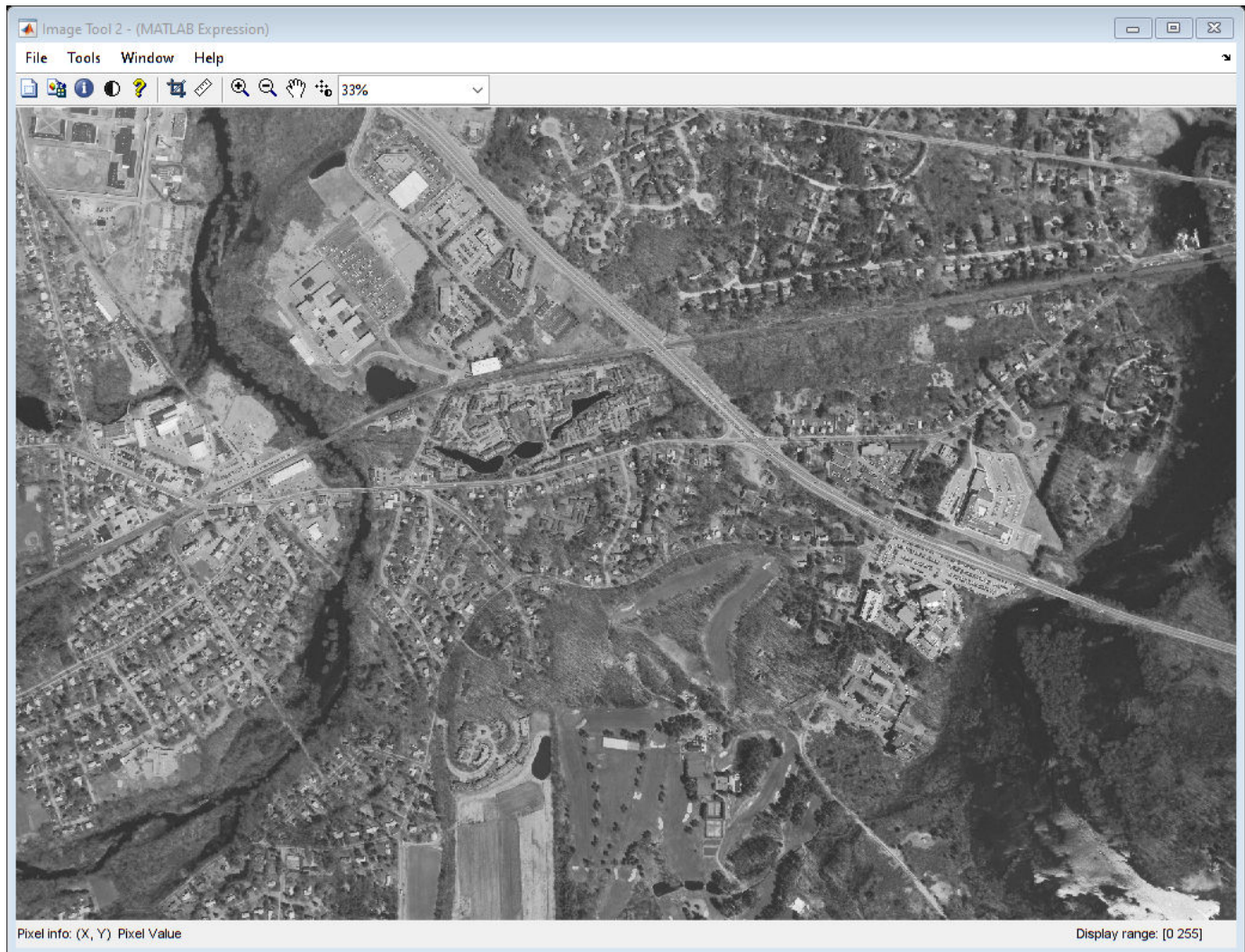
To get a new image that contains only the pixels with values of 70 or less, threshold the image on the GPU.

```
imWaterGPU = imGPUgray<70;
```

Display the thresholded image. Unlike Image Viewer, the `imshow` function supports `gpuArray` input.

```
figure
imshow(imWaterGPU)
```

Remove small objects from the image while preserving the shape and size of larger objects by using morphological opening. The `imopen` function performs morphological opening and supports `gpuArray` input.

```
imWaterMask = imopen(imWaterGPU,strel('disk',5));
imshow(imWaterMask)
```
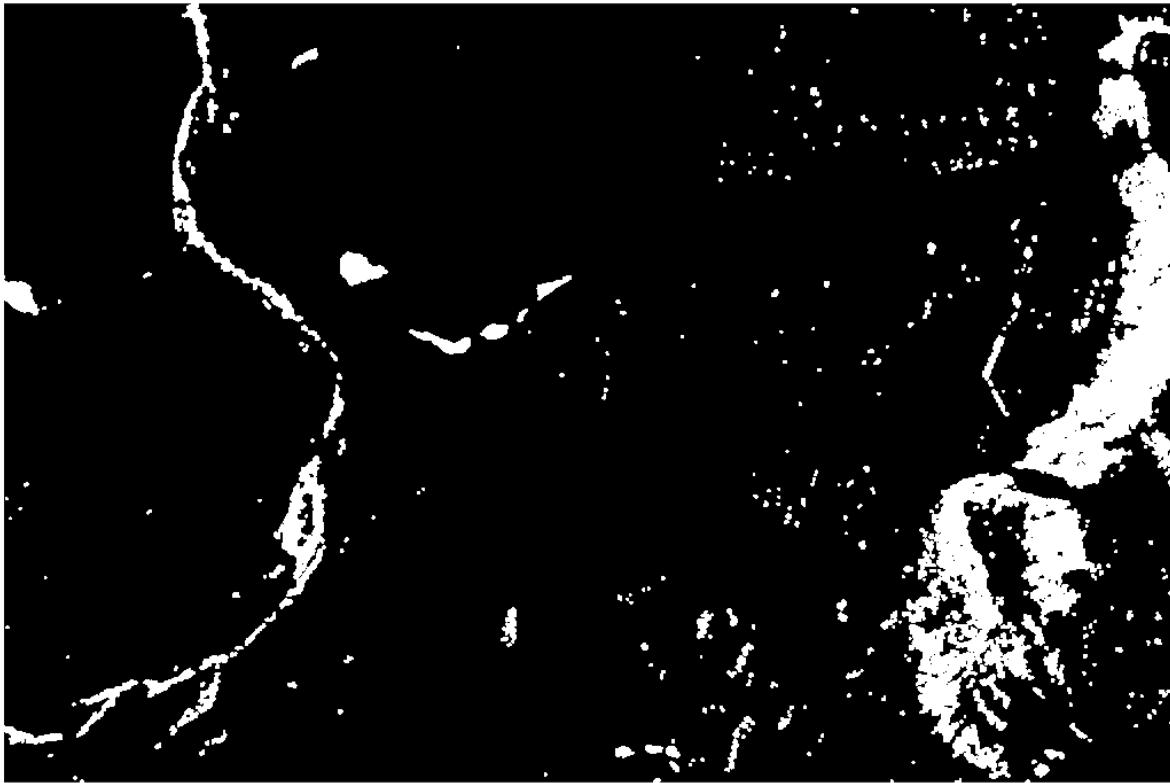
Create a copy of the original image that will contain the enhanced data. Convert the data type to `single`.

```
imGPUenhanced = im2single(imGPUoriginal);
```

Get the blue channel from the original image.

```
blueChannelOriginal = imGPUenhanced(:,:,3);
```

Enhance the saturation of the blue channel by increasing the strength of the blue channel for pixels where the mask is 1 (`true`).

```
blueChannelEnhanced = blueChannelOriginal + 0.2*single(imWaterMask);
```

The maximum value of the enhanced blue channel exceeds the maximum value expected of images of data type `single`. Rescale the data to the expected range [0, 1] by using the `rescale` function.

```
blueChannelEnhanced = rescale(blueChannelEnhanced);
```

Replace the blue channel with the enhanced blue channel.

```
imGPUenhanced(:,:,3) = blueChannelEnhanced;
```

Compare the original image with the enhanced image by displaying them in a montage. Pixels corresponding to water have a more saturated blue color in the enhanced image than in the original image.

```
montage({imCPU,imGPUenhanced})
title('Original Versus Enhanced Image')
```


Original Versus Enhanced Image
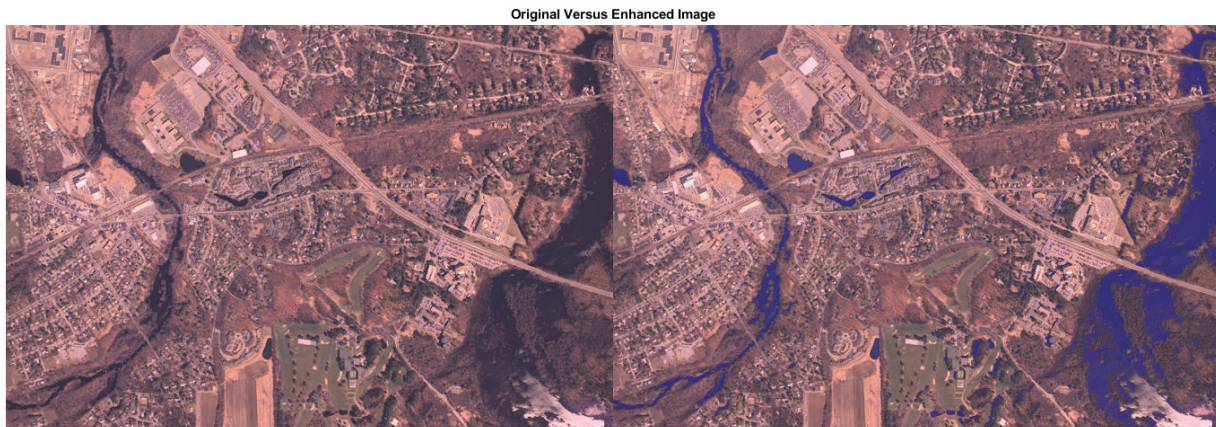
After filtering the image on the GPU, move the data back to the CPU by using the `gather` function. Write the modified image to a file.

```
outCPU = gather(imGPUenhanced);
imwrite(outCPU,'concordwater.png')
```

See Also
gather | gpuArray

Related Examples
- "Perform Pixel-Based Operations on GPU" on page 21-8

More About
- "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox)
- Functions Supporting GPU Computing

Perform Pixel-Based Operations on GPU

This example shows how to perform pixel-based operations on a GPU by using functions that send both the data and the operations to the GPU for processing. This method is most effective for element-wise operations that require two or more data sets.

Read and display an image.

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


Move the data from the CPU to the GPU by creating a `gpuArray` (Parallel Computing Toolbox) object.

```
Igpu = gpuArray(I);
```

Perform an operation on the GPU. This example defines a custom function called `rgb2gray_custom` that converts an RGB image to grayscale by using a custom weighting of the red, green, and blue color channels. This function is defined at the end of the example. Pass the handle to the custom function and data to the GPU for evaluation by the `arrayfun` (Parallel Computing Toolbox) function.

```
Igray_gpu = arrayfun(@rgb2gray_custom, ...
    Igpu(:,:,1),Igpu(:,:,2),Igpu(:,:,3));
```

Caught front-end user exception: MATLAB:m_illegal_character

Move the data back to the CPU from the GPU by using the `gather` (Parallel Computing Toolbox) function.

```
I_gpuresult = gather(Igray_gpu);
```

Display the result.

```
imshow(I_gpuresult)
```


Supporting Function

The `rgb2gray_custom` helper functions takes a linear combination of three channels and returns a single channel output image.

```
function gray = rgb2gray_custom(r,g,b)
    gray = 0.5*r + 0.25*g + 0.25*b;
end
```

See Also
`arrayfun` | `gather` | `gpuArray`

Related Examples
- "Perform Thresholding and Morphological Operations on GPU" on page 21-3

More About

- "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox)
- Functions Supporting GPU Computing