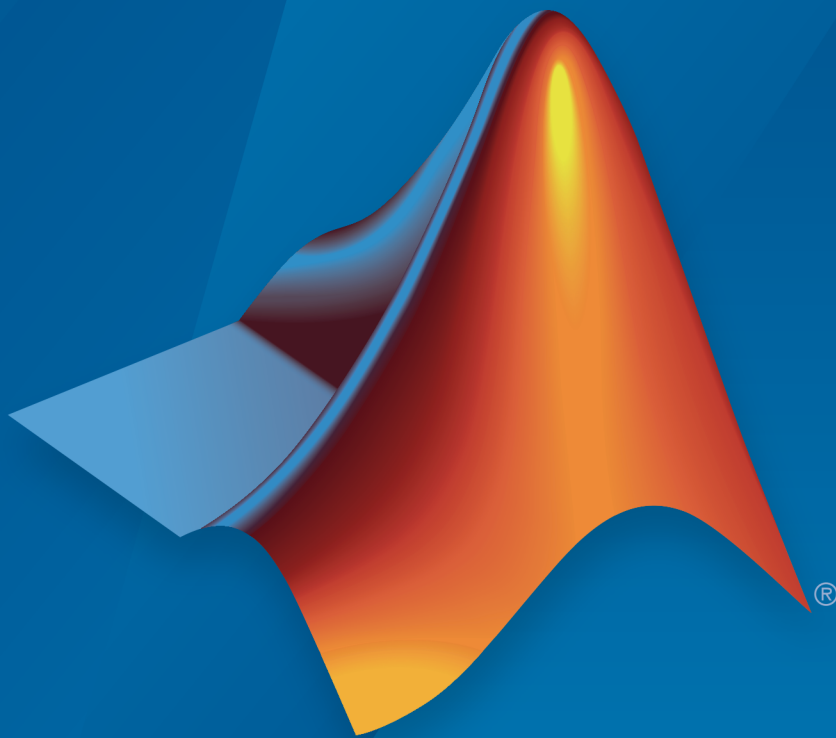


Image Processing Toolbox™

Reference



MATLAB®

R2017a

 MathWorks®

How to Contact MathWorks



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



Phone: 508-647-7000



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

Image Processing Toolbox™ Reference

© COPYRIGHT 1993–2017 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

August 1993	First printing	Version 1
May 1997	Second printing	Version 2
April 2001	Third printing	Revised for Version 3.0
June 2001	Online only	Revised for Version 3.1 (Release 12.1)
July 2002	Online only	Revised for Version 3.2 (Release 13)
May 2003	Fourth printing	Revised for Version 4.0 (Release 13.0.1)
September 2003	Online only	Revised for Version 4.1 (Release 13.SP1)
June 2004	Online only	Revised for Version 4.2 (Release 14)
August 2004	Online only	Revised for Version 5.0 (Release 14+)
October 2004	Fifth printing	Revised for Version 5.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 5.0.2 (Release 14SP2)
September 2005	Online only	Revised for Version 5.1 (Release 14SP3)
March 2006	Online only	Revised for Version 5.2 (Release 2006a)
September 2006	Online only	Revised for Version 5.3 (Release 2006b)
March 2007	Online only	Revised for Version 5.4 (Release 2007a)
September 2007	Online only	Revised for Version 6.0 (Release 2007b)
March 2008	Online only	Revised for Version 6.1 (Release 2008a)
October 2008	Online only	Revised for Version 6.2 (Release 2008b)
March 2009	Online only	Revised for Version 6.3 (Release 2009a)
September 2009	Online only	Revised for Version 6.4 (Release 2009b)
March 2010	Online only	Revised for Version 7.0 (Release 2010a)
September 2010	Online only	Revised for Version 7.1 (Release 2010b)
April 2011	Online only	Revised for Version 7.2 (Release 2011a)
September 2011	Online only	Revised for Version 7.3 (Release 2011b)
March 2012	Online only	Revised for Version 8.0 (Release 2012a)
September 2012	Online only	Revised for Version 8.1 (Release 2012b)
March 2013	Online only	Revised for Version 8.2 (Release 2013a)
September 2013	Online only	Revised for Version 8.3 (Release 2013b)
March 2014	Online only	Revised for Version 9.0 (Release 2014a)
October 2014	Online only	Revised for Version 9.1 (Release 2014b)
March 2015	Online only	Revised for Version 9.2 (Release 2015a)
September 2015	Online only	Revised for Version 9.3 (Release 2015b)
March 2016	Online only	Revised for Version 9.4 (Release 2016a)
September 2016	Online only	Revised for Version 9.5 (Release 2016b)
March 2017	Online only	Revised for Version 10.0 (Release 2017a)

1	Functions — Alphabetical List
----------	--------------------------------------

Functions — Alphabetical List

Color Thresholder

Threshold a color image

Description

The **Color Thresholder** app lets you threshold color images by manipulating the color components of these images, based on different color spaces. Using this app, you can create a segmentation mask for a color image.

Open the Color Thresholder App

- MATLAB® Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the Color Thresholder app icon.
- MATLAB command prompt: Enter `colorThresholder`.

Examples

- “Image Segmentation Using the Color Thresholder App”

Programmatic Use

`colorThresholder` opens the Color Thresholder app, which enables you to create a segmentation mask of a color image based on the exploration of different color spaces.

`colorThresholder(RGB)` opens the Color Thresholder app, loading the image `RGB` into the app.

`colorThresholder close` closes all open instances of the Color Thresholder app.

See Also

See Also

Apps

Image Segmenter

Functions

imcontrast

Topics

“Image Segmentation Using the Color Thresholder App”

Introduced in R2014a

Image Batch Processor

Apply a function to multiple images

Description

The **Image Batch Processor** app lets you process a folder of images using a function you specify. The function must have the following signature: `out = fcn(in)`. The app creates an output folder containing the processed images, using the same name and subfolder structure as the input folder.

Open the Image Batch Processor App

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the Image Batch Processor app icon.
- MATLAB command prompt: Enter `imageBatchProcessor`.

Examples

- “Batch Processing Using the Image Batch Processor App”

Programmatic Use

`imageBatchProcessor` opens the Image Batch Processor app, which enables you to process a folder of images.

`imageBatchProcessor close` closes all open instances of the Image Batch Processor app.

See Also

See Also

Functions

`imread` | `imwrite`

Topics

“Batch Processing Using the Image Batch Processor App”

Introduced in R2015a

Image Browser

Browse images using thumbnails

Description

The **Image Browser** app lets you view thumbnails of all the images in a particular folder or image datastore. Once displayed in the app, you can select an image and open it in one of several Image Processing Toolbox™ apps. You can save images displayed in the app to the MATLAB workspace as an `ImageDatastore` object. See `ImageDatastore` for more information.

Open the Image Browser App

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the Image Browser app icon.
- MATLAB command prompt: Enter `imageBrowser`.

Programmatic Use

`imageBrowser` opens the Image Browser app.

`imageBrowser(folder)` opens the Image Browser app with all images in the folder, `folder`, loaded.

`imageBrowser(imds)` opens the Image Browser app with all images in the image datastore, `imds`, loaded.

See Also

See Also

Apps

Image Batch Processor

Functions

`imageDatastore`

Topics

“View Thumbnails of Images in Folder or Datastore”

“Getting Started with Datastore” (MATLAB)

Introduced in R2016b

Image Segmenter

Segment an image by refining regions

Description

The **Image Segmenter** app lets you segment an image using the active contours algorithm. Using this app, you first create an initial segmentation that defines seed locations and then segment the image iteratively.

Open the Image Segmenter App

- MATLAB Toolstrip: Open the **Apps** tab, under **Image Processing and Computer Vision**, click the Image Segmenter app icon.
- MATLAB command prompt: Enter `imageSegmenter`.

Examples

- “Image Segmentation Using the Image Segmenter App”

Programmatic Use

`imageSegmenter` opens the Image Segmenter app, which enables you to create a segmentation mask of an image by using active contours.

`imageSegmenter(I)` opens the Image Segmenter app, loading the image `I` into the app.

`imageSegmenter close` closes all open instances of the Image Segmenter app.

See Also

See Also

Functions

`activecontour`

Topics

“Image Segmentation Using the Image Segmenter App”

Introduced in R2014b

Image Region Analyzer

Browse and filter connected components in an image

Description

The **Image Region Analyzer** app measures a set of properties for each connected component (also called an object or region) in a binary image and displays this information in a table. You can also use this app to create other binary images by filtering the image on region properties.

Open the Image Region Analyzer App

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the Image Region Analyzer app icon.
- MATLAB command prompt: Enter `imageRegionAnalyzer`.

Examples

- “Calculate Region Properties Using Image Region Analyzer”
- “Filter Images on Region Properties Using Image Region Analyzer App”

Programmatic Use

`imageRegionAnalyzer` opens the Image Region Analyzer app, which enables you to create other binary images and get information about the regions within binary images.

`imageRegionAnalyzer(I)` opens the Image Region Analyzer app, loading the image `I` into the app.

`imageRegionAnalyzer close` closes all open instances of the Image Region Analyzer app.

See Also

See Also

Functions

`bwareafilt` | `bwpropfilt` | `regionprops`

Topics

“Calculate Region Properties Using Image Region Analyzer”

“Filter Images on Region Properties Using Image Region Analyzer App”

Introduced in R2014b

Image Viewer

View and explore images

Description

The **Image Viewer** app provides image display capabilities as well as access to several tools for navigating and exploring images, and performing some common image processing tasks.

Open the Image Viewer App

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the Image Viewer app icon.
- MATLAB command prompt: Enter `imtool`.

Examples

- “Explore Images with the Image Viewer App”

Programmatic Use

```
imtool
```

See Also

See Also

Apps

Video Viewer

Functions

`imshow`

Topics

“Explore Images with the Image Viewer App”

Introduced in R2014b

Registration Estimator

Register 2-D grayscale images

Description

The **Registration Estimator** app aligns 2-D grayscale images using automatic image registration. Using this app, you can:

- Compare feature-based, intensity-based, and nonrigid registration techniques interactively
- Obtain the registered image and the geometric transformation

Feature-Based Techniques

Registration Estimator app offers six registration techniques that use feature detection and matching:

- FAST
- MinEigen
- Harris
- BRISK
- SURF
- MSER

Intensity-Based Techniques

Registration Estimator app offers three registration techniques that use intensity metric optimization:

- Monomodal intensity
- Multimodal intensity
- Phase correlation

For more details of the available techniques, see “Techniques Supported by Registration Estimator App”.

Open the Registration Estimator App

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the Registration Estimator app icon.
- MATLAB command prompt: Enter `registrationEstimator`.

Examples

- “Register Images Using the Registration Estimator App”

Programmatic Use

`registrationEstimator` opens the Registration Estimator app, which enables you to perform intensity-based, feature-based, and nonrigid image registration.

`registrationEstimator(MOVING, FIXED)` opens the Registration Estimator app, loading the grayscale images `MOVING` and `FIXED` into the app.

`registrationEstimator close` closes all open instances of the Registration Estimator app.

See Also

See Also

Functions

`imregconfig` | `imregdemons` | `imregister` | `imregtform` | `imwarp`

Topics

“Register Images Using the Registration Estimator App”

“Techniques Supported by Registration Estimator App”

Introduced in R2017a

Video Viewer

View videos and image sequences

Description

The **Video Viewer** app plays movies, videos, or image sequences. Using Video Viewer you can select the movie or image sequence that you want to play, jump to a specific frame in the sequence, change the frame rate of the display, or perform other viewing activities.

Open the Video Viewer App

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the Video Viewer app icon.
- MATLAB command prompt: Enter `implay`.

Examples

- “View Image Sequences in Video Viewer App”

Programmatic Use

`implay`

See Also

See Also

Apps

Image Viewer

Functions

`implay`

Topics

“View Image Sequences in Video Viewer App”

Introduced in R2014b

Volume Viewer

View volumetric image

Description

The **Volume Viewer** app lets you view 3-D volumetric images. Using this app, you can view 3-D image data as volumes or as plane slices, and do volume rendering, maximum intensity projection, and isosurface visualizations. Using the Rendering Editor component you can manipulate opacity to see the structures in the volume that you want to see and make transparent those structures in the volume that you do not want to see.

Open the Volume Viewer App

- MATLAB Toolstrip: Open the **Apps** tab, under **Image Processing and Computer Vision**, click the Volume Viewer app icon.
- MATLAB command prompt: Enter `volumeViewer`.

Examples

- “Explore 3-D Volumetric Data with the Volume Viewer App”

Programmatic Use

`volumeViewer` opens a volume visualization app.

`volumeViewer(I)` loads the volume *V* into the app.

`volumeViewer close` closes all open Volume Viewer apps.

See Also

See Also

Apps

Functions

Topics

“Explore 3-D Volumetric Data with the Volume Viewer App”

Introduced in R2017a

activecontour

Segment image into foreground and background using active contour

Syntax

```
bw = activecontour(A,mask)
bw = activecontour(A,mask,n)
bw = activecontour(A,mask,method)
bw = activecontour(A,mask,n,method)
bw = activecontour( ____,Name,Value)
```

Description

`bw = activecontour(A,mask)` segments the image `A` into foreground (object) and background regions using active contour segmentation. Active contour evolves the segmentation using an iterative process. By default, `activecontour` performs 100 iterations. `mask` is a binary image that specifies the initial state of the active contour. The boundaries of the object regions (white) in `mask` define the initial contour position used for contour evolution to segment the image. The output image `bw` is a binary image where the foreground is white (logical true) and the background is black (logical false).

To obtain faster and more accurate segmentation results, specify an initial contour position that is close to the desired object boundaries.

`bw = activecontour(A,mask,n)` segments the image by evolving the contour for a maximum of `n` iterations.

`bw = activecontour(A,mask,method)` specifies the active contour method used for segmentation, either `'Chan-Vese'` or `'edge'`.

`bw = activecontour(A,mask,n,method)` segments the image by evolving the contour for a maximum of `n` iterations using the specified `method`.

`bw = activecontour(____,Name,Value)` specifies parameters that control various aspects of the segmentation. Parameter names can be abbreviated, and case does not matter.

Examples

Segment an Image Specifying the Mask

This example shows how to segment an image using the default settings of the `activecontour` function.

Read a grayscale image and display it.

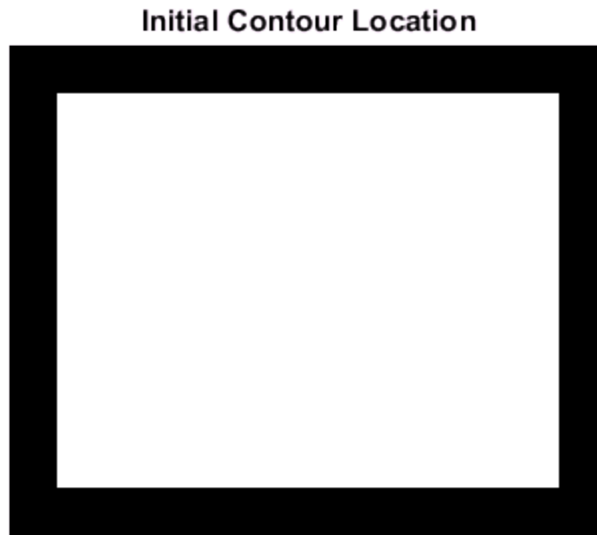
```
I = imread('coins.png');  
imshow(I)  
title('Original Image')
```



Specify the initial contour and display it.

```
mask = zeros(size(I));  
mask(25:end-25,25:end-25) = 1;
```

```
figure
imshow(mask)
title('Initial Contour Location')
```

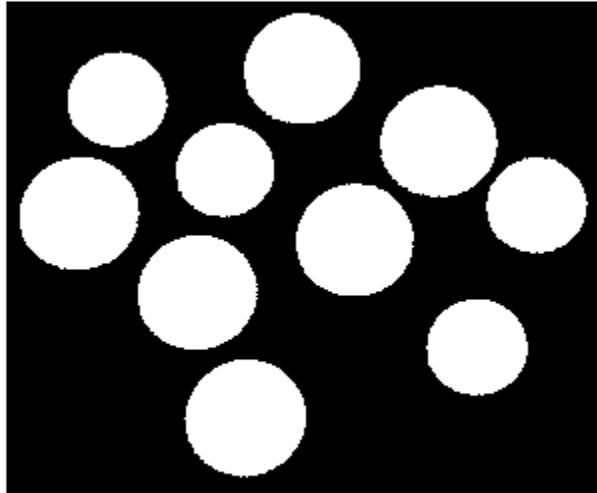


Segment the image using the default method and 300 iterations.

```
bw = activecontour(I,mask,300);
```

Display the result.

```
figure
imshow(bw)
title('Segmented Image')
```

Segmented Image**Segment Image Overlaying Mask and Contour on Original Image**

Read image and display it.

```
I = imread('toyobjects.png');  
imshow(I)  
hold on  
title('Original Image');
```

Original Image



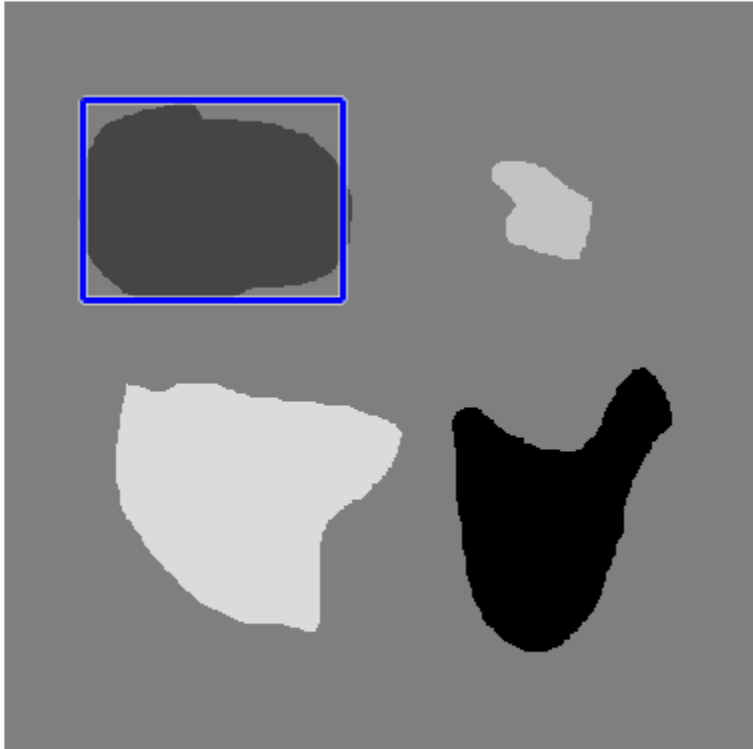
Specify initial contour location close to the object that is to be segmented.

```
mask = false(size(I));  
mask(50:150,40:170) = true;
```

Display the initial contour on the original image in blue.

```
visboundaries(mask, 'Color', 'b');
```

Original Image



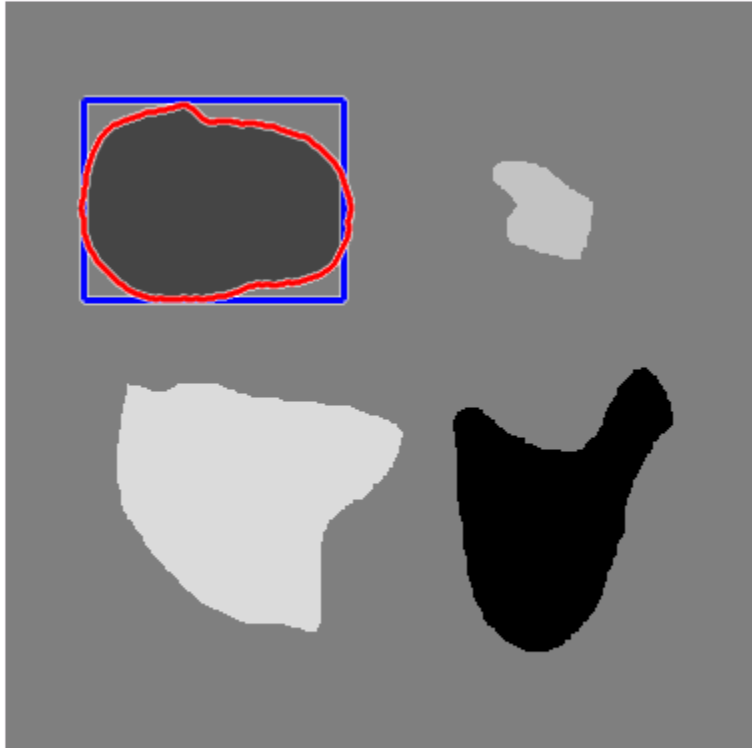
Segment the image using the 'edge' method and 200 iterations.

```
bw = activecontour(I, mask, 200, 'edge');
```

Display the final contour on the original image in red.

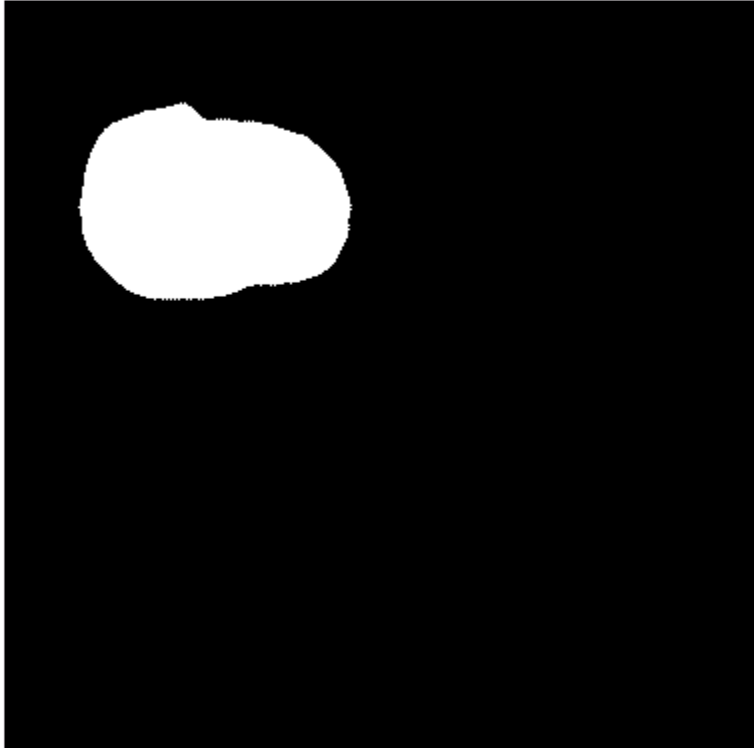
```
visboundaries(bw, 'Color', 'r');  
title('Initial contour (blue) and final contour (red)');
```

Initial contour (blue) and final contour (red)



Display segmented image.

```
figure, imshow(bw)
title('Segmented Image');
```


Segmented Image

Segment an Image Specifying a Polygonal Mask Created Interactively

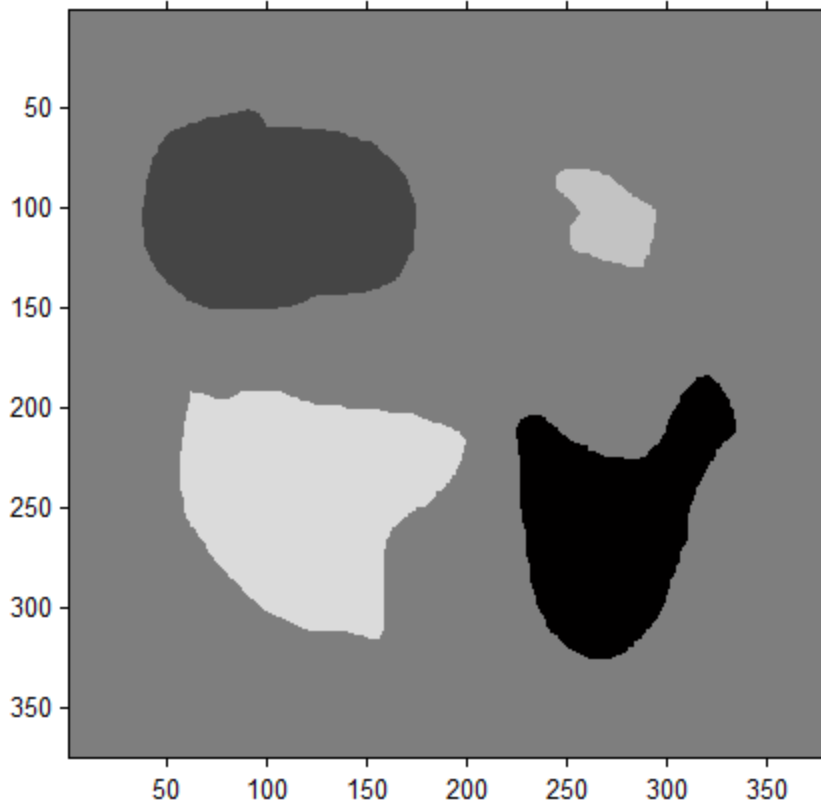
Read image into the workspace and display it. Display instructions to specify initial contour location.

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

```
str = 'Click to select initial contour location. Double-click to confirm and proceed.'
```

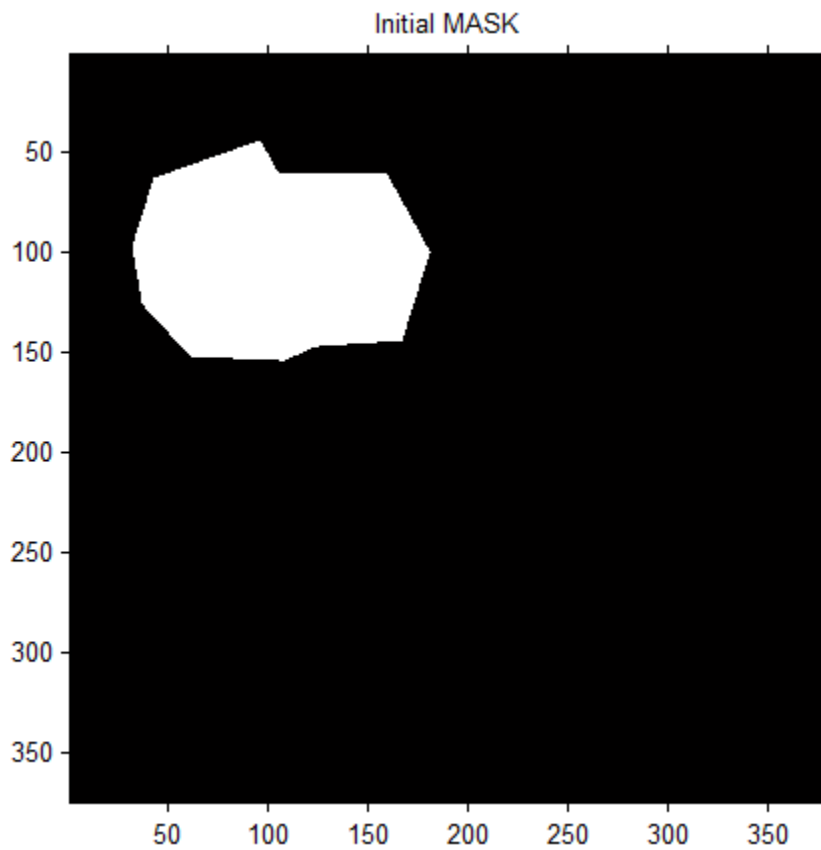
```
title(str,'Color','b','FontSize',12);  
disp(sprintf('\nNote: Click close to object boundaries for more accurate result.'))
```

Click to select initial contour location. Double-click to confirm and proceed.



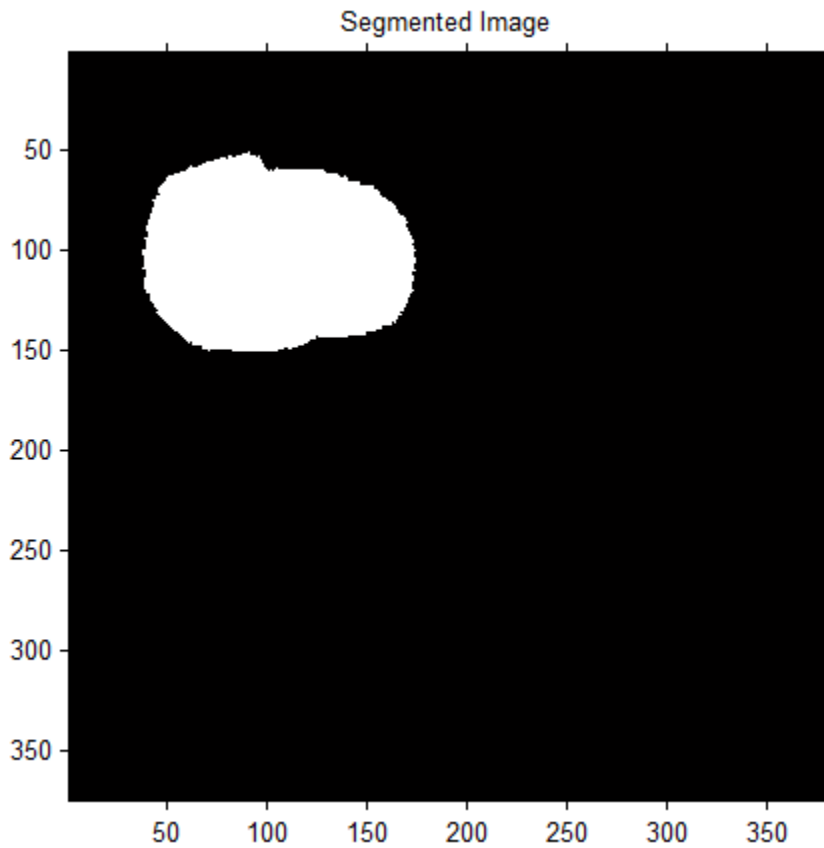
Specify initial contour interactively.

```
mask = roipoly;  
  
figure, imshow(mask)  
title('Initial MASK');
```



Segment the image, specifying 200 iterations.

```
maxIterations = 200;  
bw = activecontour(I, mask, maxIterations, 'Chan-Vese');  
  
% Display segmented image  
figure, imshow(bw)  
title('Segmented Image');
```



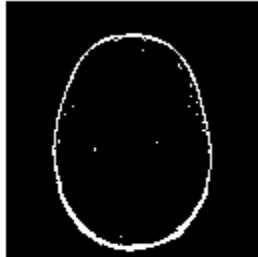
Perform 3-D Segmentation Using 2-D Initial Seed Mask

Load 3-D volumetric image data, removing the singleton dimension.

```
D = load('mri.mat');  
A = squeeze(D.D);
```

Create 2-D mask for initial seed points.

```
seedLevel = 10;  
seed = A(:,:,seedLevel) > 75;  
figure  
imshow(seed)
```



Create an empty 3-D seed mask and put the seed points into it.

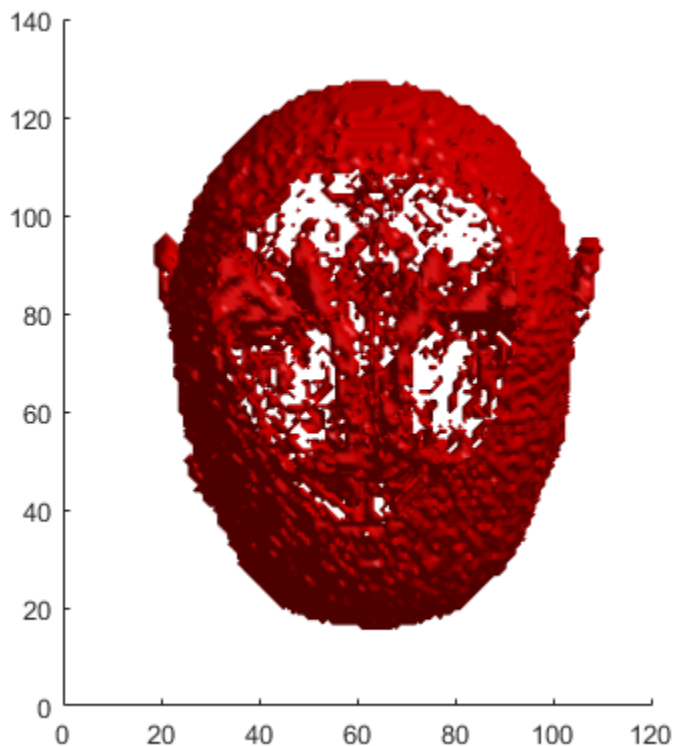
```
mask = zeros(size(A));  
mask(:,:,seedLevel) = seed;
```

Perform the segmentation using active contours, specifying the seed mask.

```
bw = activecontour(A,mask,300);
```

Display the 3-D segmented image.

```
figure;  
p = patch(isosurface(double(bw)));  
p.FaceColor = 'red';  
p.EdgeColor = 'none';  
daspect([1 1 27/128]);  
camlight;  
lighting phong
```



Input Arguments

A — Image to be segmented

nonsparse, 2-D or 3-D, numeric array

Image to segmented, specified as a nonsparse, 2-D or 3-D, numeric array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

mask — Initial contour at which the evolution of the segmentation begins

binary image

Initial contour at which the evolution of the segmentation begins, specified as a binary image the same size as `A`.

For 2-D and 3-D grayscale images, the size of `mask` must match the size of the image `A`. For color and multi-channel images, `mask` must be a 2-D logical array where the first two dimensions match the first two dimensions of the image `A`.

Data Types: logical

`n` — Maximum number of iterations to perform in evolution of the segmentation

100 (default) | numeric scalar.

Maximum number of iterations to perform in evolution of the segmentation, specified as a numeric scalar. `activecontour` stops the evolution of the active contour when it reaches the maximum number of iterations. `activecontour` also stops the evolution if the contour position in the current iteration is the same as the contour position in one of the most recent five iterations.

If the initial contour position (specified by `mask`) is far from the object boundaries, specify higher values of `n` to achieve desired segmentation results.

Data Types: double

`method` — Active contour method used for segmentation

'Chan-Vese' (default) | 'edge'

Active contour method used for segmentation, specified as the character vector 'Chan-Vese' or 'edge'. The Chan and Vese region-based energy model is described in [1]. The edge-based model, similar to Geodesic Active Contour, is described in [2].

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `bw = activecontour(I, mask, 200, 'edge', 'SmoothFactor', 1.5);`

'SmoothFactor' — Degree of smoothness or regularity of the boundaries of the segmented regions

0, for 'Chan-Vese'; 1 for 'edge' (default) | positive numeric scalar

Degree of smoothness or regularity of the boundaries of the segmented regions, specified as the comma-separated pair consisting of `'SmoothFactor'` and a positive numeric scalar. Higher values produce smoother region boundaries but can also smooth out finer details. Lower values produce more irregularities (less smoothing) in the region boundaries but allow finer details to be captured. The default smoothness value depends on the method chosen.

```
Example: bw = activecontour(I, mask, 200, 'edge', 'SmoothFactor', 1.5);
```

```
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32
```

'ContractionBias' — Tendency of the contour to grow outwards or shrink inwards

0, for `'Chan-Vese'`; 0.3 for `'edge'` (default) | scalar

Tendency of the contour to grow outwards or shrink inwards, specified as the comma-separated pair consisting of `'ContractionBias'` and a scalar. Positive values bias the contour to shrink inwards (contract). Negative values bias the contour to grow outwards (expand). This parameter does not guarantee that the contour contracts (or expands). It is possible that even with a positive value for this parameter, the contour could actually expand. However, by specifying a bias, you slow the expansion when compared to an unbiased contour. Typical values for this parameter are between -1 and 1.

```
Example: bw = activecontour(I, mask, 200,  
'edge', 'ContractionBias', 0.4);
```

```
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32
```

Output Arguments

bw — Segmented image

binary image the same size as the input image A.

Segmented image, returned as a binary image the same size as the input image A. The foreground is white (logical true) and the background is black (logical false).

Tips

- `activecontour` uses the boundaries of the regions in `mask` as the initial state of the contour from where the evolution starts. `mask` regions with holes can cause unpredictable results. Use `imfill` to fill any holes in the regions in `mask`.

- If a region touches the image borders, `activecontour` removes a single-pixel layer from the region, before further processing, so that the region does not touch the image border.
- To get faster and more accurate results, specify an initial contour position that is close to the desired object boundaries, especially for the 'edge' method.
- For the 'edge' method, the active contour is naturally biased towards shrinking inwards (collapsing). In the absence of any image gradient, the active contour shrinks on its own. Conversely, with the 'Chan-Vese' method, where the contour is unbiased, the contour is free to either shrink or expand based on the image features.
- To achieve an accurate segmentation with the 'edge' method, specify an initial contour that lies outside the boundaries of the object. The active contour with the 'edge' method is biased to shrink, by default.
- If object regions are of significantly different grayscale intensities, the 'Chan-Vese' method [1] might not segment all objects in the image. For example, if the image contains objects that are brighter than the background and some that are darker, the 'Chan-Vese' method typically segments out either the dark or the bright objects only.

Algorithms

`activecontour` uses the Sparse-Field level-set method, similar to the method described in [3], for implementing active contour evolution.

References

- [1] T. F. Chan, L. A. Vese, *Active contours without edges*. IEEE Transactions on Image Processing, Volume 10, Issue 2, pp. 266-277, 2001
- [2] V. Caselles, R. Kimmel, G. Sapiro, *Geodesic active contours*. International Journal of Computer Vision, Volume 22, Issue 1, pp. 61-79, 1997.
- [3] R. T. Whitaker, *A level-set approach to 3d reconstruction from range data*. International Journal of Computer Vision, Volume 29, Issue 3, pp.203-231, 1998.

See Also

See Also

Image Segmenter | `imellipse` | `imfreehand` | `multithresh` | `poly2mask` | `roipoly`

Introduced in R2013a

adapthisteq

Contrast-limited adaptive histogram equalization (CLAHE)

Syntax

```
J = adapthisteq(I)
J = adapthisteq(I,Name,Value,...)
```

Description

`J = adapthisteq(I)` enhances the contrast of the grayscale image `I` by transforming the values using contrast-limited adaptive histogram equalization (CLAHE) [1].

CLAHE operates on small regions in the image, called *tiles*, rather than the entire image. Each tile's contrast is enhanced, so that the histogram of the output region approximately matches the histogram specified by the 'Distribution' parameter. The neighboring tiles are then combined using bilinear interpolation to eliminate artificially induced boundaries. The contrast, especially in homogeneous areas, can be limited to avoid amplifying any noise that might be present in the image.

`J = adapthisteq(I,Name,Value,...)` specifies any of the additional parameter/value pairs. Parameter names can be abbreviated, and case does not matter.

Examples

Apply Contrast-limited Adaptive Histogram Equalization (CLAHE)

Apply Contrast-limited Adaptive Histogram Equalization (CLAHE) to an image and display the results.

```
I = imread('tire.tif');
A = adapthisteq(I,'clipLimit',0.02,'Distribution','rayleigh');
figure, imshow(I);
```



`figure, imshow(A);`



Apply CLAHE to a color image

Read the color image into the workspace.

```
[X, MAP] = imread('shadow.tif');
```

Convert the indexed image into a truecolor (RGB) image.

```
RGB = ind2rgb(X,MAP);
```

Convert the RGB image into the L*a*b* color space.

```
LAB = rgb2lab(RGB);
```

Scale values to the range expected by the `adapthisteq` function, [0 1].

```
L = LAB(:,:,1)/100;
```

Perform CLAHE. Multiply the result to get back to the range used by the L*a*b* color space.

```
LAB(:,:,1) = adapthisteq(L, 'NumTiles', ...  
                        [8 8], 'ClipLimit', 0.005)*100;
```

Convert the resultant image back into the RGB color space.

```
J = lab2rgb(LAB);
```

Display the original image and result.

```
figure, imshow(RGB);
```



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



Input Arguments

I — Input Image

2-D array

Input intensity image, specified as a numeric 2-D array.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `'NumTiles', [8 16]` divides the image into 8 rows and 16 columns of tiles.

'NumTiles' — Number of tiles

[8,8] (default) | two-element vector of positive integers

Number of tiles, specified as a two-element vector of positive integers. With the original image divided into M rows and N columns of tiles, the value of 'NumTiles' is [M N]. Both M and N must be at least 2. The total number of tiles is equal to $M*N$.

Data Types: double

'ClipLimit' — Contrast enhancement limit

0.01 (default) | real scalar

Contrast enhancement limit, specified as a real scalar in the range [0, 1]. Higher limits result in more contrast.

Data Types: double

'NBins' — Number of histogram bins used to build a contrast enhancing transformation

256 (default) | positive integer scalar

Number of histogram bins used to build a contrast enhancing transformation, specified as a positive integer scalar. Higher values result in greater dynamic range at the cost of slower processing speed.

Data Types: double

'Range' — Range of output data

'full' (default) | 'original'

Range of the output image data, specified as one of the following values:

Value	Description
'full'	Limit the range to $[\min(I(:)) \max(I(:))]$.
'original'	Use the full range of the output class (e.g. [0 255] for uint8).

'Distribution' — Desired histogram shape

'uniform' (default) | 'rayleigh' | 'exponential'

Desired histogram shape, specified as one of the following values:

Value	Description
'uniform'	Create a flat histogram.
'rayleigh'	Create a bell-shaped histogram.
'exponential'	Create a curved histogram.

'Alpha' — Distribution parameter

0.4 (default) | nonnegative real scalar

Distribution parameter, specified as a nonnegative real scalar. 'Alpha' is only used when 'Distribution' is set to 'rayleigh' or 'exponential'.

Data Types: double

Output Arguments

J — Output intensity image

2-D array

Output intensity image, returned as a 2-D array of the same class as the input image I.

Tips

- 'NumTiles' specifies the number of rectangular contextual regions (tiles) into which adapthisteq divides the image. adapthisteq calculates the contrast transform function for each of these regions individually. The optimal number of tiles depends on the type of the input image, and it is best determined through experimentation.
- 'ClipLimit' is a contrast factor that prevents over-saturation of the image specifically in homogeneous areas. These areas are characterized by a high peak in the histogram of the particular image tile due to many pixels falling inside the same gray level range. Without the clip limit, the adaptive histogram equalization technique could produce results that, in some cases, are worse than the original image.
- 'Distribution' specifies the distribution that adapthisteq uses as the basis for creating the contrast transform function. The distribution you select should depend on the type of the input image. For example, underwater imagery appears to look more natural when the Rayleigh distribution is used.

References

- [1] Zuiderveld, Karel. “Contrast Limited Adaptive Histogram Equalization.” *Graphic Gems IV*. San Diego: Academic Press Professional, 1994. 474–485.

See Also

See Also

histeq

Introduced before R2006a

adapthresh

Adaptive image threshold using local first-order statistics

Syntax

```
T = adapthresh(I)
T = adapthresh(I,sensitivity)
T = adapthresh( ____,Name,Value)
```

Description

`T = adapthresh(I)` computes a locally adaptive threshold that can be used with the `imbinarize` function to convert an intensity image to a binary image. The result, `T`, is a matrix the same size as `I` containing normalized intensity values in the range `[0, 1]`. `adapthresh` chooses the threshold based on the local mean intensity (first-order statistics) in the neighborhood of each pixel.

`T = adapthresh(I,sensitivity)` computes a locally adaptive threshold with sensitivity factor specified by `sensitivity`. `sensitivity` is a scalar in the range `[0 1]` that indicates sensitivity towards thresholding more pixels as foreground.

`T = adapthresh(____,Name,Value)` computes a locally adaptive threshold using name-value pairs to control aspects of the thresholding.

Examples

Find Threshold and Segment Bright Rice Grains from Dark Background

Read image into the workspace.

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

Use `adapthresh` to determine threshold to use in binarization operation.

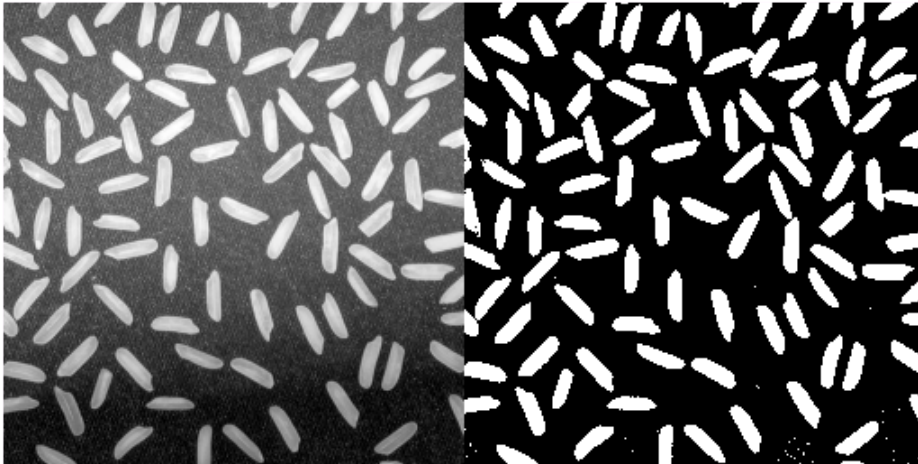
```
T = adapthresh(I, 0.4);
```

Convert image to binary image, specifying the threshold value.

```
BW = imbinarize(I,T);
```

Display the original image with the binary version, side-by-side.

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



Find Threshold and Segment Dark Text from Bright Background

Read image into the workspace.

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

Using `adaptthresh` compute adaptive threshold and display the local threshold image. This represents an estimate of average background illumination.

```
T = adaptthresh(I,0.4, 'ForegroundPolarity', 'dark');  
figure  
imshow(T)
```



Binarize image using locally adaptive threshold

```
BW = imbinarize(I,T);  
figure  
imshow(BW)
```

What Is Image Filtering in the Spatial Domain?

Filtering is a technique for modifying or enhancing an image. For example, you can filter an image to emphasize certain features or remove other features. Image processing operations implemented with filtering include smoothing, sharpening, and edge enhancement.

Filtering is a *neighborhood operation*, in which the value of any given pixel in the output image is determined by applying some algorithm to the values of the pixels in the neighborhood of the corresponding input pixel. A pixel's neighborhood is some set of pixels, defined by their locations relative to that pixel. (See *Neighborhood* or *Block Processing: An Overview* for a general discussion of neighborhood operations.) *Linear filtering* is filtering in which the value of an output pixel is a linear combination of the values of the pixels in the input pixel's neighborhood.

Convolution

Linear filtering of an image is accomplished through an operation called *convolution*. Convolution is a neighborhood operation in which each output pixel is the weighted sum of neighboring input pixels. The matrix of weights is called the *convolution kernel*, also known as the *filter*. A convolution kernel is a correlation kernel that has been rotated 180 degrees.

For example, suppose the image is

```
A = [17 24  1  8 15
      23  5  7 14 16
        4  6 13 20 22
       10 12 19 21  3
       11 10  7  5  4]
```

Input Arguments

I — Input intensity image

real, nonsparse 2-D matrix

Input intensity image, specified as a real, nonsparse, 2-D matrix. If the image contains Infs or NaNs, the behavior of `adaptthresh` is undefined. Propagation of Infs or NaNs may not be localized to the neighborhood around Inf or NaN pixels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

sensitivity — Determine which pixels get thresholded as foreground pixels

0.5 (default) | real, nonnegative numeric scalar in the range [0 1]

Determine which pixels get thresholded as foreground pixels, specified as a real, nonnegative numeric scalar in the range [0 1]. High sensitivity values lead to thresholding more pixels as foreground, at the risk of including some background pixels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `T = adapthresh(I,0.4,'ForegroundPolarity','dark');`

'NeighborhoodSize' — Size of neighborhood used to compute local statistic around each pixel

`2*floor(size(I)/16)+1` (default) | real, numeric scalar or two-element vector of positive odd integers

Size of neighborhood used to compute local statistic around each pixel, specified as a real, numeric, scalar or two-element vector of positive odd integers.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'ForegroundPolarity' — Determine which pixels are considered foreground pixels

'bright' (default) | 'dark'

Determine which pixels are considered foreground pixels, specified using either of the following:

Value	Meaning
'bright'	The foreground is brighter than the background. This is the default.
'dark'	The foreground is darker than the background

Data Types: `char`

'Statistic' — Statistic used to compute local threshold at each pixel

'mean' (default) | 'median' | 'gaussian'

Statistic used to compute local threshold at each pixel, specified as one of the following:

Value	Meaning
'mean'	The local mean intensity in the neighborhood. This is the default. This technique is also called Bradley's method.
'median'	The local median in the neighborhood. Computation of this statistic can be slow. Consider using a smaller neighborhood size to obtain faster results.
'gaussian'	The Gaussian weighted mean in the neighborhood.

Data Types: char

Output Arguments

T — Normalized intensity values

double matrix

Normalized intensity values, returned as a **double** matrix the same size as **I**.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder™. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- The **ForegroundPolarity** and **Statistic** arguments must be compile-time constants.

See Also

See Also

graythresh | imbinarize | otsuthresh

Introduced in R2016a

affine2d class

2-D affine geometric transformation

Description

An `affine2d` object encapsulates a 2-D affine geometric transformation.

You can create an `affine2d` object using the following methods:

- `imregtform` — Returns an `affine2d` object that maps a moving image to a fixed image
- `fitgeotrans` — Returns an `affine2d` object that maps control point pairs
- The `affine2d` class constructors

Construction

`tform = affine2d()` creates an `affine2d` object with default property settings that correspond to the identity transformation.

`tform = affine2d(A)` creates an `affine2d` object given an input 3-by-3 matrix `A` that specifies a valid affine transformation.

Input Arguments

A — Forward 2-D affine transformation

3-by-3 numeric matrix

Forward 2-D affine transformation, specified as a 3-by-3 numeric matrix. `A` has the form:

```
A = [a b 0;  
     c d 0;  
     e f 1];
```

Data Types: `double` | `single`

Properties

T — Forward 2-D affine transformation

3-by-3 numeric matrix

Forward 2-D affine transformation, specified as a 3-by-3 numeric matrix.

The matrix T uses the convention:

$$[x \ y \ 1] = [u \ v \ 1] * T$$

where T has the form:

$$\begin{bmatrix} a & b & 0; \\ c & d & 0; \\ e & f & 1 \end{bmatrix};$$

The default of T is the identity transformation.

Data Types: `double` | `single`

Dimensionality — Describes the dimensionality of the geometric transformation

2

Describes the dimensionality of the geometric transformation for both input and output points.

Methods

`invert`

Invert geometric transformation

`outputLimits`

Find output spatial limits given input spatial limits

`transformPointsForward`

Apply forward geometric transformation

`transformPointsInverse`

Apply inverse geometric transformation

<code>isRigid</code>	Determine if transformation is rigid transformation
<code>isTranslation</code>	Determine if transformation is pure translation
<code>isSimilarity</code>	Determine if transformation is similarity transformation

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#) in the MATLAB documentation.

Examples

Define 10-Degree Rotation in the Counter-Clockwise Direction

Create an `affine2d` object that defines the transformation.

```
theta = 10;
tform = affine2d([cosd(theta) -sind(theta) 0; sind(theta) cosd(theta) 0; 0 0 1])
tform =
    affine2d with properties:
        T: [3x3 double]
        Dimensionality: 2
```

Apply forward geometric transformation to an input (U,V) point (5,10).

```
[X,Y] = transformPointsForward(tform,5,10)
X =
    6.6605
```

```
Y =
    8.9798
```

Apply inverse geometric transformation to output (X,Y) point from the previous step to recover the original points from the inverse transformation.

```
[U,V] = transformPointsInverse(tform,X,Y)
```

```
U =
    5.0000
```

```
V =
    10
```

Apply 10-Degree Counter-Clockwise rotation to Image Using `imwarp` Function

Read image.

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

Create an `affine2d` object that defines the transformation.

```
theta = 10;
```

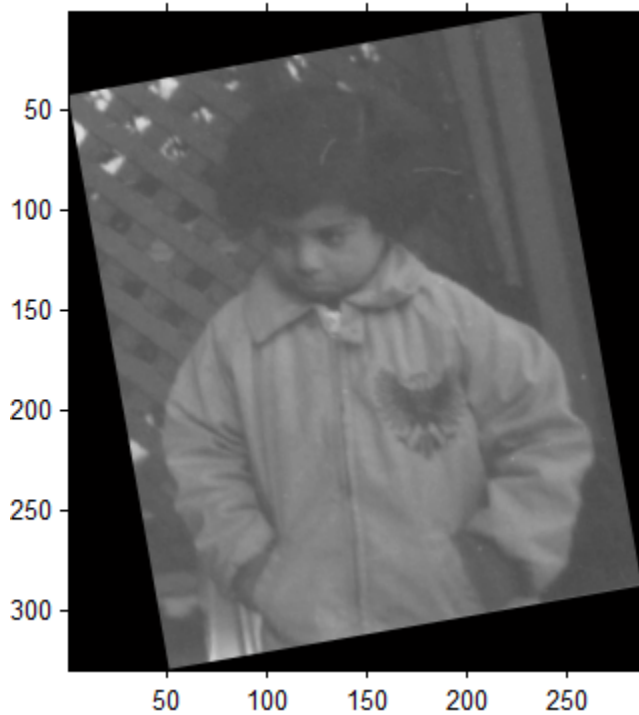
```
tform = affine2d([cosd(theta) -sind(theta) 0; sind(theta) cosd(theta) 0; 0 0 1])
```

```
tform =
```

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

Apply geometric transformation to image.

```
outputImage = imwarp(A,tform);
figure, imshow(outputImage);
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, you can only specify singular objects—arrays of objects are not supported.

See Also

See Also

Functions

[fitgeotrans](#) | [imregister](#) | [imregtform](#) | [imwarp](#)

Classes

[affine3d](#) | [projective2d](#) | [images.geotrans.PiecewiseLinearTransformation2d](#)
| [images.geotrans.PolynomialTransformation2d](#) |
[images.geotrans.LocalWeightedMeanTransformation2d](#)

invert

Class: `affine2d`

Invert geometric transformation

Syntax

```
invtfom = invert(tform)
```

Description

`invtfom = invert(tform)` returns the inverse of the geometric transformation `tform`.

Input Arguments

tform

Geometric transformation, specified as an `affine2d` geometric transformation object.

Output Arguments

invtfom

Inverse of the geometric transformation, returned as an `affine2d` geometric transformation object

Examples

Invert Geometric Transformation Object

Create an `affine2d` object that defines a rotation of 10 degrees counter-clockwise.


```
theta = 10;  
tform = affine2d([cosd(theta) -sind(theta) 0; sind(theta) cosd(theta) 0; 0 0 1]);
```

```
tform =
```

```
    affine2d with properties:
```

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

Invert the geometric transformation.

```
invtfom = invert(tform)
```

```
invtfom =
```

```
    affine2d with properties:
```

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

See Also

outputLimits

Class: affine2d

Find output spatial limits given input spatial limits

Syntax

```
[xLimitsOut,yLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn)
```

Description

`[xLimitsOut,yLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn)` estimates the output spatial limits corresponding to a given geometric transformation, `tform`, and a set of input spatial limits, `xLimitsIn` and `yLimitsIn`.

Input Arguments

tform

Geometric transformation, specified as an `affine2d` geometric transformation object.

xLimitsIn

Input spatial limits in *X* dimension, specified as a two-element vector of `doubles`.

yLimitsIn

Input spatial limits in *Y* dimension, specified as a two-element vector of `doubles`.

Output Arguments

xLimitsOut

Output spatial limits in *X* dimension, returned as a two-element vector of `doubles`.

yLimitsOut

Output spatial limits in Y dimension, returned as a two-element vector of **doubles**.

Examples

Estimate the Output Limits for a Geometric Transformation

Create an `affine2d` object that defines a rotation of 10 degrees counter-clockwise.

```
theta = 10;  
tform = affine2d([cosd(theta) -sind(theta) 0; sind(theta) cosd(theta) 0; 0 0 1]);  
tform =
```

```
    affine2d with properties:
```

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

Estimate the output spatial limits, given the geometric transformation.

```
[xlim, ylim] = outputLimits(tform,[1 240],[1 291])
```

```
xlim =
```

```
    1.1585    286.8855
```

```
ylim =
```

```
   -40.6908    286.4054
```

See Also

transformPointsForward

Class: affine2d

Apply forward geometric transformation

Syntax

```
[x,y] = transformPointsForward(tform,u,v)  
X = transformPointsForward(tform,U)
```

Description

`[x,y] = transformPointsForward(tform,u,v)` applies the forward geometric transformation of `tform` to the input 2-D point arrays `u` and `v` and outputs the point arrays `x` and `y`. The input point arrays `u` and `v` must be of the same size.

`X = transformPointsForward(tform,U)` applies the forward geometric transformation of `tform` to the input n -by-2 point matrix `U` and outputs the n -by-2 point matrix `X`. `transformPointsForward` maps the point `U(k, :)` to the point `X(k, :)`.

Input Arguments

tform

Geometric transformation, specified as an `affine2d` geometric transformation object.

u

Coordinates in X dimension of points to be transformed, specified as an array.

v

Coordinates in Y dimension of points to be transformed, specified as an array.

U

X and Y coordinates of points to be transformed, specified as an n -by-2 matrix

Output Arguments

x

Transformed coordinates in X dimension, returned as a array.

y

Transformed coordinates in Y dimension, returned as a array.

X

Transformed points in X and Y dimensions, returned as an n -by-2 point matrix

Examples

Apply Forward Geometric Transformation

Create an `affine2d` object that defines the transformation.

```
theta = 10;
```

```
tform = affine2d([cosd(theta) -sind(theta) 0; sind(theta) cosd(theta) 0; 0 0 1])
```

```
tform =
```

```
    affine2d with properties:
```

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

Apply forward geometric transformation to an input (U,V) point.

```
[X,Y] = transformPointsForward(tform,5,10)
```

```
X =
```

```
    6.6605
```

```
Y =
```

8.9798

See Also

transformPointsInverse

Class: affine2d

Apply inverse geometric transformation

Syntax

```
[u,v] = transformPointsInverse(tform,x,y)
U = transformPointsInverse(tform,X)
```

Description

`[u,v] = transformPointsInverse(tform,x,y)` applies the inverse geometric transformation of `tform` to the input 2-D point arrays `x` and `y` and outputs the point arrays `u` and `v`. The input point arrays `x` and `y` must be of the same size.

`U = transformPointsInverse(tform,X)` applies the inverse geometric transformation of `tform` to the input n -by-2 point matrix `X` and outputs the n -by-2 point matrix `U`. `transformPointsInverse` maps the point `X(k,:)` to the point `U(k,:)`.

Input Arguments

tform

Geometric transformation, specified as an `affine2d` geometric transformation object.

Default:

x

Coordinates in X dimension of points to be transformed, specified as a array.

y

Coordinates in Y dimension of points to be transformed, specified as a array.

X

X and Y coordinates of points to be transformed, specified as an n -by-2 matrix

Output Arguments

u

Transformed coordinates in X dimension, returned as an array.

v

Transformed coordinates in Y dimension, returned as an array.

U

Transformed X and Y coordinates, returned as an n -by-2 matrix

Examples

Apply Inverse Geometric Transformation

Create an `affine2d` object that defines the transformation.

```
theta = 10;
```

```
tform = affine2d([cosd(theta) -sind(theta) 0; sind(theta) cosd(theta) 0; 0 0 1])
```

```
tform =
```

```
    affine2d with properties:
```

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

Apply forward geometric transformation to an input point.

```
[X,Y] = transformPointsForward(tform,5,10)
```

```
X =
```


6.6605

Y =

8.9798

Apply inverse geometric transformation to output point from the previous step to recover the original coordinates.

```
[U,V] = transformPointsInverse(tform,X,Y)
```

U =

5.0000

V =

10

See Also

isRigid

Class: affine2d

Determine if transformation is rigid transformation

Syntax

```
TF = isRigid(tform)
```

Description

`TF = isRigid(tform)` determines whether or not the affine transformation specified by `tform` is a rigid transformation. `TF` is a scalar boolean that is true when `tform` is a rigid transformation. A `tform` is a rigid transformation when `tform.T` defines only rotation and translation.

Input Arguments

tform

Geometric transformation, specified as an `affine2d` geometric transformation object.

Output Arguments

TF

Scalar boolean, returned as `true` when the `tform` is a rigid transformation.

Examples

Check if transformation is rigid

Create an `affine2d` object that defines a pure translation.

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

tform = affine2d(A)

tform =

    affine2d with properties:

                T: [3x3 double]
    Dimensionality: 2

Test if it is a rigid transformation.

tf = isRigid(tform)

tf =

     1
```

See Also

isTranslation

Class: affine2d

Determine if transformation is pure translation

Syntax

```
TF = isTranslation(tform)
```

Description

`TF = isTranslation(tform)` determines whether the affine transformation specified in `tform` is a pure translation transformation. `TF` is a scalar boolean that is true when `tform` defines only translation.

Input Arguments

tform

Geometric transformation, specified as an `affine2d` geometric transformation object.

Output Arguments

TF

Scalar boolean, returned as `true` when the `tform` is a pure translation.

Examples

Check if transformation is a pure translation

Create an `affine2d` object that defines a pure translation.

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

```
tform = affine2d(A)
```

```
tform =
```

```
    affine2d with properties:
```

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

Check if transformation is a pure translation.

```
tf = isTranslation(tform)
```

```
tf =
```

```
    1
```

See Also

isSimilarity

Class: affine2d

Determine if transformation is similarity transformation

Syntax

```
TF = isSimilarity(tform)
```

Description

`TF = isSimilarity(tform)` determines whether or not the affine transformation specified by `tform` is a similarity transformation. `TF` is a scalar boolean that is true when `tform` is a similarity transformation. A `tform` is a similarity transformation when it defines only homogeneous scale, rotation, and translation.

Input Arguments

tform

Geometric transformation, specified as an `affine2d` geometric transformation object.

Output Arguments

TF

Scalar boolean, returned as `true` when the `tform` is a similarity transformation.

Examples

Check if transformation is a similarity transformation

Create an `affine2d` object that defines a pure translation.

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

```
tform = affine2d(A)
```

```
tform =
```

```
    affine2d with properties:
```

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

Check if transformation is a similarity transformation.

```
tf = isSimilarity(tform)
```

```
tf =
```

```
    1
```

See Also

affine3d class

3-D affine geometric transformation

Description

An `affine3d` object encapsulates a 3-D affine geometric transformation.

You can create an `affine3d` object using the following methods:

- `imregtform` — Returns an `affine3d` object that maps a moving 3-D image to a fixed 3-D image
- The `affine3d` class constructors

Construction

`tform = affine3d()` creates an `affine3d` object with default property settings that correspond to the identity transformation.

`tform = affine3d(A)` constructs an `affine3d` object given an input 4-by-4 matrix `A` that specifies a valid 4-by-4 affine transformation matrix.

Input Arguments

A — Forward 3-D affine transformation

4-by-4 numeric matrix

Forward 3-D affine transformation, specified as a 4-by-4 numeric matrix. `A` has the form:

```
A = [a b c 0;  
     d e f 0;  
     g h i 0;  
     j k l 1];
```

Data Types: `double` | `single`

Properties

T — Forward 3-D affine transformation

4-by-4 numeric matrix

Forward 3-D affine transformation, specified as a 4-by-4 numeric matrix.

The matrix T uses the convention:

$$[x \ y \ z \ 1] = [u \ v \ w \ 1] * T$$

where T has the form:

```
[ a b c 0;
  d e f 0;
  g h i 0;
  j k l 1];
```

The default of T is the identity transformation.

Data Types: `double` | `single`

Dimensionality — Describes the dimensionality of the geometric transformation

3

Describes the dimensionality of the geometric transformation for both input and output points.

Methods

<code>invert</code>	Invert geometric transformation
<code>outputLimits</code>	Find output spatial limits given input spatial limits
<code>transformPointsForward</code>	Apply forward geometric transformation
<code>transformPointsInverse</code>	Apply inverse 3-D geometric transformation to points
<code>isRigid</code>	Determine if transformation is rigid transformation

<code>isTranslation</code>	Determine if transformation is pure translation
<code>isSimilarity</code>	Determine if transformation is similarity transformation

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#) in the MATLAB documentation.

Examples

Define a Different Scale Factor in Each Dimension

Create an `affine3d` object that defines a different scale factor in each dimension.

```
Sx = 1.2;  
Sy = 1.6;  
Sz = 2.4;  
tform = affine3d([Sx 0 0 0; 0 Sy 0 0; 0 0 Sz 0; 0 0 0 1]);
```

```
tform =
```

```
    affine3d with properties:
```

```
          T: [4x4 double]  
    Dimensionality: 3
```

Apply forward geometric transformation to an input point.

```
[X,Y,Z] = transformPointsForward(tform,1,1,1)
```

```
X =
```

```
    1.2000
```

```
Y =
```

```
    1.6000
```

```
Z =
    2.4000
```

Apply inverse geometric transformation to output points from the previous step to recover the original points from the inverse transformation.

```
[U,V,W] = transformPointsInverse(tform,X,Y,Z)
```

```
U =
    1
```

```
V =
    1
```

```
W =
    1
```

Apply Scale Transformation to an MRI Volume Using `imwarp` Function

Load MRI images.

```
A = load('mri');
A = squeeze(A.D);
```

Create an `affine3d` object that defines a different scale factor in each dimension.

```
Sx = 1.2;
Sy = 1.6;
Sz = 2.4;
tform = affine3d([Sx 0 0 0; 0 Sy 0 0; 0 0 Sz 0; 0 0 0 1]);
```

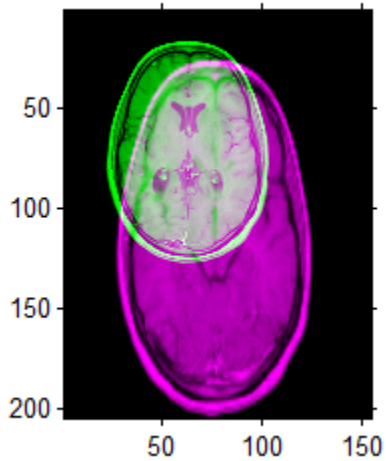
```
tform =
```

```
affine3d with properties:
```

```
          T: [4x4 double]
Dimensionality: 3
```

Apply geometric transformation to image using `imwarp` and visualize axial slice through center of transformed volume to see effect of scale transformation.

```
outputImage = imwarp(A,tform);  
figure, imshowpair(A(:,:,14),outputImage(:,:,27));
```



See Also

See Also

Functions

`imregister` | `imregtform` | `imwarp`

Classes

`affine2d`

invert

Class: `affine3d`

Invert geometric transformation

Syntax

```
invtfom = invert(tform)
```

Description

`invtfom = invert(tform)` returns the inverse of the geometric transformation `tform`.

Input Arguments

tform

Geometric transformation, specified as an `affine3d` geometric transformation object.

Output Arguments

invtfom

Inverse of the geometric transformation, returned as an `affine3d` geometric transformation object

Examples

Invert Geometric Transformation Object

Create an `affine3d` object that defines a different scale factor in each dimension.

```
Sx = 1.2;  
Sy = 1.6;  
Sz = 2.4;  
tform = affine3d([Sx 0 0 0; 0 Sy 0 0; 0 0 Sz 0; 0 0 0 1]);
```

```
tform =
```

```
    affine3d with properties:
```

```
                T: [4x4 double]  
    Dimensionality: 3
```

Invert the geometric transformation.

```
invtform = invert(tform)
```

```
invtform =
```

```
    affine3d with properties:
```

```
                T: [4x4 double]  
    Dimensionality: 3
```

See Also

outputLimits

Class: affine3d

Find output spatial limits given input spatial limits

Syntax

```
[xLimitsOut,yLimitsOut,zLimitsOut] =  
outputLimits(tform,xLimitsIn,yLimitsIn,zLimitsIn)
```

Description

[xLimitsOut,yLimitsOut,zLimitsOut] =
outputLimits(tform,xLimitsIn,yLimitsIn,zLimitsIn) estimates the output spatial limits corresponding to a given geometric transformation and a set of input spatial limits.

Input Arguments

tform

Geometric transformation, specified as an `affine3d` geometric transformation object.

xLimitsIn

Input spatial limits in X dimension, specified as a two-element vector of doubles.

yLimitsIn

Input spatial limits in Y dimension, specified as a two-element vector of doubles.

zLimitsIn

Input spatial limits in Z dimension, specified as a two-element vector of doubles.

Output Arguments

xLimitsOut

Output spatial limits in X dimension, returned as a two-element vector of **doubles**.

yLimitsOut

Output spatial limits in Y dimension, returned as a two-element vector of **doubles**.

zLimitsOut

Output spatial limits in Z dimension, returned as a two-element vector of **doubles**.

Examples

Estimate the Output Limits for a Geometric Transformation

Create an `affine3d` object that defines a different scale factor in each dimension.

```
Sx = 1.2;  
Sy = 1.6;  
Sz = 2.4;  
tform = affine3d([Sx 0 0 0; 0 Sy 0 0; 0 0 Sz 0; 0 0 0 1]);
```

```
tform =
```

```
    affine3d with properties:
```

```
          T: [4x4 double]  
    Dimensionality: 3
```

Estimate the output spatial limits, given the geometric transformation.

```
[xlim, ylim, zlim] = outputLimits(tform,[1 128],[1 128],[1 27])
```

```
xlim =
```

```
    1.2000  153.6000
```

```
ylim =
```



```
1.6000 204.8000
```

```
zlim =
```

```
2.4000 64.8000
```

See Also

transformPointsForward

Class: affine3d

Apply forward geometric transformation

Syntax

```
[x,y,z] = transformPointsForward(tform,u,v,w)  
X = transformPointsForward(tform,U)
```

Description

`[x,y,z] = transformPointsForward(tform,u,v,w)` applies the forward transformation of `tform` to the input 3-D point arrays `u,v`, and `w` and outputs the point arrays `x,y`, and `z`. The input point arrays `u,v`, and `w` must be of the same size.

`X = transformPointsForward(tform,U)` applies the forward transformation of `tform` to the input n -by-2 point matrix `U` and outputs the n -by-2 point matrix `X`. `transformPointsForward` maps the point `U(k, :)` to the point `X(k, :)`.

Input Arguments

tform

Geometric transformation, specified as an `affine3` geometric transformation object.

u

Coordinates in X dimension of points to be transformed, specified as an array.

v

Coordinates in Y dimension of points to be transformed, specified as an array.

w

Coordinates in Z dimension of points to be transformed, specified as an array.

U*n*-by-2 point matrix

Output Arguments

xTransformed coordinates in *X* dimension, returned as a array.**y**Transformed coordinates in *Y* dimension, returned as a array.**z**Transformed coordinates in *Z* dimension, returned as a array.**X**Transformed points, returned as an *n*-by-2 point matrix.

Examples

Apply Forward Geometric Transformation

Create an `affine3d` object that defines a different scale factor in each dimension.

```
Sx = 1.2;  
Sy = 1.6;  
Sz = 2.4;  
tform = affine3d([Sx 0 0 0; 0 Sy 0 0; 0 0 Sz 0; 0 0 0 1]);
```

```
tform =
```

```
    affine3d with properties:
```

```
        T: [4x4 double]  
    Dimensionality: 3
```

Apply forward geometric transformation to an input points.

```
[X,Y,Z] = transformPointsForward(tform,5,10,3)
```

```
X =
```

```
6
```

```
Y =
```

```
16
```

```
Z =
```

```
7.2000
```

See Also

transformPointsInverse

Class: affine3d

Apply inverse 3-D geometric transformation to points

Syntax

```
[u,v,w] = transformPointsInverse(tform,x,y,z)  
U = transformPointsInverse(tform,X)
```

Description

`[u,v,w] = transformPointsInverse(tform,x,y,z)` applies the inverse transformation of `tform` to the input 3-D point arrays `x`, `y`, and `z` and outputs the point arrays `u`, `v`, and `w`. The input point arrays `x`, `y`, and `z` must be of the same size.

`U = transformPointsInverse(tform,X)` applies the inverse transformation of `tform` to the input n -by-2 point matrix `X` and outputs the n -by-2 point matrix `U`. `transformPointsInverse` maps the point `X(k,:)` to the point `U(k,:)`.

Input Arguments

tform

Geometric transformation `tform`

Default:

x

Coordinates in X dimension of points to be transformed, specified as a array.

y

Coordinates in Y dimension of points to be transformed, specified as a array.

z

Coordinates in Z dimension of points to be transformed, specified as a array.

x

Coordinates of points to be transformed, specified as an n -by-2 matrix

Output Arguments

u

Transformed coordinates in X dimension, returned as an array.

v

Transformed coordinates in Y dimension, returned as an array.

w

Transformed coordinates in Z dimension, returned as an array.

U

Transformed coordinates, returned as an n -by-2 matrix

Examples

Apply Inverse Geometric Transformation

Create an `affine3d` object that defines a different scale factor in each dimension.

```
Sx = 1.2;  
Sy = 1.6;  
Sz = 2.4;  
tform = affine3d([Sx 0 0 0; 0 Sy 0 0; 0 0 Sz 0; 0 0 0 1]);
```

```
tform =
```

```
    affine3d with properties:
```

```
T: [4x4 double]
Dimensionality: 3
```

Apply forward geometric transformation to an input points.

```
[X,Y,Z] = transformPointsForward(tform,5,10,3)
```

```
X =
```

```
6
```

```
Y =
```

```
16
```

```
Z =
```

```
7.2000
```

Apply inverse geometric transformation to output (X,Y,Z) point from the previous step to recover the original coordinates.

```
[U,V,W] = transformPointsInverse(tform,X,Y,Z)
```

```
U =
```

```
5
```

```
V =
```

```
10
```

```
W =
```

```
3
```

See Also

isRigid

Class: affine3d

Determine if transformation is rigid transformation

Syntax

```
TF = isRigid(tform)
```

Description

`TF = isRigid(tform)` determines whether or not the affine transformation specified by `tform` is a rigid transformation. `TF` is a scalar boolean that is true when `tform` is a rigid transformation. A `tform` is a rigid transformation when `tform.T` defines only rotation and translation.

Input Arguments

tform

Geometric transformation, specified as an `affine3d` geometric transformation object.

Output Arguments

TF

Scalar boolean, returned as `true` when the `tform` is a rigid transformation.

Examples

Check if transformation is rigid

Create an `affine3d` object that defines a different scale factor in each dimension.


```
Sx = 1.2;  
Sy = 1.6;  
Sz = 2.4;  
tform = affine3d([Sx 0 0 0; 0 Sy 0 0; 0 0 Sz 0; 0 0 0 1])
```

```
tform =
```

```
    affine3d with properties:
```

```
                T: [4x4 double]  
    Dimensionality: 3
```

Check if the transformation is rigid.

```
TF = isRigid(tform)
```

```
TF =
```

```
    0
```

See Also

isTranslation

Class: affine3d

Determine if transformation is pure translation

Syntax

```
TF = isTranslation(tform)
```

Description

`TF = isTranslation(tform)` determines whether affine transformation specified in `tform` is a pure translation transformation. `TF` is a scalar boolean that is true when `tform` defines only translation.

Input Arguments

tform

Geometric transformation, specified as an `affine3d` geometric transformation object.

Output Arguments

TF

Scalar boolean, returned as `true` when the `tform` is a translation.

Examples

Check if transformation is translation

Create an `affine3d` object that defines a different scale factor in each dimension.

```
Sx = 1.2;  
Sy = 1.6;  
Sz = 2.4;  
tform = affine3d([Sx 0 0 0; 0 Sy 0 0; 0 0 Sz 0; 0 0 0 1]);
```

```
tf =
```

```
    affine3d with properties:
```

```
        T: [4x4 double]  
    Dimensionality: 3
```

Check if transformation is a translation.

```
tf = isTranslation(tform)
```

```
tf =
```

```
    0
```

See Also

isSimilarity

Class: affine3d

Determine if transformation is similarity transformation

Syntax

```
TF = isSimilarity(tform)
```

Description

`TF = isSimilarity(tform)` determines whether or not the affine transformation specified by `tform` is a similarity transformation. `TF` is a scalar boolean that is true when `tform` is a similarity transformation. A `tform` is a similarity transformation when it defines only homogeneous scale, rotation, and translation.

Input Arguments

tform

Geometric transformation, specified as an `affine3d` geometric transformation object.

Output Arguments

TF

Scalar boolean, returned as `true` when the `tform` is a similarity transformation.

Examples

Check if transformation is a similarity transformation

Create an `affine3d` object that defines a different scale factor in each dimension.

```
Sx = 1.2;  
Sy = 1.6;  
Sz = 2.4;  
tform = affine3d([Sx 0 0 0; 0 Sy 0 0; 0 0 Sz 0; 0 0 0 1])
```

```
tform =
```

```
    affine3d with properties:
```

```
                T: [4x4 double]  
    Dimensionality: 3
```

Check if the transformation is a similarity transformation.

```
TF = isSimilarity(tform)
```

```
TF =
```

```
    0
```

See Also

analyze75info

Read metadata from header file of Analyze 7.5 data set

Syntax

```
info = analyze75info(filename)
info = analyze75info( ____,Name,Value)
```

Description

`info = analyze75info(filename)` reads the header file of the Analyze 7.5 data set specified by the character vector `filename`. The function returns `info`, a structure whose fields contain information about the data set. Analyze 7.5 is a 3-D biomedical image visualization and analysis product developed by the Biomedical Imaging Resource of the Mayo Clinic. An Analyze 7.5 data set is made of two files, a header file and an image file. The files have the same name with different file extensions. The header file has the file extension `.hdr` and the image file has the file extension `.img`.

`info = analyze75info(____,Name,Value)` reads the Analyze 7.5 header file using name-value pairs to control different aspects of the operation.

Examples

Get Information about an Analyze 7.5 Data Set

Get information about an Analyze 7.5 data set. An Analyze 7.5 data set is made up of two files: a header file with the file extension `.hdr` and an image file with the file extension `.img`. You don't need to specify a file extension when calling `analyze75info`.

```
info = analyze75info('brainMRI');
```

Get information about an Analyze 7.5 data set, this time specifying the byte ordering of the data set. If you specify the wrong byte order, `analyze75info` attempts to read the file with the other supported byte order.

```
info = analyze75info('brainMRI', 'ByteOrder', 'ieee-le');
```

Input Arguments

filename — Name of Analyze 7.5 data set

character vector

Name of Analyze 7.5 data set, specified as a character vector. You don't need to specify a file extension.

Example: `info = analyze75info('brainMRI');`

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example: `info = analyze75info('brainMRI','ByteOrder','ieee-le');`

'ByteOrder' — Endianness of the data

character vector

Endianness of the data, specified as one of the character vectors in the following table. If the specified value results in a read error, `analyze75info` issues a warning message and attempts to read the header file with the opposite **ByteOrder** format.

Value	Meaning
'ieee-le'	Byte ordering is Little Endian
'ieee-be'	Byte ordering is Big Endian

Data Types: char

Output Arguments

info — Information about Analyze 7.5 data set

structure

Information about Analyze 7.5 data set, returned as a structure.

See Also

See Also

`analyze75read`

Introduced before R2006a

analyze75read

Read image data from image file of Analyze 7.5 data set

Syntax

```
X = analyze75read(filename)
X = analyze75read(info)
```

Description

`X = analyze75read(filename)` reads the image data from the image file of an Analyze 7.5 format data set specified by the character vector `filename`. The function returns the image data in `X`.

Analyze 7.5 is a 3-D biomedical image visualization and analysis product developed by the Biomedical Imaging Resource of the Mayo Clinic. An Analyze 7.5 data set is made of two files, a header file and an image file. The files have the same name with different file extensions. The header file has the file extension `.hdr` and the image file has the file extension `.img`.

Note By default, `analyze75read` returns image data in radiological orientation (LAS). For more information, see “Read Image Data from Analyze 7.5 File” on page 1-99.

`X = analyze75read(info)` reads the image data from the image file specified in the metadata structure `info`. `info` must be a valid metadata structure returned by the `analyze75info` function.

Examples

Read Image Data from Analyze 7.5 File

Read image data from an Analyze 7.5 file.

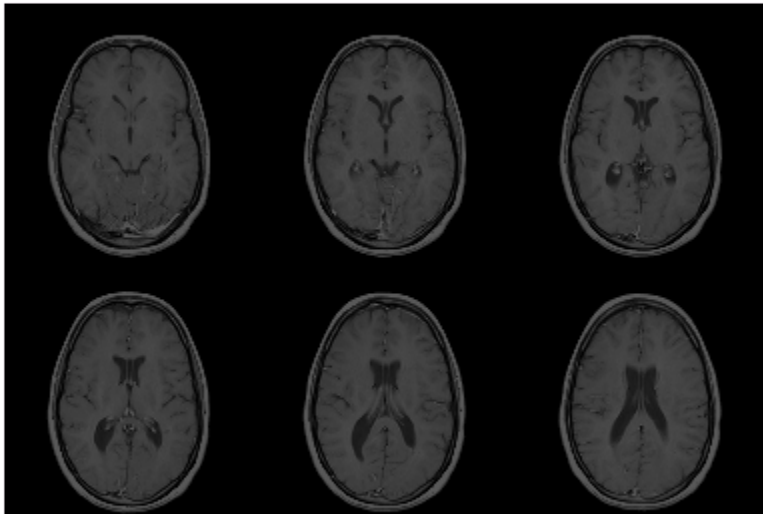
```
X = analyze75read('brainMRI');
```

View the data. First, because Analyze 7.5 format uses radiological orientation (LAS), flip the data for correct image display in MATLAB.

```
X = flip(X);
```

Then, reshape the data to create an array that can be displayed using `montage`. Select frames 12 to 17.

```
Y = reshape(X(:,:,12:17),[size(X,1) size(X,2) 1 6]);  
montage(Y);
```



Read Image Data Using the Info Structure

Read image data from an Analyze 7.5 data set, using the structure returned by `analyze75info` to specify the data set. First, use `analyze75info` to create the info structure.

```
info = analyze75info('brainMRI');
```

Call `analyze75read` to read image data from the data set, specifying the `info` structure returned by `analyze75info`.

```
X = analyze75read(info);
```

Input Arguments

filename — Name of Analyze 7.5 data set

character vector

Name of Analyze 7.5 data set, specified as a character vector. You don't need to specify a file extension.

```
Example: info = analyze75info('brainMRI');
```

Data Types: `char`

info — Information about Analyze 7.5 data set

structure

Information about the Analyze 7.5 data set, specified as a structure returned by the `analyze75info` function.

Data Types: `struct`

Output Arguments

X — Image data from Analyze 7.5 data set

array

Image data from Analyze 7.5 data set, returned as an array. `X` can be `logical`, `uint8`, `int16`, `int32`, `single`, or `double`. `analyze75read` uses a data type for `X` that is consistent with the data type specified in the data set header file. Complex and RGB data types are not supported. For single-frame, grayscale images, `X` is an m -by- n array.

See Also

See Also

`analyze75info`

Introduced before R2006a

applycform

Apply device-independent color space transformation

Syntax

```
B = applycform(A,C)
```

Description

`B = applycform(A,C)` converts the color values in `A` to the color space specified in the color transformation structure `C`. The color transformation structure specifies various parameters of the transformation. See `makecform` for details.

If `A` is two-dimensional, each row is interpreted as a color unless the color transformation structure contains a grayscale ICC profile. (See Note for this case.) `A` can have one or more columns, depending on the input color space. `B` has the same number of rows and one or more columns, depending on the output color space. (The ICC spec currently supports up to 15-channel device spaces.)

If `A` is three-dimensional, each row-column location is interpreted as a color, and `size(A,3)` is typically 1 or more, depending on the input color space. `B` has the same number of rows and columns as `A`, and `size(B,3)` is 1 or more, depending on the output color space.

Class Support

`A` is a real, nonsparse array of class `uint8`, `uint16`, or `double` or a character vector. `A` is only a character vector if `C` was created with the following syntax:

```
C = makecform('named', profile, space)
```

The output array `B` has the same class as `A`, unless the output space is `XYZ`. If the input is `XYZ` data of class `uint8`, the output is of class `uint16`, because there is no standard 8-bit encoding defined for `XYZ` color values.

Note: If the color transformation structure **C** contains a grayscale ICC profile, `applycform` interprets each pixel in **A** as a color. **A** can have any number of columns. **B** has the same size as **A**.

Examples

Convert sRGB to L*a*b* Color Space using Applycform

Read color image that uses the sRGB color space into the workspace.

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

Create a color transformation structure that defines an sRGB to L*a*b* conversion.

```
C = makecform('srgb2lab');
```

Perform the transformation with `applycform`.

```
lab = applycform(rgb,C);
```

See Also

See Also

`lab2double` | `lab2uint16` | `lab2uint8` | `makecform` | `whitepoint` | `xyz2double`
| `xyz2uint16`

Topics

“Understanding Color Spaces and Color Space Conversion”

Introduced before R2006a

applylut

Neighborhood operations on binary images using lookup tables

Compatibility

`applylut` is not recommended. Use `bwlookup` instead.

Syntax

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

Description

`A = applylut(BW,LUT)` performs a 2-by-2 or 3-by-3 neighborhood operation on binary image `BW` by using a lookup table (LUT). LUT is either a 16-element or 512-element vector returned by `makelut`. The vector consists of the output values for all possible 2-by-2 or 3-by-3 neighborhoods.

Class Support

`BW` can be numeric or logical, and it must be real, two-dimensional, and nonsparse. `LUT` can be numeric or logical, and it must be a real vector with 16 or 512 elements. If all the elements of `LUT` are 0 or 1, then `A` is logical. If all the elements of `LUT` are integers between 0 and 255, then `A` is `uint8`. For all other cases, `A` is `double`.

Examples

Perform Erosion Using a 2-by-2 Neighborhood

Create the LUT.

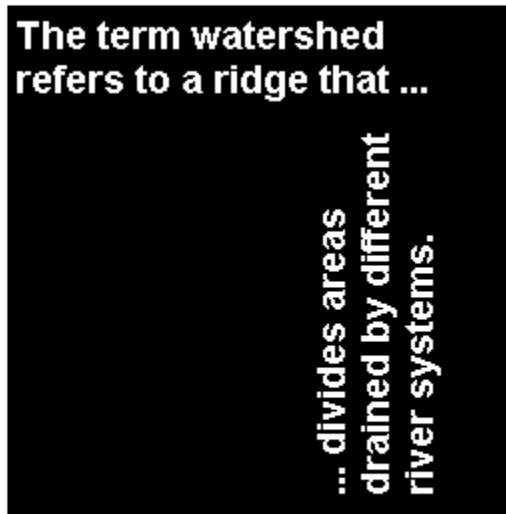
```
lutfun = @(x)(sum(x(:))==4);  
lut    = makelut(lutfun,2);
```

Read image into the workspace and then apply the LUT to the image. An output pixel is on only if all four of the input pixel's neighborhood pixels are on .

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

Show the original image and the eroded image.

```
figure, imshow(BW1);
```



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


The term watershed
refers to a ridge that ...

... divides areas
drained by different
river systems.

Algorithms

`applylut` performs a neighborhood operation on a binary image by producing a matrix of indices into `lut`, and then replacing the indices with the actual values in `lut`. The specific algorithm used depends on whether you use 2-by-2 or 3-by-3 neighborhoods.

2-by-2 Neighborhoods

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

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

```
8     2
4     1
```

The resulting convolution contains integer values in the range [0,15]. `applylut` uses the central part of the convolution, of the same size as `BW`, and adds 1 to each value to shift the range to [1,16]. It then constructs `A` by replacing the values in the cells of the index matrix with the values in `lut` that the indices point to.

3-by-3 Neighborhoods

For 3-by-3 neighborhoods, `length(lut)` is 512. There are nine pixels in each neighborhood, and two possible states for each pixel, so the total number of permutations is $2^9 = 512$.

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

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

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

See Also

`makelut`

Introduced before R2006a

axes2pix

Convert axes coordinates to pixel coordinates

Syntax

```
pixelx = axes2pix(dim, XDATA, AXESX)
```

Description

`pixelx = axes2pix(dim, XDATA, AXESX)` converts an axes coordinate into an intrinsic (“pixel”) coordinate. For example, if `pt = get(gca, 'CurrentPoint')` then `AXESX` could be `pt(1,1)` or `pt(1,2)`. `AXESX` must be in intrinsic coordinates. `XDATA` is a two-element vector returned by `get(image_handle, 'XData')` or `get(image_handle, 'YData')`. `dim` is the number of image columns for the x coordinate, or the number of image rows for the y coordinate.

Class Support

`dim`, `XDATA`, and `AXESX` can be `double`. The output is `double`.

Note

`axes2pix` performs minimal checking on the validity of `AXESX`, `DIM`, or `XDATA`. For example, `axes2pix` returns a negative coordinate if `AXESX` is less than `XDATA(1)`. The function calling `axes2pix` bears responsibility for error checking.

Examples

Convert Axes Coordinate into Intrinsic Coordinate

Display image.

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



Get the size of the image.

```
[nrows,ncols] = size(get(h,'CData'));
```

Get the image XData and YData.

```
xdata = get(h,'XData')
```

```
xdata =
```

```
1 240
```

```
ydata = get(h,'YData')
```

```
ydata =
```

```
1 291
```

Convert an axes coordinate into an intrinsic coordinate for the x and y dimensions.

```
px = axes2pix(ncols,xdata,30)
```

```
px = 30
```

```
py = axes2pix(nrows,ydata,30)
```

```
py = 30
```

Convert Axes Coordinate to Intrinsic Coordinate with Nondefault XData and YData

Display image.

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



Get the size of the image.

```
[nrows,ncols] = size(get(h,'CData'));
```

Specify non-default XData and YData.

```
xdata = [10 100]
```

```
xdata =
```

```
    10    100
```

```
ydata = [20 90]
```

```
ydata =
```

```
    20    90
```

Convert the axes coordinate to an intrinsic (pixel) coordinate.

```
px = axes2pix(ncols,xdata,30)
```

```
px = 54.1111
```

```
py = axes2pix(nrows,ydata,30)
```

```
py = 42.4286
```

See Also

See Also

[bwselect](#) | [imfill](#) | [impixel](#) | [impixelinfo](#) | [improfile](#) | [roipoly](#)

Introduced before R2006a

bestblk

Determine optimal block size for block processing

Syntax

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

Description

`siz = bestblk([m n],k)` returns, for an m -by- n image, the optimal block size for block processing. The optimal block size is the size required along the outer partial blocks. k is a scalar specifying the maximum row and column dimensions for the block. If you omit this argument, the default is 100. The return value `siz` is a 1-by-2 vector containing the row and column dimensions for the block.

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

Examples

Determine Optimal Block Size

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

```
siz =
```

```
    64    50
```

Algorithms

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

- If m is less than or equal to k , return m .
- If m is greater than k , consider all values between $\min(m/10, k/2)$ and k . Return the value that minimizes the padding required.

The same algorithm is then repeated for n .

See Also

`blockproc`

Introduced before R2006a

blockproc

Distinct block processing for image

Syntax

```
B = blockproc(A, blockSize, fun)
B = blockproc(src_filename, blockSize, fun)
B = blockproc(adapter, blockSize, fun)
blockproc( ___, Name, Value, ...)
```

Description

`B = blockproc(A, blockSize, fun)` processes the image `A` by applying the function `fun` to each distinct block of `A` and concatenating the results into `B`, the output matrix. `blockSize` is a two-element vector, `[rows cols]`, that specifies the size of the block. `fun` is a handle to a function that accepts a *block struct* as input and returns a matrix, vector, or scalar `Y`. For example, `Y = fun(block_struct)`. (For more information about a *block struct*, see “Definitions” on page 1-128.) For each block of data in the input image, `A`, `blockproc` passes the block in a *block struct* to the user function, `fun`, to produce `Y`, the corresponding block in the output image. If `Y` is empty, `blockproc` does not generate any output and returns empty after processing all blocks. Choosing an appropriate block size can significantly improve performance. For more information, see “Choosing Block Size”. For more information about function handles, see “Create Function Handle” (MATLAB).

`B = blockproc(src_filename, blockSize, fun)` processes the image `src_filename`, reading and processing one block at a time. This syntax is useful for processing large images since only one block of the image is read into memory at a time. If the output matrix `B` is too large to fit into memory, omit the output argument and instead use the 'Destination' parameter/value pair to write the output to a file.

`B = blockproc(adapter, blockSize, fun)` processes the source image specified by `adapter`, an `ImageAdapter` object. An `ImageAdapter` is a user-defined class that provides `blockproc` with a common API for reading and writing to a particular image file format. For more information, see “Read and Write Data in Unsupported Formats Using `blockproc`”.

`blockproc(____, Name, Value, ...)` processes the input image, specifying parameters and corresponding values that control various aspects of the block behavior. Parameter names are not case-sensitive.

Input Arguments

A

Input image.

blockSize

Size of the block, specified as a two-element vector, `[rows cols]`.

fun

Function handle to a function that accepts a *block struct* as input and returns a matrix, vector, or scalar.

src_filename

Input image.

adapter

A user-defined class that provides `blockproc` with a common API for reading and writing to a particular image file format.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

'BorderSize'

Number of border pixels to add to each block, specified as a two-element vector, `[V H]`. The function adds `V` rows above and below each block and `H` columns left and right of each block. The size of each resulting block is:

[M + 2*V, N + 2*H]

By default, the function automatically removes the border from the result of `fun`. See the `'TrimBorder'` parameter for more information. The function pads blocks with borders extending beyond the image edges with zeros.

Default: [0 0] (no border)

'Destination'

Destination for the output, specified as a character vector or an `ImageAdapter` object. The character vector is the name of a TIFF file which must include the `'.tif'` file extension. If a file with this name exists, it is overwritten. `ImageAdapters` provide an interface for reading and writing to arbitrary image file formats.

The `'Destination'` parameter is useful when you expect your output to be too large to fit into memory. It provides a workflow for file-to-file image processing for arbitrarily large images. When you specify the `'Destination'` parameter, `blockproc` does not return the processed image as an output argument, but instead writes the output to the `'Destination'`. (You cannot request an output argument when the `'Destination'` parameter is specified.)

'PadPartialBlocks'

Pad partial blocks to make them full-sized, specified as the logical scalar `true` or `false`. When set to `true`, `blockproc` pads partial blocks to make them full-sized (M-by-N) blocks. Partial blocks arise when the image size is not exactly divisible by the block size. If they exist, partial blocks lie along the right and bottom edge of the image. The default is `false`, meaning that the function does not pad the partial blocks, but processes them as-is. `blockproc` uses zeros to pad partial blocks when necessary.

Default: `false`

'PadMethod'

Method used to pad the image boundary, specified as any of these values. `'PadMethod'` determines how `blockproc` will pad the image boundary. Options are:

Value	Description
<code>'replicate'</code>	Repeat border elements.

Value	Description
'symmetric'	Pad image with mirror reflections of itself.
X	Pad image with the scalar value X. By default X == 0.

'TrimBorder'

Remove border pixels from the output of the user function, specified as the logical scalar `true` or `false`. When set to `true`, the `blockproc` function trims off border pixels from the output of the user function, `fun`. The function removes `V` rows from the top and bottom of the output of `fun`, and `H` columns from the left and right edges. The `'BorderSize'` parameter defines `V` and `H`. The default is `true`, meaning that the `blockproc` function automatically removes borders from the `fun` output.

Default: `true`

'UseParallel'

Enable parallel mode, specified as the logical scalar `true` or `false`. This mode of image processing requires the Parallel Computing Toolbox™. When set to `true`, `blockproc` attempts to run in parallel mode, distributing the processing across multiple workers (MATLAB sessions) in an open MATLAB pool. In parallel mode, the input image cannot be an `ImageAdapter` on page 1-632 object. See `parpool` for information on configuring your parallel environment.

Default: `false`

'DisplayWaitbar'

Display wait bar, specified as the logical scalar `true` or `false`. When set to `true`, `blockproc` displays a waitbar to indicate progress for long-running operations. To prevent `blockproc` from displaying a waitbar, set `DisplayWaitbar` to `false`.

Default: `true`

File Format Support: Input and output files for `blockproc` (as specified by `src_filename` and the `'Destination'` parameter) must have one of the following file types and must be named with one of the listed file extensions:

- Read/Write File Formats: TIFF (*.tif, *.tiff), JPEG2000 (*.jp2, *.j2c, *.j2k)

- Read-Only File Formats: JPEG2000 (*.jpf, *.jpx)

Output Arguments

B

Output matrix.

Examples

Create Thumbnail of Image

Read image into the workspace.

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

Create block processing function.

```
fun = @(block_struct) imresize(block_struct.data,0.15);
```

Process the image, block-by-block.

```
I2 = blockproc(I,[100 100],fun);
```

Display the original image and the processed image.

```
figure;  
imshow(I);
```



```
figure;  
imshow(I2);
```



Set Pixels in 32-by-32 blocks to Standard Deviation

Create block processing function.

```
fun = @(block_struct) ...  
    std2(block_struct.data) * ones(size(block_struct.data));
```

Perform the block processing operation, specifying the input image by filename.

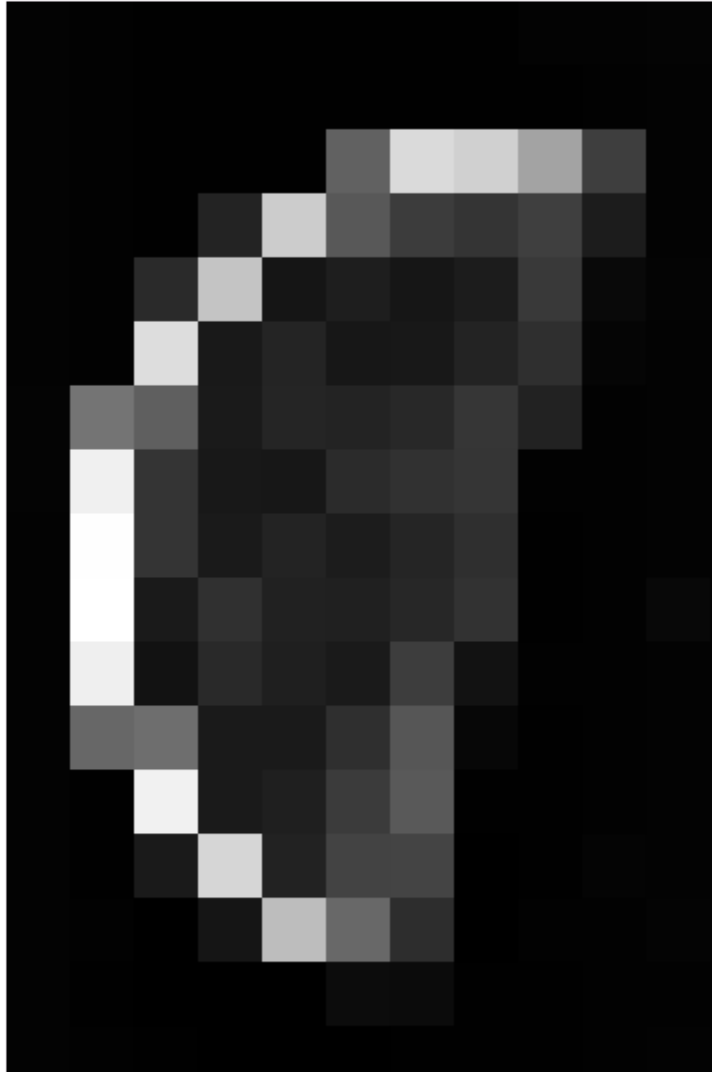
```
I2 = blockproc('moon.tif',[32 32],fun);
```

Display the original image and the processed version.

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




```
figure;  
imshow(I2,[]);
```



Switch Red and Green Bands of RGB Image

Read image into the workspace.

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

Create block processing function.

```
fun = @(block_struct) block_struct.data(:,:, [2 1 3]);
```

Perform the block processing operation.

```
blockproc(I, [200 200], fun, 'Destination', 'grb_peppers.tif');
```

Display original image and the processed image.

```
figure;  
imshow('peppers.png');
```



```
figure;  
imshow('grb_peppers.tif');
```



Convert Large TIFF Image into JPEG2000 Image

Note: To run this example, you must replace 'largeImage.tif' with the name of your file.

Create block processing function.

```
fun = @(block_struct) block_struct.data;
```

Convert a TIFF image into a new JPEG2000 image. Replace 'largeImage.tif' with the name of an actual image file.

```
blockproc('largeImage.tif',[1024 1024],fun,'Destination','New.jp2');
```

Definitions

Block Struct

A *block struct* is a MATLAB structure that contains the block data and other information about the block. Fields in the *block struct* are:

- `block_struct.border`: A two-element vector, [V H], that specifies the size of the vertical and horizontal padding around the block of data. (See the 'BorderSize' parameter in the Inputs section.)
- `block_struct.blockSize`: A two-element vector, [rows cols], that specifies the size of the block data. If a border has been specified, the size does not include the border pixels.
- `block_struct.data`: M-by-N or M-by-N-by-P matrix of block data
- `block_struct.imageSize`: A two-element vector, [rows cols], that specifies the full size of the input image.
- `block_struct.location`: A two-element vector, [row col], that specifies the position of the first pixel (minimum-row, minimum-column) of the block data in the input image. If a border has been specified, the location refers to the first pixel of the discrete block data, not the added border pixels.

See Also

See Also

[ImageAdapter](#) | [colfilt](#) | [nlfilter](#)

Topics

“Anonymous Functions” (MATLAB)

“Parameterizing Functions” (MATLAB)

“Distinct Block Processing”

Introduced in R2009b

boundarymask

Find region boundaries of segmentation

Syntax

```
mask = boundarymask(L)
mask = boundarymask(BW)
mask = boundarymask( ____, conn)
```

Description

`mask = boundarymask(L)` computes a mask that represents the region boundaries for the input label matrix `L`. The output, `mask`, is a logical image that is `true` at boundary locations and `false` at non-boundary locations.

`mask = boundarymask(BW)` computes the region boundaries for the input binary image `BW`.

`mask = boundarymask(____, conn)` computes the region boundaries using a connectivity specified by `conn`.

Examples

Create Rasterized Grid of Region Boundaries

Read image into the workspace.

```
A = imread('kobi.png');
```

Create a superpixel representation of the image, returned as a label matrix.

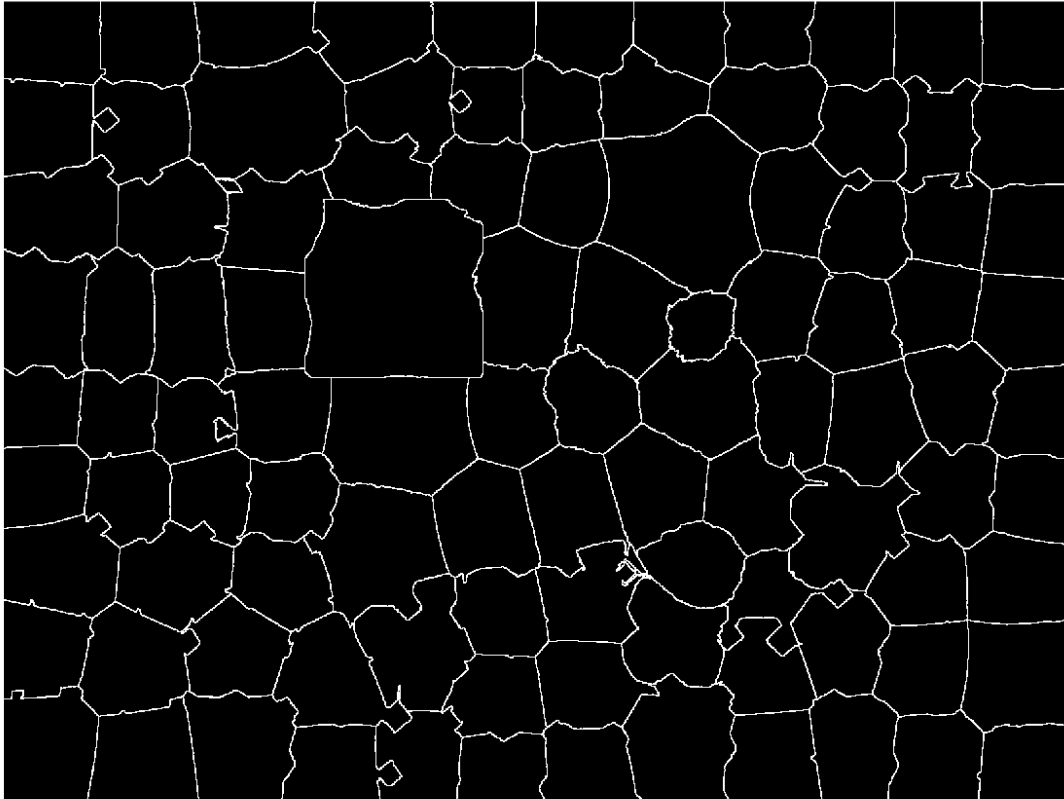
```
L = superpixels(A,100);
```

Create the rasterized grid of the regions in the label matrix.

```
mask = boundarymask(L);
```

Display the boundary mask binary image.

```
figure  
imshow(mask, 'InitialMagnification',67)
```



Input Arguments

L — Input label matrix

finite, nonnegative, nonsparse, 2-D numeric or logical array

Input label matrix, specified as a finite, nonnegative, nonsparse, 2-D numeric or logical array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

BW — Input binary image

numeric or logical array

Input binary image, specified as a numeric or logical array the same size as `L`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `logical`

conn — Connectivity

8 (default) | 4

Connectivity, specified as the numeric scalar `4` or `8`. For a given pixel P in the input image, the corresponding output `mask(P)` is `true` if any of the pixels in the 4-connected or 8-connected neighborhood of P have a value different than P .

Value	Meaning
4	4-connected neighborhood
8	8-connected neighborhood

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output Arguments

mask — Rasterized grid of region boundaries

2-D logical matrix

Rasterized grid of region boundaries, specified as a 2-D logical matrix the same size as the input image.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the input argument `conn` must be a compile-time constant.

See Also

See Also

`imoverlay` | `label2idx` | `superpixels`

Introduced in R2016a

bwarea

Area of objects in binary image

Syntax

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

Description

`total = bwarea(BW)` estimates the area of the objects in binary image `BW`. `total` is a scalar whose value corresponds roughly to the total number of `on` pixels in the image, but might not be exactly the same because different patterns of pixels are weighted differently.

Class Support

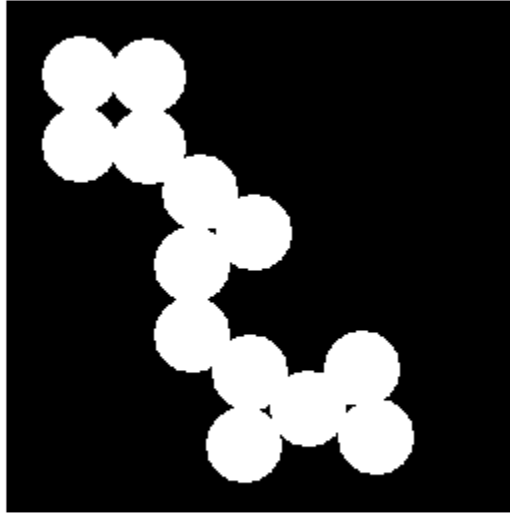
`BW` can be numeric or logical. For numeric input, any nonzero pixels are considered to be `on`. The return value `total` is of class `double`.

Examples

Calculate the Area of Objects in a Binary Image

Read the image and display it.

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



Calculate the area of objects in the image.

```
bwarea(BW)
```

```
ans = 1.4187e+04
```

Algorithms

`bwarea` estimates the area of all of the `on` pixels in an image by summing the areas of each pixel in the image. The area of an individual pixel is determined by looking at its 2-by-2 neighborhood. There are six different patterns, each representing a different area:

- Patterns with zero `on` pixels (area = 0)
- Patterns with one `on` pixel (area = 1/4)
- Patterns with two adjacent `on` pixels (area = 1/2)

- Patterns with two diagonal `on` pixels (area = $3/4$)
- Patterns with three `on` pixels (area = $7/8$)
- Patterns with all four `on` pixels (area = 1)

Keep in mind that each pixel is part of four different 2-by-2 neighborhoods. This means, for example, that a single `on` pixel surrounded by `off` pixels has a total area of 1.

References

- [1] Pratt, William K., *Digital Image Processing*, New York, John Wiley & Sons, Inc., 1991, p. 634.

See Also

See Also

bweuler | bwperim

Introduced before R2006a

bwareafilt

Extract objects from binary image by size

Syntax

```
BW2 = bwareafilt(BW,range)
BW2 = bwareafilt(BW,n)
BW2 = bwareafilt(BW,n,keep)
```

Description

`BW2 = bwareafilt(BW,range)` extracts all connected components (objects) from the binary image `BW`, where the area of the objects is in `range`, producing another binary image `BW2`. `range` is a 2-by-1 vector of minimum and maximum sizes (inclusive). `bwareafilt` removes objects that do not meet the criterion. The default connectivity is 8.

`BW2 = bwareafilt(BW,n)` keeps the `n` largest objects. In the event of a tie for `n`-th place, only the first `n` objects are included in `BW2`.

`BW2 = bwareafilt(BW,n,keep)` retains the `n` largest objects if `keep` is 'largest' (the default) or the `n` smallest if `keep` is 'smallest'.

Examples

Filter Binary Image by Area of Objects

Read image.

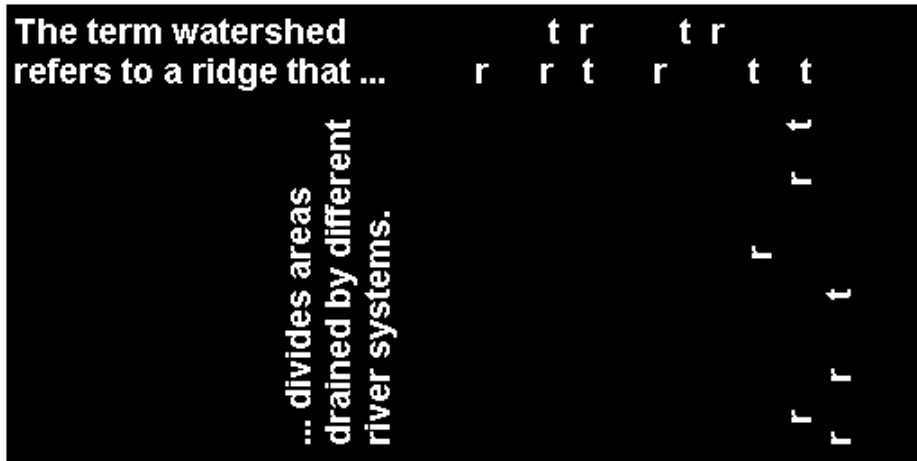
```
BW = imread('text.png');
```

Filter image, retaining only those objects with areas between 40 and 50.

```
BW2 = bwareafilt(BW,[40 50]);
```

Display the original image and filtered image side by side.

```
imshowpair(BW,BW2, 'montage')
```



Filter Binary Image by Size of Objects

Read image.

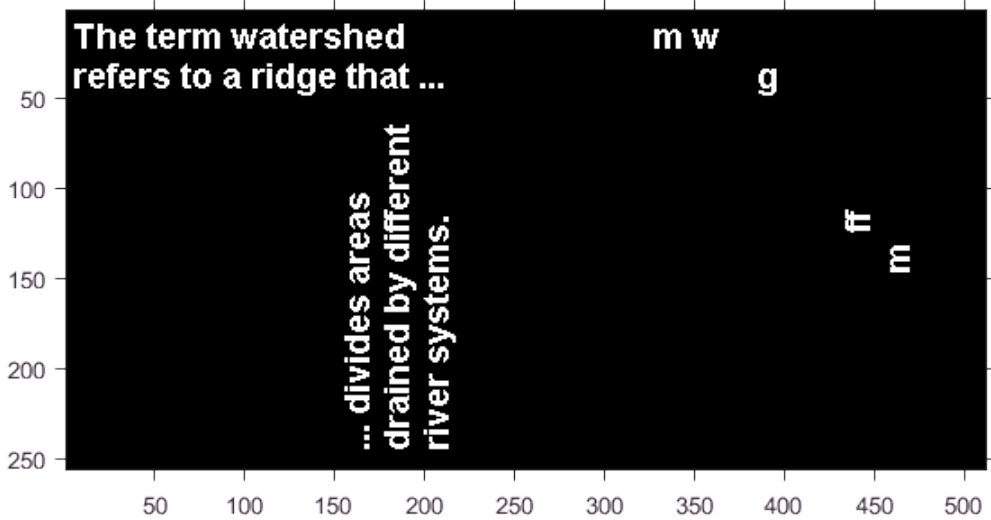
```
BW = imread('text.png');
```

Filter image, retaining only the 5 objects with the largest areas.

```
BW2 = bwareafilt(BW,5);
```

Display the original image and the filtered image side by side.

```
imshowpair(BW,BW2, 'montage')
```



- “Filter Images on Region Properties Using Image Region Analyzer App”

Input Arguments

BW — Image to be filtered

binary image

Image to be filtered, specified as a binary image.

Data Types: `logical`

range — Minimum and maximum values of property inclusive

2-by-1 numeric vector

Minimum and maximum values of the property inclusive, specified as a 2-by-1 numeric vector of the form `[low high]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

n — Number of objects to include when filtering image objects by size

scalar double

Number of objects to include when filtering image objects by size, specified as a scalar double.

Data Types: double

keep — Size of objects to include in the output image

'largest' (default) | 'smallest'

Size of objects to include in the output image, specified as 'largest' or 'smallest'. In the event of a tie for n-th place, `bwareafilt` includes only the first n objects.

Data Types: char

Output Arguments

BW2 — Filtered image

binary image

Filtered image, returned as a binary image, the same size and class as the input image BW.

See Also

See Also

bwpropfilt | conndef | regionprops

Topics

“Filter Images on Region Properties Using Image Region Analyzer App”

Introduced in R2014b

bwareaopen

Remove small objects from binary image

Syntax

```
BW2 = bwareaopen(BW,P)  
BW2 = bwareaopen(BW,P,conn)
```

Description

`BW2 = bwareaopen(BW,P)` removes all connected components (objects) that have fewer than `P` pixels from the binary image `BW`, producing another binary image, `BW2`. The default connectivity is 8 for two dimensions, 26 for three dimensions, and `conndef(ndims(BW), 'maximal')` for higher dimensions. This operation is known as an *area opening*.

`BW2 = bwareaopen(BW,P,conn)` removes all connected components, where `conn` specifies the desired connectivity.

Examples

Remove Objects in Image Containing Fewer Than 50 Pixels

Read binary image.

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

Remove objects containing fewer than 50 pixels using `bwareaopen` function.

```
BW2 = bwareaopen(BW, 50);
```

Display original image next to morphologically opened image.

```
imshowpair(BW,BW2,'montage')
```



Input Arguments

BW – Binary image

real, nonsparse, logical or numeric array of any dimension

Binary image, specified as a nonsparse, logical or numeric array of any dimension.

Example: `BW2 = bwareaopen(BW, 50);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

P – Maximum number of pixels in objects

nonnegative, integer-valued, numeric scalar

Maximum number of pixels in objects, specified as a nonnegative, integer-valued, numeric scalar.

Example: `BW2 = bwareaopen(BW, 50);`

Data Types: `double`

conn — Connectivity

8 (2-D), 26 (3-D) or calculated using `conndef(ndims(BW), 'maximal')` for higher dimensions (default) | 4 | 8 | 6 | 18 | 26 | 3-by-3-by-...-by-3 matrix of 0s and 1s.

Connectivity, specified as one of the following numeric scalars:

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can also be defined in a more general way for any dimension by specifying a 3-by-3-by-...-by-3 matrix of 0s and 1s. The 1-valued elements define neighborhood locations relative to the central element of `conn`. Note that `conn` must be symmetric about its central element.

Example: `BW2 = bwareaopen(BW, 50, 4);`

Data Types: `double` | `logical`

Output Arguments

BW2 — Image that has been morphologically opened

nonsparse, logical array of any dimension

Image that has been morphologically opened, specified as a nonsparse, logical array the same size as `BW`.

Algorithms

The basic steps are

- 1 Determine the connected components:

```
CC = bwconncomp(BW, conn);
```

- 2 Compute the area of each component:

```
S = regionprops(CC, 'Area');
```

- 3 Remove small objects:

```
L = labelmatrix(CC);  
BW2 = ismember(L, find([S.Area] >= P));
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- `BW` must be a 2-D binary image. N-D arrays are not supported.
- `conn` can only be one of the two-dimensional connectivities (4 or 8) or a 3-by-3 matrix. The 3-D connectivities (6, 18, and 26) are not supported. Matrices of size 3-by-3-by-...-by-3 are not supported.
- `conn` must be a compile-time constant.

See Also

See Also

`bwconncomp` | `conndef`

Introduced before R2006a

bwboundaries

Trace region boundaries in binary image

Syntax

```
B = bwboundaries(BW)
B = bwboundaries(BW,conn)
B = bwboundaries(BW,conn,options)
[B,L]= bwboundaries( ___ )
[B,L,N,A] = bwboundaries( ___ )
```

Description

`B = bwboundaries(BW)` traces the exterior boundaries of objects, as well as boundaries of holes inside these objects, in the binary image `BW`. `bwboundaries` also descends into the outermost objects (parents) and traces their children (objects completely enclosed by the parents). Returns `B`, a cell array of boundary pixel locations.

`B = bwboundaries(BW,conn)` traces the exterior boundaries of objects, where `conn` specifies the connectivity to use when tracing parent and child boundaries.

`B = bwboundaries(BW,conn,options)` traces the exterior boundaries of objects, where `options` is either `'holes'` or `'noholes'`, specifying whether you want to include the boundaries of holes inside other objects.

`[B,L]= bwboundaries(___)` returns a label matrix `L` where objects and holes are labeled.

`[B,L,N,A] = bwboundaries(___)` returns `N`, the number of objects found, and `A`, an adjacency matrix.

Examples

Overlay Region Boundaries on Image

Read grayscale image into the workspace.

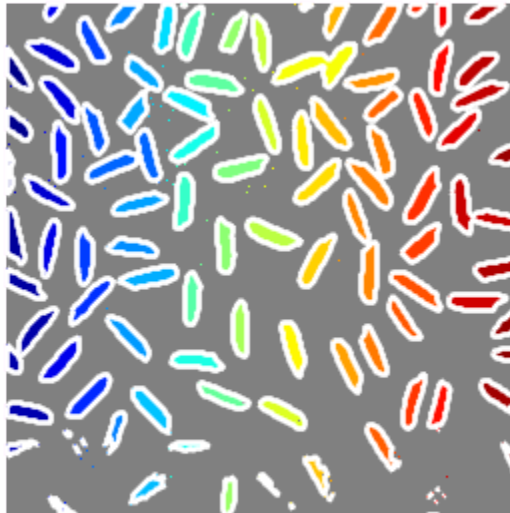
```
I = imread('rice.png');
```

Convert grayscale image to binary image using local adaptive thresholding.

```
BW = imbinarize(I);
```

Calculate boundaries of regions in image and overlay the boundaries on the image.

```
[B,L] = bwboundaries(BW,'noholes');  
imshow(label2rgb(L, @jet, [.5 .5 .5]))  
hold on  
for k = 1:length(B)  
    boundary = B{k};  
    plot(boundary(:,2), boundary(:,1), 'w', 'LineWidth', 2)  
end
```



Overlay Region Boundaries on Image and Annotate with Region Numbers

Read binary image into the workspace.

```
BW = imread('blobs.png');
```

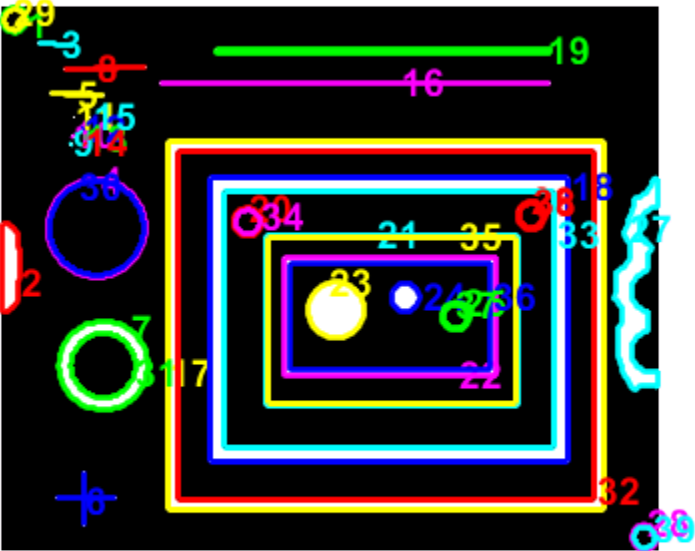
Calculate boundaries of regions in the image.

```
[B,L,N,A] = bwboundaries(BW);
```

Display the image with the boundaries overlaid. Add the region number next to every boundary (based on the label matrix). Use the zoom tool to read individual labels.

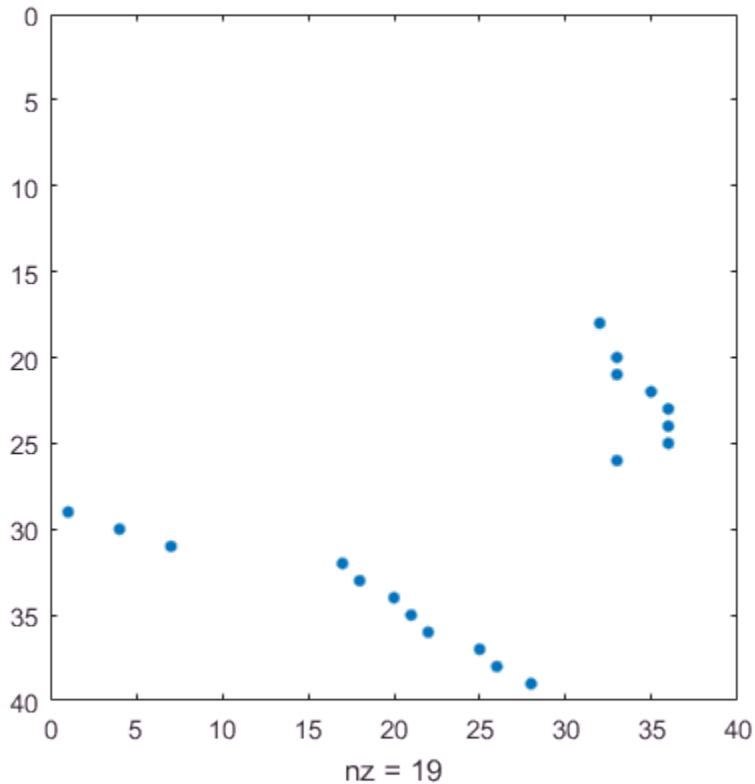
```
imshow(BW); hold on;
colors=['b' 'g' 'r' 'c' 'm' 'y'];
for k=1:length(B),
    boundary = B{k};
    cidx = mod(k,length(colors))+1;
    plot(boundary(:,2), boundary(:,1),...
         colors(cidx),'LineWidth',2);

    %randomize text position for better visibility
    rndRow = ceil(length(boundary)/(mod(rand*k,7)+1));
    col = boundary(rndRow,2); row = boundary(rndRow,1);
    h = text(col+1, row-1, num2str(L(row,col)));
    set(h,'Color',colors(cidx),'FontSize',14,'FontWeight','bold');
end
```

Display the adjacency matrix using the spy function.

```
figure  
spy(A);
```



Display Object Boundaries in Red and Hole Boundaries in Green

Read binary image into workspace.

```
BW = imread('blobs.png');
```

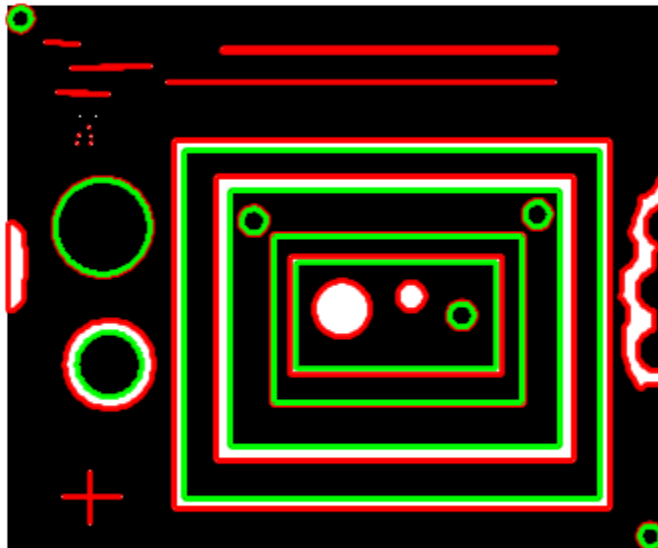
Calculate boundaries.

```
[B,L,N] = bwboundaries(BW);
```

Display object boundaries in red and hole boundaries in green.

```
imshow(BW); hold on;
```

```
for k=1:length(B),  
    boundary = B{k};  
    if(k > N)  
        plot(boundary(:,2), boundary(:,1), 'g','LineWidth',2);  
    else  
        plot(boundary(:,2), boundary(:,1), 'r','LineWidth',2);  
    end  
end
```



Display Parent Boundaries in Red and Holes in Green

Read image into workspace.

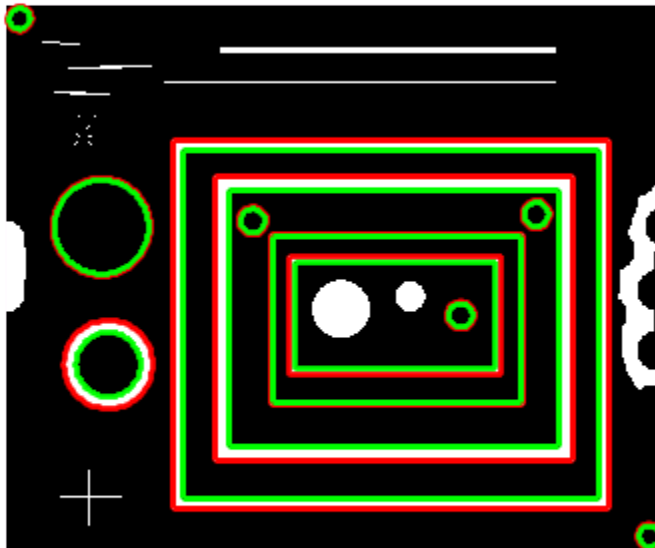
```
BW = imread('blobs.png');
```

Display parent boundaries in red and their holes in green.

```

[B,L,N,A] = bwboundaries(BW);
figure; imshow(BW); hold on;
% Loop through object boundaries
for k = 1:N
    % Boundary k is the parent of a hole if the k-th column
    % of the adjacency matrix A contains a non-zero element
    if (nnz(A(:,k)) > 0)
        boundary = B{k};
        plot(boundary(:,2),...
            boundary(:,1),'r','LineWidth',2);
        % Loop through the children of boundary k
        for l = find(A(:,k))'
            boundary = B{l};
            plot(boundary(:,2),...
                boundary(:,1),'g','LineWidth',2);
        end
    end
end
end

```

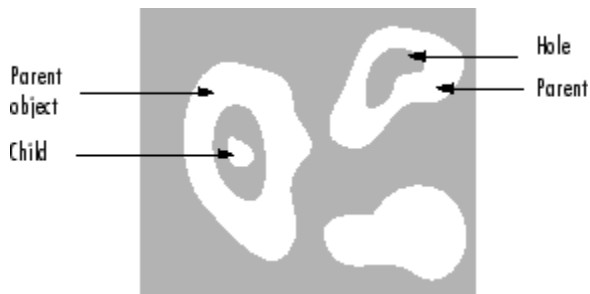


Input Arguments

BW — Input binary image

real, nonsparse, 2-D logical or numeric array

Binary input image, specified as a real, nonsparse, 2-D logical or numeric array. **BW** must be a binary image where nonzero pixels belong to an object and 0 pixels constitute the background. The following figure illustrates these components.



Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

conn — Connectivity

8 (default) | 4

Connectivity, specified as either of the following scalar values:

Value	Meaning
4	4-connected neighborhood
8	8-connected neighborhood. This is the default.

Data Types: double

options — Determine whether to search for both parent and child boundaries

'holes' (default) | 'noholes'

Determine whether to search for both parent and child boundaries, specified as either of the following:

Option	Meaning
'holes'	Search for both object and hole boundaries. This is the default.
'noholes'	Search only for object (parent and child) boundaries. This can provide better performance.

Data Types: char

Output Arguments

B — Row and column coordinates of boundary pixels

P-by-1 cell array

Row and column coordinates of boundary pixels, returned as a *P*-by-1 cell array, where *P* is the number of objects and holes. Each cell in the cell array contains a *Q*-by-2 matrix. Each row in the matrix contains the row and column coordinates of a boundary pixel. *Q* is the number of boundary pixels for the corresponding region.

L — Label matrix of contiguous regions

2-D array of nonnegative integers

Label matrix of contiguous regions, returned as a 2-D array of nonnegative integers of class `double`. The *k*th region includes all elements in **L** that have value *k*. The number of objects and holes represented by **L** is equal to $\max(\mathbf{L}(:))$. The zero-valued elements of **L** make up the background.

N — Number of objects found

numeric scalar

Number of objects found, returned as a numeric scalar of class `double`.

A — Parent-child dependencies between boundaries and holes

square, sparse, logical matrix

Parent-child dependencies between boundaries and holes, returned as a square, sparse, logical matrix of class `double` with side of length $\max(\mathbf{L}(:))$. The rows and columns of **A** correspond to the positions of boundaries stored in **B**. The first **N** cells in **B** are object

boundaries. $A(i, j) = 1$ means that object i is a child of object j . The boundaries that enclose or are enclosed by the k -th boundary can be found using A as follows:

```
enclosing_boundary = find(A(m,:));  
enclosed_boundaries = find(A(:,m));
```

Algorithms

The `bwboundaries` function implements the Moore-Neighbor tracing algorithm modified by Jacob's stopping criteria. This function is based on the `boundaries` function presented in the first edition of *Digital Image Processing Using MATLAB*, by Gonzalez, R. C., R. E. Woods, and S. L. Eddins, New Jersey, Pearson Prentice Hall, 2004.

References

- [1] Gonzalez, R. C., R. E. Woods, and S. L. Eddins, *Digital Image Processing Using MATLAB*, New Jersey, Pearson Prentice Hall, 2004.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- The parameter `conn` must be a compile-time constant.
- The parameter `options` must be a compile-time constant.
- The return value A can only be a full matrix, not a sparse matrix.

See Also

See Also

`bwlabel` | `bwlabeln` | `bwperim` | `bwtraceboundary`

Introduced before R2006a

bwconncomp

Find connected components in binary image

Syntax

```
CC = bwconncomp(BW)
CC = bwconncomp(BW,conn)
```

Description

`CC = bwconncomp(BW)` returns the connected components `CC` found in the binary image `BW`. `bwconncomp` uses a default connectivity of 8 for two dimensions, 26 for three dimensions, and `conndef(ndims(BW), 'maximal')` for higher dimensions.

`CC = bwconncomp(BW,conn)` returns the connected components where `conn` specifies the desired connectivity for the connected components.

Examples

Calculate Centroids of 3-D Objects

Create a small sample 3-D array.

```
BW = cat(3, [1 1 0; 0 0 0; 1 0 0],...
            [0 1 0; 0 0 0; 0 1 0],...
            [0 1 1; 0 0 0; 0 0 1]);
```

Find the connected components in the array.

```
CC = bwconncomp(BW)
```

```
CC = struct with fields:
    Connectivity: 26
    ImageSize: [3 3 3]
    NumObjects: 2
    PixelIdxList: {[5×1 double] [3×1 double]}
```

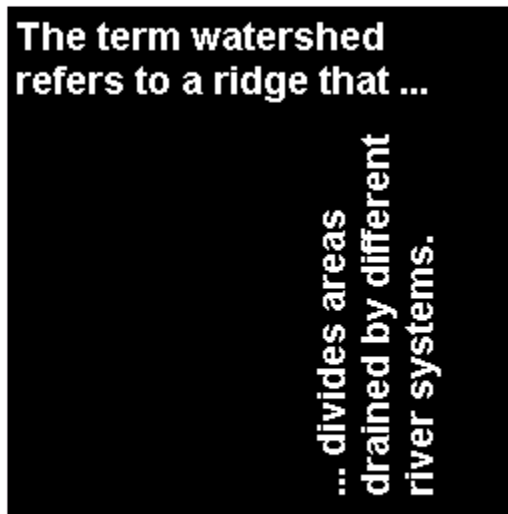
Calculate centroids of the objects in the array.

```
S = regionprops(CC, 'Centroid')  
  
S = 2×1 struct array with fields:  
Centroid
```

Erase Largest Component from Image

Read image into the workspace and display it.

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



Find the number of connected components in the image.

```
CC = bwconncomp(BW)
```

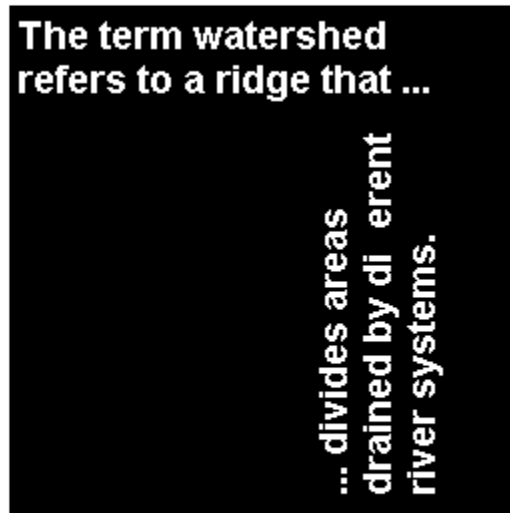
```
CC = struct with fields:
  Connectivity: 8
  ImageSize: [256 256]
  NumObjects: 88
  PixelIdxList: {1×88 cell}
```

Determine which is the largest component in the image and erase it (set all the pixels to 0).

```
numPixels = cellfun(@numel,CC.PixelIdxList);
[biggest,idx] = max(numPixels);
BW(CC.PixelIdxList{idx}) = 0;
```

Display the image, noting that the largest component happens to be the two consecutive 'f's in the word different.

```
figure
imshow(BW)
```



Input Arguments

BW — Input binary image

real, nonsparse, numeric or logical array of any dimension

Input binary image, specified as a real, nonsparse, numeric or logical array of any dimension.

Example: `BW = imread('text.png');` `CC = bwconncomp(BW);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

conn — Connectivity for the connected components

8 for two dimensions; 26 for three dimensions (default) | 4 | 8 | 6 | 18 | 26

Connectivity for the connected components, specified as one of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

To calculate the default connectivity for higher dimensions, `bwconncomp` uses `conndef(ndims(BW), 'maximal')`.

Connectivity can be defined in a more general way for any dimension using a 3-by-3-by-...-by-3 matrix of 0s and 1s. `conn` must be symmetric about its center element. The 1-valued elements define neighborhood locations relative to `conn`.

Example: `BW = imread('text.png');` `CC = bwconncomp(BW,4);`

Data Types: `double` | `logical`

Output Arguments

CC — Connected components

struct

Connected components, returned as a structure with four fields.

Field	Description
Connectivity	Connectivity of the connected components (objects)
ImageSize	Size of BW
NumObjects	Number of connected components (objects) in BW
PixelIdxList	1-by-NumObjects cell array where the k -th element in the cell array is a vector containing the linear indices of the pixels in the k -th object.

Tips

- The functions `bwlabel`, `bwlabeln`, and `bwconncomp` all compute connected components for binary images. `bwconncomp` replaces the use of `bwlabel` and `bwlabeln`. It uses significantly less memory and is sometimes faster than the other functions.

Function	Input Dimension	Output Form	Memory Use	Connectivity
<code>bwlabel</code>	2-D	Label matrix with double-precision	High	4 or 8
<code>bwlabeln</code>	N-D	Double-precision label matrix	High	Any
<code>bwconncomp</code>	N-D	CC struct	Low	Any

- To extract features from a binary image using `regionprops` with default connectivity, just pass `BW` directly into `regionprops` (i.e., `regionprops(BW)`).
- To compute a label matrix having more memory-efficient data type (e.g., `uint8` versus `double`), use the `labelmatrix` function on the output of `bwconncomp`. See the documentation for each function for more information.

Algorithms

The basic steps in finding the connected components are:

- 1 Search for the next unlabeled pixel, p .
- 2 Use a flood-fill algorithm to label all the pixels in the connected component containing p .
- 3 Repeat steps 1 and 2 until all the pixels are labelled.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- `bwconncomp` only supports 2-D inputs.
- The `conn` arguments must be a compile-time constant and the only connectivities supported are 4 or 8. You can also specify connectivity as a 3-by-3 matrix, but it can only be `[0 1 0;1 1 1;0 1 0]` or `ones(3)`
- The `PixelIdxList` field in the `CC` struct return value is not supported.

See Also

See Also

`bwlabel` | `bwlabeln` | `labelmatrix` | `regionprops`

Introduced in R2009a

bwconvhull

Generate convex hull image from binary image

Syntax

```
CH = bwconvhull(BW)
CH = bwconvhull(BW,method)
CH = bwconvhull(BW,'objects',conn)
```

Description

`CH = bwconvhull(BW)` computes the convex hull of all objects in `BW` and returns `CH`, a binary convex hull image.

`CH = bwconvhull(BW,method)` specifies the desired method for computing the convex hull image.

`CH = bwconvhull(BW,'objects',conn)` specifies the desired connectivity used when defining individual foreground objects. The `conn` parameter is only valid when the method is `'objects'`.

Examples

Display Binary Convex Hull of Image

Read a grayscale image into the workspace. Convert it into a binary image and calculate the union binary convex hull. Finally, calculate the objects convex hull and display all the images in one figure window.

```
subplot(2,2,1);
I = imread('coins.png');
imshow(I);
title('Original');

subplot(2,2,2);
```

```
BW = I > 100;
imshow(BW);
title('Binary');

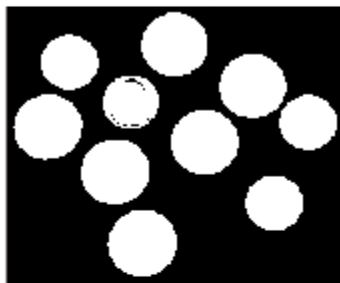
subplot(2,2,3);
CH = bwconvhull(BW);
imshow(CH);
title('Union Convex Hull');

subplot(2,2,4);
CH_objects = bwconvhull(BW, 'objects');
imshow(CH_objects);
title('Objects Convex Hull');
```

Original



Binary



Union Convex Hull



Objects Convex Hull



Input Arguments

BW — Input binary image

2-D logical array

Input binary image, specified as a 2-D logical array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

method — Method used to compute the convex hull

'union' (default) | 'objects'

Method used to compute the convex hull, specified as one of the following character vectors.

Value	Description
'union'	Compute the convex hull of all foreground objects, treating them as a single object
'objects'	Compute the convex hull of each connected component of BW individually. CH contains the convex hulls of each connected component.

Data Types: `char`

conn — Connectivity (only used when the method specified is 'objects')

8 (default) | 4 | 3-by-3 matrix of 0s and 1s

Connectivity, specified as either of the following scalar values. The **conn** parameter is only valid when the **method** is 'objects'.

Value	Description
4	Two-dimensional, four-connected neighborhood
8	Two-dimensional, eight-connected neighborhood.

You can also define connectivity in a more general way by using a 3-by-3 matrix of 0s and 1s. The 1-valued elements define neighborhood locations relative to **conn**'s center element. **conn** must be symmetric about its center element.

Data Types: `double`

Output Arguments

CH — Binary mask of the convex hull of all foreground objects in the input image

2-D logical array

Binary mask of the convex hull of all foreground objects in the input image, returned as a 2-D logical array.

See Also

See Also

`bwconncomp` | `bwlabel` | `labelmatrix` | `regionprops`

Introduced in R2011a

bwdist

Distance transform of binary image

Syntax

```
D = bwdist(BW)
[D,IDX] = bwdist(BW)
[D,IDX] = bwdist(BW,method)
[gpuarrayD, gpuarrayIDX]= bwdist(gpuarrayBW)
```

Description

`D = bwdist(BW)` computes the Euclidean distance transform of the binary image `BW`. For each pixel in `BW`, the distance transform assigns a number that is the distance between that pixel and the nearest nonzero pixel of `BW`. `bwdist` uses the Euclidean distance metric by default. `BW` can have any dimension. `D` is the same size as `BW`.

`[D,IDX] = bwdist(BW)` also computes the closest-pixel map in the form of an index array, `IDX`. (The closest-pixel map is also called the feature map, feature transform, or nearest-neighbor transform.) `IDX` has the same size as `BW` and `D`. Each element of `IDX` contains the linear index of the nearest nonzero pixel of `BW`.

`[D,IDX] = bwdist(BW,method)` computes the distance transform, where `method` specifies an alternate distance metric. `method` can take any of the following values.

Method	Description
'chessboard'	In 2-D, the chessboard distance between (x_1,y_1) and (x_2,y_2) is $\max(x_1 - x_2 , y_1 - y_2)$.
'cityblock'	In 2-D, the cityblock distance between (x_1,y_1) and (x_2,y_2) is $ x_1 - x_2 + y_1 - y_2 $.
'euclidean'	In 2-D, the Euclidean distance between (x_1,y_1) and (x_2,y_2) is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. This is the default method.

Method	Description
'quasi-euclidean'	<p>In 2-D, the quasi-Euclidean distance between (x_1, y_1) and (x_2, y_2) is</p> $ x_1 - x_2 + (\sqrt{2} - 1) y_1 - y_2 , \quad x_1 - x_2 > y_1 - y_2 $ $(\sqrt{2} - 1) x_1 - x_2 + y_1 - y_2 , \quad \text{otherwise.}$

[gpuarrayD, gpuarrayIDX]= bwdist(gpuarrayBW) computes the Euclidean distance transform of the binary image `gpuarrayBW`, performing the operation on a GPU. The images must be 2-D and have less than 2^{32-1} elements. In addition, you can only compute the Euclidean distance metric. This syntax requires the Parallel Computing Toolbox.

Class Support

BW can be numeric or logical, and it must be nonsparse. D is a single matrix with the same size as BW. The class of IDX depends on the number of elements in the input image, and is determined using the following table.

Class	Range
'uint32'	<code>numel(BW) <= 2³² - 1</code>
'uint64'	<code>numel(BW) >= 2³²</code>

`gpuarrayBW` can be a 2-D `gpuArray` of type `uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`, `single`, `double` or `logical`. `gpuarrayD` is a `gpuArray` with the same size as `gpuarrayBW` and underlying class `single`. `gpuarrayIDX` is a `gpuArray` with the same size as `gpuarrayBW` and underlying class `uint32`.

Examples

Compute the Euclidean Distance Transform

This example shows how to compute the Euclidean distance transform of a binary image, and the closest-pixel map of the image.

Create a binary image.

```
bw = zeros(5,5);
bw(2,2) = 1;
bw(4,4) = 1
```

bw =

```

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

Calculate the distance transform.

```
[D,IDX] = bwdist(bw)
```

D = 5×5 single matrix

```

1.4142   1.0000   1.4142   2.2361   3.1623
1.0000         0   1.0000   2.0000   2.2361
1.4142   1.0000   1.4142   1.0000   1.4142
2.2361   2.0000   1.0000         0   1.0000
3.1623   2.2361   1.4142   1.0000   1.4142
```

IDX = 5×5 uint32 matrix

```

7   7   7   7   7
7   7   7   7  19
7   7   7  19  19
7   7  19  19  19
7  19  19  19  19
```

In the nearest-neighbor matrix `IDX` the values 7 and 19 represent the position of the nonzero elements using linear matrix indexing. If a pixel contains a 7, its closest nonzero neighbor is at linear position 7.

Compute the Euclidean distance transform on a GPU

Create an image.

```
bw = gpuArray.zeros(5,5);  
bw(2,2) = 1;  
bw(4,4) = 1;
```

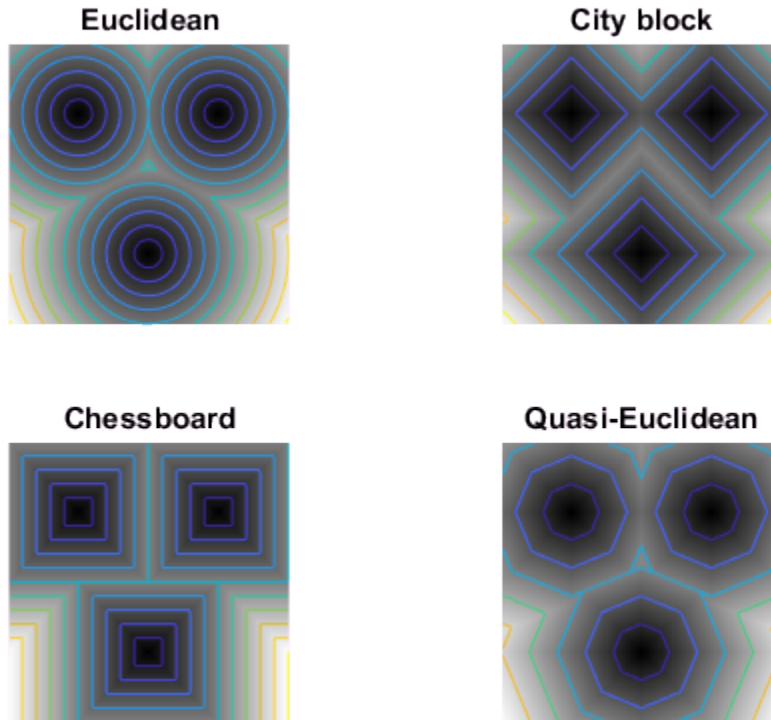
Calculate the distance transform.

```
[D,IDX] = bwdist(bw)
```

Compare 2-D Distance Transforms for Supported Distance Methods

This example shows how to compare the 2-D distance transforms for supported distance methods. In the figure, note how the quasi-Euclidean distance transform best approximates the circular shape achieved by the Euclidean distance method.

```
bw = zeros(200,200);  
bw(50,50) = 1; bw(50,150) = 1; bw(150,100) = 1;  
D1 = bwdist(bw, 'euclidean');  
D2 = bwdist(bw, 'cityblock');  
D3 = bwdist(bw, 'chessboard');  
D4 = bwdist(bw, 'quasi-euclidean');  
RGB1 = repmat(mat2gray(D1), [1 1 3]);  
RGB2 = repmat(mat2gray(D2), [1 1 3]);  
RGB3 = repmat(mat2gray(D3), [1 1 3]);  
RGB4 = repmat(mat2gray(D4), [1 1 3]);  
figure  
subplot(2,2,1), imshow(RGB1), title('Euclidean')  
hold on, imcontour(D1)  
subplot(2,2,2), imshow(RGB2), title('City block')  
hold on, imcontour(D2)  
subplot(2,2,3), imshow(RGB3), title('Chessboard')  
hold on, imcontour(D3)  
subplot(2,2,4), imshow(RGB4), title('Quasi-Euclidean')  
hold on, imcontour(D4)
```



Compare Isosurface Plots for Distance Transforms of 3-D Image

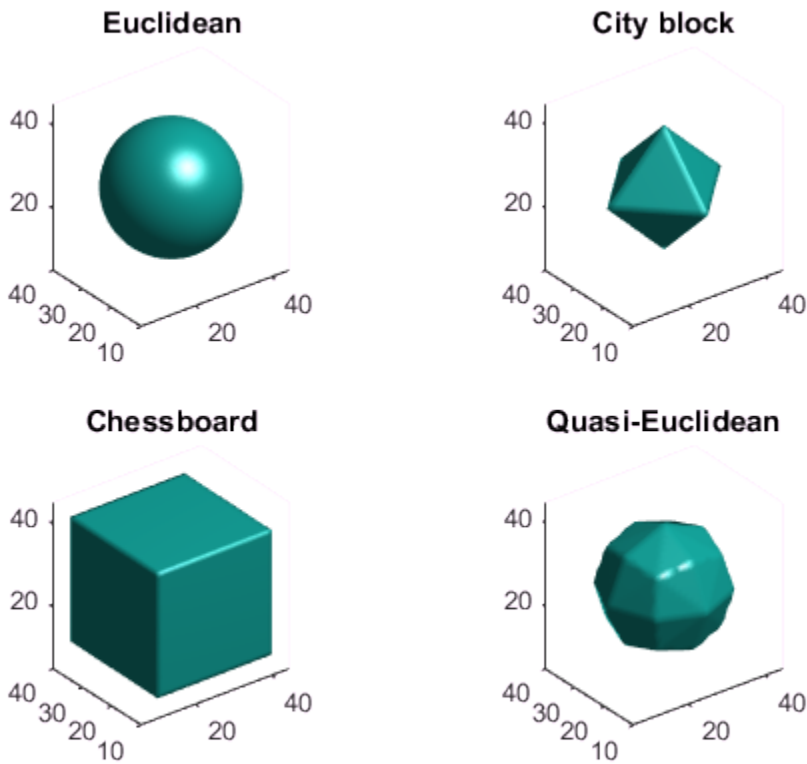
This example shows how to compare isosurface plots for the distance transforms of a 3-D image containing a single nonzero pixel in the center.

```

bw = zeros(50,50,50); bw(25,25,25) = 1;
D1 = bwdist(bw);
D2 = bwdist(bw, 'cityblock');
D3 = bwdist(bw, 'chessboard');
D4 = bwdist(bw, 'quasi-euclidean');
figure
subplot(2,2,1), isosurface(D1,15), axis equal, view(3)
camlight, lighting gouraud, title('Euclidean')

```

```
subplot(2,2,2), isosurface(D2,15), axis equal, view(3)
camlight, lighting gouraud, title('City block')
subplot(2,2,3), isosurface(D3,15), axis equal, view(3)
camlight, lighting gouraud, title('Chessboard')
subplot(2,2,4), isosurface(D4,15), axis equal, view(3)
camlight, lighting gouraud, title('Quasi-Euclidean')
```



Tips

- `bwdist` uses fast algorithms to compute the true Euclidean distance transform, especially in the 2-D case. The other methods are provided primarily for pedagogical

reasons. However, the alternative distance transforms are sometimes significantly faster for multidimensional input images, particularly those that have many nonzero elements.

- The function `bwdist` changed in version 6.4 (R2009b). Previous versions of the Image Processing Toolbox used different algorithms for computing the Euclidean distance transform and the associated label matrix. If you need the same results produced by the previous implementation, use the function `bwdist_old`.

Algorithms

For Euclidean distance transforms, `bwdist` uses the fast algorithm described in

[1] Maurer, Calvin, Rensheng Qi, and Vijay Raghavan, "A Linear Time Algorithm for Computing Exact Euclidean Distance Transforms of Binary Images in Arbitrary Dimensions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 25, No. 2, February 2003, pp. 265-270.

For cityblock, chessboard, and quasi-Euclidean distance transforms, `bwdist` uses the two-pass, sequential scanning algorithm described in

[2] Rosenfeld, Azriel and John Pfaltz, "Sequential operations in digital picture processing," *Journal of the Association for Computing Machinery*, Vol. 13, No. 4, 1966, pp. 471-494.

The different distance measures are achieved by using different sets of weights in the scans, as described in

[3] Paglieroni, David, "Distance Transforms: Properties and Machine Vision Applications," *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, Vol. 54, No. 1, January 1992, pp. 57-58.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the optional second input argument, `method`, must be a compile-time constant. Input images must have fewer than 2^{32} pixels.

See Also

See Also

`bwulterode` | `watershed`

Topics

“Distance Transform”

Introduced before R2006a

bwdistgeodesic

Geodesic distance transform of binary image

Syntax

```
D = bwdistgeodesic(BW,mask)
D = bwdistgeodesic(BW,C,R)
D = bwdistgeodesic(BW,ind)
D = bwdistgeodesic(...,method)
```

Description

`D = bwdistgeodesic(BW,mask)` computes the geodesic distance transform, given the binary image `BW` and the seed locations specified by `mask`. Regions where `BW` is `true` represent valid regions that can be traversed in the computation of the distance transform. Regions where `BW` is `false` represent constrained regions that cannot be traversed in the distance computation. For each `true` pixel in `BW`, the geodesic distance transform assigns a number that is the constrained distance between that pixel and the nearest `true` pixel in `mask`. Output matrix `D` contains geodesic distances.

`D = bwdistgeodesic(BW,C,R)` computes the geodesic distance transform of the binary image `BW`. Vectors `C` and `R` contain the column and row coordinates of the seed locations.

`D = bwdistgeodesic(BW,ind)` computes the geodesic distance transform of the binary image `BW`. `ind` is a vector of linear indices of seed locations.

`D = bwdistgeodesic(...,method)` specifies an alternate distance metric.

Input Arguments

BW

Binary image.

mask

Logical image the same size as `BW` that specifies seed locations.

C, R

Numeric vectors that contain the positive integer column and row coordinates of the seed locations. Coordinate values are valid C,R subscripts in BW.

ind

Numeric vector of positive integer, linear indices of seed locations.

method

Type of distance metric. **method** can have any of these values.

Method	Description
'cityblock'	In 2-D, the cityblock distance between (x_1, y_1) and (x_2, y_2) is $ x_1 - x_2 + y_1 - y_2 $.
'chessboard'	The chessboard distance is $\max(x_1 - x_2 , y_1 - y_2)$.
'quasi-euclidean'	The quasi-Euclidean distance is $ x_1 - x_2 + (\sqrt{2} - 1) y_1 - y_2 , x_1 - x_2 > y_1 - y_2 $ $(\sqrt{2} - 1) x_1 - x_2 + y_1 - y_2 , \text{ otherwise.}$

Default: 'chessboard'

Output Arguments

D

Numeric array of class **single**, with the same size as input **BW**, that contains geodesic distances.

Class Support

BW is a logical matrix. **C**, **R**, and **ind** are numeric vectors that contain positive integer values. **D** is a numeric array of class **single** that has the same size as the input **BW**.

Examples

Compute Geodesic Distance Transformation of Binary Image

Create a sample binary image for this example.

```
BW = [1 1 1 1 1 1 1 1 1 1;...
      1 1 1 1 1 1 0 0 1 1;...
      1 1 1 1 1 1 0 0 1 1;...
      1 1 1 1 1 1 0 0 1 1;...
      0 0 0 0 0 1 0 0 1 0;...
      0 0 0 0 1 1 0 1 1 0;...
      0 1 0 0 1 1 0 0 0 0;...
      0 1 1 1 1 1 1 0 1 0;...
      0 1 1 0 0 0 1 1 1 0;...
      0 0 0 0 1 0 0 0 0 0];
BW = logical(BW);
```

Create two vectors of seed locations.

```
C = [1 2 3 3 3];
R = [3 3 3 1 2];
```

Calculate the geodesic distance transform. Output pixels for which BW is false have undefined geodesic distance and contain NaN values. Because there is no connected path from the seed locations to element BW(10,5), the output D(10,5) has a value of Inf.

```
D = bwdistgeodesic(BW,C,R)
```

D = 10×10 single matrix

```

     2     1     0     1     2     3     4     5     6     7
     1     1     0     1     2     3  NaN  NaN     6     7
     0     0     0     1     2     3  NaN  NaN     7     7
     1     1     1     1     2     3  NaN  NaN     8     8
  NaN  NaN  NaN  NaN  NaN     3  NaN  NaN     9  NaN
  NaN  NaN  NaN  NaN     4     4  NaN    10    10  NaN
  NaN     8  NaN  NaN     5     5  NaN  NaN    NaN  NaN
  NaN     8     7     6     6     6     6  NaN     8  NaN
  NaN     8     7  NaN  NaN  NaN     7     7     8  NaN
  NaN  NaN  NaN  NaN  Inf  NaN  NaN  NaN  NaN  NaN
```

Algorithms

`bwdistgeodesic` uses the geodesic distance algorithm described in Soille, P., *Morphological Image Analysis: Principles and Applications, 2nd Edition*, Secaucus, NJ, Springer-Verlag, 2003, pp. 219–221.

See Also

See Also

`bwdist` | `graydist`

Introduced in R2011b

bweuler

Euler number of binary image

Syntax

```
eul = bweuler(BW,n)
```

Description

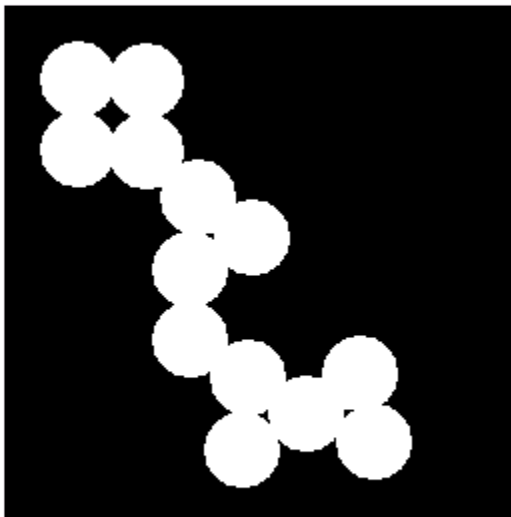
`eul = bweuler(BW,n)` returns the Euler number for the binary image `BW`. The Euler number is the total number of objects in the image minus the total number of holes in those objects. `n` specifies the connectivity. Objects are connected sets of `on` pixels, that is, pixels having a value of 1.

Examples

Calculate Euler Number for Binary Image

Read binary image into workspace, and display it.

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



Calculate the Euler number. In this example, all the circles touch so they create one object. The object contains four "holes", which are the black areas created by the touching circles. Thus the Euler number is 1 minus 4, or -3.

```
bweuler(BW)
```

```
ans = -3
```

Input Arguments

BW — Input binary image

logical or numeric matrix that must be 2-D, real, and nonsparse

Input binary image, specified as a logical or numeric matrix that must be 2-D, real, and nonsparse.

Example: `BW = imread('circles.png'); eul = bweuler(BW,4);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

n — Connectivity

8 (default) | 4

Connectivity, specified as either the value 4 or 8.

Value	Description
4	4-connected objects
8	8-connected objects

Example: `BW2 = bweuler(BW,4);`

Data Types: `double`

Output Arguments

eu1 — Euler number

numeric scalar value

Euler number, returned as a numeric scalar value of class `double`.

Algorithms

`bweuler` computes the Euler number by considering patterns of convexity and concavity in local 2-by-2 neighborhoods. See [2] for a discussion of the algorithm used.

References

- [1] Horn, Berthold P. K., *Robot Vision*, New York, McGraw-Hill, 1986, pp. 73-77.
- [2] Pratt, William K., *Digital Image Processing*, New York, John Wiley & Sons, Inc., 1991, p. 633.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

See Also

See Also

`bwmorph` | `bwperim`

Introduced before R2006a

bwhitmiss

Binary hit-miss operation

Syntax

```
BW2 = bwhitmiss(BW1,SE1,SE2)
BW2 = bwhitmiss(BW1,INTERVAL)
```

Description

`BW2 = bwhitmiss(BW1,SE1,SE2)` performs the hit-miss operation defined by the structuring elements `SE1` and `SE2`. The hit-miss operation preserves pixels whose neighborhoods match the shape of `SE1` and don't match the shape of `SE2`. `SE1` and `SE2` can be flat structuring element objects, created by `strel`, or neighborhood arrays. The neighborhoods of `SE1` and `SE2` should not have any overlapping elements. The syntax `bwhitmiss(BW1,SE1,SE2)` is equivalent to `imerode(BW1,SE1) & imerode(~BW1,SE2)`.

`BW2 = bwhitmiss(BW1,INTERVAL)` performs the hit-miss operation defined in terms of a single array, called an *interval*. An interval is an array whose elements can contain 1, 0, or -1. The 1-valued elements make up the domain of `SE1`, the -1-valued elements make up the domain of `SE2`, and the 0-valued elements are ignored. The syntax `bwhitmiss(BW1,INTERVAL)` is equivalent to `bwhitmiss(BW1,INTERVAL == 1, INTERVAL == -1)`.

Class Support

`BW1` can be a logical or numeric array of any dimension, and it must be nonsparse. `BW2` is always a logical array the same size as `BW1`. `SE1` and `SE2` must be flat STREL objects or they must be logical or numeric arrays containing 1's and 0's. `INTERVAL` must be an array containing 1's, 0's, and -1's.

Examples

Perform Hit-miss Operation on Binary Image

Create sample binary image for this example.

```
bw = [0 0 0 0 0 0
      0 0 1 1 0 0
      0 1 1 1 1 0
      0 1 1 1 1 0
      0 0 1 1 0 0
      0 0 1 0 0 0]
```

```
bw =
```

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

Define an interval.

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

Perform hit-miss operation.

```
bw2 = bwhitmiss(bw,interval)
```

```
bw2 = 6x6 logical array
0 0 0 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

See Also

See Also

strel | offsetstrel | imdilate | imerode

Introduced before R2006a

bwlabel

Label connected components in 2-D binary image

Syntax

```
L = bwlabel(BW)
L = bwlabel(BW,n)
[L,num] = bwlabel( ___ )
[gpuarrayL,num] = bwlabel(gpuarrayBW,n)
```

Description

`L = bwlabel(BW)` returns the label matrix `L` that contains labels for the 8-connected objects found in `BW`. The label matrix, `L`, is the same size as `BW`.

`L = bwlabel(BW,n)` returns a label matrix, where the variable `n` specifies the connectivity.

`[L,num] = bwlabel(___)` also returns `num`, the number of connected objects found in `BW`.

`[gpuarrayL,num] = bwlabel(gpuarrayBW,n)` performs the labeling operation on a GPU. The input image and output image are `gpuArrays`. The variable `n` can be a numeric array or a `gpuArray`. This syntax requires the Parallel Computing Toolbox.

Examples

Label Components Using 4-connected Objects

Create a small binary image.

```
BW = logical ([1 1 1 0 0 0 0 0
               1 1 1 0 1 1 0 0
               1 1 1 0 1 1 0 0
               1 1 1 0 0 0 1 0
               1 1 1 0 0 0 1 0])
```

```

    1   1   1   0   0   0   1   0
    1   1   1   0   0   1   1   0
    1   1   1   0   0   0   0   0]);

```

Create the label matrix using 4-connected objects.

```
L = bwlabel(BW,4)
```

```
L =
```

```

    1   1   1   0   0   0   0   0
    1   1   1   0   2   2   0   0
    1   1   1   0   2   2   0   0
    1   1   1   0   0   0   3   0
    1   1   1   0   0   0   3   0
    1   1   1   0   0   0   3   0
    1   1   1   0   0   3   3   0
    1   1   1   0   0   0   0   0

```

Use the `find` command to get the row and column coordinates of the object labeled "2".

```
[r, c] = find(L==2);
rc = [r c]
```

```
rc =
```

```

    2   5
    3   5
    2   6
    3   6

```

Label Components Using 4-connected Objects on a GPU

Create a small binary image and create a `gpuArray` object to contain it.

```

BW = gpuArray(logical([1 1 1 0 0 0 0 0
    1 1 1 0 1 1 0 0
    1 1 1 0 1 1 0 0
    1 1 1 0 0 0 1 0
    1 1 1 0 0 0 1 0
    1 1 1 0 0 0 1 0
    1 1 1 0 0 1 1 0
    1 1 1 0 0 0 0 0]));

```

Create the label matrix using 4-connected objects.

```
L = bwlabel(BW,4)
```

Use the `find` command to get the row and column coordinates of the object labeled "2".

```
[r,c] = find(L == 2)
```

Input Arguments

BW — Input binary image

2-D, real, nonsparse, numeric or logical array

Input binary image, specified as a 2-D, real, nonsparse, numeric or logical array.

```
Example: BW = imread('text.png'); L = bwlabel(BW);
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

n — Connectivity

8 (default) | 4

Connectivity, specified as the values **4**, for 4-connected objects, or **8**, for 8-connected objects.

```
Example: BW = imread('text.png'); L = bwlabel(BW,4);
```

Data Types: `double`

gpuarrayBW — Input binary image for processing on GPU

`gpuArray`

Input binary image for processing on GPU, specified as a `gpuArray`.

```
Example: BW = gpuArray(imread('text.png')); L = bwlabel(BW);
```

Output Arguments

L — Label matrix

array class `double`

Label matrix, returned as an array of class `double`, the same size as the input image.

num — Number of connected objects found

array class double

Label matrix, returned as an array of class double.

gpuarrayL — Label matrix

gpuArray

Label matrix, returned as a gpuArray when processed on a GPU.

Tips

- The functions `bwlabel`, `bwlabeln`, and `bwconncomp` all compute connected components for binary images. `bwconncomp` replaces the use of `bwlabel` and `bwlabeln`. It uses significantly less memory and is sometimes faster than the other functions.

	Input Dimension	Output Form	Memory Use	Connectivity
<code>bwlabel</code>	2-D	Double-precision label matrix	High	4 or 8
<code>bwlabeln</code>	N-D	Double-precision label matrix	High	Any
<code>bwconncomp</code>	N-D	CC struct	Low	Any

- You can use the MATLAB `find` function in conjunction with `bwlabel` to return vectors of indices for the pixels that make up a specific object. For example, to return the coordinates for the pixels in object 2, enter the following:

```
[r, c] = find(bwlabel(BW)==2)
```

You can display the output matrix as a pseudocolor indexed image. Each object appears in a different color, so the objects are easier to distinguish than in the original image. For more information, see `label2rgb`.

- To compute a label matrix having a more memory-efficient data type (e.g., `uint8` versus `double`), use the `labelmatrix` function on the output of `bwconncomp`. For more information, see the reference page for each function.
- To extract features from a binary image using `regionprops` with default connectivity, just pass `BW` directly into `regionprops`, i.e., `regionprops(BW)`.

- The `bwlabel` function can take advantage of hardware optimization for data types `logical`, `uint8`, and `single` to run faster. Hardware optimization requires `marker` and `mask` to be 2-D images and `conn` to be either 4 or 8.

Algorithms

`bwlabel` uses the general procedure outlined in reference [1], pp. 40-48:

- 1 Run-length encode the input image.
- 2 Scan the runs, assigning preliminary labels and recording label equivalences in a local equivalence table.
- 3 Resolve the equivalence classes.
- 4 Relabel the runs based on the resolved equivalence classes.

References

- [1] Haralick, Robert M., and Linda G. Shapiro, *Computer and Robot Vision, Volume I*, Addison-Wesley, 1992, pp. 28-48.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, the parameter `n` must be a compile-time constant.

See Also

See Also

`bwconncomp` | `bwlabeln` | `bwselect` | `label2rgb` | `labelmatrix` | `regionprops`

Introduced before R2006a

bwlabeln

Label connected components in binary image

Syntax

```
L = bwlabeln(BW)
[L, NUM] = bwlabeln(BW)
[L, NUM] = bwlabeln(BW, conn)
```

Description

`L = bwlabeln(BW)` returns a label matrix, `L`, containing labels for the connected components in `BW`. The input image `BW` can have any dimension; `L` is the same size as `BW`. The elements of `L` are integer values greater than or equal to 0. The pixels labeled 0 are the background. The pixels labeled 1 make up one object; the pixels labeled 2 make up a second object; and so on. The default connectivity is 8 for two dimensions, 26 for three dimensions, and `conndef(ndims(BW), 'maximal')` for higher dimensions.

`[L, NUM] = bwlabeln(BW)` returns in `NUM` the number of connected objects found in `BW`.

`[L, NUM] = bwlabeln(BW, conn)` specifies the desired connectivity. `conn` can have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can also be defined in a more general way for any dimension by using for `conn` a 3-by-3-by-...-by-3 matrix of 0s and 1s. The 1-valued elements define

neighborhood locations relative to the central element of `conn`. Note that `conn` must be symmetric about its central element.

The functions `bwlabel`, `bwlabeln`, and `bwconncomp` all compute connected components for binary images. `bwconncomp` replaces the use of `bwlabel` and `bwlabeln`. It uses significantly less memory and is sometimes faster than the older functions.

Function	Input Dimension	Output Form	Memory Use	Connectivity
<code>bwlabel</code>	2-D	Double-precision label matrix	High	4 or 8
<code>bwlabeln</code>	N-D	Double-precision label matrix	High	Any
<code>bwconncomp</code>	N-D	CC struct	Low	Any

Class Support

BW can be numeric or logical, and it must be real and nonsparse. L is of class `double`.

Examples

Calculate Centroids of 3-D Objects

Create simple sample 3-D binary image.

```
BW = cat(3, [1 1 0; 0 0 0; 1 0 0],...
           [0 1 0; 0 0 0; 0 1 0],...
           [0 1 1; 0 0 0; 0 0 1])
```

BW =

(:,:,1) =

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

(:,:,2) =

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

```
(:,: ,3) =
```

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

Label connected components in the image.

```
bwlabeln(BW)
```

```
ans =
```

```
(:,: ,1) =
```

```
1  1  0
0  0  0
2  0  0
```

```
(:,: ,2) =
```

```
0  1  0
0  0  0
0  2  0
```

```
(:,: ,3) =
```

```
0  1  1
0  0  0
0  0  2
```

Tips

To extract features from a binary image using `regionprops` with default connectivity, just pass `BW` directly into `regionprops`, i.e., `regionprops(BW)`.

To compute a label matrix having a more memory-efficient data type (e.g., `uint8` versus `double`), use the `labelmatrix` function on the output of `bwconncomp`:

```
C = bwconncomp(BW);  
L = labelmatrix(CC);  
  
CC = bwconncomp(BW, conn);  
S = regionprops(CC);
```

Algorithms

`bwlabeln` uses the following general procedure:

- 1 Scan all image pixels, assigning preliminary labels to nonzero pixels and recording label equivalences in a union-find table.
- 2 Resolve the equivalence classes using the union-find algorithm [1].
- 3 Relabel the pixels based on the resolved equivalence classes.

References

[1] Sedgewick, Robert, *Algorithms in C*, 3rd Ed., Addison-Wesley, 1998, pp. 11-20.

See Also

See Also

`bwconncomp` | `bwlabel` | `label2rgb` | `labelmatrix` | `regionprops`

Introduced before R2006a

bwlookup

Nonlinear filtering using lookup tables

Syntax

```
A = bwlookup(BW,lut)
gpuarrayA = bwlookup(gpuarrayBW,lut)
```

Description

`A = bwlookup(BW,lut)` performs a 2-by-2 or 3-by-3 nonlinear neighborhood filtering operation on binary or grayscale image `BW` and returns the results in output image `A`. The neighborhood processing determines an integer index value used to access values in lookup table `lut`. The fetched `lut` value becomes the pixel value in output image `A` at the targeted position.

- `A` is the same size as `BW`
- `A` is the same data type as `lut`

`gpuarrayA = bwlookup(gpuarrayBW,lut)` performs the filtering operation on a GPU. The input image and output image are `gpuArrays`. `lut` can be a numeric or `gpuArray` vector. This syntax requires the Parallel Computing Toolbox.

Examples

Perform Erosion Along Edges of Binary Image

Construct the vector `lut` such that the filtering operation places a 1 at the targeted pixel location in the input image only when all four pixels in the 2-by-2 neighborhood of `BW` are set to 1.

```
lutfun = @(x)(sum(x(:))==4);
lut = makelut(lutfun,2)
```

```
lut =
```



```
0
0
0
0
0
0
0
0
0
0
```

Load binary image.

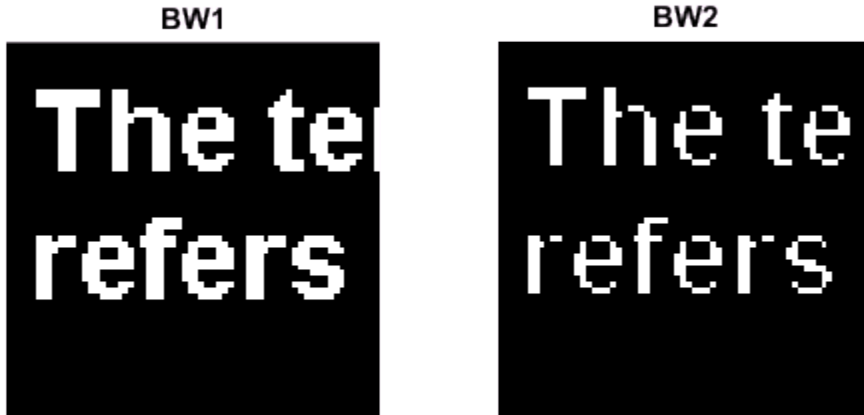
```
BW1 = imread('text.png');
```

Perform 2-by-2 neighborhood processing with 16-element vector lut .

```
BW2 = bwlookup(BW1,lut);
```

Show zoomed before and after images.

```
figure;
h1 = subplot(1,2,1); imshow(BW1), axis off; title('BW1')
h2 = subplot(1,2,2); imshow(BW2); axis off; title('BW2')
% 16X zoom to see effects of erosion on text
set(h1,'Ylim',[.5 64.5]); set(h1,'Xlim',[.5 64.5]);
set(h2,'Ylim',[.5 64.5]); set(h2,'Xlim',[.5 64.5]);
```



2-by-2 Neighborhood Erosion of Binary Image Using GPU

Perform an erosion along the edges of a binary image using a 2-by-2 neighborhood, running the code on a graphics processing unit (GPU).

Construct `lut` so it is true only when all four 2-by-2 locations equal 1

```
lut = makelut('sum(x(:))==4',2);
```

Load binary image.

```
BW1 = imread('text.png');
```

Perform 2-by-2 neighborhood processing with 16-element vector LUT. To run the code on a GPU, create a `gpuArray` to contain the image.

```
BW2 = bwlookup(gpuArray(BW1),lut);
```

Show zoomed before and after images.

```
figure;
h1 = subplot(1,2,1); imshow(BW1), axis off; title('BW1')
h2 = subplot(1,2,2); imshow(BW2); axis off; title('BW2')

% 16X zoom to see effects of erosion on text
set(h1,'Ylim',[.5 64.5]); set(h1,'Xlim',[.5 64.5]);
set(h2,'Ylim',[.5 64.5]); set(h2,'Xlim',[.5 64.5]);
```

Input Arguments

BW — Input image

binary image | grayscale image

Input image transformed by nonlinear neighborhood filtering operation, specified as either a grayscale or binary (logical) image. In the case of numeric values, non-zero pixels are considered `true` which is equivalent to logical 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

gpuarrayBW — Input image for processing on a GPU

A `gpuArray` containing a binary image

Input image for processing on a GPU, specified as a `gpuArray` containing a binary image.

lut — Lookup table of output pixel values

16- or 512-element vector

Lookup table of output pixel values, specified as a 16- or 512-element vector. The size of `lut` determines which of the two neighborhood operations is performed.

- If `lut` contains 16 data elements, then the neighborhood matrix is 2-by-2.
- If `lut` contains 512 data elements, then the neighborhood matrix is 3-by-3.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Output Arguments

A — Output image

binary image | grayscale image

Output image, returned as a grayscale or binary image whose size matches `BW`, and whose distribution of pixel values are determined by the content of `lut`.

- `A` is the same size as `BW`
- `A` is the same data type as `lut`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

gpuarrayA — Output image

`gpuArray` containing a grayscale or binary image

Output image, returned as `gpuArray` containing a grayscale or binary image.

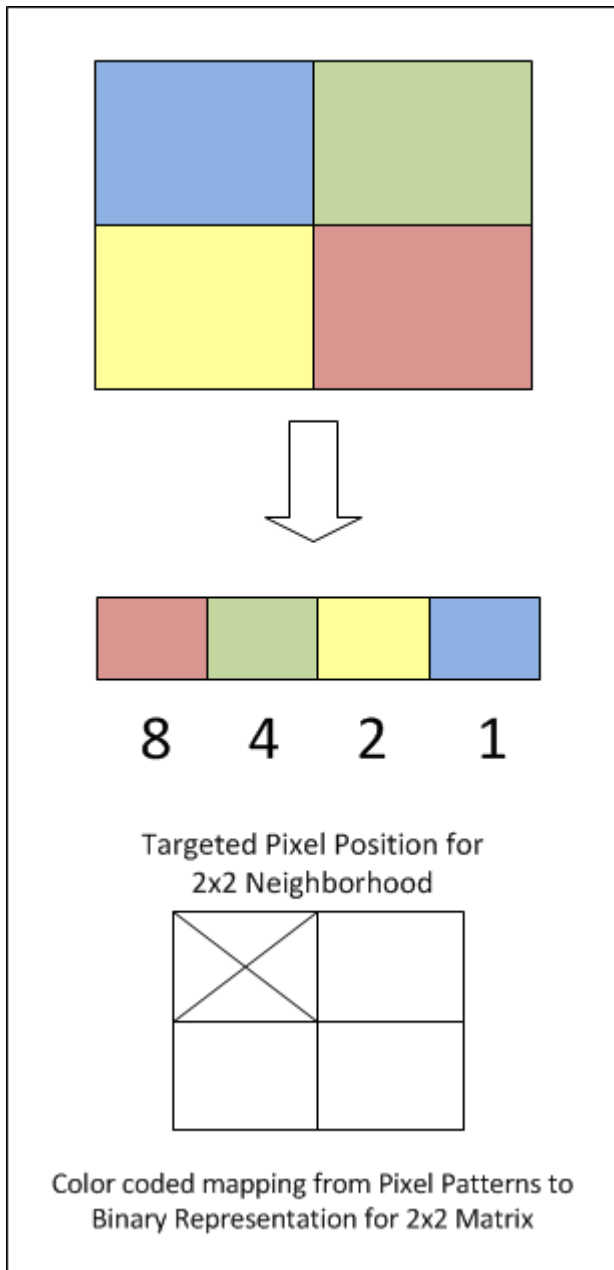
Algorithm

The first step in each iteration of the filtering operation performed by `bwlookup` entails computing the `index` into vector `lut` based on the binary pixel pattern of the neighborhood matrix on image `BW`. The value in `lut` accessed at `index`, `lut(index)`, is inserted into output image `A` at the targeted pixel location. This results in image `A` being the same data type as vector `lut`.

Since there is a 1-to-1 correspondence in targeted pixel locations, image `A` is the same size as image `BW`. If the targeted pixel location is on an edge of image `BW` and if any part of the 2-by-2 or 3-by-3 neighborhood matrix extends beyond the image edge, then these non-image locations are padded with 0 in order to perform the filtering operation.

The following figures show the mapping from binary 0 and 1 patterns in the neighborhood matrices to its binary representation. Adding 1 to the binary representation yields `index` which is used to access `lut`.

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



To illustrate, this example shows how the pixel pattern in a 2-by-2 matrix determines which entry in lut is placed in the targeted pixel location.

- 1 Create random 16-element lut vector containing uint8 data.

```
scurr = rng;    % save current random number generator seed state
rng('default') % always generate same set of random numbers
lut = uint8( round( 255*rand(16,1) ) ) % generate lut
rng(scurr);    % restore
```

```
lut =
```

```
208
231
 32
233
161
 25
 71
139
244
246
 40
248
244
124
204
 36
```

- 2 Create a 2-by-2 image and assume for this example that the targeted pixel location is location BW(1,1).

```
BW = [1 0; 0 1]
```

```
BW =
```

```
 1    0
 0    1
```

- 3 By referring to the color coded mapping figure above, the binary representation for this 2-by-2 neighborhood can be computed as shown in the code snippet below. The logical 1 at BW(1,1) corresponds to blue in the figure which maps to the Least Significant Bit (LSB) at position 0 in the 4-bit binary representation ($2^0=1$). The logical 1 at BW(2,2) is red which maps to the Most Significant Bit (MSB) at position 3 in the 4-bit binary representation ($2^3=8$).

```
% BW(1,1): blue square; sets bit position 0 on right
% BW(2,2): red square; sets bit position 3 on left
binNot = '1 0 0 1'; % binary representation of 2x2 neighborhood matrix

X = bin2dec( binNot ); % convert from binary to decimal
index = X + 1 % add 1 to compute index value for uint8 vector lut
A11 = lut(index) % value at A(1,1)

index =

    10

A11 =

    246
```

- 4** The above calculation predicts that output image A should contain the value 246 at targeted position A(1,1).

```
A = bwlookup(BW,lut) % perform filtering

A =

    246    32
    161   231
```

A(1,1) does in fact equal 246.

Note: For a more robust way to perform image erosion, see function `imerode`.

For 3-by-3 neighborhoods, `length(lut)` is 512. There are nine pixels in each neighborhood, and two possible states for each pixel, so the total number of permutations is $2^9 = 512$.

The process for computing the binary representation of 3-by-3 neighborhood processing is the same as shown above for 2-by-2 neighborhoods.

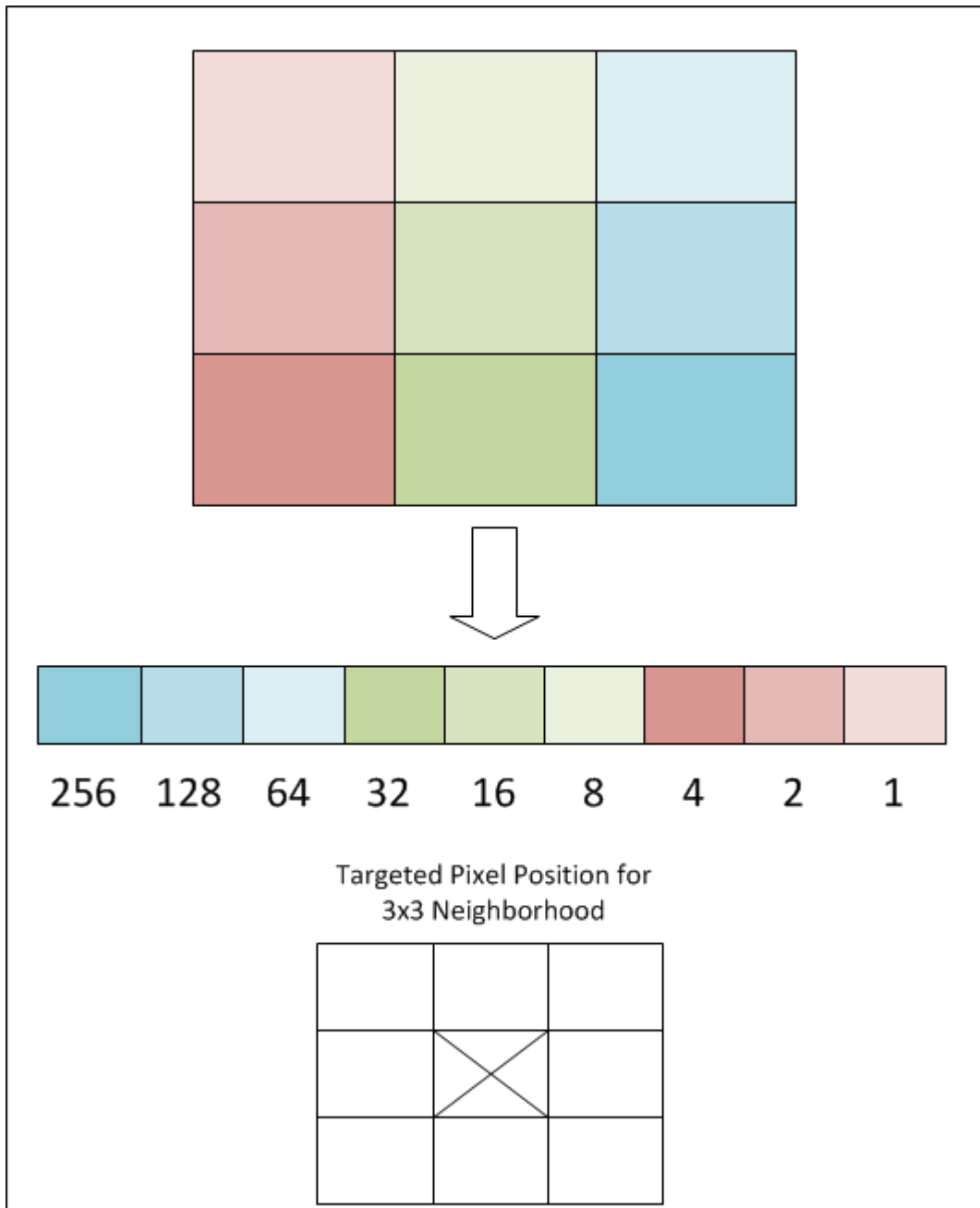


Figure 1: Color coded mapping from Pixel Patterns to Binary Representation for 3x3 Matrix

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, specify an input image of class `logical`.

See Also

See Also

`makelut`

Introduced in R2012b

bwmorph

Morphological operations on binary images

Syntax

```
BW2 = bwmorph(BW,operation)
BW2 = bwmorph(BW,operation,n)
gpuarrayBW2 = bwmorph(gpuarrayBW, ___ )
```

Description

`BW2 = bwmorph(BW,operation)` applies a specific morphological operation to the binary image `BW`.

`BW2 = bwmorph(BW,operation,n)` applies the operation `n` times. `n` can be `Inf`, in which case the operation is repeated until the image no longer changes.

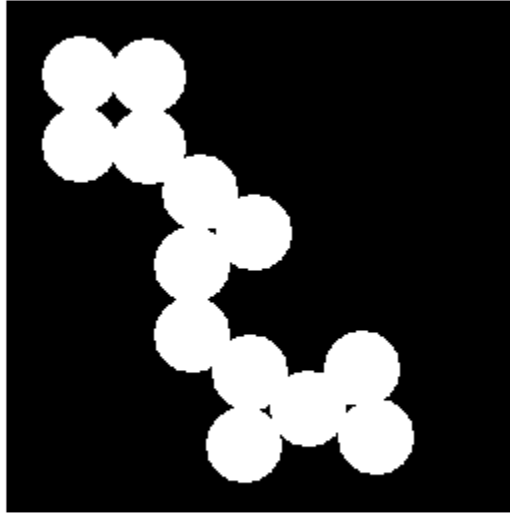
`gpuarrayBW2 = bwmorph(gpuarrayBW, ___)` performs the morphological operation on a GPU. The input image and output image are `gpuArrays`. This syntax requires the Parallel Computing Toolbox.

Examples

Perform Morphological Operations on Binary Image

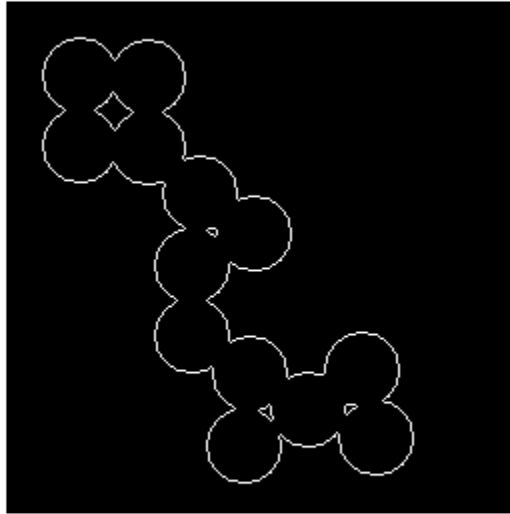
Read binary image and display it.

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



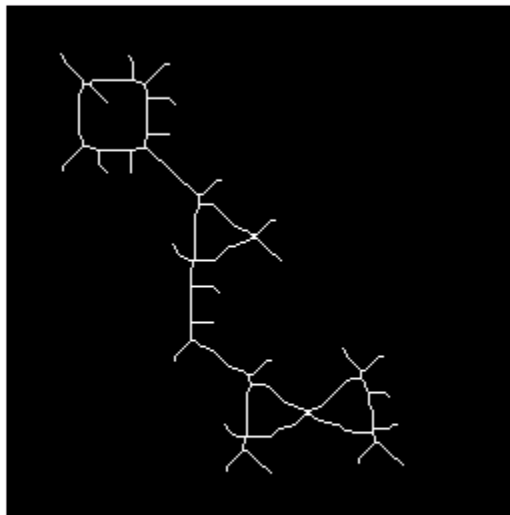
Remove interior pixels to leave an outline of the shapes.

```
BW2 = bwmorph(BW, 'remove');  
figure  
imshow(BW2)
```



Get the image skeleton.

```
BW3 = bwmorph(BW, 'skel', Inf);  
figure  
imshow(BW3)
```



Perform Morphological Operations on a GPU

This example performs the same operations as the previous example but performs them on a GPU. The example starts by reading the image into a `gpuArray`.

```
BW1 = gpuArray(imread('circles.png'));  
figure  
imshow(BW1)
```

```
BW2 = bwmorph(BW1, 'remove');  
figure  
imshow(BW2)
```

```
BW3 = bwmorph(BW1, 'skel', Inf);  
figure  
imshow(BW3)
```

Input Arguments

BW — Input image

binary image

Input image, specified as a binary image. The input image can be numeric or logical, but must be 2-D, real and nonsparse.

Example: `BW = imread('circles.png');`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

operation — Morphological operation to perform

character vector

Morphological operation to perform, specified as one of the following.

Operation	Description
'bothat'	Performs the morphological “bottom hat” operation, returning the image minus the morphological closing of the image (dilation followed by erosion).
'branchpoints'	Find branch points of skeleton. For example: <pre> 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 becomes 0 0 0 0 0 1 1 1 1 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 </pre> <p>Note: To find branch points, the image must be skeletonized. To create a skeletonized image, use <code>bwmorph(BW, 'skel')</code>.</p>
'bridge'	Bridges unconnected pixels, that is, sets 0-valued pixels to 1 if they have two nonzero neighbors that are not connected. For example: <pre> 1 0 0 1 1 0 1 0 1 becomes 1 1 1 0 0 1 0 1 1 </pre>
'clean'	Removes isolated pixels (individual 1s that are surrounded by 0s), such as the center pixel in this pattern.

Operation	Description
	<pre>0 0 0 0 1 0 0 0 0</pre>
'close'	Performs morphological closing (dilation followed by erosion).
'diag'	<p>Uses diagonal fill to eliminate 8-connectivity of the background. For example:</p> <pre>0 1 0 0 1 0 1 0 0 becomes 1 1 0 0 0 0 0 0 0</pre>
'endpoints'	<p>Finds end points of skeleton. For example:</p> <pre>1 0 0 0 1 0 0 0 0 1 0 0 becomes 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0</pre> <p>Note: To find end points, the image must be skeletonized. To create a skeletonized image, use <code>bwmorph(BW, 'skel')</code>.</p>
'fill'	<p>Fills isolated interior pixels (individual 0s that are surrounded by 1s), such as the center pixel in this pattern.</p> <pre>1 1 1 1 0 1 1 1 1</pre>
'hbreak'	<p>Removes H-connected pixels. For example:</p> <pre>1 1 1 1 1 1 0 1 0 becomes 0 0 0 1 1 1 1 1 1</pre>
'majority'	Sets a pixel to 1 if five or more pixels in its 3-by-3 neighborhood are 1s; otherwise, it sets the pixel to 0.
'open'	Performs morphological opening (erosion followed by dilation).
'remove'	Removes interior pixels. This option sets a pixel to 0 if all its 4-connected neighbors are 1, thus leaving only the boundary pixels on.

Operation	Description
'shrink'	With $n = \text{Inf}$, shrinks objects to points. It removes pixels so that objects without holes shrink to a point, and objects with holes shrink to a connected ring halfway between each hole and the outer boundary. This option preserves the Euler number.
'skel'	With $n = \text{Inf}$, removes pixels on the boundaries of objects but does not allow objects to break apart. The pixels remaining make up the image skeleton. This option preserves the Euler number.
'spur'	Removes spur pixels. For example: <pre> 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 becomes 0 0 0 0 0 1 0 0 0 1 0 0 1 1 0 0 1 1 0 0 </pre>
'thicken'	With $n = \text{Inf}$, thickens objects by adding pixels to the exterior of objects until doing so would result in previously unconnected objects being 8-connected. This option preserves the Euler number.
'thin'	With $n = \text{Inf}$, thins objects to lines. It removes pixels so that an object without holes shrinks to a minimally connected stroke, and an object with holes shrinks to a connected ring halfway between each hole and the outer boundary. This option preserves the Euler number. See “Algorithms” on page 1-212 for more detail.
'tophat'	Performs morphological "top hat" operation, returning the image minus the morphological opening of the image (erosion followed by dilation).

Example: `BW3 = bwmorph(BW, 'skel');`

Data Types: char

n — Number of times to perform the operation

numeric value

Number of times to perform the operation, specified as a numeric value. n can be `Inf`, in which case `bwmorph` repeats the operation until the image no longer changes.

Example: `BW3 = bwmorph(BW, 'skel', 100);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

gpuarrayBW — Input image

binary image in a `gpuArray`

Input image, specified as a binary image of class `logical` in a `gpuArray`. The input image can be numeric or logical, but must be 2-D, real and nonsparse.

Example: `BW = imread('circles.png');`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Output Arguments

BW2 — Output image

binary image

Output image returned as a binary image of class `logical`.

gpuarrayBW2 — Output image

binary image in a `gpuArray`

Output image returned as a binary image of class `logical` in a `gpuArray`.

Tips

- To perform erosion or dilation, use the `imerode` or `imdilate` functions. If you want to duplicate the dilation or erosion performed by `bwmorph`, specify the structuring element `ones(3)` with these functions.

Algorithms

When used with the `'thin'` option, `bwmorph` uses the following algorithm [3]:

- 1 In the first subiteration, delete pixel p if and only if the conditions G_1 , G_2 , and G_3 are all satisfied.

- 2 In the second subiteration, delete pixel p if and only if the conditions G_1 , G_2 , and G_3' are all satisfied.

Condition G1:

$$X_H(p) = 1$$

where

$$X_H(p) = \sum_{i=1}^4 b_i$$

$$b_i = \begin{cases} 1, & \text{if } x_{2i-1} = 0 \text{ and } (x_{2i} = 1 \text{ or } x_{2i+1} = 1) \\ 0, & \text{otherwise} \end{cases}$$

x_1, x_2, \dots, x_8 are the values of the eight neighbors of p , starting with the east neighbor and numbered in counter-clockwise order.

Condition G2:

$$2 \leq \min\{n_1(p), n_2(p)\} \leq 3$$

where

$$n_1(p) = \sum_{k=1}^4 x_{2k-1} \vee x_{2k}$$

$$n_2(p) = \sum_{k=1}^4 x_{2k} \vee x_{2k+1}$$

Condition G3:

$$(x_2 \vee x_3 \vee \bar{x}_8) \wedge x_1 = 0$$

Condition G3':

$$(x_6 \vee x_7 \vee \bar{x}_4) \wedge x_5 = 0$$

The two subiterations together make up one iteration of the thinning algorithm. When the user specifies an infinite number of iterations (`n=Inf`), the iterations are repeated until the image stops changing. The conditions are all tested using `applylut` with precomputed lookup tables.

References

- [1] Haralick, Robert M., and Linda G. Shapiro, *Computer and Robot Vision*, Vol. 1, Addison-Wesley, 1992.
- [2] Kong, T. Yung and Azriel Rosenfeld, *Topological Algorithms for Digital Image Processing*, Elsevier Science, Inc., 1996.
- [3] Lam, L., Seong-Whan Lee, and Ching Y. Suen, "Thinning Methodologies-A Comprehensive Survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol 14, No. 9, September 1992, page 879, bottom of first column through top of second column.
- [4] Pratt, William K., *Digital Image Processing*, John Wiley & Sons, Inc., 1991.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

- When generating code, the character vectors specifying the operation must be a compile-time constant and, for best results, the input image must be of class `logical`.

See Also

See Also

`bweuler` | `bwperim` | `gpuArray` | `imdilate` | `imerode`

Introduced before R2006a

bwpack

Pack binary image

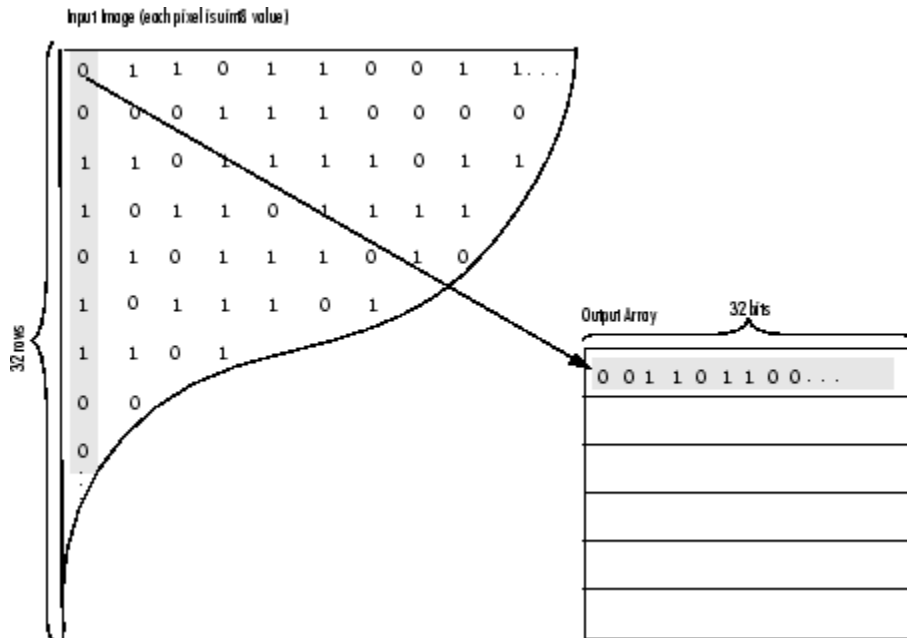
Syntax

`BWP = bwpack(BW)`

Description

`BWP = bwpack(BW)` packs the `uint8` binary image `BW` into the `uint32` array `BWP`, which is known as a *packed binary image*. Because each 8-bit pixel value in the binary image has only two possible values, 1 and 0, `bwpack` can map each pixel to a single bit in the packed output image.

`bwpack` processes the image pixels by column, mapping groups of 32 pixels into the bits of a `uint32` value. The first pixel in the first row corresponds to the least significant bit of the first `uint32` element of the output array. The first pixel in the 32nd input row corresponds to the most significant bit of this same element. The first pixel of the 33rd row corresponds to the least significant bit of the second output element, and so on. If `BW` is `M`-by-`N`, then `BWP` is `ceil(M/32)`-by-`N`. This figure illustrates how `bwpack` maps the pixels in a binary image to the bits in a packed binary image.



Binary image packing is used to accelerate some binary morphological operations, such as dilation and erosion. If the input to `imdilate` or `imerode` is a packed binary image, the functions use a specialized routine to perform the operation faster.

Use `bwunpack` to unpack packed binary images.

Class Support

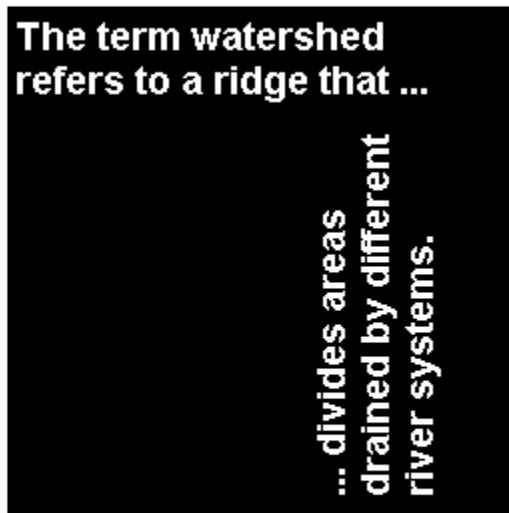
BW can be logical or numeric, and it must be 2-D, real, and nonsparse. BWP is of class `uint32`.

Examples

Pack, Dilate, and Unpack Binary Image

Read binary image into the workspace.

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



Pack the image.

```
BWp = bwpack(BW);
```

Dilate the packed image.

```
BWp_dilated = imdilate(BWp, ones(3,3), 'ispacked');
```

Unpack the dilated image and display it.

```
BW_dilated = bwunpack(BWp_dilated, size(BW,1));  
imshow(BW_dilated)
```




Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. The code generated for this function uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

See Also

See Also

`bwunpack` | `imdilate` | `imerode`

Introduced before R2006a

bwperim

Find perimeter of objects in binary image

Syntax

```
BW2 = bwperim(BW)
BW2 = bwperim(BW,conn)
```

Description

`BW2 = bwperim(BW)` returns a binary image that contains only the perimeter pixels of objects in the input image `BW`. A pixel is part of the perimeter if it is nonzero and it is connected to at least one zero-valued pixel. The default connectivity is 4 for two dimensions, 6 for three dimensions, and `conndef(ndims(BW), 'minimal')` for higher dimensions. If you do not specify a return value, `bwperim` displays the result in a figure window.

`BW2 = bwperim(BW,conn)` where `conn` specifies the desired connectivity.

Examples

Find Perimeter of Objects in Binary Image

Read binary image into workspace.

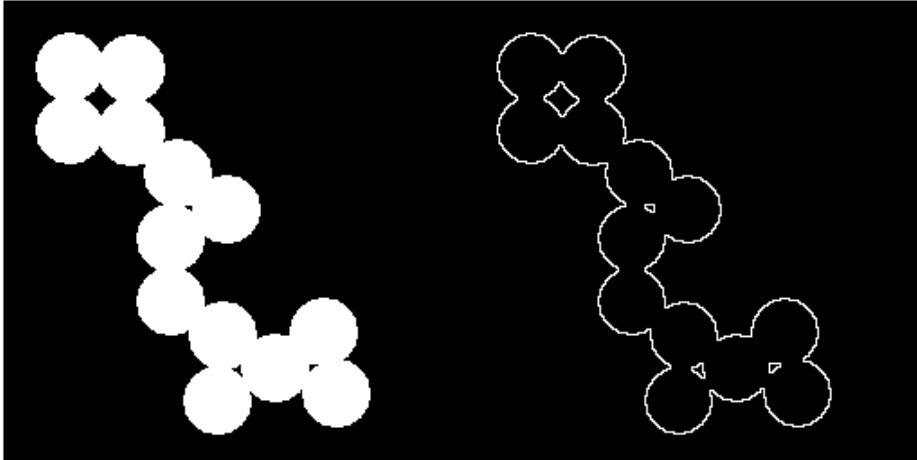
```
BW = imread('circles.png');
```

Calculate the perimeters of objects in the image.

```
BW2 = bwperim(BW,8);
```

Display the original image and the perimeters side-by-side.

```
imshowpair(BW,BW2,'montage')
```



Input Arguments

BW — Input binary image

logical or numeric matrix that must be 2-D, real, and nonsparse

Input binary image, specified as a logical or numeric matrix that must be 2-D, real, and nonsparse.

```
Example: BW = imread('circles.png'); BW2 = bwperim(BW);
```

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

conn — Connectivity

4 for 2-D (default) | 6 | 8 | 18 | 26 | 3-by-3-by-...-by-3 array of zeroes and ones

Connectivity, specified as one of the values in this table or a 3-by-3-by-...-by-3 array of 0s and 1s. The 1-valued elements define neighborhood locations relative to the center element of **conn**. Note that **conn** must be symmetric around its center element.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Example: `BW2 = bwperim(BW,8);`

Data Types: `double` | `logical`

Output Arguments

BW2 — Output binary image containing only perimeter pixels of objects

logical array

Output image containing only perimeter pixels of objects, returned as a logical array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

- `bwperim` supports only 2-D images.
- `bwperim` does not support a no-output-argument syntax.
- The connectivity matrix input argument, `conn`, must be a constant.

See Also

See Also

`bwarea` | `bwboundaries` | `bweuler` | `bwtraceboundary` | `conndef` | `imfill`

Introduced before R2006a

bwpropfilt

Extract objects from binary image using properties

Syntax

```
BW2 = bwpropfilt(BW,attrib,range)
BW2 = bwpropfilt(BW,attrib,n)
BW2 = bwpropfilt(BW,attrib,n,keep)
BW2 = bwpropfilt(BW,I,attrib,___)
BW2 = bwpropfilt(BW,___, conn)
```

Description

`BW2 = bwpropfilt(BW,attrib,range)` extracts all connected components (objects) from a binary image `BW` that meet the criteria specified by `attrib` and `range`. `attrib` is a character vector that identifies a particular property of the objects, such as their area. `range` is a 1-by-2 row vector that specifies the range of values (low and high) of the property. `bwpropfilt` returns a binary image `BW2` containing only those objects that meet the criteria.

`BW2 = bwpropfilt(BW,attrib,n)` sorts the objects based on the value of the specified property, `attrib`, returning a binary image that contains only the top `n` largest objects. In the event of a tie for `n`-th place, `bwpropfilt` keeps only the first `n` objects in `BW2`.

`BW2 = bwpropfilt(BW,attrib,n,keep)` sorts the objects based on `attrib` values, keeping the `n` largest values if `keep` is 'largest' (the default) and the `n` smallest if `keep` is 'smallest'.

`BW2 = bwpropfilt(BW,I,attrib,___)` sorts objects based on the intensity values in the grayscale image `I` and the property `attrib`.

`BW2 = bwpropfilt(BW,___, conn)` specifies the desired connectivity. Connectivity can be either 4, 8, or a 3-by-3 matrix of 0s and 1. The 1-valued elements define neighborhood locations relative to the center element of `conn` and `conn` must be symmetric about its center element.

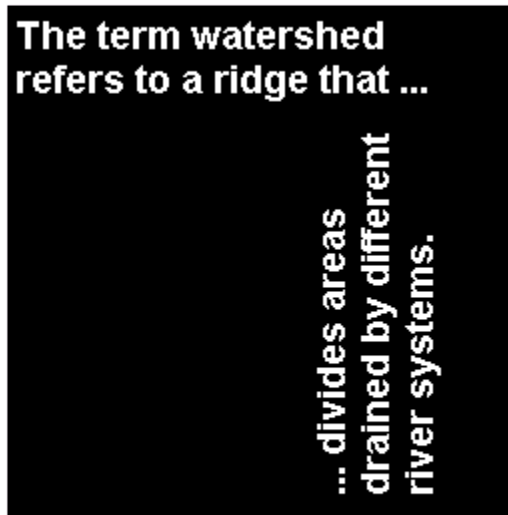
Examples

Find Regions Without Holes

Read image and display it.

```
BW = imread('text.png');  
figure  
imshow(BW)  
title('Original Image')
```

Original Image

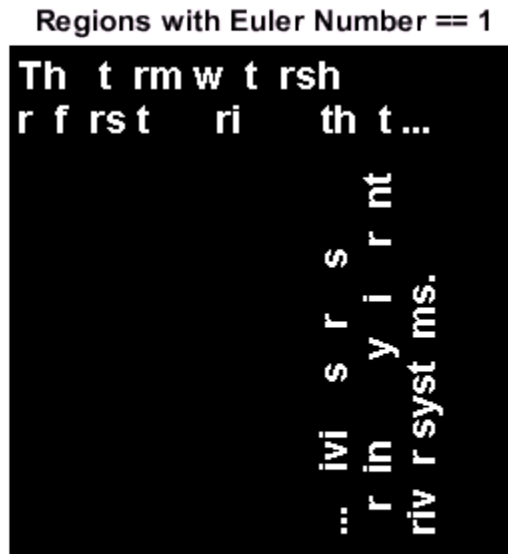


Use filtering to create a second image that contains only those regions in the original image that do not have holes. For these regions, the Euler number property is equal to 1. Display filtered image.

```
BW2 = bwpropfilt(BW, 'EulerNumber', [1 1]);  
figure
```



```
imshow(BW2)
title('Regions with Euler Number == 1')
```



Find Which Ten Objects Have Largest Perimeters

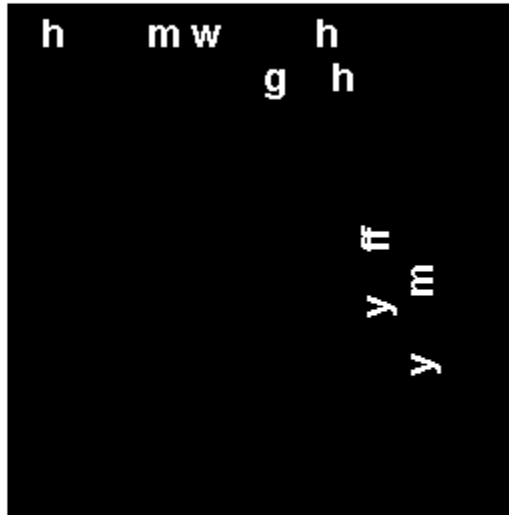
Read image.

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

Find the ten objects in the image with the largest perimeters and display filtered image.

```
BW2 = bwpropfilt(BW, 'perimeter', 10);
figure;
imshow(BW2)
title('Objects with the Largest Perimeters')
```

Objects with the Largest Perimeters



- “Filter Images on Region Properties Using Image Region Analyzer App”

Input Arguments

BW — Image to be filtered

binary image

Image to be filtered, specified as a binary image.

Data Types: `logical`

attrib — Name of attribute on which to filter

character vector

Name of attribute on which to filter, specified as one of the following character vectors. For detailed information about these attributes, see `regionprops`.

Area	EulerNumber	MinorAxisLength
ConvexArea	Extent	Orientation
Eccentricity	FilledArea	Perimeter
EquivDiameter	MajorAxisLength	Solidity

If you specify a grayscale image, `attrib` can have one of these additional values.

MaxIntensity	MeanIntensity	MinIntensity
--------------	---------------	--------------

Data Types: char

range — Minimum and maximum values of the property inclusive

1-by-2 numeric row vector

Minimum and maximum values of the property inclusive, specified as a 1-by-2 numeric vector of the form [low high].

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

conn — Connectivity of objects

3-by-3 matrix of 0s and 1s (default) | 4 | 8

Connectivity of objects in the image, specified as the scalar values 4 or 8, or a 3-by-3 matrix of 0s and 1s. 1-valued elements define neighborhood locations relative to the center element of `conn`, which must be symmetric about its center element.

Data Types: double | logical

n — Number of objects to return

scalar double

Number of object to return, specified as a scalar double.

Data Types: double

keep — Objects to retain

'largest' (default) | 'smallest'

Objects to retain, specified as 'largest' or 'smallest'.

Data Types: char

I — Marker image

grayscale image

Marker image, specified as a grayscale image, the same size as the input binary image. Intensity values in the grayscale image define regions in the input binary image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

BW2 — Filtered image

binary image

Filtered image, returned as a binary image the same size as BW.

See Also

See Also

`bwareafilt` | `bwareaopen` | `bwconncomp` | `conndef` | `regionprops`

Topics

“Filter Images on Region Properties Using Image Region Analyzer App”

Introduced in R2014b

bwselect

Select objects in binary image

Syntax

```
BW2 = bwselect(BW,c,r,n)
BW2 = bwselect(BW,n)
[BW2, idx] = bwselect( ___ )
BW2 = bwselect(x,y,BW,xi,yi,n)
[x,y,BW2,idx,xi,yi] = bwselect( ___ )
```

Description

`BW2 = bwselect(BW,c,r,n)` returns a binary image containing the objects that overlap the pixel (r,c) , where n specifies the connectivity. Objects are connected sets of on pixels, that is, pixels having a value of 1. By default, `bwselect` looks for 4-connected objects.

`BW2 = bwselect(BW,n)` displays the image `BW` on the screen and lets you select the (r,c) coordinates using the mouse. If you omit `BW`, `bwselect` operates on the image in the current axes. Use normal button clicks to add points. Press **Backspace** or **Delete** to remove the previously selected point. A shift-click, right-click, or double-click selects the final point; press **Return** to finish the selection without adding a point.

`[BW2, idx] = bwselect(___)` returns the linear indices of the pixels belonging to the selected objects.

`BW2 = bwselect(x,y,BW,xi,yi,n)` uses the vectors `x` and `y` to establish a nondefault spatial coordinate system for `BW`. The arguments `xi` and `yi` are scalars or equal-length vectors that specify locations in this coordinate system.

`[x,y,BW2,idx,xi,yi] = bwselect(___)` returns the `XData` and `YData` in `x` and `y`, the output image in `BW2`, linear indices of all the pixels belonging to the selected objects in `idx`, and the specified spatial coordinates in `xi` and `yi`.

Examples

Select Objects in Binary Image

Select objects in a binary image and create a new image containing only those objects.

Read binary image into the workspace.

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

Specify the locations of objects in the image using row and column indices.

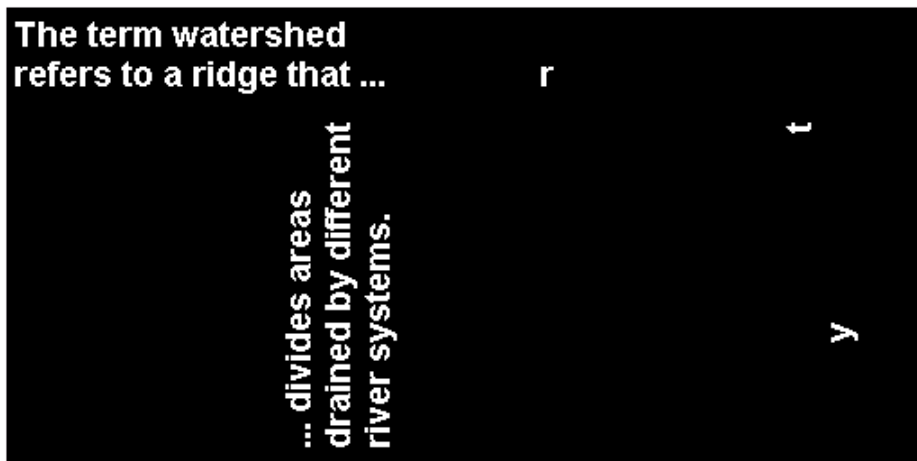
```
c = [43 185 212];  
r = [38 68 181];
```

Create a new binary image containing only the selected objects. This example specifies 4-connected objects.

```
BW2 = bwselect(BW,c,r,4);
```

Display the original image and the new image side-by-side.

```
imshowpair(BW,BW2,'montage');
```



Input Arguments

BW — Input binary image

2-D, nonsparse, logical or numeric matrix

Input binary image, specified as a 2-D, nonsparse, logical or numeric matrix. If you do not specify an output argument, `bwselect` displays the output image in a new figure.

Example: `BW = imread('text.png');`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

c — Column index

numeric scalar or vector

Column index, specified as a numeric scalar or vector. If `c` and `r` are equal-length vectors, `BW2` contains the sets of objects overlapping with any of the pixels $(r(k), c(k))$.

Example: `c = [43 185 212];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

r — Row index

numeric scalar or vector

Row index, specified as a numeric scalar or vector. If `r` and `c` are equal-length vectors, `BW2` contains the sets of objects overlapping with any of the pixels $(r(k), c(k))$.

Example: `r = [38 68 181];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

n — Connectivity

8 (default) | 4

Connectivity, specified as either the value 4 or 8.

Value	Description
4	4-connected objects

Value	Description
8	8-connected objects

Example: `BW2 = bwselect(BW,c,r,4);`

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

x — x coordinates of nondefault coordinate system

numeric scalar or vector

x coordinates of nondefault coordinate system, specified as a numeric scalar or vector.

Example: `x = [19.5 23.5];`

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

y — y coordinates of nondefault coordinate system

numeric scalar or vector

y coordinates of nondefault coordinate system, specified as a numeric scalar or vector.

Example: `y = [8.0 12.0];`

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

xi — x coordinates of locations in nondefault coordinate system

numeric scalar or vector

x coordinates of locations in nondefault coordinate system, specified as a numeric scalar or vector.

Example: `x = [19.5 23.5];`

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

yi — y coordinates of locations in nondefault coordinate system

numeric scalar or vector

y coordinates of locations in nondefault coordinate system, specified as a numeric scalar or vector.

Example: `y = [8.0 12.0];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

BW2 — Binary image containing objects that overlap the specified pixels

logical array

Binary image containing objects that overlap the specified pixels, returned as a logical array.

If you do not specify an output argument, `bwselect` displays the output image in a new figure.

idx — Linear indices of the pixels belonging to the selected objects

numeric vector

Linear indices of the pixels belonging to the selected objects, returned as a numeric vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, `bwselect` only supports the following syntaxes:

- `BW2 = bwselect(BW, c, r)`
- `[BW2, idx] = bwselect(BW, c, r)`
- `BW2 = bwselect(BW, c, r, n)`
- `[BW2, idx] = bwselect(BW, c, r, n)`
- In addition, the optional fourth input argument, `n`, must be a compile-time constant.

See Also

See Also

`bwlabel` | `imfill` | `impixel` | `regionfill` | `roipoly`

Introduced before R2006a

bwtraceboundary

Trace object in binary image

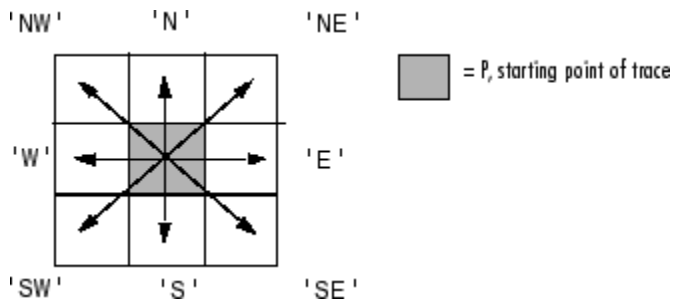
Syntax

```
B = bwtraceboundary(BW,P,fstep)
B = bwtraceboundary(bw,P,fstep,conn)
B = bwtraceboundary(bw,P,fstep,conn,n,dir)
```

Description

`B = bwtraceboundary(BW,P,fstep)` traces the outline of an object in binary image `bw`. Nonzero pixels belong to an object and 0 pixels constitute the background. `P` is a two-element vector specifying the row and column coordinates of the point on the object boundary where you want the tracing to begin.

`fstep` is a character vector specifying the initial search direction for the next object pixel connected to `P`. The following figure illustrates all the possible values for `fstep`.



`bwtraceboundary` returns `B`, a `Q`-by-2 matrix, where `Q` is the number of boundary pixels for the region. `B` holds the row and column coordinates of the boundary pixels.

`B = bwtraceboundary(bw,P,fstep,conn)` specifies the connectivity to use when tracing the boundary. `conn` can have either of the following scalar values.

Value	Meaning
4	4-connected neighborhood Note: With this connectivity, <code>fstep</code> is limited to the following values: 'N', 'E', 'S', and 'W'.
8	8-connected neighborhood. This is the default.

`B = bwtraceboundary(bw,P,fstep,conn,n,dir)` specifies `n`, the maximum number of boundary pixels to extract, and `dir`, the direction in which to trace the boundary. When `n` is set to `Inf`, the default value, the algorithm identifies all the pixels on the boundary. `dir` can have either of the following values:

Value	Meaning
'clockwise'	Search in a clockwise direction. This is the default.
'counterclockwise'	Search in counterclockwise direction.

Class Support

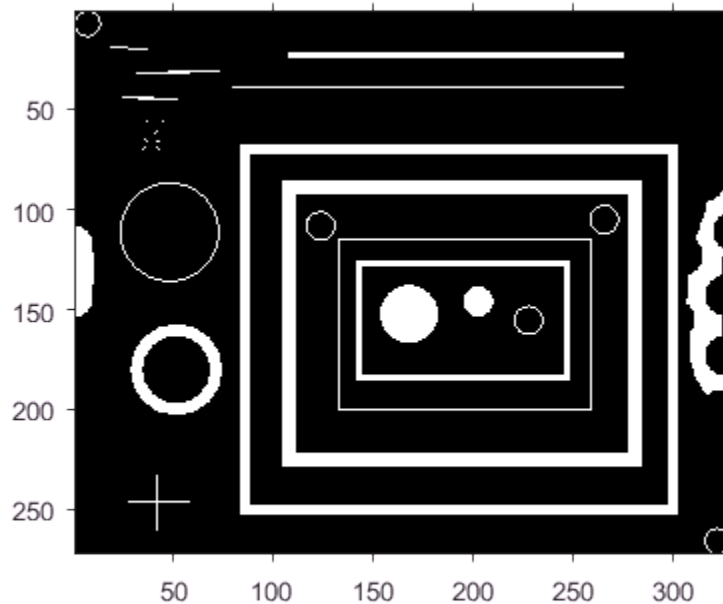
`BW` can be logical or numeric and it must be real, 2-D, and nonsparse. `B`, `P`, `conn`, and `N` are of class `double`. `dir` and `fstep` are character vectors.

Examples

Trace Boundary and Visualize Contours

Read image and display it.

```
BW = imread('blobs.png');
imshow(BW,[]);
```

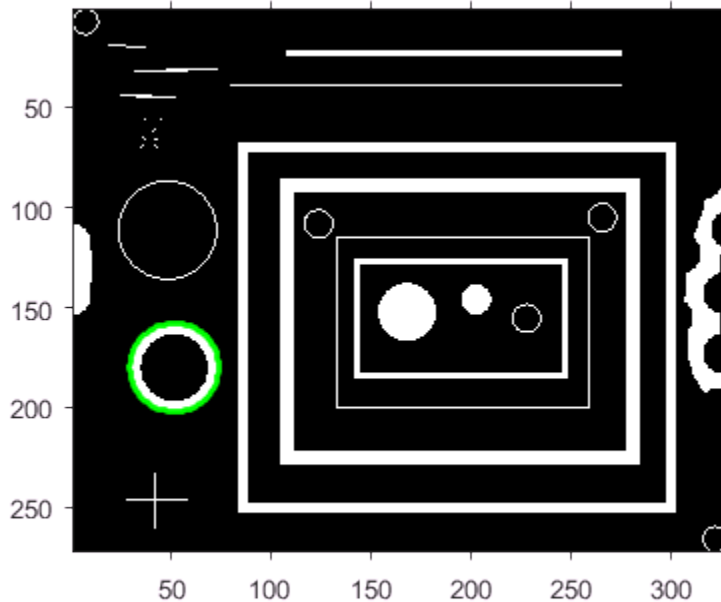


Pick an object in the image and trace the boundary. To select an object, specify a pixel on its boundary. This example uses the coordinates of a pixel on the boundary of the thick white circle, obtained through visual inspection using `impixelinfo`. The example specifies the initial search direction, the connectivity, how many boundary pixels should be returned, and the direction in which to perform the search.

```
r = 163;
c = 37;
contour = bwtraceboundary(BW,[r c], 'W',8,Inf, 'counterclockwise');
```

Plot the contour on the image.

```
hold on;
plot(contour(:,2),contour(:,1), 'g', 'LineWidth',2);
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms

for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

- When generating code, the `dir`, `fstep`, and `conn` arguments must be compile-time constants.

See Also

See Also

`bwboundaries` | `bwperim`

Introduced before R2006a

bwulterode

Ultimate erosion

Syntax

```
BW2 = bwulterode(BW)
BW2 = bwulterode(BW,method,conn)
```

Description

`BW2 = bwulterode(BW)` computes the ultimate erosion of the binary image `BW`. The ultimate erosion of `BW` consists of the regional maxima of the Euclidean distance transform of the complement of `BW`. The default connectivity for computing the regional maxima is 8 for two dimensions, 26 for three dimensions, and `conndef(ndims(BW), 'maximal')` for higher dimensions.

`BW2 = bwulterode(BW,method,conn)` specifies the distance transform method and the regional maxima connectivity. *method* can be one of the following values: 'euclidean', 'cityblock', 'chessboard', and 'quasi-euclidean'.

`conn` can have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can be defined in a more general way for any dimension by using for `conn` a 3-by-3-by... - by-3 matrix of 0's and 1's. The 1-valued elements define neighborhood

locations relative to the center element of `conn`. Note that `conn` must be symmetric about its center element.

Class Support

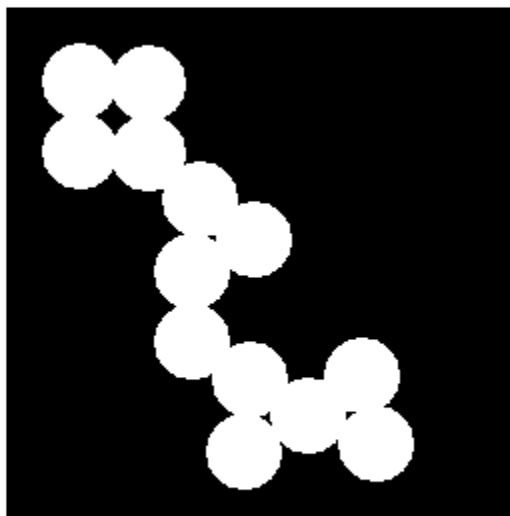
BW can be numeric or logical and it must be nonsparse. It can have any dimension. The return value BW2 is always a logical array.

Examples

Perform Ultimate Erosion of Binary Image

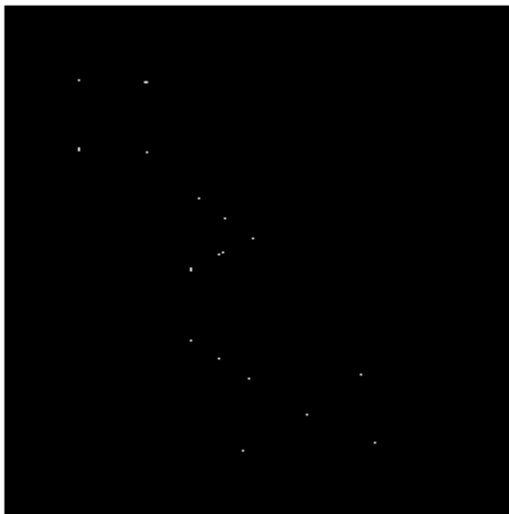
Read a binary image into the workspace and display it.

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



Perform the ultimate erosion of the image and display it.

```
ultimateErosion = bwulterode(originalBW);  
figure, imshow(ultimateErosion)
```



See Also

See Also

[bwdist](#) | [conndef](#) | [imregionalmax](#)

Introduced before R2006a

bwunpack

Unpack binary image

Syntax

```
BW = bwunpack(BWP,m)
```

Description

`BW = bwunpack(BWP,m)` unpacks the packed binary image `BWP`. `BWP` is a `uint32` array. When it unpacks `BWP`, `bwunpack` maps the least significant bit of the first row of `BWP` to the first pixel in the first row of `BW`. The most significant bit of the first element of `BWP` maps to the first pixel in the 32nd row of `BW`, and so on. `BW` is `M`-by-`N`, where `N` is the number of columns of `BWP`. If `m` is omitted, its default value is `32*size(BWP,1)`.

Binary image packing is used to accelerate some binary morphological operations, such as dilation and erosion. If the input to `imdilate` or `imerode` is a packed binary image, the functions use a specialized routine to perform the operation faster.

`bwpack` is used to create packed binary images.

Class Support

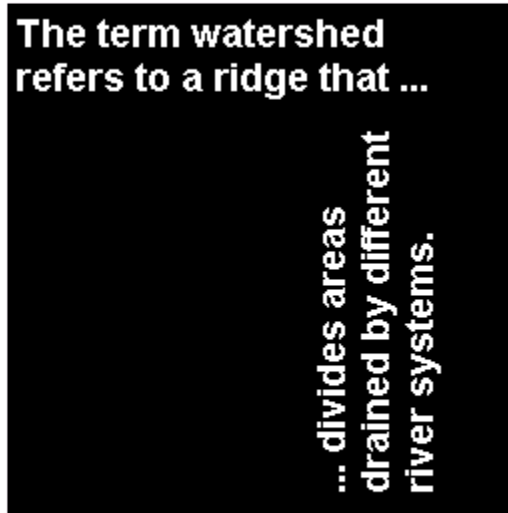
`BWP` is of class `uint32` and must be real, 2-D, and nonsparse. The return value `BW` is of class `uint8`.

Examples

Pack, Dilate, and Unpack Binary Image

Read binary image into the workspace.

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



Pack the image.

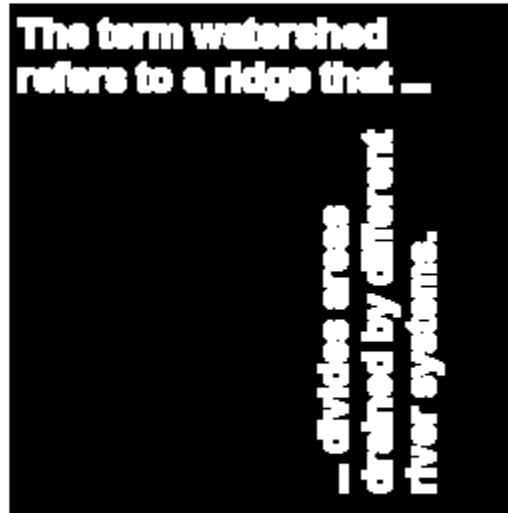
```
BWp = bwpack(BW);
```

Dilate the packed image.

```
BWp_dilated = imdilate(BWp,ones(3,3),'ispacked');
```

Unpack the dilated image and display it.

```
BW_dilated = bwunpack(BWp_dilated, size(BW,1));  
imshow(BW_dilated)
```



Tips

- This function supports the generation of C code using MATLAB Coder. The code generated for this function uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

When generating code, all input arguments must be compile-time constants.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, all input arguments must be compile-time constants.

See Also

`bwpack` | `imdilate` | `imerode`

Introduced before R2006a

checkerboard

Create checkerboard image

Syntax

I = checkerboard
 I = checkerboard(n)
 I = checkerboard(n,p,q)

Description

I = checkerboard creates an 8-by-8 square checkerboard image that has four identifiable corners. The checkerboard pattern is made up of tiles. Each tile contains four squares, each with a default of 10 pixels per side. The light squares on the left half of the checkerboard are white. The light squares on the right half of the checkerboard are gray.

TILE = [DARK LIGHT; LIGHT DARK]



I = checkerboard(n) creates an 8-by-8 square checkerboard image where each square has n pixels per side.

I = checkerboard(n,p,q) creates a rectangular checkerboard image where p specifies the number of rows of tiles and q specifies the number of columns of tiles. If you omit q, the number of columns defaults to p and the checkerboard is square. Each square has n pixels per side.

Examples

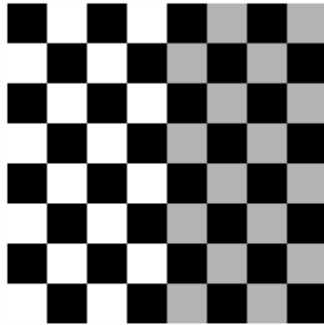
Create Square Checkerboard

Create a checkerboard where the side of every square is 20 pixels in length.

```
I = checkerboard(20);
```

Display the checkerboard.

```
imshow(I)
```



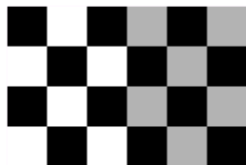
Create Rectangular Checkerboard

Create a rectangular checkerboard that is 2 tiles high and 3 tiles wide. The side of every square is 20 pixels in length.

```
J = checkerboard(20,2,3);
```

Display the checkerboard.

```
figure  
imshow(J)
```



Create Black and White Checkerboard

Create a black and white checkerboard with the default tile size and the default number of rows and columns.

```
K = (checkerboard > 0.5);
```

Display the checkerboard.

```
figure  
imshow(K)
```



Input Arguments

n — Side length in pixels of each square in the checkerboard pattern

10 (default) | positive integer

Side length in pixels of each square in the checkerboard pattern, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

p — Number of rows of tiles in the checkerboard pattern

8 (default) | positive integer

Number of rows of tiles in the checkerboard pattern, specified as a positive integer. Since there are four squares per tile, there are $2 \cdot p$ rows of squares in the checkerboard.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

q — **Number of columns of tiles in the checkerboard pattern**

positive integer

Number of columns of tiles in the checkerboard pattern, specified as a positive integer. If you omit `q`, the value defaults to `p` and the checkerboard is square. Since there are four squares per tile, there are $2*q$ columns of squares in the checkerboard.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

I — **Rectangular image with a checkerboard pattern**

2-D numeric array

Rectangular image with a checkerboard pattern, returned as a 2-D numeric array. The light squares on the left half of the checkerboard are white. The light squares on the right half of the checkerboard are gray.

Data Types: `double`

See Also

See Also

`fitgeotrans` | `imwarp`

Introduced before R2006a

col2im

Rearrange matrix columns into blocks

Syntax

```
A = col2im(B,[m n],[mm nn],'distinct')
A = col2im(B,[m n],[mm nn],'sliding')
```

Description

`A = col2im(B,[m n],[mm nn],'distinct')` rearranges each column of `B` into a distinct `m`-by-`n` block to create the matrix `A` of size `mm`-by-`nn`. If `B = [A11(:) A21(:) A12(:) A22(:)]`, where each column has length `m*n`, then `A = [A11 A12; A21 A22]` where each `Aij` is `m`-by-`n`.

`A = col2im(B,[m n],[mm nn],'sliding')` rearranges the row vector `B` into a matrix of size `(mm-m+1)`-by-`(nn-n+1)`. `B` must be a vector of size `1`-by-`(mm-m+1)*(nn-n+1)`. `B` is usually the result of processing the output of `im2col(...,'sliding')` using a column compression function (such as `sum`).

`col2im(B,[m n],[mm nn])` is the same as `col2im(B,[m n],[mm nn],'sliding')`.

Class Support

`B` can be logical or numeric. The return value `A` is of the same class as `B`.

Examples

Rearrange Matrix Values into Row-wise Orientation

Create a matrix.

```
B = reshape(uint8(1:25),[5 5])'
```

```
B = 5×5 uint8 matrix
```

```
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

Rearrange the values in the matrix into a column-wise arrangement.

```
C = im2col(B,[1 5])
```

```
C = 5×5 uint8 matrix
```

```
 1  6 11 16 21
 2  7 12 17 22
 3  8 13 18 23
 4  9 14 19 24
 5 10 15 20 25
```

Rearrange the values in the matrix back into their original row-wise orientation.

```
A = col2im(C,[1 5],[5 5],'distinct')
```

```
A = 5×5 uint8 matrix
```

```
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

See Also

[blockproc](#) | [im2col](#) | [colfilt](#) | [nlfilter](#)

Introduced before R2006a

colfilt

Columnwise neighborhood operations

Syntax

```
B = colfilt(A,[m n],block_type,fun)
B = colfilt(A,[m n],[mblock nblock],block_type,fun)
B = colfilt(A,'indexed',...)
```

Description

`B = colfilt(A,[m n],block_type,fun)` processes the image `A` by rearranging each `m`-by-`n` block of `A` into a column of a temporary matrix, and then applying the function `fun` to this matrix. `fun` must be a function handle. The function `colfilt` zero-pads `A`, if necessary.

Before calling `fun`, `colfilt` calls `im2col` to create the temporary matrix. After calling `fun`, `colfilt` rearranges the columns of the matrix back into `m`-by-`n` blocks using `col2im`.

`block_type` is one of the values listed in this table.

Note: `colfilt` can perform operations similar to `blockproc` and `nlfilter`, but often executes much faster.

Value	Description
'distinct'	Rearranges each <code>m</code> -by- <code>n</code> distinct block of <code>A</code> into a column in a temporary matrix, and then applies the function <code>fun</code> to this matrix. <code>fun</code> must return a matrix the same size as the temporary matrix. <code>colfilt</code> then rearranges the columns of the matrix returned by <code>fun</code> into <code>m</code> -by- <code>n</code> distinct blocks.
'sliding'	Rearranges each <code>m</code> -by- <code>n</code> sliding neighborhood of <code>A</code> into a column in a temporary matrix, and then applies the function <code>fun</code> to this matrix. <code>fun</code> must return a row vector containing a single value for each column

Value	Description
	in the temporary matrix. (Column compression functions such as <code>sum</code> return the appropriate type of output.) <code>colfilt</code> then rearranges the vector returned by <code>fun</code> into a matrix the same size as <code>A</code> .

`B = colfilt(A,[m n],[mblock nblock],block_type,fun)` processes the matrix `A` as above, but in blocks of size `mblock`-by-`nblock` to save memory. Note that using the `[mblock nblock]` argument does not change the result of the operation.

`B = colfilt(A,'indexed',...)` processes `A` as an indexed image, padding with 0's if the class of `A` is `uint8` or `uint16`, or 1's if the class of `A` is `double` or `single`.

Note To save memory, the `colfilt` function might divide `A` into subimages and process one subimage at a time. This implies that `fun` may be called multiple times, and that the first argument to `fun` may have a different number of columns each time.

Class Support

The input image `A` can be of any class supported by `fun`. The class of `B` depends on the class of the output from `fun`.

Examples

Perform Columnwise Neighborhood Filtering on Image

This example shows how to set each output pixel to the mean value of the input pixel's 5-by-5 neighborhood using columnwise neighborhood processing.

Read a grayscale image into the workspace.

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

Perform columnwise filtering. The function `mean` is called on each 5-by-5 pixel neighborhood.

```
I2 = uint8(colfilt(I,[5 5],'sliding',@mean));
```

Display the original image and the filtered image.

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

Original Image



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

Filtered Image



See Also

`blockproc` | `col2im` | `im2col` | `nlfilter`

Topics

“Anonymous Functions” (MATLAB)
“Parameterizing Functions” (MATLAB)
“Create Function Handle” (MATLAB)

Introduced before R2006a

colorcloud

Display 3-D color gamut as point cloud in specified color space

Syntax

```
colorcloud(rgb)
colorcloud(rgb,colorspace)
colorcloud( ____,Name,Value)
hPanel = colorcloud( ____ )
```

Description

`colorcloud(rgb)` displays the full color gamut of the color image `rgb` as a point cloud. By default, `colorcloud` uses the RGB color space.

`colorcloud(rgb,colorspace)` displays the full color gamut of the color image `rgb` as a point cloud in the color space specified by `colorspace`.

`colorcloud(____,Name,Value)` displays the full color gamut using name-value pairs to control aspects of the visualization.

`hPanel = colorcloud(____)` returns the uipanel object created by `colorcloud`.

Examples

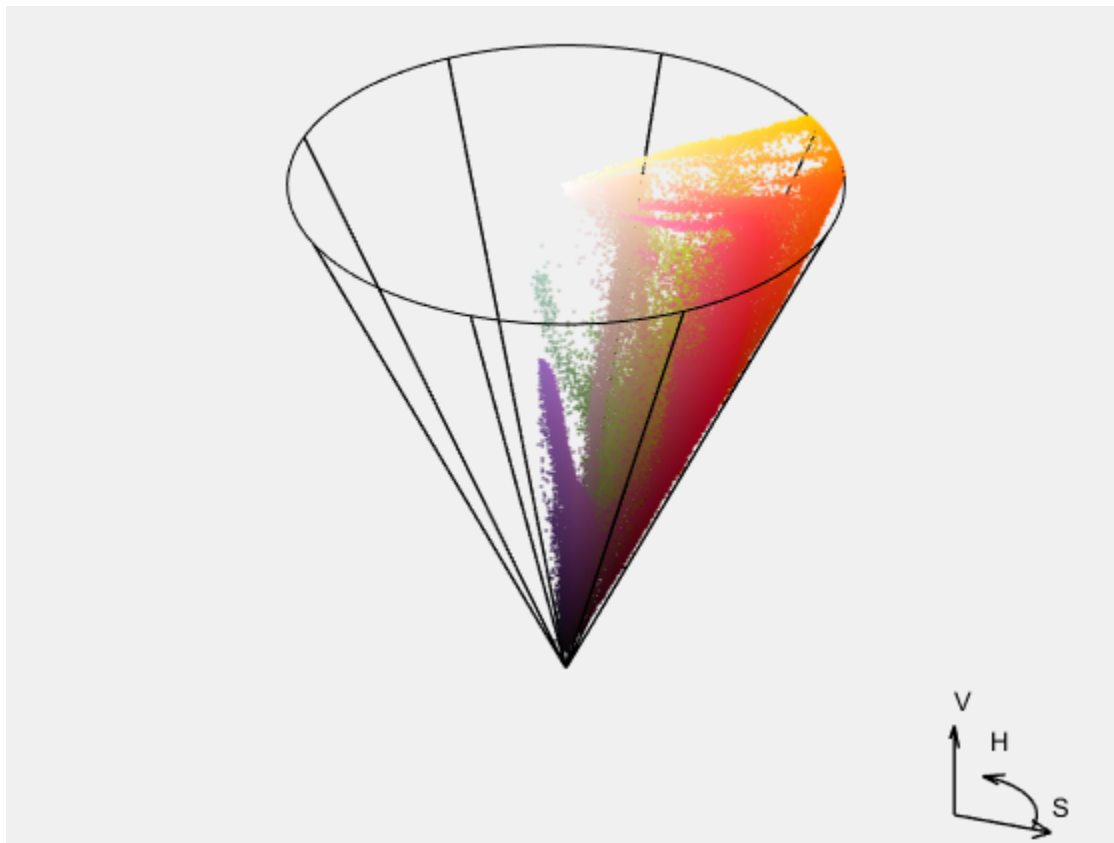
View 3D Color Gamut of RGB Image in HSV Color Space

Read in RGB image

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

View color gamut

```
colorcloud(RGB,'hsv');
```



Input Arguments

rgb — Color image

m-by-*n*-by-3 array

Color image, specified as an *m*-by-*n*-by-3 array.

Data Types: single | double | uint8 | uint16

colorspace — Colorspace name

'rgb' (default) | 'hsv' | 'ycbcr' | 'lab'

Colorspace name, specified as one of the following values:

Value	Description
'hsv'	Color gamut in HSV color space
'lab'	Color gamut in CIE 1976 L*a*b* color space
'rgb'	Color gamut in RGB color space
'ycbcr'	Color gamut in YCbCr color space

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:

'Parent' — Parent of the object created by `colorcloud`
new figure (default)

Parent of the object created by `colorcloud`, specified as a figure or uipanel object. If you do not specify a valid object, `colorcloud` creates a new figure window.

'BackgroundColor' — Color used as background to the color cloud
[0.94 0.94 0.94] (default) | colorspec

Color used as background to the color cloud, specified as a MATLAB `ColorSpec`.

'WireFrameColor' — Color of the color space wire frame
'black' (default) | colorspec

Color of the color space wire frame, defined as MATLAB `ColorSpec`. If you specify the value 'none', `colorcloud` deletes the wire frame.

'OrientationAxesColor' — Color of the orientation axes and labels
'black' (default) | colorspec

Color of the orientation axes and labels, specified as a MATLAB `ColorSpec`. If you specify the value 'none', `colorcloud` deletes the labels.

Output Arguments

hPane1 — Color gamut point cloud

uipanel object

Color gamut point cloud, returned as a uipanel object.

See Also

Introduced in R2016b

conndef

Create connectivity array

Syntax

```
conn = conndef(num_dims,type)
```

Description

`conn = conndef(num_dims,type)` returns the connectivity array defined by `type` for `num_dims` dimensions. Several Image Processing Toolbox functions use `conndef` to create the default connectivity input argument.

Examples

Create 2-D Connectivity Array with Minimal Connectivity

Create a 2-D connectivity array.

```
conn = conndef(2,'minimal')
```

```
conn =
```

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

Create 2-D Connectivity Array with Maximal Connectivity

Create a 2-D connectivity array.

```
conn = conndef(2,'maximal')
```

```
conn =
```

```
1    1    1
1    1    1
1    1    1
```

Create 3-D Connectivity Array with Minimal Connectivity

Create a 3-D connectivity array.

```
conndef(3, 'minimal')
```

```
ans =
```

```
(:,: ,1) =
```

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

```
(:,: ,2) =
```

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

```
(:,: ,3) =
```

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

Input Arguments

num_dims — Number of dimensions

numeric scalar

Number of dimensions, specified as a numeric scalar.

Example: `conn1 = conndef(2, 'minimal')`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

type — Type of neighborhood connectivity

'minimal' | 'maximal'

Type of neighborhood connectivity, specified as either 'minimal' or 'maximal'

Value	Description
'minimal'	Defines a neighborhood whose neighbors are touching the central element on an (N-1)-dimensional surface, for the N-dimensional case.
'maximal'	Defines a neighborhood including neighbors that touch the central element in any way; it is <code>ones(repmat(3,1,NUM_DIMS))</code> .

Example: `conn1 = conndef(2,'minimal')`

Data Types: `char`

Output Arguments

conn — Connectivity matrix

3-by-3-by...-3 logical array

Connectivity matrix, returned as a 3-by-3-...-by-3 logical array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, the `num_dims` and `type` arguments must be compile-time constants.

See Also

Introduced before R2006a

convmtx2

2-D convolution matrix

Syntax

```
T = convmtx2(H,m,n)
T = convmtx2(H,[m n])
```

Description

`T = convmtx2(H,m,n)` returns the convolution matrix `T` for the matrix `H`. If `X` is an `m`-by-`n` matrix, then `reshape(T*X(:),size(H)+[m n]-1)` is the same as `conv2(X,H)`.

`T = convmtx2(H,[m n])` returns the convolution matrix, where the dimensions `m` and `n` are a two-element vector.

Class Support

The inputs are all of class `double`. The output matrix `T` is of class `sparse`. The number of nonzero elements in `T` is no larger than `prod(size(H))*m*n`.

See Also

See Also

`conv2` | `convmtx`

Introduced before R2006a

corner

Find corner points in image

Compatibility

`corner` is not recommended. Use `detectHarrisFeatures` or `detectMinEigenFeatures` in Computer Vision System Toolbox™ instead.

Syntax

```
C = corner(I)
C = corner(I,method)
C = corner(I,N)
C = corner(I,method,N)
C = corner( ____,Name,Value,...)
```

Description

`C = corner(I)` detects corners in image `I` and returns them in matrix `C`.

`C = corner(I,method)` detects corners in image `I` using the specified `method`.

`C = corner(I,N)` detects corners in image `I` and returns a maximum of `N` corners.

`C = corner(I,method,N)` detects corners using the specified `method` and maximum number of corners.

`C = corner(____,Name,Value,...)` specifies parameters and corresponding values that control various aspects of the corner detection algorithm.

Input Arguments

I

A grayscale or binary image.

method

The algorithm used to detect corners. Supported methods are:

- 'Harris': The Harris corner detector.
- 'MinimumEigenvalue': Shi & Tomasi's minimum eigenvalue method.

Default: 'Harris'

N

The maximum number of corners the `corner` function can return.

Default: 200

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

'FilterCoefficients'

A vector, `V`, of filter coefficients for the separable smoothing filter. The outer product, `V*V'`, gives the full filter kernel. The length of the vector must be odd and at least 3.

Default: `fspecial('gaussian',[5 1],1.5)`

'QualityLevel'

A scalar value, `Q`, where $0 < Q < 1$, specifying the minimum accepted quality of corners. When candidate corners have corner metric values less than $Q * \max(\text{corner metric})$, the toolbox rejects them. Use larger values of `Q` to remove erroneous corners.

Default: 0.01

'SensitivityFactor'

A scalar value, `K`, where $0 < K < 0.25$, specifying the sensitivity factor used in the Harris detection algorithm. The smaller the value of `K`, the more likely the algorithm is to detect sharp corners. Use this parameter with the 'Harris' method only.

Default: 0.04

Output Arguments

C

An M -by-2 matrix containing the X and Y coordinates of the corner points detected in **I**.

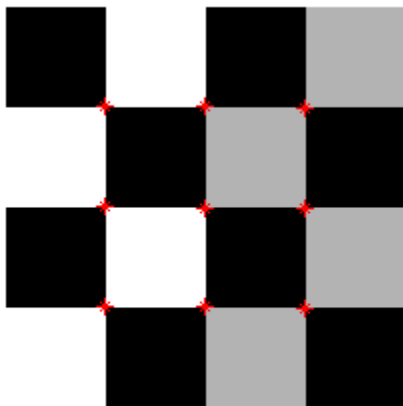
Class Support

I is a nonsparse numeric array. **C** is a matrix of class `double`.

Examples

Find and plot corner points in a checkerboard image.

```
I = checkerboard(50,2,2);  
C = corner(I);  
imshow(I);  
hold on  
plot(C(:,1), C(:,2), 'r*');
```



Tips

The `corner` and `cornermetric` functions both detect corners in images. For most applications, use the streamlined `corner` function to find corners in one step. If you want greater control over corner selection, use the `cornermetric` function to compute a corner metric matrix and then write your own algorithm to find peak values.

Algorithms

The `corner` function performs nonmaxima suppression on candidate corners, and corners are at least two pixels apart.

Introduced in R2010b

cornermetric

Create corner metric matrix from image

Compatibility

`cornermetric` is not recommended. Use `detectHarrisFeatures` or `detectMinEigenFeatures` and the `cornerPoints` class in Computer Vision System Toolbox instead.

Description

`C = cornermetric(I)` generates a corner metric matrix for the grayscale or logical image `I`. The corner metric, `C`, is used to detect corner features in `I` and is the same size as `I`. Larger values in `C` correspond to pixels in `I` with a higher likelihood of being a corner feature.

`C = cornermetric(I, method)` generates a corner metric matrix for the grayscale or logical image `I` using the specified `method`. Valid values for `method` are:

Value	Description
'Harris'	The Harris corner detector. This is the default method.
'MinimumEigenvalue'	Shi and Tomasi's minimum eigenvalue method.

`C = cornermetric(..., param1, val1, param2, val2, ...)` generates a corner metric matrix for `I`, specifying parameters and corresponding values that control various aspects of the corner metric calculation algorithm. Parameters include:

Parameter	Description
'FilterCoefficients'	A vector, <code>V</code> , of filter coefficients for the separable smoothing filter. This parameter is valid with the 'Harris' and 'MinimumEigenvalue' methods.

Parameter	Description
	The outer product, $V \cdot V'$, gives the full filter kernel. The length of the vector must be odd and at least 3. The default is <code>fspecial('gaussian',[5 1],1.5)</code> .
'SensitivityFactor'	A scalar k , $0 < k < 0.25$, specifying the sensitivity factor used in the Harris detection algorithm. For smaller values of k , the algorithm is more likely to detect sharper corners. This parameter is only valid with the 'Harris' method. Default value: 0.04

Class Support

I is a nonsparse numeric array. C is a matrix of class `double`.

Examples

Find Corner Features in Grayscale Image

Read image and use part of it for processing.

```
I = imread('pout.tif');
I = I(1:150,1:120);
subplot(1,3,1);
imshow(I);
title('Original Image');
```

Original Image

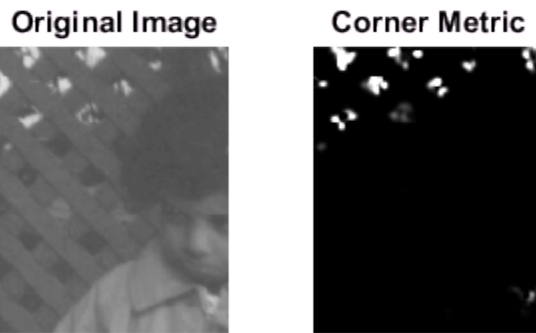


Generate a corner metric matrix.

```
C = cornermetric(I);
```

Adjust the corner metric for viewing.

```
C_adjusted = imadjust(C);  
subplot(1,3,2);  
imshow(C_adjusted);  
title('Corner Metric');
```

Find and display corner features.

```
corner_peaks = imregionalmax(C);  
corner_idx = find(corner_peaks == true);  
[r g b] = deal(I);  
r(corner_idx) = 255;  
g(corner_idx) = 255;  
b(corner_idx) = 0;  
RGB = cat(3,r,g,b);  
subplot(1,3,3);  
imshow(RGB);  
title('Corner Points');
```

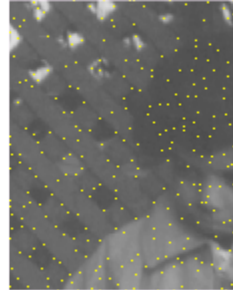
Original Image



Corner Metric



Corner Points



Tips

The `corner` and `cornermetric` functions both detect corners in images. For most applications, use the streamlined `corner` function to find corners in one step. If you want greater control over corner selection, use the `cornermetric` function to compute a corner metric matrix and then write your own algorithm to find peak values.

See Also

See Also

edge

Introduced in R2008b

corr2

2-D correlation coefficient

Syntax

```
r = corr2(A,B)
r = corr2(gpuarrayA,gpuarrayB)
```

Description

`r = corr2(A,B)` returns the correlation coefficient `r` between `A` and `B`, where `A` and `B` are matrices or vectors of the same size. `r` is a scalar **double**.

`r = corr2(gpuarrayA,gpuarrayB)` performs the operation on a GPU. The input images are 2-D `gpuArrays` of the same size. `r` is a scalar **double** `gpuArray`. This syntax requires the Parallel Computing Toolbox.

Class Support

`A` and `B` can be numeric or logical. The return value `r` is a scalar **double**.

`gpuarrayA` and `gpuarrayB` must be real, 2-D `gpuArrays`. If either `A` or `B` is not a `gpuArray`, it must be numeric or logical and nonsparse. `corr2` moves any data not already on the GPU to the GPU. `R` is a scalar **double** `gpuArray`.

Examples

Compute the correlation coefficient

Compute the correlation coefficient between an image and the same image processed with a median filter.

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

```
J = medfilt2(I);
R = corr2(I,J)

R = 0.9959
```

Compute the Correlation Coefficient on a GPU

Compute the correlation coefficient on a GPU between an image and the same image processed using standard deviation filtering.

```
I = gpuArray(imread('pout.tif'));
J = stdfilt(I);
R = corr2(I,J)

R =

    0.2762
```

Algorithms

corr2 computes the correlation coefficient using

$$r = \frac{\sum_m \sum_n (A_{mn} - \bar{A})(B_{mn} - \bar{B})}{\sqrt{\left(\sum_m \sum_n (A_{mn} - \bar{A})^2\right) \left(\sum_m \sum_n (B_{mn} - \bar{B})^2\right)}}$$

where $\bar{A} = \text{mean2}(A)$, and $\bar{B} = \text{mean2}(B)$.

See Also

See Also

corrcoef | gpuArray | std2

Introduced before R2006a

cp2tform

Infer spatial transformation from control point pairs

Compatibility

cp2tform is not recommended. Use `fitgeotrans` instead.

Syntax

```
TFORM = cp2tform(movingPoints, fixedPoints, transformtype)
TFORM = cp2tform(CPSTRUCT, transformtype)
[TFORM, movingPoints, fixedPoints] = cp2tform(CPSTRUCT, ...)
TFORM = cp2tform(movingPoints, fixedPoints, 'polynomial', order)
TFORM = cp2tform(CPSTRUCT, 'polynomial', order)
TFORM = cp2tform(movingPoints, fixedPoints, 'piecewise linear')
TFORM = cp2tform(CPSTRUCT, 'piecewise linear')
TFORM = cp2tform(movingPoints, fixedPoints, 'lwm', N)
TFORM = cp2tform(CPSTRUCT, 'lwm', N)
[TFORM, movingPoints, fixedPoints, movingPoints_bad,
fixedPoints_bad] = cp2tform(movingPoints, fixedPoints, 'piecewise
linear')
[TFORM, movingPoints, fixedPoints, movingPoints_bad,
fixedPoints_bad] = cp2tform(CPSTRUCT, 'piecewise linear')
```

Description

`TFORM = cp2tform(movingPoints, fixedPoints, transformtype)` infers a spatial transformation from control point pairs and returns this transformation as a `TFORM` structure.

`TFORM = cp2tform(CPSTRUCT, transformtype)` works on a `CPSTRUCT` structure that contains the control point matrices for the moving and fixed images. The Control Point Selection Tool, `cpselect`, creates the `CPSTRUCT`.

[TFORM, movingPoints, fixedPoints] = cp2tform(CPSTRUCT, ...) returns the control points that were used in movingPoints and fixedPoints. Unmatched and predicted points are not used. See cpstruct2pairs.

TFORM = cp2tform(movingPoints, fixedPoints, 'polynomial', order) lets you specify the order of the polynomials to use.

TFORM = cp2tform(CPSTRUCT, 'polynomial', order) works on a CPSTRUCT structure.

TFORM = cp2tform(movingPoints, fixedPoints, 'piecewise linear') creates a Delaunay triangulation of the fixed control points, and maps corresponding moving control points to the fixed control points. The mapping is linear (affine) for each triangle and continuous across the control points but not continuously differentiable as each triangle has its own mapping.

TFORM = cp2tform(CPSTRUCT, 'piecewise linear') works on a CPSTRUCT structure.

TFORM = cp2tform(movingPoints, fixedPoints, 'lwm', N) creates a mapping by inferring a polynomial at each control point using neighboring control points. The mapping at any location depends on a weighted average of these polynomials. You can optionally specify the number of points, N, used to infer each polynomial. The N closest points are used to infer a polynomial of order 2 for each control point pair. If you omit N, it defaults to 12. N can be as small as 6, but making N small risks generating ill-conditioned polynomials.

TFORM = cp2tform(CPSTRUCT, 'lwm', N) works on a CPSTRUCT structure.

[TFORM, movingPoints, fixedPoints, movingPoints_bad, fixedPoints_bad] = cp2tform(movingPoints, fixedPoints, 'piecewise linear') returns the control points that were used in movingPoints and fixedPoints and the control points that were eliminated because they were middle vertices of degenerate fold-over triangles in movingPoints_bad and fixedPoints_bad.

[TFORM, movingPoints, fixedPoints, movingPoints_bad, fixedPoints_bad] = cp2tform(CPSTRUCT, 'piecewise linear') works on a CPSTRUCT structure.

Input Arguments

movingPoints

m-by-2, **double** matrix containing the *x*- and *y*-coordinates of control points in the image you want to transform.




fixedPoints



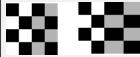

m-by-2, **double** matrix containing the *x*- and *y*-coordinates of control points in the fixed image.

transformtype

Specifies the type of spatial transformation to infer. The `cp2tform` function requires a minimum number of control point pairs to infer a structure of each transform type. The following table lists all the transformation types supported by `cp2tform` in order of complexity. The `'lwm'` and `'polynomial'` transform types can each take an optional, additional parameter.

Transformation Types

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	

Transformation Type	Description	Minimum Number of Control Point Pairs	Example
'projective'	Use this transformation when the scene appears tilted. Straight lines remain straight, but parallel lines converge toward vanishing points that might or might not fall within the image.	4	
'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)	

CPSTRUCT

Structure containing control point matrices for the moving and fixed images. Use the Control Point Selection Tool (`cpselect`) to create the CPSTRUCT.

'polynomial', order

Specifies the order of polynomials to use. `order` can be 2, 3, or 4.

Default: 3

'piecewise linear'

Linear for each piece and continuous, not continuously differentiable.

'lwm'

Local weighted mean.

N

Number of points.

Output Arguments

TFORM

Structure containing the spatial transformation.

movingPoints

Moving control points that were used to infer the spatial transformation. Unmatched and predicted points are not used.

fixedPoints

Fixed control points that were used to infer the spatial transformation. Unmatched and predicted points are not used.

movingPoints_bad

moving control points that were eliminated because they were determined to be outliers.

fixedPoints_bad

fixed control points that were eliminated because they were determined to be outliers.

Examples

Transform an image, use the `cp2tform` function to return the transformation, and compare the angle and scale of the `TFORM` to the angle and scale of the original transformation:

```
I = checkerboard;
J = imrotate(I,30);
fixedPoints = [11 11; 41 71];
movingPoints = [14 44; 70 81];
cpselect(J,I,movingPoints,fixedPoints);

t = cp2tform(movingPoints,fixedPoints,'nonreflective similarity');

% Recover angle and scale by checking how a unit vector
% parallel to the x-axis is rotated and stretched.
u = [0 1];
v = [0 0];
```

```
[x, y] = tformfwd(t, u, v);
dx = x(2) - x(1);
dy = y(2) - y(1);
angle = (180/pi) * atan2(dy, dx)
scale = 1 / sqrt(dx^2 + dy^2)
```

Tips

- When `transformtype` is 'nonreflective similarity', 'similarity', 'affine', 'projective', or 'polynomial', and `movingPoints` and `fixedPoints` (or `CPSTRUCT`) have the minimum number of control points needed for a particular transformation, `cp2tform` finds the coefficients exactly.
- If `movingPoints` and `fixedPoints` have more than the minimum number of control points, a least-squares solution is found. See `mldivide`.
- When either `movingPoints` or `fixedPoints` has a large offset with respect to their origin (relative to range of values that it spans), `cp2tform` shifts the points to center their bounding box on the origin before fitting a `TFORM` structure. This enhances numerical stability and is handled transparently by wrapping the origin-centered `TFORM` within a custom `TFORM` that automatically applies and undoes the coordinate shift as needed. As a result, `fields(T)` can give different results for different coordinate inputs, even for the same transformation type.

Algorithms

`cp2tform` uses the following general procedure:

- 1 Use valid pairs of control points to infer a spatial transformation or an inverse mapping from output space (x,y) to input space (x,y) according to `transformtype`.
- 2 Return the `TFORM` structure containing spatial transformation.

The procedure varies depending on the `transformtype`.

Nonreflective Similarity

Nonreflective similarity transformations can include a rotation, a scaling, and a translation. Shapes and angles are preserved. Parallel lines remain parallel. Straight lines remain straight.

Let

```
sc = scale*cos(angle)
ss = scale*sin(angle)
```

$$[u \ v] = [x \ y \ 1] * \begin{bmatrix} sc & -ss \\ ss & sc \\ tx & ty \end{bmatrix}$$

Solve for `sc`, `ss`, `tx`, and `ty`.

Similarity

Similarity transformations can include rotation, scaling, translation, and reflection. Shapes and angles are preserved. Parallel lines remain parallel. Straight lines remain straight.

Let

```
sc = s*cos(theta)
ss = s*sin(theta)
```

$$[u \ v] = [x \ y \ 1] * \begin{bmatrix} sc & -a*ss \\ ss & a*sc \\ tx & ty \end{bmatrix}$$

Solve for `sc`, `ss`, `tx`, `ty`, and `a`. If `a = -1`, reflection is included in the transformation. If `a = 1`, reflection is not included in the transformation.

Affine

In an affine transformation, the *x* and *y* dimensions can be scaled or sheared independently and there can be a translation. Parallel lines remain parallel. Straight lines remain straight. Nonreflective similarity transformations are a subset of affine transformations.

For an affine transformation,

$$[u \ v] = [x \ y \ 1] * T_{inv}$$

`Tinv` is a 3-by-2 matrix. Solve for the six elements of `Tinv`:

```
t_affine = cp2tform(movingPoints,fixedPoints,'affine');
```

The coefficients of the inverse mapping are stored in `t_affine.tdata.Tinv`.

At least three control-point pairs are needed to solve for the six unknown coefficients.

Projective

In a projective transformation, quadrilaterals map to quadrilaterals. Straight lines remain straight. Affine transformations are a subset of projective transformations.

For a projective transformation,

$$[u\ v\ w] = [x\ y\ w] * Tinv$$

where

$$u = up/wp$$

$$v = vp/wp$$

`Tinv` is a 3-by-3 matrix.

Assuming

$$Tinv = \begin{bmatrix} A & D & G; \\ B & E & H; \\ C & F & I \end{bmatrix};$$

$$u = (Ax + By + C)/(Gx + Hy + I)$$

$$v = (Dx + Ey + F)/(Gx + Hy + I)$$

Solve for the nine elements of `Tinv`:

```
t_proj = cp2tform(movingPoints, fixedPoints, 'projective');
```

The coefficients of the inverse mapping are stored in `t_proj.tdata.Tinv`.

At least four control-point pairs are needed to solve for the nine unknown coefficients.

Note: An affine or projective transformation can also be expressed like this, for a 3-by-2 `Tinv`:

$$[u\ v]' = Tinv' * [x\ y\ 1]'$$

Or, like this, for a 3-by-3 `Tinv`:

$$[u\ v\ 1]' = Tinv' * [x\ y\ 1]'$$

Polynomial

In a polynomial transformation, polynomial functions of x and y determine the mapping.

Second-Order Polynomials

For a second-order polynomial transformation,

$$[u \ v] = [1 \ x \ y \ x*y \ x^2 \ y^2] * T_{inv}$$

Both u and v are second-order polynomials of x and y . Each second-order polynomial has six terms. To specify all coefficients, T_{inv} has size 6-by-2.

```
t_poly_ord2 = cp2tform(movingPoints, fixedPoints, 'polynomial');
```

The coefficients of the inverse mapping are stored in `t_poly_ord2.tdata`.

At least six control-point pairs are needed to solve for the 12 unknown coefficients.

Third-Order Polynomials

For a third-order polynomial transformation:

$$[u \ v] = [1 \ x \ y \ x*y \ x^2 \ y^2 \ y*x^2 \ x*y^2 \ x^3 \ y^3] * T_{inv}$$

Both u and v are third-order polynomials of x and y . Each third-order polynomial has 10 terms. To specify all coefficients, T_{inv} has size 10-by-2.

```
t_poly_ord3 = cp2tform(movingPoints, fixedPoints, 'polynomial',3);
```

The coefficients of the inverse mapping are stored in `t_poly_ord3.tdata`.

At least ten control-point pairs are needed to solve for the 20 unknown coefficients.

Fourth-Order Polynomials

For a fourth-order polynomial transformation:

$$[u \ v] = [1 \ x \ y \ x*y \ x^2 \ y^2 \ y*x^2 \ x*y^2 \ x^3 \ y^3 \ x^3*y \ x^2*y^2 \ x*y^3 \ x^4 \ y^4] * T_{inv}$$

Both u and v are fourth-order polynomials of x and y . Each fourth-order polynomial has 15 terms. To specify all coefficients, T_{inv} has size 15-by-2.

```
t_poly_ord4 = cp2tform(movingPoints, fixedPoints, 'polynomial',4);
```

Fourth-Order Polynomials

The coefficients of the inverse mapping are stored in `t_poly_ord4.tdata`.

At least 15 control-point pairs are needed to solve for the 30 unknown coefficients.

Piecewise Linear

In a piecewise linear transformation, linear (affine) transformations are applied separately to each triangular region of the image[1].

- 1 Find a Delaunay triangulation of the fixed control points.
- 2 Using the three vertices of each triangle, infer an affine mapping from fixed to moving coordinates.

Note At least four control-point pairs are needed. Four pairs result in two triangles with distinct mappings.

Local Weighted Mean

For each control point in `fixedPoints`:

- 1 Find the N closest control points.
- 2 Use these N points and their corresponding points in `movingPoints` to infer a second-order polynomial.
- 3 Calculate the radius of influence of this polynomial as the distance from the center control point to the farthest point used to infer the polynomial (using `fixedPoints`)[2].

Note At least six control-point pairs are needed to solve for the second-order polynomial. Ill-conditioned polynomials might result if too few pairs are used.

References

- [1] Goshtasby, Ardeshir, "Piecewise linear mapping functions for image registration," *Pattern Recognition*, Vol. 19, 1986, pp. 459-466.

[2] Goshtasby, Ardeshir, "Image registration by local approximation methods," *Image and Vision Computing*, Vol. 6, 1988, pp. 255-261.

See Also

`cpcorr` | `cpselect` | `cpstruct2pairs` | `imtransform` | `tformfwd` | `tforminv`

Introduced before R2006a

cpcorr

Tune control point locations using cross-correlation

Syntax

```
movingPointsAdjusted = cpcorr(movingPoints,fixedPoints,moving,fixed)
```

Description

`movingPointsAdjusted = cpcorr(movingPoints,fixedPoints,moving,fixed)` uses normalized cross-correlation to adjust each pair of control points specified in `movingPoints` and `fixedPoints`. `moving` and `fixed` are images. `cpcorr` returns the adjusted control points in `movingPointsAdjusted`.

Note: The `moving` and `fixed` images must have the same scale for `cpcorr` to be effective. If `cpcorr` cannot correlate a pair of control points, `movingPointsAdjusted` contains the same coordinates as `movingPoints` for that pair.

Examples

Fine-Tune Control-Point Locations using Cross Correlation

Read two images into the workspace.

```
moving = imread('onion.png');  
fixed = imread('peppers.png');
```

Define sets of control points for both images.

```
movingPoints = [118 42;99 87];  
fixedPoints = [190 114;171 165];
```

Display the images, and display the control points in white.

```
figure; imshow(fixed)
hold on
plot(fixedPoints(:,1),fixedPoints(:,2),'xw')
title('fixed')
```

fixed



```
figure; imshow(moving)
hold on
plot(movingPoints(:,1),movingPoints(:,2),'xw')
title('moving')
```

moving



Observe the slight errors in the position of the moving points.

Adjust the moving control points using cross correlation.

```
movingPointsAdjusted = cpcorr(movingPoints, fixedPoints, ...
                              moving(:, :, 1), fixed(:, :, 1))
```

```
movingPointsAdjusted =
```

```
115.9000    39.1000
 97.0000    89.9000
```

Display the adjusted moving points in yellow. Compared to the original moving points (in white), the adjusted points more closely match the positions of the fixed points.

```
plot(movingPointsAdjusted(:, 1), movingPointsAdjusted(:, 2), 'xy')
```

moving



Input Arguments

movingPoints — Coordinates of control points in the image to be transformed

M-by-2 double matrix

Coordinates of control points in the image to be transformed, specified as an *M*-by-2 double matrix.

Example: `movingPoints = [127 93; 74 59];`

Data Types: double

fixedPoints — Coordinates of control points in the reference image

M-by-2 double matrix

Coordinates of control points in the reference image, specified as an *M*-by-2 double matrix.

Example: `fixedPoints = [323 195; 269 161];`

Data Types: double

moving — Image to be registered

numeric array of finite values

Image to be registered, specified as a numeric array of finite values.

fixed — Reference image in the target orientation

numeric array of finite values

Reference image in the target orientation, specified as a numeric array of finite values.

Output Arguments

movingPointsAdjusted — Adjusted coordinates of control points in the image to be transformed

double matrix the same size as `movingPoints`

Adjusted coordinates of control points in the image to be transformed, returned as a double matrix the same size as `movingPoints`.

Tips

`cpcorr` cannot adjust a point if any of the following occur:

- points are too near the edge of either image
- regions of images around points contain `Inf` or `NaN`
- region around a point in moving image has zero standard deviation
- regions of images around points are poorly correlated

Algorithms

`cpcorr` only moves the position of a control point by up to four pixels. Adjusted coordinates are accurate up to one-tenth of a pixel. `cpcorr` is designed to get subpixel accuracy from the image content and coarse control point selection.

See Also

See Also

`cpselect` | `fitgeotrans` | `imwarp` | `normxcorr2`

Introduced before R2006a

cpselect

Control Point Selection tool

Syntax

```
cpselect(moving, fixed)
cpselect(moving, fixed, cpstruct_in)
cpselect(moving, fixed, initialMovingPoints, initialFixedPoints)
h = cpselect( ___ )
h = cpselect( ___, 'Wait', false)
[selectedMovingPoints, selectedFixedPoints] =
cpselect( ___, 'Wait', true)
```

Description

`cpselect(moving, fixed)` starts the Control Point Selection Tool, a user interface that enables you to select control points in two related images. `moving` is the image to be warped, which brings it into the coordinate system of the `fixed` image. `moving` and `fixed` can be either variables that contain grayscale, truecolor, or binary images, or the names of files containing these images. The Control Point Selection Tool returns the control points in a `cpstruct` structure.

`cpselect(moving, fixed, cpstruct_in)` starts `cpselect` with an initial set of control points that are stored in `cpstruct_in`. This syntax allows you to restart `cpselect` with the state of control points, including unpaired and predicted control points, previously saved in `cpstruct_in`.

`cpselect(moving, fixed, initialMovingPoints, initialFixedPoints)` starts `cpselect` with an initial set of valid control point pairs. `initialMovingPoints` and `initialFixedPoints` are m -by-2 matrices that store `moving` and `fixed` control point coordinates, respectively. The two columns represent the x - and y -coordinates of the control points.

`h = cpselect(___)` returns a handle `h` to the Control Point Selection tool. You can use the `close(h)` syntax to close the tool from the command line.

`h = cpselect(____, 'Wait', false)` returns a handle `h` to the Control Point Selection tool. You can use the `close(h)` syntax to close the tool from the command line. In contrast to setting 'Wait' as `true`, this syntax lets you run `cpselect` at the same time as you run other programs in MATLAB.

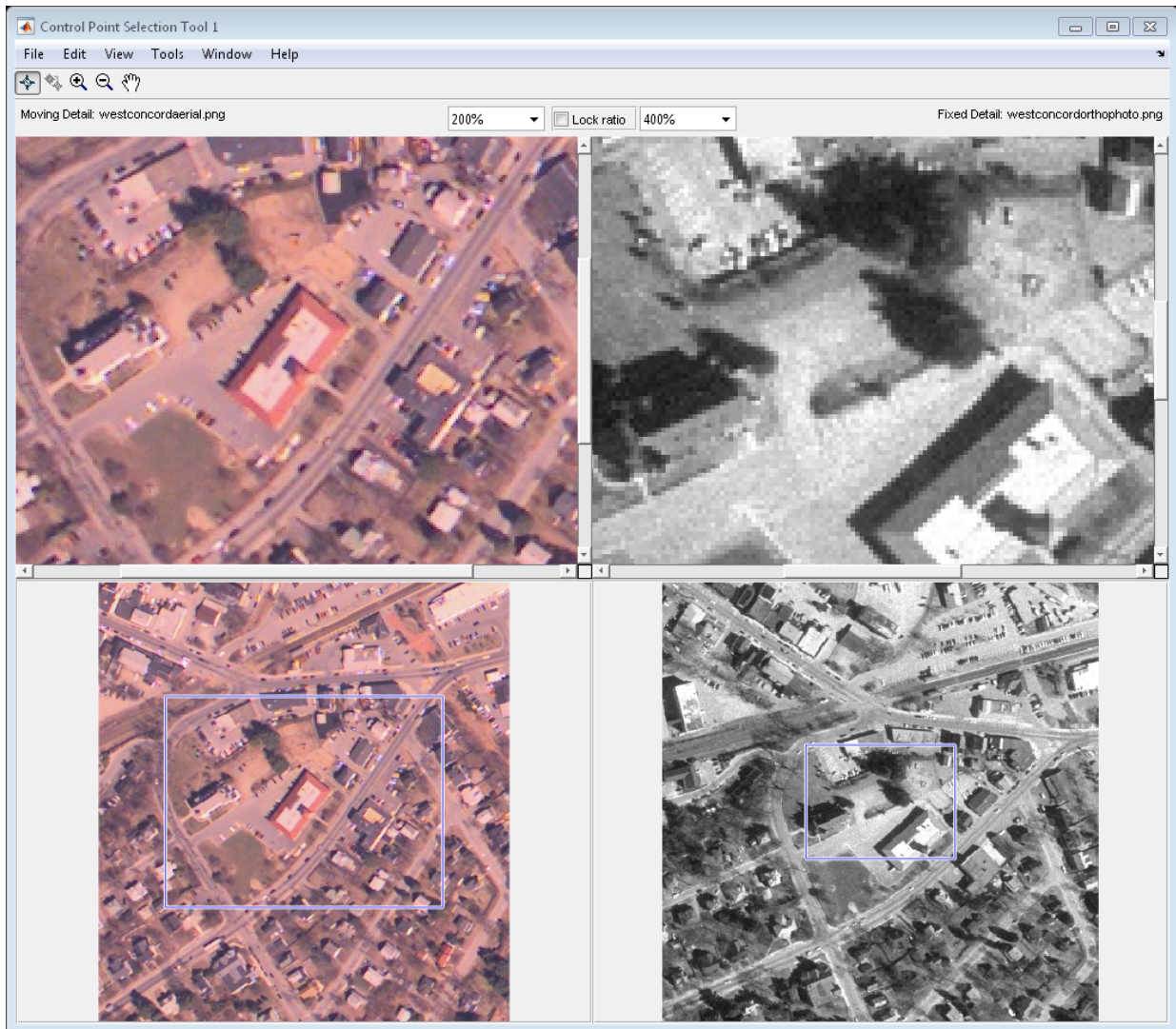
`[selectedMovingPoints,selectedFixedPoints] = cpselect(____, 'Wait', true)` takes control of the MATLAB command line until you finish selecting control points. `cpselect` returns valid selected pairs of points. `selectedMovingPoints` and `selectedFixedPoints` are p -by-2 matrices that store the coordinates in the moving and fixed images, respectively. The two columns represent the x - and y -coordinates of the selected control points.

Examples

Start Control Point Selection Tool with Saved Images

Call `cpselect` , specifying the names of the image you want to register and the reference image. This example uses the optional syntax that returns a handle to the tool that is created so that you can close the tool programmatically.

```
h = cpselect('westconcordaerial.png', 'westconcordorthophoto.png');
```

Close the tool.

```
close(h)
```

Open Control Point Selection Tool with Predefined Control Points

Create a sample reference image.

```
I = checkerboard;
```

Create a copy of the sample image, rotating it to create a sample image that needs registering.

```
J = imrotate(I,30);
```

Specify two sets of control points for the fixed and moving images.

```
fixedPoints = [11 11; 41 71];  
movingPoints = [14 44; 70 81];
```

Open the Control Point Selection Tool, specifying the sample fixed and moving images and the two sets of saved control points.

```
cpselect(J,I,movingPoints,fixedPoints);
```

When the tool opens, you are prompted to save the control points.

Register an Aerial Photo to an Orthophoto

Read an aerial photo and an orthophoto into the workspace and display them.

```
aerial = imread('westconcordaerial.png');  
figure, imshow(aerial)  
ortho = imread('westconcordorthophoto.png');  
figure, imshow(ortho)
```

Load some points that have already been picked.

```
load westconcordpoints
```

Open the Control Point Selection tool, specifying the two images and the preselected points. Use the 'Wait' parameter to make `cpselect` wait for you to pick some more points.

```
[aerial_points,ortho_points] = ...  
    cpselect(aerial,'westconcordorthophoto.png',...  
            movingPoints,fixedPoints,...
```

```
'Wait',true);
```

When control returns to the command line, perform the registration. First use `fitgeotrans` to estimate the geometric transformation required to bring the moving image into alignment with the fixed image. You specify the control points you selected and the type of transformation you want. Next use `imwarp` to perform the transformation. Finally, display the transformed image over the original orthophoto to see how well the registration succeeded.

```
t_concord = fitgeotrans(aerial_points,ortho_points,'projective');
ortho_ref = imref2d(size(ortho)); %relate intrinsic and world coordinates
aerial_registered = imwarp(aerial,t_concord,'OutputView',ortho_ref);
figure, imshowpair(aerial_registered,ortho,'blend')
```

- “Control Point Selection Procedure”
- “Export Control Points to the Workspace”

Input Arguments

moving — Input image to be aligned

grayscale image | truecolor image | binary image | character vector

Input image to be aligned, specified as a grayscale, truecolor, or binary image, or a character vector. A grayscale image can be `uint8`, `uint16`, `int16`, `single`, or `double`. A truecolor image can be `uint8`, `uint16`, `single`, or `double`. A binary image is of class `logical`. If `moving` is a character vector, it must identify files containing these same types of images.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

fixed — Reference image

grayscale image | truecolor image | binary image | character vector

Reference image, specified as a grayscale, truecolor, or binary image. A grayscale image can be `uint8`, `uint16`, `int16`, `single`, or `double`. A truecolor image can be `uint8`, `uint16`, `single`, or `double`. A binary image is of class `logical`. If `fixed` is a character vector, it must identify files containing these same types of images.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

cpstruct_in — Preselected control points

structure

Preselected control points, specified as a structure (`cpstruct`). `cpstruct_in` contains information about x - and y -coordinates of all control points in the moving and fixed images, including unpaired and predicted control points. `cpstruct_in` also contains indexing information that allows the Control Point Selection tool to restore the state of the control points.

Create a `cpstruct` by exporting points from the Control Point Selection tool, described in “Export Control Points to the Workspace”.

Data Types: `struct`

initialMovingPoints — Preselected control points on the moving image

m -by-2 numeric array

Preselected control points on the moving image, specified as an m -by-2 numeric array. The two columns represent the x - and y -coordinates of the control points.

Data Types: `double`

initialFixedPoints — Preselected control points on the fixed image

m -by-2 numeric array

Preselected control points on the fixed image, specified as an m -by-2 numeric array. The two columns represent the x - and y -coordinates of the control points.

Data Types: `double`

Output Arguments

h — Control Point Selection tool

handle

Control Point Selection tool, returned as a handle.

selectedMovingPoints — Selected control points on the moving image

p -by-2 numeric array

Selected control points on the moving image, specified as a p -by-2 numeric array. The two columns represent the x - and y -coordinates of the control points.

Data Types: `double`

selectedFixedPoints — Selected control points on the fixed image

p-by-2 numeric array

Selected control points on the fixed image, specified as a *p*-by-2 numeric array. The two columns represent the *x*- and *y*-coordinates of the control points.

Data Types: double

Tips

- When calling `cpselect` in a script, specify the 'Wait' option as `true`. The 'Wait' option causes `cpselect` to block the MATLAB command line until control points have been selected and returned. If you do not use the 'Wait' option, `cpselect` returns control immediately and your script continues without allowing time for control point selection. Additionally, without the 'Wait' option, `cpselect` does not return the control points as return values.

Algorithms

`cpselect` uses the following general procedure for control-point prediction.

- 1 Find all valid pairs of control points.
- 2 Infer a spatial transformation between `moving` and `fixed` control points using method that depends on the number of valid pairs, as follows:

2 pairs	Nonreflective similarity
3 pairs	Affine
4 or more pairs	Projective
- 3 Apply spatial transformation to the new point. This transformation generates the predicted point.
- 4 Display predicted point.

See Also

See Also

`cpcorr` | `cpstruct2pairs` | `fitgeotrans` | `imtool` | `imwarp`

Topics

“Control Point Selection Procedure”

“Export Control Points to the Workspace”

Introduced before R2006a

cpstruct2pairs

Extract valid control point pairs from `cpstruct` structure

Syntax

```
[movingPoints, fixedPoints] = cpstruct2pairs(cpstruct_in)
```

Description

`[movingPoints, fixedPoints] = cpstruct2pairs(cpstruct_in)` extracts the valid control point pairs from `cpstruct_in`, returning two arrays `movingPoints` and `fixedPoints`.

Examples

Convert `cpstruct` to Sets of Control Point Pairs

Read an aerial photograph and an orthoregistered image into the workspace.

```
aerial = imread('westconcordaerial.png');
ortho = imread('westconcordorthophoto.png');
```

Load some preselected control points for these images.

```
load westconcordpoints
whos
```

Name	Size	Bytes	Class	Attributes
aerial	394x369x3	436158	uint8	
fixedPoints	4x2	64	double	
movingPoints	4x2	64	double	
ortho	366x364	133224	uint8	

Open the Control Point Selection tool, specifying the two images along with the predefined control points.

```
cpselect(aerial, ortho, movingPoints, fixedPoints);
```

Create the `cpstruct`. Using the Control Point Selection tool, select **Export Points to Workspace** from the **File** menu to save the points to the workspace. On the **Export Points to Workspace** dialog box, check the **Structure with all points** check box and clear **Moving points of valid pairs** and **Fixed points of valid pairs**. Click **OK**. Close the Control Point Selection tool.

Use `cpstruct2pairs` to extract the moving and fixed points from the `cpstruct`.

```
[mPoints,fPoints] = cpstruct2pairs(cpstruct);
```

Compare the stored set of points with the set of points you exported.

```
fixedPoints, fpoints
```

```
fixedPoints =
```

```
164.5639 113.2890
353.5325 130.0798
143.4046 284.8935
353.5325 311.9810
```

```
fpoints =
```

```
164.5639 113.2890
353.5325 130.0798
143.4046 284.8935
353.5325 311.9810
```

Input Arguments

`cpstruct_in` — Preselected control points

structure

Preselected control points, specified as a structure (`cpstruct`). `cpstruct_in` contains information about the x - and y -coordinates of all control points in the moving and fixed images, including unpaired and predicted control points. `cpstruct2pairs` eliminates unmatched and predicted control points, and returns the set of valid control point pairs.

`cpstruct_in` is a structure produced by the Control Point Selection tool (`cpselect`) when you choose the **Export Points to Workspace** option. For more information, see “Export Control Points to the Workspace”.

Data Types: struct

Output Arguments

movingPoints — Control point pairs from moving image being aligned

m-by-2 numeric array

Control point pairs from image being aligned, returned as an *m*-by-2 numeric array.

Data Types: double

fixedPoints — Control point pairs from reference image

m-by-2 numeric array

Control point pairs from reference image, returned as an *m*-by-2 numeric array.

Data Types: double

See Also

See Also

cpselect | fitgeotrans

Topics

“Export Control Points to the Workspace”

Introduced before R2006a

dct2

2-D discrete cosine transform

Syntax

```
B = dct2(A)
B = dct2(A,m,n)
B = dct2(A,[m n])
```

Description

`B = dct2(A)` returns the two-dimensional discrete cosine transform of `A`. The matrix `B` is the same size as `A` and contains the discrete cosine transform coefficients $B(k_1,k_2)$.

`B = dct2(A,m,n)` pads the matrix `A` with 0's to size `m`-by-`n` before transforming. If `m` or `n` is smaller than the corresponding dimension of `A`, `dct2` truncates `A`.

`B = dct2(A,[m n])` same as above.

Class Support

`A` can be numeric or logical. The returned matrix `B` is of class `double`.

Examples

Remove High Frequencies in Image using DCT

This example shows how to remove high frequencies from an image using the two-dimensional discrete cosine transfer (DCT).

Read an image into the workspace, then convert the image to grayscale.

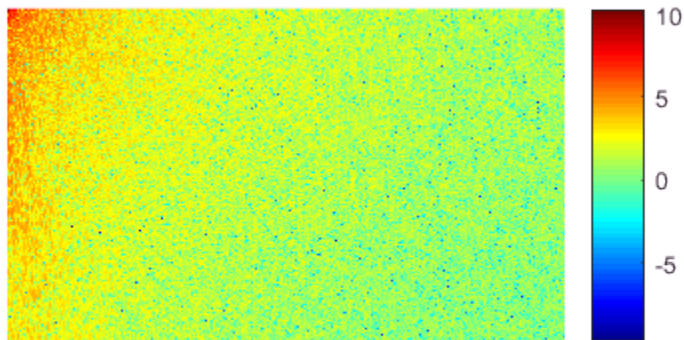
```
RGB = imread('autumn.tif');
I = rgb2gray(RGB);
```

Perform a 2-D DCT of the grayscale image using the `dct2` function.

```
J = dct2(I);
```

Display the transformed image using a logarithmic scale. Notice that most of the energy is in the upper left corner.

```
figure  
imshow(log(abs(J)),[])  
colormap(gca, jet(64))  
colorbar
```



Set values less than magnitude 10 in the DCT matrix to zero.

```
J(abs(J) < 10) = 0;
```

Reconstruct the image using the inverse DCT function `idct2`.

```
K = idct2(J);
```

Display the original grayscale image alongside the processed image.

```
figure
```

```
imshowpair(I,K,'montage')
title('Original Grayscale Image (Left) and Processed Image (Right)');
```

Original Grayscale Image (Left) and Processed Image (Right)



Algorithms

The discrete cosine transform (DCT) is closely related to the discrete Fourier transform. It is a separable linear transformation; that is, the two-dimensional transform is equivalent to a one-dimensional DCT performed along a single dimension followed by a one-dimensional DCT in the other dimension. The definition of the two-dimensional DCT for an input image A and output image B is

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

where

$$\alpha_p = \begin{cases} \frac{1}{\sqrt{M}}, & p = 0 \\ \sqrt{\frac{2}{M}}, & 1 \leq p \leq M-1 \end{cases}$$

and

$$\alpha_q = \begin{cases} \frac{1}{\sqrt{N}}, & q = 0 \\ \sqrt{\frac{2}{N}}, & 1 \leq q \leq N-1 \end{cases}$$

M and N are the row and column size of \mathbf{A} , respectively. If you apply the DCT to real data, the result is also real. The DCT tends to concentrate information, making it useful for image compression applications.

This transform can be inverted using `idct2`.

References

- [1] Jain, Anil K., *Fundamentals of Digital Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1989, pp. 150-153.
- [2] Pennebaker, William B., and Joan L. Mitchell, *JPEG: Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993.

See Also

`fft2` | `idct2` | `ifft2`

Introduced before R2006a

dctmtx

Discrete cosine transform matrix

Syntax

```
D = dctmtx(n)
```

Description

`D = dctmtx(n)` returns the n -by- n DCT (discrete cosine transform) matrix. $D*A$ is the DCT of the columns of A and $D'*A$ is the inverse DCT of the columns of A (when A is n -by- n).

Class Support

n is an integer scalar of class `double`. D is returned as a matrix of class `double`.

Examples

Calculate Discrete Cosine Transform Matrix

Read image into the workspace and cast it to class `double`.

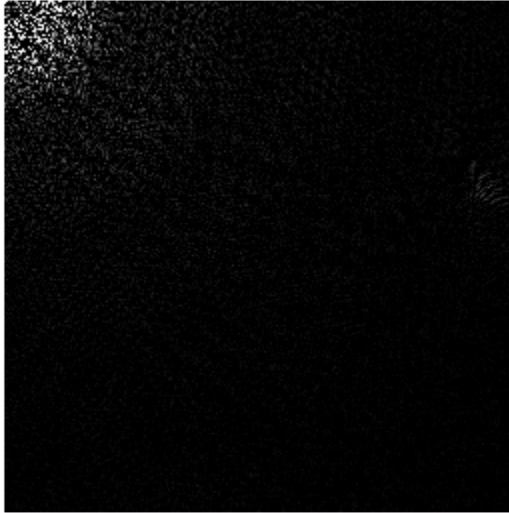
```
A = im2double(imread('rice.png'));
```

Calculate discrete cosine transform matrix.

```
D = dctmtx(size(A,1));
```

Multiply the input image A by D to get the DCT of the columns of A and by D' to get the inverse DCT of the columns of A .

```
dct = D*A*D';  
figure, imshow(dct)
```



Tips

If A is square, the two-dimensional DCT of A can be computed as $D \cdot A \cdot D'$. This computation is sometimes faster than using `dct2`, especially if you are computing a large number of small DCTs, because D needs to be determined only once.

For example, in JPEG compression, the DCT of each 8-by-8 block is computed. To perform this computation, use `dctmtx` to determine D , and then calculate each DCT using $D \cdot A \cdot D'$ (where A is each 8-by-8 block). This is faster than calling `dct2` for each individual block.

See Also

`dct2`

Introduced before R2006a

deconvblind

Deblur image using blind deconvolution

Syntax

```
[J,PSF] = deconvblind(I, INITPSF)
[J,PSF] = deconvblind(I, INITPSF, NUMIT)
[J,PSF] = deconvblind(I, INITPSF, NUMIT, DAMPAR)
[J,PSF] = deconvblind(I, INITPSF, NUMIT, DAMPAR, WEIGHT)
[J,PSF] = deconvblind(I, INITPSF, NUMIT, DAMPAR, WEIGHT, READOUT)
[J,PSF] = deconvblind(..., FUN, P1, P2, ..., PN)
```

Description

`[J,PSF] = deconvblind(I, INITPSF)` deconvolves image `I` using the maximum likelihood algorithm, returning both the deblurred image `J` and a restored point-spread function `PSF`. The restored `PSF` is a positive array that is the same size as `INITPSF`, normalized so its sum adds up to 1. The `PSF` restoration is affected strongly by the size of the initial guess `INITPSF` and less by the values it contains. For this reason, specify an array of 1's as your `INITPSF`.

`I` can be an `N`-dimensional array.

To improve the restoration, `deconvblind` supports several optional parameters, described below. Use `[]` as a placeholder if you do not specify an intermediate parameter.

`[J,PSF] = deconvblind(I, INITPSF, NUMIT)` specifies the number of iterations (default is 10).

`[J,PSF] = deconvblind(I, INITPSF, NUMIT, DAMPAR)` specifies the threshold deviation of the resulting image from the input image `I` (in terms of the standard deviation of Poisson noise) below which damping occurs. The iterations are suppressed for the pixels that deviate within the `DAMPAR` value from their original value. This suppresses the noise generation in such pixels, preserving necessary image details elsewhere. The default value is 0 (no damping).

`[J,PSF] = deconvblind(I, INITPSF, NUMIT, DAMPAR, WEIGHT)` specifies which pixels in the input image `I` are considered in the restoration. By default, `WEIGHT` is a unit

array, the same size as the input image. You can assign a value between 0.0 and 1.0 to elements in the `WEIGHT` array. The value of an element in the `WEIGHT` array determines how much the pixel at the corresponding position in the input image is considered. For example, to exclude a pixel from consideration, assign it a value of 0 in the `WEIGHT` array. You can adjust the weight value assigned to each pixel according to the amount of flat-field correction.

`[J,PSF] = deconvblind(I, INITPSF, NUMIT, DAMPAR, WEIGHT, READOUT)`, where `READOUT` is an array (or a value) corresponding to the additive noise (e.g., background, foreground noise) and the variance of the read-out camera noise. `READOUT` has to be in the units of the image. The default value is 0.

`[J,PSF] = deconvblind(..., FUN, P1, P2, ..., PN)`, where `FUN` is a function describing additional constraints on the PSF. `FUN` must be a function handle.

`FUN` is called at the end of each iteration. `FUN` must accept the PSF as its first argument and can accept additional parameters `P1`, `P2`, ..., `PN`. The `FUN` function should return one argument, PSF, that is the same size as the original PSF and that satisfies the positivity and normalization constraints. The function `colfilt` zero-pads `A`, if necessary.

Note The output image `J` could exhibit ringing introduced by the discrete Fourier transform used in the algorithm. To reduce the ringing, use `I = edgetaper(I,PSF)` before calling `deconvblind`.

Resuming Deconvolution

You can use `deconvblind` to perform a deconvolution that starts where a previous deconvolution stopped. To use this feature, pass the input image `I` and the initial guess at the PSF, `INITPSF`, as cell arrays: `{I}` and `{INITPSF}`. When you do, the `deconvblind` function returns the output image `J` and the restored point-spread function, `PSF`, as cell arrays, which can then be passed as the input arrays into the next `deconvblind` call. The output cell array `J` contains four elements:

`J{1}` contains `I`, the original image.

`J{2}` contains the result of the last iteration.

`J{3}` contains the result of the next-to-last iteration.

$J\{4\}$ is an array generated by the iterative algorithm.

Class Support

I and $INITPSF$ can be `uint8`, `uint16`, `int16`, `single`, or `double`. `DAMPAR` and `READOUT` must have the same class as the input image. Other inputs have to be `double`. The output image J (or the first array of the output cell) has the same class as the input image I . The output PSF is `double`.

Examples

Deblur an Image Using Blind Deconvolution

Create a sample image with noise.

```
% Set the random number generator back to its default settings for
% consistency in results.
rng default;

I = checkerboard(8);
PSF = fspecial('gaussian',7,10);
V = .0001;
BlurredNoisy = imnoise(imfilter(I,PSF),'gaussian',0,V);
```

Create a weight array to specify which pixels are included in processing.

```
WT = zeros(size(I));
WT(5:end-4,5:end-4) = 1;
INITPSF = ones(size(PSF));
```

Perform blind deconvolution.

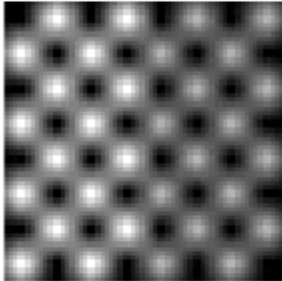
```
[J P] = deconvblind(BlurredNoisy,INITPSF,20,10*sqrt(V),WT);
```

Display the results.

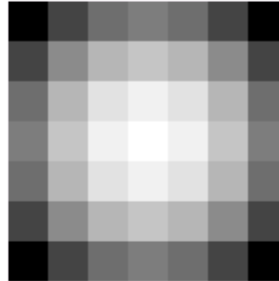
```
subplot(221);imshow(BlurredNoisy);
title('A = Blurred and Noisy');
subplot(222);imshow(PSF,[]);
title('True PSF');
subplot(223);imshow(J);
```

```
title('Deblurred Image');  
subplot(224);imshow(P,[]);  
title('Recovered PSF');
```

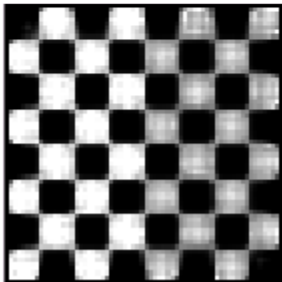
A = Blurred and Noisy



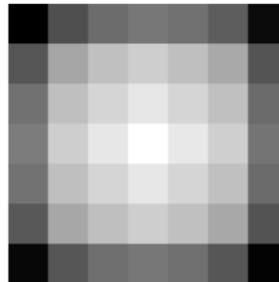
True PSF



Deblurred Image



Recovered PSF



See Also

[deconvlucy](#) | [deconvreg](#) | [deconvwnr](#) | [edgetaper](#) | [imnoise](#) | [otf2psf](#) | [padarray](#) | [psf2otf](#)

Topics

“Anonymous Functions” (MATLAB)

“Parameterizing Functions” (MATLAB)
“Create Function Handle” (MATLAB)

Introduced before R2006a

deconvlucy

Deblur image using Lucy-Richardson method

Syntax

```
J = deconvlucy(I, PSF)
J = deconvlucy(I, PSF, NUMIT)
J = deconvlucy(I, PSF, NUMIT, DAMPAR)
J = deconvlucy(I, PSF, NUMIT, DAMPAR, WEIGHT)
J = deconvlucy(I, PSF, NUMIT, DAMPAR, WEIGHT, READOUT)
J = deconvlucy(I, PSF, NUMIT, DAMPAR, WEIGHT, READOUT, SUBSMPL)
```

Description

`J = deconvlucy(I, PSF)` restores image `I` that was degraded by convolution with a point-spread function `PSF` and possibly by additive noise. The algorithm is based on maximizing the likelihood of the resulting image `J`'s being an instance of the original image `I` under Poisson statistics.

`I` can be a `N`-dimensional array.

To improve the restoration, `deconvlucy` supports several optional parameters. Use `[]` as a placeholder if you do not specify an intermediate parameter.

`J = deconvlucy(I, PSF, NUMIT)` specifies the number of iterations the `deconvlucy` function performs. If this value is not specified, the default is 10.

`J = deconvlucy(I, PSF, NUMIT, DAMPAR)` specifies the threshold deviation of the resulting image from the image `I` (in terms of the standard deviation of Poisson noise) below which damping occurs. Iterations are suppressed for pixels that deviate beyond the `DAMPAR` value from their original value. This suppresses the noise generation in such pixels, preserving necessary image details elsewhere. The default value is 0 (no damping).

`J = deconvlucy(I, PSF, NUMIT, DAMPAR, WEIGHT)` specifies the weight to be assigned to each pixel to reflect its recording quality in the camera. A bad pixel is

excluded from the solution by assigning it zero weight value. Instead of giving a weight of unity for good pixels, you can adjust their weight according to the amount of flat-field correction. The default is a unit array of the same size as input image `I`.

`J = deconvlucy(I, PSF, NUMIT, DAMPAR, WEIGHT, READOUT)` specifies a value corresponding to the additive noise (e.g., background, foreground noise) and the variance of the readout camera noise. `READOUT` has to be in the units of the image. The default value is 0.

`J = deconvlucy(I, PSF, NUMIT, DAMPAR, WEIGHT, READOUT, SUBSMPL)`, where `SUBSMPL` denotes subsampling and is used when the `PSF` is given on a grid that is `SUBSMPL` times finer than the image. The default value is 1.

Note The output image `J` could exhibit ringing introduced by the discrete Fourier transform used in the algorithm. To reduce the ringing, use `I = edgetaper(I, PSF)` before calling `deconvlucy`.

Resuming Deconvolution

If `I` is a cell array, it can contain a single numerical array (the blurred image) or it can be the output from a previous run of `deconvlucy`.

When you pass a cell array to `deconvlucy` as input, it returns a 1-by-4 cell array `J`, where

`J{1}` contains `I`, the original image.

`J{2}` contains the result of the last iteration.

`J{3}` contains the result of the next-to-last iteration.

`J{4}` is an array generated by the iterative algorithm.

Class Support

`I` can be `uint8`, `uint16`, `int16`, `double`, or `single`. `PSF` can be `uint8`, `uint16`, `int16`, `double`, or `single`. Note, however, that `deconvlucy` converts the `PSF` to `double`

without normalization. DAMPAR and READOUT must have the same class as the input image. Other inputs have to be double. The output image J (or the first array of the output cell) has the same class as the input image I.

Examples

Remove Blur Using Several deconvlucy Optional Syntaxes

Create a sample image and blur it.

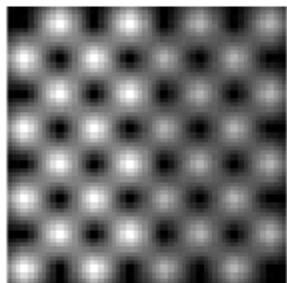
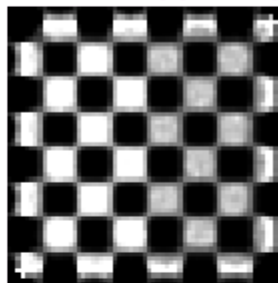
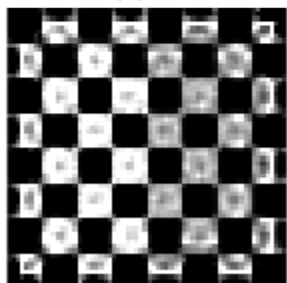
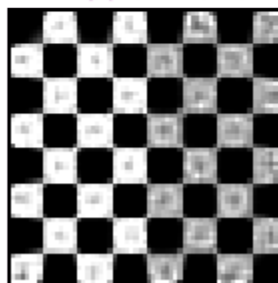
```
I = checkerboard(8);
PSF = fspecial('gaussian',7,10);
V = .0001;
BlurredNoisy = imnoise(imfilter(I,PSF),'gaussian',0,V);
```

Create a weight array and call deconvlucy using several optional parameters.

```
WT = zeros(size(I));
WT(5:end-4,5:end-4) = 1;
J1 = deconvlucy(BlurredNoisy,PSF);
J2 = deconvlucy(BlurredNoisy,PSF,20,sqrt(V));
J3 = deconvlucy(BlurredNoisy,PSF,20,sqrt(V),WT);
```

Display the results.

```
subplot(221);imshow(BlurredNoisy);
title('A = Blurred and Noisy');
subplot(222);imshow(J1);
title('deconvlucy(A,PSF)');
subplot(223);imshow(J2);
title('deconvlucy(A,PSF,NI,DP)');
subplot(224);imshow(J3);
title('deconvlucy(A,PSF,NI,DP,WT)');
```


A = Blurred and Noisy**deconvlucy(A,PSF)****deconvlucy(A,PSF,NI,DP)****deconvlucy(A,PSF,NI,DP,WT)**

References

- [1] Biggs, D.S.C. "Acceleration of Iterative Image Restoration Algorithms." *Applied Optics*. Vol. 36. Number 8, 1997, pp. 1766–1775.
- [2] Hanisch, R.J., R.L. White, and R.L. Gilliland. "Deconvolution of Hubble Space Telescope Images and Spectra." *Deconvolution of Images and Spectra* (P.A. Jansson, ed.). Boston, MA: Academic Press, 1997, pp. 310–356.

See Also

`deconvblind` | `deconvreg` | `deconvwnr` | `otf2psf` | `padarray` | `psf2otf`

Introduced before R2006a

deconvreg

Deblur image using regularized filter

Syntax

```
J = deconvreg(I, PSF)
J = deconvreg(I, PSF, NOISEPOWER)
J = deconvreg(I, PSF, NOISEPOWER, LRANGE)
J = deconvreg(I, PSF, NOISEPOWER, LRANGE, REGOP)
[J, LAGRA] = deconvreg(I, PSF, ...)
```

Description

`J = deconvreg(I, PSF)` deconvolves image `I` using the regularized filter algorithm, returning deblurred image `J`. The assumption is that the image `I` was created by convolving a true image with a point-spread function `PSF` and possibly by adding noise. The algorithm is a constrained optimum in the sense of least square error between the estimated and the true images under requirement of preserving image smoothness.

`I` can be a `N`-dimensional array.

`J = deconvreg(I, PSF, NOISEPOWER)` where `NOISEPOWER` is the additive noise power. The default value is 0.

`J = deconvreg(I, PSF, NOISEPOWER, LRANGE)` where `LRANGE` is a vector specifying range where the search for the optimal solution is performed. The algorithm finds an optimal Lagrange multiplier `LAGRA` within the `LRANGE` range. If `LRANGE` is a scalar, the algorithm assumes that `LAGRA` is given and equal to `LRANGE`; the `NP` value is then ignored. The default range is between `[1e-9 and 1e9]`.

`J = deconvreg(I, PSF, NOISEPOWER, LRANGE, REGOP)` where `REGOP` is the regularization operator to constrain the deconvolution. The default regularization operator is the Laplacian operator, to retain the image smoothness. The `REGOP` array dimensions must not exceed the image dimensions; any nonsingleton dimensions must correspond to the nonsingleton dimensions of `PSF`.

`[J, LAGRA] = deconvreg(I, PSF, ...)` outputs the value of the Lagrange multiplier `LAGRA` in addition to the restored image `J`.

Note The output image **J** could exhibit ringing introduced by the discrete Fourier transform used in the algorithm. To reduce the ringing, process the image with the `edgetaper` function prior to calling the `deconvreg` function. For example, `I = edgetaper(I,PSF)`.

Class Support

I can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. Other inputs have to be of class `double`. **J** has the same class as **I**.

Examples

Deblur Image Using Regularized Filter

Create sample image.

```
I = checkerboard(8);
```

Create PSF and use it to create a blurred and noisy version of the input image.

```
PSF = fspecial('gaussian',7,10);  
V = .01;  
BlurredNoisy = imnoise(imfilter(I,PSF),'gaussian',0,V);  
NOISEPOWER = V*prod(size(I));
```

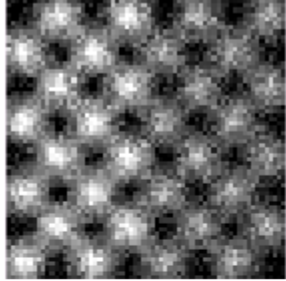
Deblur the image.

```
[J LAGRA] = deconvreg(BlurredNoisy,PSF,NOISEPOWER);
```

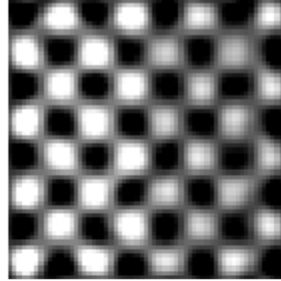
Display the various versions of the image.

```
subplot(221); imshow(BlurredNoisy);  
title('A = Blurred and Noisy');  
subplot(222); imshow(J);  
title('[J LAGRA] = deconvreg(A,PSF,NP)');  
subplot(223); imshow(deconvreg(BlurredNoisy,PSF,[],LAGRA/10));  
title('deconvreg(A,PSF,[],0.1*LAGRA)');  
subplot(224); imshow(deconvreg(BlurredNoisy,PSF,[],LAGRA*10));  
title('deconvreg(A,PSF,[],10*LAGRA)');
```

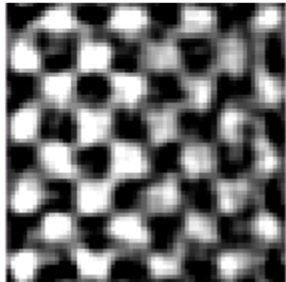
A = Blurred and Noisy



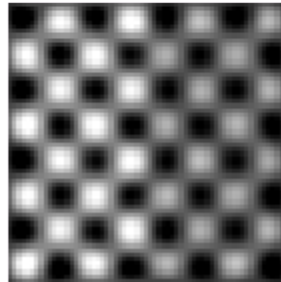
[J LAGRA] = deconvreg(A,PSF,NP)



deconvreg(A,PSF,[],0.1*LAGRA)



deconvreg(A,PSF,[],10*LAGRA)



See Also

[deconvblind](#) | [deconvlucy](#) | [deconvwnr](#) | [otf2psf](#) | [padarray](#) | [psf2otf](#)

Introduced before R2006a

deconvwnr

Deblur image using Wiener filter

Syntax

`J = deconvwnr(I,PSF,NSR)`
`J = deconvwnr(I,PSF,NCORR,ICORR)`

Description

`J = deconvwnr(I,PSF,NSR)` deconvolves image `I` using the Wiener filter algorithm, returning deblurred image `J`. Image `I` can be an N-dimensional array. `PSF` is the point-spread function with which `I` was convolved. `NSR` is the noise-to-signal power ratio of the additive noise. `NSR` can be a scalar or a spectral-domain array of the same size as `I`. Specifying 0 for the `NSR` is equivalent to creating an ideal inverse filter.

The algorithm is optimal in a sense of least mean square error between the estimated and the true images.

`J = deconvwnr(I,PSF,NCORR,ICORR)` deconvolves image `I`, where `NCORR` is the autocorrelation function of the noise and `ICORR` is the autocorrelation function of the original image. `NCORR` and `ICORR` can be of any size or dimension, not exceeding the original image. If `NCORR` or `ICORR` are N-dimensional arrays, the values correspond to the autocorrelation within each dimension. If `NCORR` or `ICORR` are vectors, and `PSF` is also a vector, the values represent the autocorrelation function in the first dimension. If `PSF` is an array, the 1-D autocorrelation function is extrapolated by symmetry to all non-singleton dimensions of `PSF`. If `NCORR` or `ICORR` is a scalar, this value represents the power of the noise of the image.

Note The output image `J` could exhibit ringing introduced by the discrete Fourier transform used in the algorithm. To reduce the ringing, use

`I = edgetaper(I,PSF)`
prior to calling `deconvwnr`.

Class Support

I can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. Other inputs have to be of class `double`. J has the same class as I.

Examples

Deblur Image Using Wiener Filter

Read image into the workspace and display it.

```
I = im2double(imread('cameraman.tif'));  
imshow(I);  
title('Original Image (courtesy of MIT)');
```

Original Image (courtesy of MIT)



Simulate a motion blur.

```
LEN = 21;
THETA = 11;
PSF = fspecial('motion', LEN, THETA);
blurred = imfilter(I, PSF, 'conv', 'circular');
figure, imshow(blurred)
```



Simulate additive noise.

```
noise_mean = 0;
noise_var = 0.0001;
blurred_noisy = imnoise(blurred, 'gaussian', ...
                        noise_mean, noise_var);
figure, imshow(blurred_noisy)
title('Simulate Blur and Noise')
```

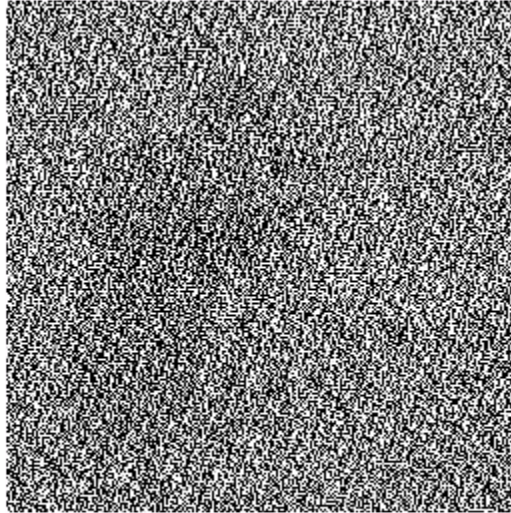

Simulate Blur and Noise



Try restoration assuming no noise.

```
estimated_nsr = 0;  
wnr2 = deconvwnr(blurred_noisy, PSF, estimated_nsr);  
figure, imshow(wnr2)  
title('Restoration of Blurred, Noisy Image Using NSR = 0')
```

Restoration of Blurred, Noisy Image Using NSR = 0



Try restoration using a better estimate of the noise-to-signal-power ratio.

```
estimated_nsr = noise_var / var(I(:));  
wnr3 = deconvwnr(blurred_noisy, PSF, estimated_nsr);  
figure, imshow(wnr3)  
title('Restoration of Blurred, Noisy Image Using Estimated NSR');
```

Restoration of Blurred, Noisy Image Using Estimated NSR



References

"Digital Image Processing", R. C. Gonzalez & R. E. Woods, Addison-Wesley Publishing Company, Inc., 1992.

See Also

[deconvblind](#) | [deconvlucy](#) | [deconvreg](#) | [edgetaper](#) | [otf2psf](#) | [padarray](#) | [psf2otf](#)

Introduced before R2006a

decorrstretch

Apply decorrelation stretch to multichannel image

Syntax

```
S = decorrstretch(A)
S = decorrstretch(A,name,value...)
```

Description

`S = decorrstretch(A)` applies a decorrelation stretch to an `m`-by-`n`-by-`nBands` image `A` and returns the result in `S`. `S` has the same size and class as `A`, and the mean and variance in each band are the same as in `A`. `A` can be an RGB image (`nBands = 3`) or can have any number of spectral bands.

The primary purpose of decorrelation stretch is visual enhancement. Decorrelation stretching is a way to enhance the color differences in an image.

`S = decorrstretch(A,name,value...)` applies a decorrelation stretch to the image `A`, subject to optional control parameters.

Examples

Highlight Color Differences in Forest Scene

This example shows how to use decorrelation stretching to highlight elements in a forest image by exaggerating the color differences.

Read an image into the workspace.

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

Apply decorrelation stretching using `decorrstretch`.

```
S = decorrstretch(ind2rgb(X,map), 'tol',0.01);
```

Display the original image and the enhanced image.

```
figure
imshow(X,map)
title('Original Image')
```

Original Image



```
figure
imshow(S)
title('Enhanced Image')
```

Enhanced Image



Input Arguments

A — Image to be enhanced
nonsparse, real, N-D array

Image to be stretched, specified as a nonsparse, real, N-D array. The image **A** is a multichannel image, such as, an RGB image (`nBands = 3`) or an image with any number of spectral bands.

Example:

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'Mode', 'covariance'

'Mode' — Decorrelation method

'correlation' (default) | 'correlation' or 'covariance'

Decorrelation method, specified as the values 'correlation' or 'covariance'. 'correlation' uses the eigen decomposition of the band-to-band correlation matrix. 'covariance' uses the eigen decomposition of the band-to-band covariance matrix.

Data Types: char

'TargetMean' — Values that the band-means of the output image must match

1-by-nBands vector containing the sample mean of each band (preserving the band-wise means) (default) | real scalar or vector of class `double` and of length nBands.

Values that the band-means of the output image must match, specified as a real scalar or vector of class `double` and of length nBands. If values need to be clamped to the standard range of the input/output image class, it can impact the results.

`targetmean` must be of class `double`, but uses the same values as the pixels in the input image. For example, if `A` is class `uint8`, then `127.5` would be reasonable value.

Data Types: `double`

'TargetSigma' — Values that the standard deviations of the individual bands of the output image must match

1-by-nBands vector containing the standard deviation of each band (preserving the band-wise variances) (default) | real, positive scalar or vector of class `double` and of length nBands

Values that the standard deviations of the individual bands of the output image must match, specified as a real, positive scalar or vector of class `double` and of length nBands. If values need to be clamped to the standard range of the input/output image class, it can impact the results. Ignored for uniform (zero-variance) bands.

`targetsigma` must be class `double`, but uses the same values and the pixels in the input image. For example, if `A` is of class `uint8`, then `and` and `50.0` would be reasonable value.

Data Types: `double`

'Tol' — Linear contrast stretch to be applied following the decorrelation stretch

one- or two-element real vector of class `double`

Linear contrast stretch to be applied following the decorrelation stretch, specified as a one- or two-element real vector of class `double`. Overrides use of `TargetMean` or `TargetSigma`. `TOL` has the same meaning as in `stretchlim`, where `TOL = [LOW_FRACT HIGH_FRACT]` specifies the fraction of the image to saturate at low and high intensities. If you specify `TOL` as a scalar value, then `LOW_FRACT = TOL` and `HIGH_FRACT = 1 - TOL`, saturating equal fractions at low and high intensities. If you do not specify a value for `TOL`, `decorrstretch` omits the linear contrast stretch.

Small adjustments to `TOL` can strongly affect the visual appearance of the output.

Data Types: `double`

'SampleSubs' — Subset of `A` used to compute the band-means, covariance, and correlation

cell array containing two arrays of pixel subscripts `{rowsubs, colsubs}`

Subset of `A` used to compute the band-means, covariance, and correlation, specified as a cell array containing two arrays of pixel subscripts `{rowsubs, colsubs}`. `rowsubs` and `colsubs` are vectors or matrices of matching size that contain row and column subscripts, respectively.

Use this option to reduce the amount of computation, to keep invalid or non-representative pixels from affecting the transformation, or both. For example, you can use `rowsubs` and `colsubs` to exclude areas of cloud cover. If not specified, `decorrstretch` uses all the pixels in `A`.

Data Types: `double`

Output Arguments

S — Output image

nonsparse, real, N-D array

S has the same size and class as **A**. The mean and variance in each band in **S** are the same as in **A**.

Tips

- The results of a straight decorrelation (without the contrast stretch option) may include values that fall outside the numerical range supported by the class `uint8` or `uint16` (negative values, or values exceeding $2^8 - 1$ or $2^{16} - 1$, respectively). In these cases, `decorrstretch` clamps its output to the supported range.
- For class `double`, `decorrstretch` clamps the output only when you provide a value for `TOL`, specifying a linear contrast stretch followed by clamping to the interval `[0 1]`.
- The optional parameters do not interact, except that a linear stretch usually alters both the band-wise means and band-wise standard deviations. Thus, while you can specify `targetmean` and `targetsigma` along with `TOL`, their effects will be modified.

Algorithms

A decorrelation stretch is a linear, pixel-wise operation in which the specific parameters depend on the values of actual and desired (target) image statistics. The vector **a** containing the value of a given pixel in each band of the input image **A** is transformed into the corresponding pixel **b** in output image **B** as follows:

$$\mathbf{b} = \mathbf{T} * (\mathbf{a} - \mathbf{m}) + \mathbf{m_target}.$$

a and **b** are `nBands`-by-1 vectors, **T** is an `nBands`-by-`nBands` matrix, and **m** and **m_target** are `nBands`-by-1 vectors such that

- **m** contains the mean of each band in the image, or in a subset of image pixels that you specify
- **m_target** contains the desired output mean in each band. The default choice is `m_target = m`.

The linear transformation matrix **T** depends on the following:

- The band-to-band sample covariance of the image, or of a subset of the image that you specify (the same subset as used for **m**), represented by matrix **COV**

- A desired output standard deviation in each band. This is conveniently represented by a diagonal matrix, `SIGMA_target`. The default choice is `SIGMA_target = SIGMA`, where `SIGMA` is the diagonal matrix containing the sample standard deviation of each band. `SIGMA` should be computed from the same pixels that were used for `m` and `Cov`, which means simply that:

$$\text{SIGMA}(k,k) = \text{sqrt}(\text{Cov}(k,k), k = 1, \dots, \text{nBands}).$$

`Cov`, `SIGMA`, and `SIGMA_target` are `nBands`-by-`nBands`, as are the matrices `Corr`, `LAMBDA`, and `V`, defined below.

The first step in computing `T` is to perform an eigen-decomposition of either the covariance matrix `Cov` or the correlation matrix

$$\text{Corr} = \text{inv}(\text{SIGMA}) * \text{Cov} * \text{inv}(\text{SIGMA}).$$

- In the correlation-based method, `Corr` is decomposed: `Corr = V LAMBDA V'`.
- In the covariance-based method, `Cov` is decomposed: `Cov = V LAMBDA V'`.

`LAMBDA` is a diagonal matrix of eigenvalues and `V` is the orthogonal matrix that transforms either `Corr` or `Cov` to `LAMBDA`.

The next step is to compute a stretch factor for each band, which is the inverse square root of the corresponding eigenvalue. It is convenient to define a diagonal matrix `S` containing the stretch factors, such that:

$$\text{S}(k,k) = 1 / \text{sqrt}(\text{LAMBDA}(k,k)).$$

Finally, matrix `T` is computed from either

$$\text{T} = \text{SIGMA_target} \text{ V S V' inv}(\text{SIGMA}) \quad (\text{correlation-based method})$$

or

$$\text{T} = \text{SIGMA_target} \text{ V S V' } \quad (\text{covariance-based method}).$$

The two methods yield identical results if the band variances are uniform.

Substituting `T` into the expression for `b`:

$$\text{b} = \text{m_target} + \text{SIGMA_target} \text{ V S V' inv}(\text{SIGMA}) * (\text{a} - \text{m})$$

or

$$b = m_target + SIGMA_target V S V' * (a - m)$$

and reading from right to left, you can see that the decorrelation stretch:

- 1 Removes a mean from each band
- 2 Normalizes each band by its standard deviation (correlation-based method only)
- 3 Rotates the bands into the eigenspace of `Corr` or `Cov`
- 4 Applies a stretch `S` in the eigenspace, leaving the image decorrelated and normalized in the eigenspace
- 5 Rotates back to the original band-space, where the bands remain decorrelated and normalized
- 6 Rescales each band according to `SIGMA_target`
- 7 Restores a mean in each band.

See Also

See Also

`imadjust` | `stretchlim`

Introduced before R2006a

demosaic

Convert Bayer pattern encoded image to truecolor image

Syntax

```
RGB = demosaic(I,sensorAlignment)
```

Description

`RGB = demosaic(I,sensorAlignment)` converts the Bayer pattern encoded image, `I`, to the truecolor image, `RGB`, using gradient-corrected linear interpolation. `sensorAlignment` specifies the Bayer pattern.

A Bayer filter mosaic, or color filter array, refers to the arrangement of color filters that let each sensor in a single-sensor digital camera record only red, green, or blue data. The patterns emphasize the number of green sensors to mimic the human eye's greater sensitivity to green light. The `demosaic` function uses interpolation to convert the two-dimensional Bayer-encoded image into the truecolor image.

Examples

Convert a Bayer Pattern Encoded Image To an RGB Image

Convert a Bayer pattern encoded image that was photographed by a camera with a sensor alignment of `'bggr'`.

```
I = imread('mandi.tif');  
J = demosaic(I,'bggr');  
imshow(I);  
figure, imshow(J);
```





Input Arguments

I — Bayer-pattern encoded image

M-by-*N* array of intensity values

Bayer-pattern encoded image, specified as an *M*-by-*N* array of intensity values. I must have at least 5 rows and 5 columns.

Data Types: uint8 | uint16 | uint32

sensorAlignment — Bayer pattern

'gbrg' | 'grbg' | 'bggr' | 'rggb'

Bayer pattern, specified as one of the values in the following table. Each value represents the order of the red, green, and blue sensors by describing the four pixels in the upper-left corner of the image (left-to-right, top-to-bottom).

Pattern	2-by-2 Sensor Alignment				
'gbrg'	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr> <td style="padding: 5px;">Green</td> <td style="padding: 5px;">Blue</td> </tr> <tr> <td style="padding: 5px;">Red</td> <td style="padding: 5px;">Green</td> </tr> </table>	Green	Blue	Red	Green
Green	Blue				
Red	Green				
'grbg'	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr> <td style="padding: 5px;">Green</td> <td style="padding: 5px;">Red</td> </tr> <tr> <td style="padding: 5px;">Blue</td> <td style="padding: 5px;">Green</td> </tr> </table>	Green	Red	Blue	Green
Green	Red				
Blue	Green				
'bggr'	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr> <td style="padding: 5px;">Blue</td> <td style="padding: 5px;">Green</td> </tr> <tr> <td style="padding: 5px;">Green</td> <td style="padding: 5px;">Red</td> </tr> </table>	Blue	Green	Green	Red
Blue	Green				
Green	Red				
'rggb'	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr> <td style="padding: 5px;">Red</td> <td style="padding: 5px;">Green</td> </tr> <tr> <td style="padding: 5px;">Green</td> <td style="padding: 5px;">Blue</td> </tr> </table>	Red	Green	Green	Blue
Red	Green				
Green	Blue				

Data Types: char

Output Arguments

RGB — RGB image

M-by-*N*-by-3 numeric array

RGB image, returned as an *M*-by-*N*-by-3 numeric array the same class as **I**.

References

- [1] Malvar, H.S., L. He, and R. Cutler, *High quality linear interpolation for demosaicing of Bayer-patterned color images*. ICASPP, Volume 34, Issue 11, pp. 2274-2282, May 2004.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- `sensorAlignment` must be a compile-time constant.

See Also

Introduced in R2007b

dicomanon

Anonymize DICOM file

Syntax

```
dicomanon(file_in, file_out)
dicomanon(..., 'keep', FIELDS)
dicomanon(..., 'update', ATTRS)
dicomanon(..., 'WritePrivate', TF)
dicomanon(..., 'UseVRHeuristic', TF)
```

Description

`dicomanon(file_in, file_out)` removes confidential medical information from the DICOM file `file_in` and creates a new file `file_out` with the modified values. Image data and other attributes are unmodified.

`dicomanon(..., 'keep', FIELDS)` modifies all of the confidential data except for those listed in `FIELDS`, which is a cell array of field names. This syntax is useful for keeping metadata that does not uniquely identify the patient but is useful for diagnostic purposes (e.g., `PatientAge`, `PatientSex`, etc.).

Note Keeping certain fields might compromise patient confidentiality.

`dicomanon(..., 'update', ATTRS)` modifies the confidential data and updates particular confidential data. `ATTRS` is a structure whose fields are the names of the attributes to preserve. The structure values are the attribute values. Use this syntax to preserve the Study/Series/Image hierarchy or to replace a specific value with a more generic property (e.g., remove `PatientBirthDate` but keep a computed `PatientAge`).

`dicomanon(..., 'WritePrivate', TF)` specifies whether `dicomanon` should write nonstandard attributes to the anonymized file. If `TF` is true, `dicomanon` includes private extensions in the file, which could compromise patient confidentiality. The default value is `false`.

`dicomanon(..., 'UseVRHeuristic', TF)` instructs the parser to use a heuristic to help read certain noncompliant files which switch value representation (VR) modes incorrectly. `dicomanon` displays a warning if the heuristic is employed. When `TF` is true (the default), a small number of compliant files will not be read correctly. Set `TF` to false to read these compliant files. Compliant files are always written.

For information about the fields that will be modified or removed, see DICOM Supplement 55 from <http://medical.nema.org/>.

Examples

Remove All Confidential Metadata from DICOM File

Create a version of a DICOM file with all the personal information removed.

```
dicomanon('US-PAL-8-10x-echo.dcm', 'US-PAL-anonymized.dcm');
```

Create a version of a DICOM file with personal information removed, keeping certain fields that could be useful for training.

```
dicomanon('US-PAL-8-10x-echo.dcm', 'US-PAL-anonymized.dcm', 'keep', ...  
          {'PatientAge', 'PatientSex', 'StudyDescription'})
```

Anonymize a series of images, keeping the hierarchy.

```
values.StudyInstanceUID = dicomuid;  
values.SeriesInstanceUID = dicomuid;  
  
d = dir('*.dcm');  
for p = 1:numel(d)  
    dicomanon(d(p).name, sprintf('anon%d.dcm', p), ...  
             'update', values)  
end
```

See Also

`dicominfo` | `dicomdict` | `dicomdisp` | `dicomwrite` | `dicomlookup` | `dicomread` | `dicomuid`

Introduced before R2006a

dicomdict

Get or set active DICOM data dictionary

Syntax

```
dicomdict('set',dictionary)
dictionary = dicomdict('get')
dicomdict('factory')
```

Description

`dicomdict('set',dictionary)` sets the Digital Imaging and Communications in Medicine (DICOM) data dictionary to the value stored in `dictionary`, a character vector containing the filename of the dictionary. DICOM-related functions use this dictionary by default, unless a different dictionary is provided at the command line.

`dictionary = dicomdict('get')` returns a character vector containing the filename of the stored DICOM data dictionary.

`dicomdict('factory')` resets the DICOM data dictionary to its default startup value.

Note The default data dictionary is a MAT-file, `dicom-dict.mat`. The toolbox also includes a text version of this default data dictionary, `dicom-dict.txt`. If you want to create your own DICOM data dictionary, open the `dicom-dict.txt` file in a text editor, modify it, and save it under another name.

Examples

Return Filename of Stored DICOM Dictionary

Determine the name of the stored DICOM dictionary.

```
dictionary = dicomdict('get');
```

See Also

dicomanon | dicominfo | dicomdisp | dicomwrite | dicomlookup | dicomread |
dicomuid

Introduced before R2006a

dicomdisp

Display DICOM file structure

Syntax

```
dicomdisp(filename)
dicomdisp( ____,Name,Value)
```

Description

`dicomdisp(filename)` reads the metadata from the compliant DICOM file specified in the character vector `filename` and displays the metadata at the command prompt. `dicomdisp` can be helpful when debugging issues with DICOM files.

`dicomdisp(____,Name,Value)` reads the metadata using name-value pairs to control aspects of the operation.

Examples

View Metadata from DICOM File

Read the metadata from DICOM file.

```
dicomdisp('CT-MON02-16-ankle.dcm');
```

```
File: B:\matlab\toolbox\images\imdata\CT-MON02-16-ankle.dcm (525436 bytes)
```

```
Read on an IEEE little-endian machine.
```

```
File begins with group 0002 metadata at byte 132.
```

```
Transfer syntax: 1.2.840.10008.1.2 (Implicit VR Little Endian).
```

```
DICOM Information object: 1.2.840.10008.5.1.4.1.1.7 (Secondary Capture Image Storage).
```

Location	Level	Tag	VR	Size	Name	Data
0000132	0	(0002,0000)	UL	4 bytes	- FileMetaInformationGroupLength	*Bin
0000144	0	(0002,0001)	OB	2 bytes	- FileMetaInformationVersion	*Bin
0000158	0	(0002,0002)	UI	26 bytes	- MediaStorageSOPClassUID	[1.2
0000192	0	(0002,0003)	UI	50 bytes	- MediaStorageSOPInstanceUID	[1.2
0000250	0	(0002,0010)	UI	18 bytes	- TransferSyntaxUID	[1.2
0000276	0	(0002,0012)	UI	18 bytes	- ImplementationClassUID	[1.2

0000302	0	(0002,0013)	SH	6 bytes	- ImplementationVersionName	[1_2
0000316	0	(0002,0016)	AE	12 bytes	- SourceApplicationEntityTitle	[CTN
0000336	0	(0008,0000)	UL	4 bytes	- IdentifyingGroupLength	*Bina
0000348	0	(0008,0008)	CS	20 bytes	- ImageType	[DER
0000376	0	(0008,0016)	UI	26 bytes	- SOPClassUID	[1.2
0000410	0	(0008,0018)	UI	50 bytes	- SOPInstanceUID	[1.2
0000468	0	(0008,0020)	DA	10 bytes	- StudyDate	[199
0000486	0	(0008,0021)	DA	10 bytes	- SeriesDate	[199
0000504	0	(0008,0023)	DA	10 bytes	- ContentDate	[199
0000522	0	(0008,0030)	TM	8 bytes	- StudyTime	[11:2
0000538	0	(0008,0031)	TM	8 bytes	- SeriesTime	[11:2
0000554	0	(0008,0033)	TM	8 bytes	- ContentTime	[11:2
0000570	0	(0008,0060)	CS	2 bytes	- Modality	[CT]
0000580	0	(0008,0064)	CS	4 bytes	- ConversionType	[WSD
0000592	0	(0008,0070)	LO	18 bytes	- Manufacturer	[GE M
0000618	0	(0008,0080)	LO	18 bytes	- InstitutionName	[JFK
0000644	0	(0008,0090)	PN	10 bytes	- ReferringPhysicianName	[Anon
0000662	0	(0008,1010)	SH	8 bytes	- StationName	[CTO
0000678	0	(0008,1030)	LO	8 bytes	- StudyDescription	[RT A
0000694	0	(0008,1060)	PN	10 bytes	- PhysicianReadingStudy	[Anon
0000712	0	(0008,1070)	PN	10 bytes	- OperatorName	[Anon
0000730	0	(0008,1090)	LO	12 bytes	- ManufacturerModelName	[GENE
0000750	0	(0010,0000)	UL	4 bytes	- PatientGroupLength	*Bina
0000762	0	(0010,0010)	PN	10 bytes	- PatientName	[Anon
0000780	0	(0018,0000)	UL	4 bytes	- AcquisitionGroupLength	*Bina
0000792	0	(0018,1020)	LO	2 bytes	- SoftwareVersion	[03]
0000802	0	(0020,0000)	UL	4 bytes	- RelationshipGroupLength	*Bina
0000814	0	(0020,000D)	UI	48 bytes	- StudyInstanceUID	[1.2
0000870	0	(0020,000E)	UI	48 bytes	- SeriesInstanceUID	[1.2
0000926	0	(0020,0011)	IS	4 bytes	- SeriesNumber	[365
0000938	0	(0020,0013)	IS	2 bytes	- InstanceNumber	[1]
0000948	0	(0028,0000)	UL	4 bytes	- ImagePresentationGroupLength	*Bina
0000960	0	(0028,0002)	US	2 bytes	- SamplesPerPixel	*Bina
0000970	0	(0028,0004)	CS	12 bytes	- PhotometricInterpretation	[MONO
0000990	0	(0028,0010)	US	2 bytes	- Rows	*Bina
0001000	0	(0028,0011)	US	2 bytes	- Columns	*Bina
0001010	0	(0028,0100)	US	2 bytes	- BitsAllocated	*Bina
0001020	0	(0028,0101)	US	2 bytes	- BitsStored	*Bina
0001030	0	(0028,0102)	US	2 bytes	- HighBit	*Bina
0001040	0	(0028,0103)	US	2 bytes	- PixelRepresentation	*Bina
0001050	0	(0028,0106)	US	2 bytes	- SmallestImagePixelValue	*Bina
0001060	0	(0028,0120)	US	2 bytes	- PixelPaddingValue	*Bina
0001070	0	(0028,1050)	DS	4 bytes	- WindowCenter	[102-
0001082	0	(0028,1051)	DS	4 bytes	- WindowWidth	[409

0001094	0	(0028,1052)	DS	6 bytes	- RescaleIntercept	[-102
0001108	0	(0028,1053)	DS	2 bytes	- RescaleSlope	[1]
0001118	0	(0028,1054)	LO	2 bytes	- RescaleType	[US]
0001128	0	(7FE0,0000)	UL	4 bytes	- PixelDataGroupLength	*Bin
0001140	0	(7FE0,0010)	OW	524288 bytes	- PixelData	[]

Input Arguments

filename — Name of DICOM file

character vector

Name of DICOM file, specified as a character vector .

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, . . . , **NameN**, **ValueN**.

Example: `dicomdisp('CT-MON02-16-ankle.dcm','UseVRHeuristic',false)`

'dictionary' — Name of DICOM data dictionary

dicom-dict.txt (default) | character vector

Name of DICOM data dictionary, specified as a character vector. When specified, `dicomdisp` uses the data dictionary to read the DICOM file. The file must be on the MATLAB search path.

Data Types: char

'UseVRHeuristic' — Read noncompliant DICOM files that switch VR modes incorrectly

true (default) | false

Read noncompliant DICOM files that switch value representation (VR) modes incorrectly, specified as the Boolean value `true` or `false`. When set to `true`, `dicomdisp` uses a heuristic to help read certain noncompliant DICOM files which switch value representation (VR) modes incorrectly. When `dicomdisp` uses this heuristic, it displays

a warning. When set to `true` (the default), `dicomdisp` might not read some compliant DICOM files correctly. To read these compliant files, set `UseVRHeuristic` to `false`.

Data Types: `logical`

See Also

See Also

`dicomanon` | `dicomdict` | `dicominfo` | `dicomlookup` | `dicomread` | `dicomuid` | `dicomwrite`

Topics

“Explicit Versus Implicit VR Attributes”

Introduced in R2015a

dicominfo

Read metadata from DICOM message

Syntax

```
info = dicominfo(filename)
info = dicominfo(filename, 'dictionary', D)
info = dicominfo( ____, 'UseVRHeuristic', TF)
info = dicominfo( ____, 'UseDictionaryVR', TF)
```

Description

`info = dicominfo(filename)` reads the metadata from the compliant Digital Imaging and Communications in Medicine (DICOM) file specified in the character vector `filename`.

`info = dicominfo(filename, 'dictionary', D)` uses the data dictionary file given in the character vector `D` to read the DICOM message. The file in `D` must be on the MATLAB search path. The default file is `dicom-dict.mat`.

`info = dicominfo(____, 'UseVRHeuristic', TF)` instructs the parser to use a heuristic to help read certain noncompliant files which switch value representation (VR) modes incorrectly. `dicominfo` displays a warning if the heuristic is used. When `TF` is `true` (the default), a small number of compliant files will not be read correctly. Set `TF` to `false` to read these compliant files.

`info = dicominfo(____, 'UseDictionaryVR', TF)` specifies whether the data types in `INFO` should conform to the data dictionary, regardless of what information is present in the file. The default value is `false`, which uses the file's VR codes even if they differ from the data dictionary. Most of the time it is unnecessary to set this field, since file contents and the data dictionary almost always agree. When `TF` is `false` (the default), `dicominfo` issues a warning when they do not agree. Set `TF` to `true` when the warning is issued and providing `info` to `dicomwrite` causes errors.

Examples

Read metadata from DICOM Message

Read metadata from DICOM message.

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

```
info = struct with fields:
```

```
    Filename: 'B:\matlab\toolbox\images\imdata\CT-MON02-16-ankle
    FileModDate: '18-Dec-2000 12:06:42'
    FileSize: 525436
    Format: 'DICOM'
    FormatVersion: 3
    Width: 512
    Height: 512
    BitDepth: 16
    ColorType: 'grayscale'
    FileMetaInformationGroupLength: 192
    FileMetaInformationVersion: [2×1 uint8]
    MediaStorageSOPClassUID: '1.2.840.10008.5.1.4.1.1.7'
    MediaStorageSOPInstanceUID: '1.2.840.113619.2.1.2411.1031152382.365.1.73616924
    TransferSyntaxUID: '1.2.840.10008.1.2'
    ImplementationClassUID: '1.2.840.113619.6.5'
    ImplementationVersionName: '1_2_5'
    SourceApplicationEntityTitle: 'CTN_STORAGE'
    IdentifyingGroupLength: 414
    ImageType: 'DERIVED\SECONDARY\3D'
    SOPClassUID: '1.2.840.10008.5.1.4.1.1.7'
    SOPInstanceUID: '1.2.840.113619.2.1.2411.1031152382.365.1.73616924
    StudyDate: '1993.04.30'
    SeriesDate: '1993.04.30'
    ContentDate: '1993.04.30'
    StudyTime: '11:27:24'
    SeriesTime: '11:27:24'
    ContentTime: '11:27:24'
    Modality: 'CT'
    ConversionType: 'WSD'
    Manufacturer: 'GE MEDICAL SYSTEMS'
    InstitutionName: 'JFK IMAGING CENTER'
    ReferringPhysicianName: [1×1 struct]
    StationName: 'CT010C0'
    StudyDescription: 'RT ANKLE'
    PhysicianReadingStudy: [1×1 struct]
```

```
      OperatorName: [1×1 struct]
    ManufacturerModelName: 'GENESIS_ZEUS'
      PatientGroupLength: 18
        PatientName: [1×1 struct]
    AcquisitionGroupLength: 10
      SoftwareVersion: '03'
    RelationshipGroupLength: 134
      StudyInstanceUID: '1.2.840.113619.2.1.1.322987881.621.736170080.681'
      SeriesInstanceUID: '1.2.840.113619.2.1.2411.1031152382.365.736169244'
      SeriesNumber: 365
      InstanceNumber: 1
    ImagePresentationGroupLength: 168
      SamplesPerPixel: 1
    PhotometricInterpretation: 'MONOCHROME2'
      Rows: 512
      Columns: 512
      BitsAllocated: 16
      BitsStored: 16
      HighBit: 15
      PixelRepresentation: 1
    SmallestImagePixelValue: 0
      PixelPaddingValue: 0
      WindowCenter: 1024
      WindowWidth: 4095
      RescaleIntercept: -1024
      RescaleSlope: 1
      RescaleType: 'US'
    PixelDataGroupLength: 524296
```

See Also

dicomanon | dicomdict | dicomdisp | dicomwrite | dicomlookup | dicomread |
dicomuid

Introduced before R2006a

dicomlookup

Find attribute in DICOM data dictionary

Syntax

```
name = dicomlookup(group, element)
[group, element] = dicomlookup(name)
```

Description

`name = dicomlookup(group, element)` looks into the current DICOM data dictionary for the attribute with the specified `group` and `element` tag and returns a character vector containing the `name` of the attribute. `group` and `element` can contain either a decimal value or hexadecimal value.

`[group, element] = dicomlookup(name)` looks into the current DICOM data dictionary for the attribute specified byname and returns the `group` and `element` tags associated with the attribute. The values are returned as decimal values.

Examples

Find Names of DICOM attributes Using Their Tags

Find the names of DICOM attributes using their tags.

```
name1 = dicomlookup('7FE0', '0010')
```

```
name1 =  
'PixelData'
```

```
name2 = dicomlookup(40, 4)
```

```
name2 =  
'PhotometricInterpretation'
```

Look up a DICOM attribute's tag (GROUP and ELEMENT) using its name.

```
[group, element] = dicomlookup('TransferSyntaxUID')
group = 2
element = 16
```

Examine the metadata of a DICOM file. This returns the same value even if the data dictionary changes.

```
metadata = dicominfo('CT-MON02-16-ankle.dcm');
metadata.(dicomlookup('0028', '0004'))

ans =
'MONOCHROME2'
```

See Also

[dicomanon](#) | [dicomdict](#) | [dicomdisp](#) | [dicominfo](#) | [dicomwrite](#) | [dicomread](#) | [dicomuid](#)

Introduced in R2006b

dicomread

Read DICOM image

Syntax

```
X = dicomread(filename)
X = dicomread(info)
[X,map] = dicomread(...)
[X,map,alpha] = dicomread(...)
[X,map,alpha,overlays] = dicomread(...)
[...] = dicomread(filename, 'frames', v)
[...] = dicomread( ____, 'UseVRHeuristic',TF)
```

Description

`X = dicomread(filename)` reads the image data from the compliant Digital Imaging and Communications in Medicine (DICOM) file `filename`. For single-frame grayscale images, `X` is an M-by-N array. For single-frame true-color images, `X` is an M-by-N-by-3 array. Multiframe images are always 4-D arrays.

`X = dicomread(info)` reads the image data from the message referenced in the DICOM metadata structure `info`. The `info` structure is produced by the `dicominfo` function.

`[X,map] = dicomread(...)` returns the image `X` and the colormap `map`. If `X` is a grayscale or true-color image, `map` is empty.

`[X,map,alpha] = dicomread(...)` returns the image `X`, the colormap `map`, and an alpha channel matrix for `X`. The values of `alpha` are 0 if the pixel is opaque; otherwise they are row indices into `map`. The RGB value in `map` should be substituted for the value in `X` to use `alpha`. `alpha` has the same height and width as `X` and is 4-D for a multiframe image.

`[X,map,alpha,overlays] = dicomread(...)` returns the image `X`, the colormap `map`, an alpha channel matrix for `X`, and any overlays from the DICOM file. Each overlay is a 1-bit black and white image with the same height and width as `X`. If multiple

overlays are present in the file, `overlays` is a 4-D multiframe image. If no overlays are in the file, `overlays` is empty.

`[...] = dicomread(filename, 'frames', v)` reads only the frames in the vector `v` from the image. `v` must be an integer scalar, a vector of integers, or `'all'`. The default value is `'all'`.

`[...] = dicomread(___, 'UseVRHeuristic', TF)` instructs the parser to use a heuristic to help read certain noncompliant files which switch value representation (VR) modes incorrectly. `dicomread` displays a warning if the heuristic is used. When `TF` is true (the default), a small number of compliant files will not be read correctly. Set `TF` to false to read these compliant files.

Class Support

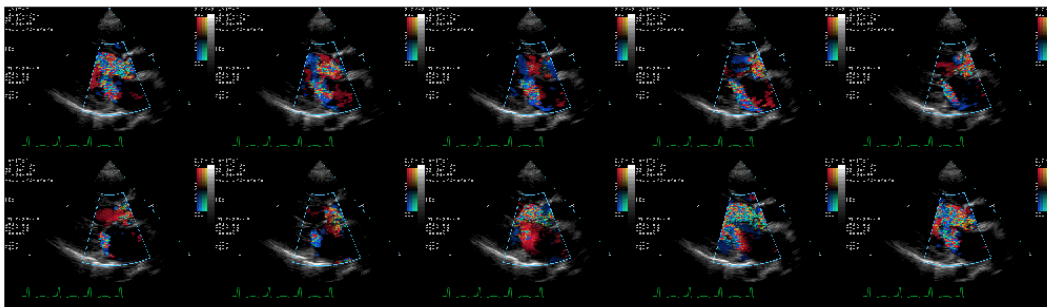
`X` can be `uint8`, `int8`, `uint16`, or `int16`. `map` must be `double`. `alpha` has the same size and type as `X`. `overlays` is a logical array.

Examples

Read DICOM Files

Read indexed image from DICOM file and display it using `montage`.

```
[X, map] = dicomread('US-PAL-8-10x-echo.dcm');
montage(X, map, 'Size', [2 5]);
```



Read image from DICOM file and display it using `imshow`.

```
info = dicominfo('CT-MON02-16-ankle.dcm');  
Y = dicomread(info);  
figure  
imshow(Y,[]);
```




Tips

- This function reads imagery from files with one of these pixel formats:

- Little-endian, implicit VR, uncompressed
- Little-endian, explicit VR, uncompressed
- Big-endian, explicit VR, uncompressed
- JPEG (lossy or lossless)
- JPEG2000 (lossy or lossless)
- Run-length Encoding (RLE)
- GE implicit VR, LE with uncompressed BE pixels (1.2.840.113619.5.2)

See Also

`dicomanon` | `dicomdict` | `dicomdisp` | `dicominfo` | `dicomlookup` | `dicomwrite` | `dicomuid`

Introduced before R2006a

dicomuid

Generate DICOM unique identifier

Syntax

```
UID = dicomuid
```

Description

`UID = dicomuid` returns a character vector containing a new DICOM unique identifier, UID.

Multiple calls to `dicomuid` produce globally unique values. Two calls to `dicomuid` always return different values.

See Also

`dicomanon` | `dicomdict` | `dicomdisp` | `dicominfo` | `dicomlookup` | `dicomread` | `dicomwrite`

Introduced before R2006a

dicomwrite

Write images as DICOM files

Syntax

```
dicomwrite(X, filename)
dicomwrite(X, map, filename)
dicomwrite(..., param1, value1, param2, value2, ...)
dicomwrite(..., 'ObjectType', IOD,...)
dicomwrite(..., 'SOPClassUID', UID,...)
dicomwrite(..., meta_struct,...)
dicomwrite(..., info,...)
status = dicomwrite(...)
```

Description

`dicomwrite(X, filename)` writes the binary, grayscale, or truecolor image `X` to the file `filename`, where `filename` is a character vector specifying the name of the Digital Imaging and Communications in Medicine (DICOM) file to create.

`dicomwrite(X, map, filename)` writes the indexed image `X` with colormap `map`.

`dicomwrite(..., param1, value1, param2, value2, ...)` specifies optional metadata to write to the DICOM file or parameters that affect how the file is written. `param1` is a character vector containing the metadata attribute name or a `dicomwrite`-specific option. `value1` is the corresponding value for the attribute or option.

To find a list of the DICOM attributes that you can specify, see the data dictionary file, `dicom-dict.txt`, included with the Image Processing Toolbox software. The following table lists the options that you can specify, in alphabetical order. Default values are enclosed in braces (`{}`).

Option Name	Description
'CompressionMode'	Type of compression to use when storing the image. Possible values: { 'None' }

Option Name	Description
	'JPEG lossless' 'JPEG lossy' 'JPEG2000 lossy' 'JPEG2000 lossless' 'RLE'
'CreateMode'	Specifies the method used for creating the data to put in the new file. Possible values: {'Create'} — Verify input values and generate missing data values. 'Copy' — Copy all values from the input and do not generate missing values.
'Dictionary'	Name of a DICOM data dictionary, specified as a character vector.
'Endian'	Byte ordering of the file. Possible values: 'Big' {'Little'} <hr/> Note: If VR is set to 'Explicit', 'Endian' must be 'Big'. dicomwrite ignores this value if 'CompressionMode' or 'TransferSyntax' is set.
'MultiframeSingleFile'	Logical value indicating whether multiframe imagery should be written to one file. When true (default), one file is created regardless of how many frames X contains. When false , one file is written for each frame in the image.

Option Name	Description
'TransferSyntax'	<p>A DICOM UID specifying the 'Endian', 'VR', and 'CompressionMode' options.</p> <hr/> <p>Note: If specified, <code>dicomwrite</code> ignores any values specified for the 'Endian', 'VR', and 'CompressionMode' options. The <code>TransferSyntax</code> value encodes values for these options.</p>
'UseMetadataBitDepths'	<p>Logical value that indicates whether to preserve the metadata values 'BitStored', 'BitsAllocated', and 'HighBit'. When <code>true</code>, <code>dicomwrite</code> preserves existing values. When <code>false</code> (default), <code>dicomwrite</code> computes these values based on the datatype of the pixel data. When 'CreateMode' is 'Create', <code>dicomwrite</code> ignores this field.</p>
'VR'	<p>Write two-letter value representation (VR) code to file. Possible values:</p> <p>'explicit' — Write VR to file.</p> <p>{'implicit'} — Infer from data dictionary.</p> <hr/> <p>Note: If you specify the 'Endian' value 'Big', you must specify 'Explicit'.</p>
'WritePrivate'	<p>Logical value indicating whether private data should be written to the file. Possible values: <code>true</code> — Write private data to file.</p> <p>{false} — Do not write private data.</p>

`dicomwrite(..., 'ObjectType', IOD, ...)` writes a file containing the necessary metadata for a particular type of DICOM Information Object (IOD). Supported IODs are

- 'Secondary Capture Image Storage' (default)
- 'CT Image Storage'
- 'MR Image Storage'

`dicomwrite(..., 'SOPClassUID', UID, ...)` provides an alternate method for specifying the IOD to create. UID is the DICOM unique identifier corresponding to one of the IODs listed above.

`dicomwrite(..., meta_struct, ...)` specifies optional metadata or file options in structure `meta_struct`. The names of fields in `meta_struct` must be the names of DICOM file attributes or options. The value of a field is the value you want to assign to the attribute or option.

`dicomwrite(..., info, ...)` specifies metadata in the metadata structure `info`, which is produced by the `dicominfo` function. For more information about this structure, see `dicominfo`.

`status = dicomwrite(...)` returns information about the metadata and the descriptions used to generate the DICOM file. This syntax can be useful when you specify an `info` structure that was created by `dicominfo` to the `dicomwrite` function. An `info` structure can contain many fields. If no metadata was specified, `dicomwrite` returns an empty matrix (`[]`).

The structure returned by `dicomwrite` contains these fields:

Field	Description
'BadAttribute'	The attribute's internal description is bad. It might be missing from the data dictionary or have incorrect data in its description.
'MissingCondition'	The attribute is conditional but no condition has been provided for when to use it.
'MissingData'	No data was provided for an attribute that must appear in the file.
'SuspectAttribute'	Data in the attribute does not match a list of enumerated values in the DICOM specification.

Examples

Write Data to DICOM File

Read a CT image from the sample DICOM file included with the toolbox.

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

Write the CT image to a file, creating a secondary capture image.

```
dicomwrite(X, 'sc_file.dcm');
```

Write the CT image, X, to a DICOM file along with its metadata. Use the `dicominfo` function to retrieve metadata from a DICOM file.

```
metadata = dicominfo('CT-MONO2-16-ankle.dcm');  
dicomwrite(X, 'ct_file.dcm', metadata);
```

Copy all metadata from one file to another. When you set the 'CreateMode' parameter to 'copy', `dicomwrite` does not verify the metadata written to the file.

```
dicomwrite(X, 'ct_copy.dcm', metadata, 'CreateMode', 'copy');
```

Tips

The DICOM format specification lists several Information Object Definitions (IODs) that can be created. These IODs correspond to images and metadata produced by different real-world modalities (e.g., MR, X-ray, Ultrasound, etc.). For each type of IOD, the DICOM specification defines the set of metadata that must be present and possible values for other metadata.

`dicomwrite` fully implements a limited number of these IODs, listed above in the `ObjectType` syntax. For these IODs, `dicomwrite` verifies that all required metadata attributes are present, creates missing attributes if necessary, and specifies default values where possible. Using these supported IODs is the best way to ensure that the files you create conform to the DICOM specification. This is `dicomwrite` default behavior and corresponds to the `CreateMode` option value of 'Create'.

To write DICOM files for IODs that `dicomwrite` doesn't implement, use the 'Copy' value for the `CreateMode` option. In this mode, `dicomwrite` writes the image data to a file including the metadata that you specify as a parameter, shown above in the `info` syntax. The purpose of this option is to take metadata from an existing file of the same modality or IOD and use it to create a new DICOM file with different image pixel data.

Note Because `dicomwrite` copies metadata to the file without verification in 'copy' mode, it is possible to create a DICOM file that does not conform to the DICOM standard. For example, the file may be missing required metadata, contain superfluous metadata,

or the metadata may no longer correspond to the modality settings used to generate the original image. When using 'Copy' mode, make sure that the metadata you use is from the same modality and IOD. If the copy you make is unrelated to the original image, use `dicomuid` to create new unique identifiers for series and study metadata. See the IOD descriptions in Part 3 of the DICOM specification for more information on appropriate IOD values.

See Also

`dicomanon` | `dicomdict` | `dicomdisp` | `dicominfo` | `dicomlookup` | `dicomread` | `dicomuid`

Introduced before R2006a

dpxinfo

Read metadata from DPX file

Syntax

```
metadata = dpxinfo(filename)
```

Description

`metadata = dpxinfo(filename)` reads information about the image contained in the DPX file specified by `filename`. `metadata` is a structure containing the file details.

Digital Picture Exchange (DPX) is an ANSI standard file format commonly used for still-frame storage in digital intermediate post-production facilities and film labs.

Examples

Read Metadata from DPX File

Read metadata from DPX file into the workspace.

```
m = dpxinfo('peppers.dpx')
```

```
m = struct with fields:
```

```
    Filename: 'B:\matlab\toolbox\images\imdata\peppers.dpx'  
    FileModDate: '16-Mar-2015 09:57:26'  
    FileSize: 892828  
    Format: 'DPX'  
    FormatVersion: '2.0'  
    Width: 512  
    Height: 384  
    BitDepth: 36  
    ColorType: 'R,G,B'  
    FormatSignature: [88 80 68 83]  
    ByteOrder: 'Little-endian'  
    Orientation: 'Left-to-right, Top-to-bottom'  
    NumberOfImageElements: 1
```

```
        DataSign: {'Unsigned'}
AmplitudeTransferFunction: {'ITU-R 709-4'}
        Colorimetry: {'ITU-R 709-4'}
ChannelBitDepths: 12
PackingMethod: 0
        Encoding: {'None'}
```

Input Arguments

filename — Name of the DPX file

character vector

Name of a DPX file, specified as a character vector. **filename** can contain the absolute path to the file, the name of a file on the MATLAB path, or a relative path.

Data Types: char

Output Arguments

metadata — Information about the DPX image data

structure

Information about the DPX image data, returned as a structure.

See Also

See Also

dpxread

Introduced in R2015b

dpxread

Read DPX image

Syntax

```
X = dpxread(filename)
```

Description

`X = dpxread(filename)` reads image data from the DPX file specified by `filename`, returning the image `X`.

Digital Picture Exchange (DPX) is an ANSI standard file format commonly used for still-frame storage in digital intermediate post-production facilities and film labs.

Examples

Read and Visualize 12-bit RGB Image

Read image from DPX file into the workspace.

```
RGB = dpxread('peppers.dpx');
```

Create a scale factor based on the data range of the image data. The image needs to be scaled to span the 16-bit data range expected by `imshow`.

```
maxOfDataRange = 2^12 - 1;  
scaleFactor = intmax('uint16') / maxOfDataRange;
```

Display the image.

```
figure  
imshow(RGB * scaleFactor)
```



Input Arguments

filename — Name of the DPX file
character vector

Name of a DPX file, specified as a character vector. **filename** can contain the absolute path to the file, the name of a file on the MATLAB path, or a relative path.

Example: `RGB = dpxread('peppers.dpx');`

Data Types: `char`

Output Arguments

X — Image data from DPX file

real, nonsparse numeric array

Image data from DPX file, returned as a real, nonsparse numeric array of class `uint8` or `uint16`, depending on the bit depth of the pixels in `filename`.

See Also

See Also

`dpxinfo`

Introduced in R2015b

edge

Find edges in intensity image

Syntax

```
BW = edge(I)
```

```
BW = edge(I, 'Sobel')
```

```
BW = edge(I, 'Sobel', threshold)
```

```
BW = edge(I, 'Sobel', threshold, direction)
```

```
BW = edge(I, 'Sobel', threshold, direction, 'nothinning')
```

```
[BW, threshOut] = edge(I, 'Sobel', ___)
```

```
BW = edge(I, 'Prewitt')
```

```
BW = edge(I, 'Prewitt', threshold)
```

```
BW = edge(I, 'Prewitt', threshold, direction)
```

```
BW = edge(I, 'Prewitt', threshold, direction, 'nothinning')
```

```
[BW, threshOut] = edge(I, 'Prewitt', ___)
```

```
BW = edge(I, 'Roberts')
```

```
BW = edge(I, 'Roberts', threshold)
```

```
BW = edge(I, 'Roberts', threshold, 'nothinning')
```

```
[BW, threshOut] = edge(I, 'Roberts', threshold, 'nothinning')
```

```
BW = edge(I, 'log')
```

```
BW = edge(I, 'log', threshold)
```

```
BW = edge(I, 'log', threshold, sigma)
```

```
[BW, threshOut] = edge(I, 'log', ___)
```

```
BW = edge(I, 'zerocross', threshold, h)
```

```
[BW, threshOut] = edge(I, 'zerocross', ___)
```

```
BW = edge(I, 'Canny')
```

```
BW = edge(I, 'Canny', threshold)
```

```
BW = edge(I, 'Canny', threshold, sigma)
```

```
[BW, threshOut] = edge(I, 'Canny', ___)
```

```
BW = edge(I, 'approxCanny')
```

```
BW = edge(I, 'approxcanny', threshold)
```

```
[gpuarrayBW, threshOut] = edge(gpuarrayI, ___)
```

Description

`BW = edge(I)` returns a binary image `BW` containing 1s where the function finds edges in the input image `I` and 0s elsewhere. The input image `I` is an intensity or a binary image. `BW` is the same size as `I`.

By default, `edge` uses the Sobel edge detection method, but you can specify any of these other methods: Canny (or a Canny approximation), Laplacian of Gaussian (`log`), Prewitt, Roberts, or Zero-crossings. The parameters you specify vary depending on the method you choose. The following section detail the parameters supported by each method.

`BW = edge(I, 'Sobel')` detect edges using the Sobel method. This method finds edges using the Sobel approximation to the derivative. It returns edges at those points where the gradient of `I` is maximum.

`BW = edge(I, 'Sobel', threshold)` return all edges that are stronger than `threshold`. If you do not specify `threshold`, or is you specify empty brackets (`[]`), `edge` chooses the value automatically.

`BW = edge(I, 'Sobel', threshold, direction)` specify the direction in which the function looks for edges in the image: `'horizontal'`, `'vertical'`, or `'both'`.

`BW = edge(I, 'Sobel', threshold, direction, 'nothinning')` specify whether to skip the additional edge-thinning stage, `'nothinning'`. Skipping this stage can improve performance. The default value is `'thinning'`.

`[BW, threshOut] = edge(I, 'Sobel', ___)` returns the threshold value.

`BW = edge(I, 'Prewitt')` detect edges using the Prewitt method. This method finds edges using the Prewitt approximation to the derivative. It returns edges at those points where the gradient of `I` is maximum.

`BW = edge(I, 'Prewitt', threshold)` return all edges that are stronger than `threshold`. If you do not specify `threshold`, or is you specify empty brackets (`[]`), `edge` chooses the value automatically.

`BW = edge(I, 'Prewitt', threshold, direction)` specify the direction in which the function looks for edges in the image: 'horizontal', 'vertical', or 'both'.

`BW = edge(I, 'Prewitt', threshold, direction, 'nothinning')` specify whether to skip the additional edge-thinning stage, 'nothinning'. Skipping this stage can improve performance. The default value is 'thinning'.

`[BW, threshOut] = edge(I, 'Prewitt', ___)` returns the threshold value.

`BW = edge(I, 'Roberts')` detect edges using the Roberts method. This method finds edges using the Roberts approximation to the derivative. It returns edges at those points where the gradient of `I` is maximum.

`BW = edge(I, 'Roberts', threshold)` return all edges that are stronger than `threshold`. If you do not specify `threshold`, or is you specify empty brackets (`[]`), `edge` chooses the value automatically.

`BW = edge(I, 'Roberts', threshold, 'nothinning')` specify whether to skip the additional edge-thinning stage, 'nothinning'. Skipping this stage can improve performance. The default value is 'thinning'.

`[BW, threshOut] = edge(I, 'Roberts', threshold, 'nothinning')` returns the threshold value.

`BW = edge(I, 'log')` detect edges using the Laplacian of Gaussian ('log') method. This method finds edges by looking for zero-crossings after filtering `I` with a Laplacian of Gaussian filter.

`BW = edge(I, 'log', threshold)` return all edges that are stronger than `threshold`. If you do not specify `threshold`, or is you specify empty brackets (`[]`), `edge` chooses the value automatically.

`BW = edge(I, 'log', threshold, sigma)` specify `sigma`, the standard deviation of the 'log' filter. The default `sigma` is 2; the size of the filter is n -by- n , where $n = \text{ceil}(\text{sigma} * 3) * 2 + 1$.

`[BW, threshOut] = edge(I, 'log', ___)` returns the threshold value.

`BW = edge(I, 'zerocross', threshold, h)` detect edges using the 'zerocross' method. This method finds edges by looking for zero-crossings after filtering `I` with a filter that you specify, `h`. The `edge` function returns edges that are stronger than

`threshold`. If you do not specify `threshold`, or if you specify empty brackets (`[]`), `edge` chooses the threshold value automatically.

`[BW, threshOut] = edge(I, 'zerocross', ___)` returns the threshold value.

`BW = edge(I, 'Canny')` detect edges using the Canny method. The Canny method finds edges by looking for local maxima of the gradient of `I`. The `edge` function calculates the gradient using the derivative of a Gaussian filter. This method uses two thresholds to detect strong and weak edges, including weak edges in the output if they are connected to strong edges. By using two thresholds, the Canny method is less likely than the other methods to be fooled by noise, and more likely to detect true weak edges.

The Canny method is not supported on a GPU.

`BW = edge(I, 'Canny', threshold)` return all edges that are stronger than `threshold`. If you do not specify `threshold`, or if you specify empty brackets (`[]`), `edge` chooses the value automatically. `threshold` is a two-element vector in which the first element is the low threshold, and the second element is the high threshold. If you specify a scalar, `edge` uses this value for the high value and uses `threshold*0.4` for the low threshold.

`BW = edge(I, 'Canny', threshold, sigma)` specify `sigma`, the standard deviation of the Gaussian filter. The default `sigma` is `sqrt(2)`. `edge` chooses the size of the filter automatically, based on `sigma`.

`[BW, threshOut] = edge(I, 'Canny', ___)` returns the threshold values as a two-element vector.

`BW = edge(I, 'approxcanny')` detect edges using the approximate Canny method. The `'approxcanny'` method is an approximate version of the Canny edge detection algorithm that provides faster execution time at the expense of less precise detection. For the `approxcanny` method, floating point images are expected to be normalized in the range `[0 1]`.

`BW = edge(I, 'approxcanny', threshold)` specifies sensitivity thresholds for the `'approxcanny'` method. `threshold` is a two element vector, the first element of which specifies the lower threshold for edge strength, below which all edges are disregarded. The second element specifies the higher threshold, above which all edge pixels are preserved. The range of values allowed is between `[0 1]`. If you specify a scalar, `edge` uses this value for the high value and uses `threshold*0.4` for the low threshold. If you do not specify `threshold`, or if `threshold` is empty (`[]`), `edge` chooses low and high values automatically.

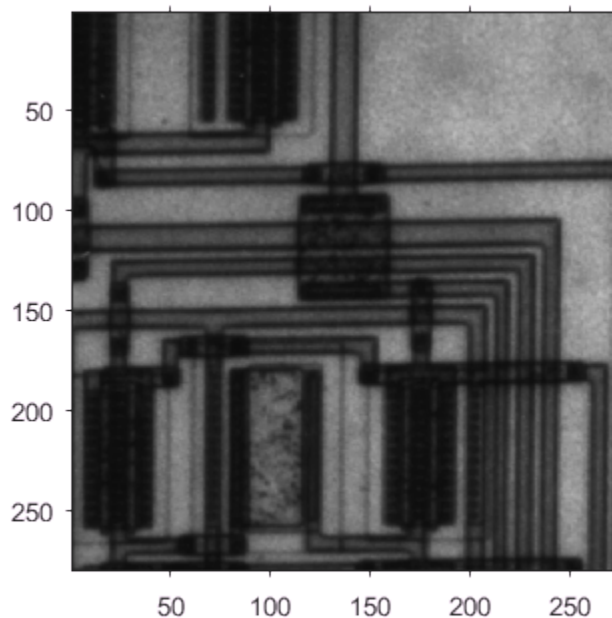
`[gpuarrayBW, threshOut] = edge(gpuarrayI, ___)` performs the edge detection operation on a GPU. The input image and the output image are `gpuArrays`. This syntax requires Parallel Computing Toolbox.

Examples

Compare Edge Detection Using Canny and Prewitt Methods

Read a grayscale image into the workspace and display it.

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



Find edges using the Canny method.

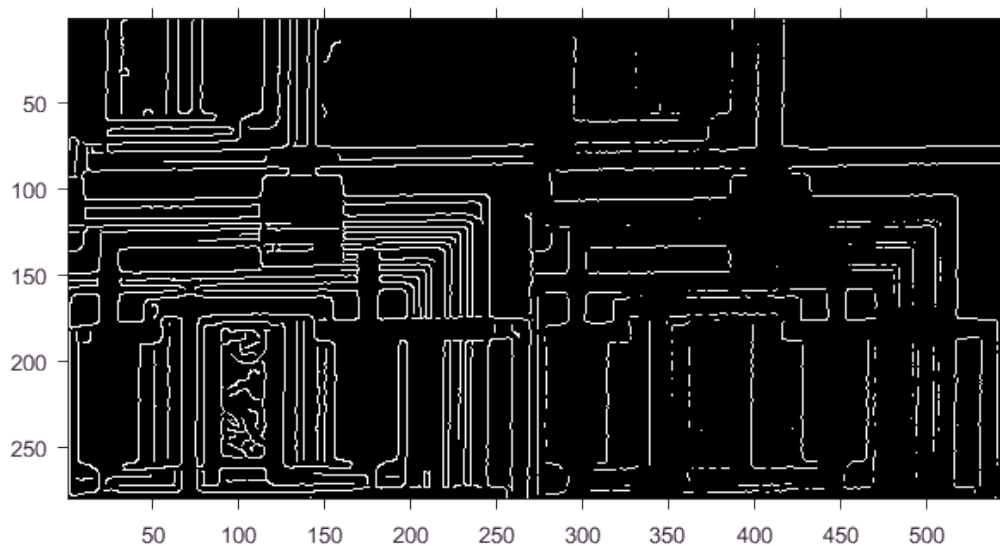
```
BW1 = edge(I, 'Canny');
```

Find edges using the Prewitt method.

```
BW2 = edge(I, 'Prewitt');
```

Display both results side-by-side.

```
imshowpair(BW1,BW2, 'montage')
```



Find Edges Using Prewitt Method on a GPU

Read grayscale image, creating a gpuArray.

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

Find edges using the Prewitt method.

```
BW = edge(I, 'prewitt');
```

Display results.

figure, imshow(BW)

Input Arguments

I — Input intensity or binary image

2-D, real, nonsparse numeric or logical array

Input intensity or binary image, specified as a 2-D, real, nonsparse, numeric, or logical array.

For the 'approxcanny' method, edge expects floating-point images to be normalized in the range [0 1].

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

threshold — Sensitivity threshold

numeric scalar | two-element vector (Canny and approxcanny methods only)

Sensitivity threshold, specified as a numeric scalar or, for the Canny and approxcanny methods only, a two-element vector. edge ignores all edges that are not stronger than threshold. If you do not specify threshold, or if you specify an empty array ([]), edge chooses the value automatically. For more information about this parameter, see “Tips” on page 1-385.

For the 'log' (Laplacian of Gaussian) and 'zerocross' methods, if you specify the threshold value 0, the output image has closed contours because it includes all the zero-crossings in the input image.

For the 'Canny' and 'zerocross' methods, if you specify the threshold value 0, the output image has closed contours because it includes all the zero-crossings in the input image.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

direction — Direction of edges to detect

'both' (default) | 'horizontal' | 'vertical'

Direction of edges to detect, specified as 'horizontal', 'vertical', or 'both'. This is only used with the Sobel and Prewitt methods.

Data Types: char

h — Filter

matrix

Filter, specified as a matrix.

Data Types: double

sigma — Standard deviation of the filter

scalar

Standard deviation of the filter, specified as a scalar. Supported by the Canny and log methods only.

Method	Description
'Canny'	Scalar value that specifies the standard deviation of the Gaussian filter. The default is <code>sqrt(2)</code> . <code>edge</code> chooses the size of the filter automatically, based on <code>sigma</code> .
'log' (Laplacian of Gaussian)	Scalar value that specifies the standard deviation of the Laplacian of Gaussian filter. The default is 2. The size of the filter is n-by-n, where <code>n=ceil(sigma*3)*2+1</code> .

Data Types: double

gpuarrayI — Input image

gpuArray

Input image, specified as a gpuArray.

Output Arguments

BW — Output binary image

logical array

Output binary image, returned as a logical array, the same size as `I`, with 1s where the function finds edges in `I` and 0s elsewhere.

threshOut — Threshold value used in the computation

numeric scalar

Threshold value used in the computation, returned as a numeric scalar.

gpuarrayBW — Output binary image when run on a GPU

gpuArray

Output binary image when run on a GPU, returned as a gpuArray.

Tips

- Notes about the `threshold` parameter:
 - For the gradient-magnitude edge detection methods (Sobel, Prewitt, Roberts), `edge` uses `threshold` to threshold the calculated gradient magnitude. For the zero-crossing methods, including Laplacian of Gaussian, `edge` uses `threshold` as a threshold for the zero-crossings. In other words, a large jump across zero is an edge, while a small jump is not.
 - The Canny method applies two thresholds to the gradient: a high threshold for low edge sensitivity and a low threshold for high edge sensitivity. `edge` starts with the low sensitivity result and then grows it to include connected edge pixels from the high sensitivity result. This helps fill in gaps in the detected edges.
 - In all cases, `edge` chooses the default threshold heuristically, depending on the input data. The best way to vary the threshold is to run `edge` once, capturing the calculated threshold as the second output argument. Then, starting from the value calculated by `edge`, adjust the threshold higher (fewer edge pixels) or lower (more edge pixels).
- The function `edge` changed in Version 7.2 (R2011a). Previous versions of the Image Processing Toolbox used a different algorithm for computing the Canny method. If you need the same results produced by the previous implementation, use the following syntax: `BW = edge(I, 'canny_old', ___)`
- The syntax `BW = edge(___,K)` has been removed. Use the `BW = edge(___,direction)` syntax instead.
- The syntax `edge(I, 'marr-hildreth', ___)` has been removed. Use the `edge(I, 'log', ___)` syntax instead.

References

- [1] Canny, John, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-8, No. 6, 1986, pp. 679-698.

[2] Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, pp. 478-488.

[3] Parker, James R., *Algorithms for Image Processing and Computer Vision*, New York, John Wiley & Sons, Inc., 1997, pp. 23-29.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- The `method`, `direction`, and `sigma` arguments must be compile-time constants.
- Nonprogrammatic syntaxes are not supported. For example, if you do not specify a return value, `edge` displays an image. This syntax is not supported.

See Also

See Also

`fspecial` | `gpuArray` | `imgradient` | `imgradientxy`

Introduced before R2006a

edgetaper

Taper discontinuities along image edges

Syntax

```
J = edgetaper(I,PSF)
```

Description

`J = edgetaper(I,PSF)` blurs the edges of the input image `I` using the point spread function `PSF`. The size of the `PSF` cannot exceed half of the image size in any dimension.

The output image `J` is the weighted sum of the original image `I` and its blurred version. The weighting array, determined by the autocorrelation function of `PSF`, makes `J` equal to `I` in its central region, and equal to the blurred version of `I` near the edges.

The `edgetaper` function reduces the ringing effect in image deblurring methods that use the discrete Fourier transform, such as `deconvwnr`, `deconvreg`, and `deconvlucy`.

Class Support

`I` and `PSF` can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. `J` is of the same class as `I`.

Examples

Blur the Edges of an Image

```
original = imread('cameraman.tif');  
PSF = fspecial('gaussian',60,10);  
edgesTapered = edgetaper(original,PSF);  
figure, imshow(original,[]);
```



```
figure, imshow(edgesTapered,[]);
```



See Also

[deconvlucy](#) | [deconvreg](#) | [deconvwnr](#) | [otf2psf](#) | [padarray](#) | [psf2otf](#)

Introduced before R2006a

entropy

Entropy of grayscale image

Syntax

`E = entropy(I)`

Description

`E = entropy(I)` returns `E`, a scalar value representing the entropy of grayscale image `I`. Entropy is a statistical measure of randomness that can be used to characterize the texture of the input image. Entropy is defined as

$$-\sum(p.\log_2(p))$$

where `p` contains the histogram counts returned from `imhist`. By default, `entropy` uses two bins for logical arrays and 256 bins for `uint8`, `uint16`, or `double` arrays.

`I` can be a multidimensional image. If `I` has more than two dimensions, the `entropy` function treats it as a multidimensional grayscale image and not as an RGB image.

Class Support

`I` can be `logical`, `uint8`, `uint16`, or `double` and must be real, nonempty, and nonsparse. `E` is `double`.

Notes

`entropy` converts any class other than `logical` to `uint8` for the histogram count calculation so that the pixel values are discrete and directly correspond to a bin value.

Examples

Calculate Entropy of Grayscale Image

Read image into the workspace.

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

Calculate the entropy.

```
J = entropy(I)
```

```
J = 6.9439
```

References

- [1] Gonzalez, R.C., R.E. Woods, S.L. Eddins, *Digital Image Processing Using MATLAB*, New Jersey, Prentice Hall, 2003, Chapter 11.

See Also

imhist | entropyfilt

Introduced before R2006a

entropyfilt

Local entropy of grayscale image

Syntax

```
J = entropyfilt(I)
J = entropyfilt(I,NHOOD)
```

Description

`J = entropyfilt(I)` returns the array `J`, where each output pixel contains the entropy value of the 9-by-9 neighborhood around the corresponding pixel in the input image `I`. `I` can have any dimension. If `I` has more than two dimensions, `entropyfilt` treats it as a multidimensional grayscale image and not as a truecolor (RGB) image. The output image `J` is the same size as the input image `I`.

For pixels on the borders of `I`, `entropyfilt` uses symmetric padding. In symmetric padding, the values of padding pixels are a mirror reflection of the border pixels in `I`.

`J = entropyfilt(I,NHOOD)` performs entropy filtering of the input image `I` where you specify the neighborhood in `NHOOD`. `NHOOD` is a multidimensional array of zeros and ones where the nonzero elements specify the neighbors. `NHOOD`'s size must be odd in each dimension.

By default, `entropyfilt` uses the neighborhood `true(9)`. `entropyfilt` determines the center element of the neighborhood by `floor((size(NHOOD) + 1)/2)`. To specify neighborhoods of various shapes, such as a disk, use the `strel` function to create a structuring element object and then use the `getnhood` function to extract the neighborhood from the structuring element object.

Class Support

`I` can be `logical`, `uint8`, `uint16`, or `double`, and must be real and nonsparse. `NHOOD` can be `logical` or `numeric` and must contain zeros or ones. The output array `J` is of class `double`.

`entropyfilt` converts any class other than logical to `uint8` for the histogram count calculation so that the pixel values are discrete and directly correspond to a bin value.

Examples

Perform Entropy Filtering

This example shows how to perform entropy filtering using `entropyfilt`. Brighter pixels in the filtered image correspond to neighborhoods in the original image with higher entropy.

Read an image into the workspace.

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

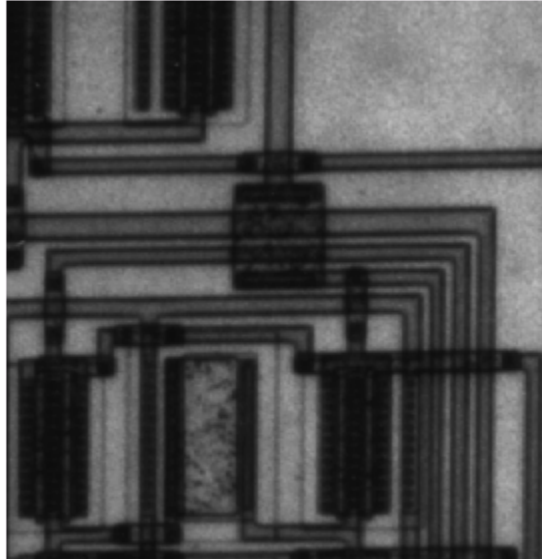
Perform entropy filtering using `entropyfilt`.

```
J = entropyfilt(I);
```

Show the original image and the processed image.

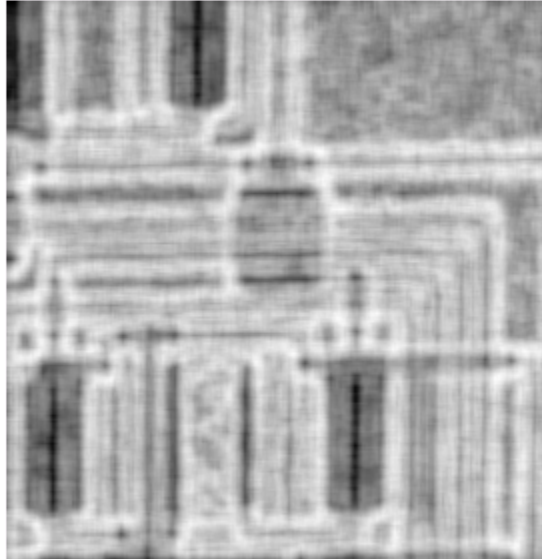
```
imshow(I)  
title('Original Image')
```

Original Image



```
figure  
imshow(J,[])  
title('Result of Entropy Filtering')
```


Result of Entropy Filtering



References

- [1] Gonzalez, R.C., R.E. Woods, S.L. Eddins, *Digital Image Processing Using MATLAB*, New Jersey, Prentice Hall, 2003, Chapter 11.

See Also

`entropy` | `imhist` | `rangefilt` | `stdfilt`

Introduced before R2006a

fan2para

Convert fan-beam projections to parallel-beam

Syntax

```
P = fan2para(F,D)
```

```
P = fan2para(..., param1, val1, param2, val2,...)
```

```
[P ,parallel_locations, parallel_rotation_angles] = fan2para(...)
```

Description

`P = fan2para(F,D)` converts the fan-beam data `F` to the parallel-beam data `P`. `D` is the distance in pixels from the fan-beam vertex to the center of rotation that was used to obtain the projections.

`P = fan2para(..., param1, val1, param2, val2,...)` specifies parameters that control various aspects of the `fan2para` conversion, listed in the following table. Parameter names can be abbreviated, and case does not matter.

Parameter	Description
'FanCoverage'	Range through which the beams are rotated, specified as one of the following: 'cycle' — Rotate through the full range [0,360). This is the default. 'minimal' — Rotate the minimum range necessary to represent the object.
'FanRotationIncrement'	Positive real scalar specifying the increment of the rotation angle of the fan-beam projections, measured in degrees. Default value is 1.

Parameter	Description
'FanSensorGeometry'	<p>Positioning of sensors, specified as one of the following:</p> <p>'arc' — Sensors are spaced equally along a circular arc at distance D from the center of rotation. Default value is 'arc'</p> <p>'line' — Sensors are spaced equally along a line, the closest point of which is distance D from the center of rotation.</p> <p>See <code>fanbeam</code> for details.</p>
'FanSensorSpacing'	<p>Positive real scalar specifying the spacing of the fan-beam sensors. Interpretation of the value depends on the setting of 'FanSensorGeometry'.</p> <p>If 'FanSensorGeometry' is set to 'arc' (the default), the value defines the angular spacing in degrees. Default value is 1.</p> <p>If 'FanSensorGeometry' is 'line', the value specifies the linear spacing. Default value is 1. See <code>fanbeam</code> for details.</p> <hr/> <p>Note: This linear spacing is measured on the x' axis. The x' axis for each column, <code>COL</code>, of <code>F</code> is oriented at <code>fan_rotation_angles(COL)</code> degrees counterclockwise from the x-axis. The origin of both axes is the center pixel of the image.</p>

Parameter	Description
'Interpolation'	<p>Type of interpolation used between the parallel-beam and fan-beam data, specified as one of the following:</p> <p>'nearest' — Nearest-neighbor</p> <p>{'linear'} — Linear</p> <p>'spline' — Piecewise cubic spline</p> <p>'pchip' — Piecewise cubic Hermite (PCHIP)</p> <p>'v5cubic' — The cubic interpolation from MATLAB 5</p>
'ParallelCoverage'	<p>Range of rotation, specified as one of the following:</p> <p>'cycle' — Parallel data covers 360 degrees</p> <p>{'halfcycle'} — Parallel data covers 180 degrees</p>
'ParallelRotationIncrement'	<p>Positive real scalar specifying the parallel-beam rotation angle increment, measured in degrees. Parallel beam angles are calculated to cover [0,180) degrees with increment PAR_ROT_INC, where PAR_ROT_INC is the value of 'ParallelRotationIncrement'.</p> <p>180/PAR_ROT_INC must be an integer.</p> <p>If 'ParallelRotationIncrement' is not specified, the increment is assumed to be the same as the increment of the fan-beam rotation angles.</p>

Parameter	Description
'ParallelSensorSpacing'	<p>Positive real scalar specifying the spacing of the parallel-beam sensors in pixels. The range of sensor locations is implied by the range of fan angles and is given by</p> $[D \cdot \sin(\min(\text{FAN_ANGLES})), \dots, D \cdot \sin(\max(\text{FAN_ANGLES}))]$ <p>If 'ParallelSensorSpacing' is not specified, the spacing is assumed to be uniform and is set to the minimum spacing implied by the fan angles and sampled over the range implied by the fan angles.</p>

`[P ,parallel_locations, parallel_rotation_angles] = fan2para(...)`
returns the parallel-beam sensor locations in `parallel_locations` and rotation angles in `parallel_rotation_angles`.

Class Support

The input arguments, `F` and `D`, can be `double` or `single`, and they must be nonsparse. All other numeric inputs are `double`. The output `P` is `double`.

Examples

Recover Parallel-beam Data from Fan-beam Data

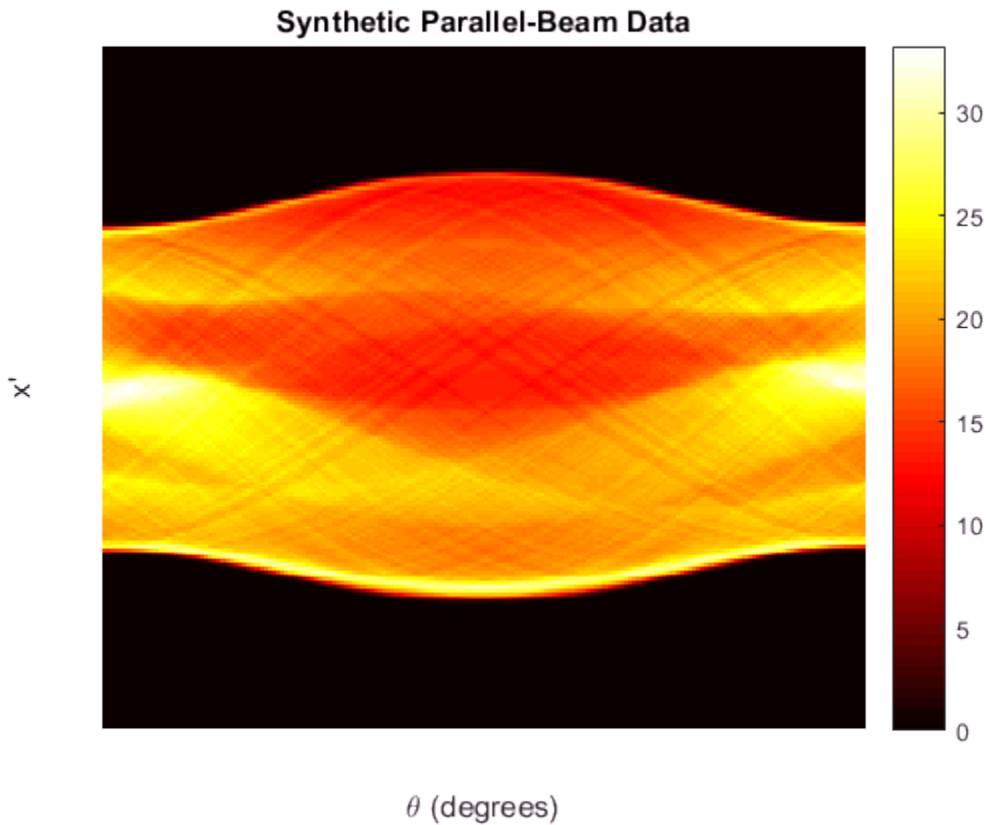
Create synthetic parallel-beam data.

```
ph = phantom(128);
```

Calculate the parallel beam transform and display it.

```
theta = 0:179;
[Psynthetic, xp] = radon(ph, theta);
imshow(Psynthetic, [], ...
       'XData', theta, 'YData', xp, 'InitialMagnification', 'fit')
axis normal
title('Synthetic Parallel-Beam Data')
xlabel('\theta (degrees)')
```

```
ylabel('x''')
colormap(gca,hot), colorbar
```



Convert the parallel-beam data to fan-beam.

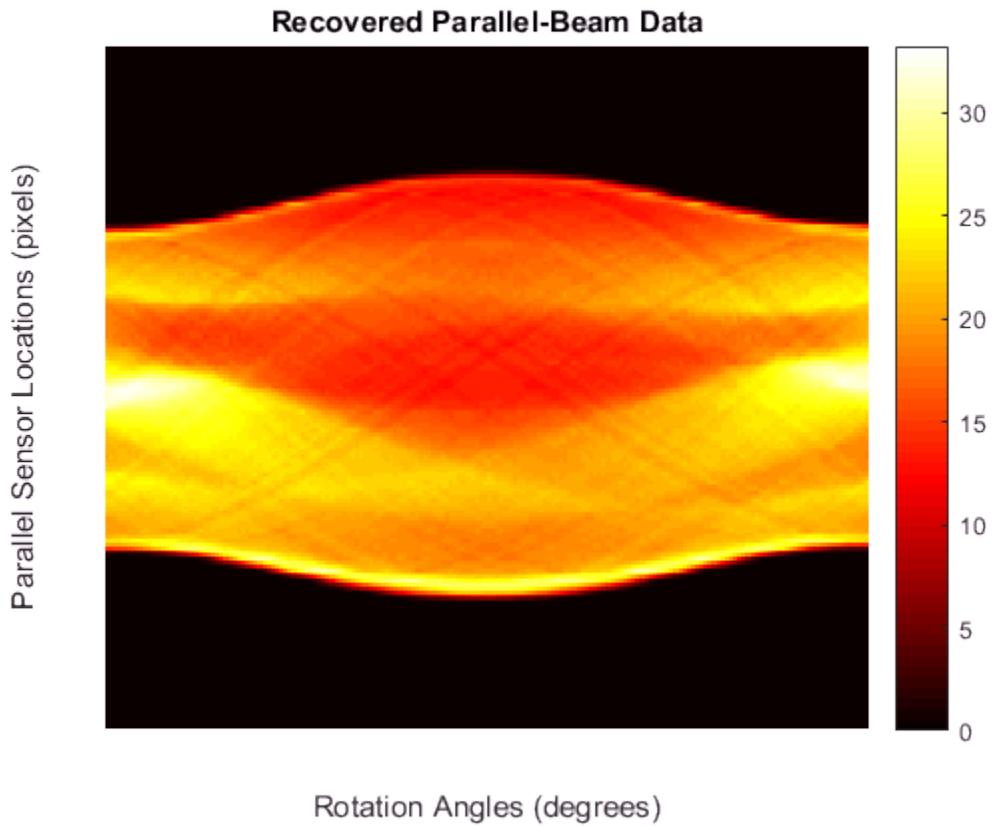
```
Fsynthetic = para2fan(Psynthetic,100,'FanSensorSpacing',1);
```

Recover original parallel-beam data.

```
[Precovered,Ploc,Pangles] = fan2para(Fsynthetic,100,...
                                     'FanSensorSpacing',1,...
                                     'ParallelSensorSpacing',1);
```

```
figure
imshow(Precovered,[],...)
```

```
    'XData',Pangles, 'YData',Ploc, 'InitialMagnification', 'fit')  
axis normal  
title('Recovered Parallel-Beam Data')  
xlabel('Rotation Angles (degrees)')  
ylabel('Parallel Sensor Locations (pixels)')  
colormap(gca,hot), colorbar
```



See Also

fanbeam | iradon | ifanbeam | para2fan | phantom | radon

Introduced before R2006a

fanbeam

Fan-beam transform

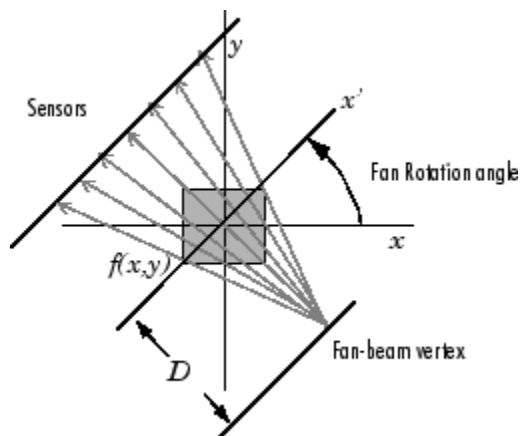
Syntax

```
F = fanbeam(I,D)
F = fanbeam(..., param1, val1, param1, val2,...)
[F, fan_sensor_positions, fan_rotation_angles] = fanbeam(...)
```

Description

$F = \text{fanbeam}(I, D)$ computes the fan-beam projection data (sinogram) F from the image I .

D is the distance in pixels from the fan-beam vertex to the center of rotation. The center of rotation is the center pixel of the image, defined as $\text{floor}((\text{size}(I)+1)/2)$. D must be large enough to ensure that the fan-beam vertex is outside of the image at all rotation angles. See “Tips” on page 1-408 for guidelines on specifying D . The following figure illustrates D in relation to the fan-beam geometry. See the `FanSensorGeometry` parameter for more information.



Each column of F contains the fan-beam sensor samples at one rotation angle. The number of columns in F is determined by the fan rotation increment. By default, the fan rotation increment is 1 degree so F has 360 columns.

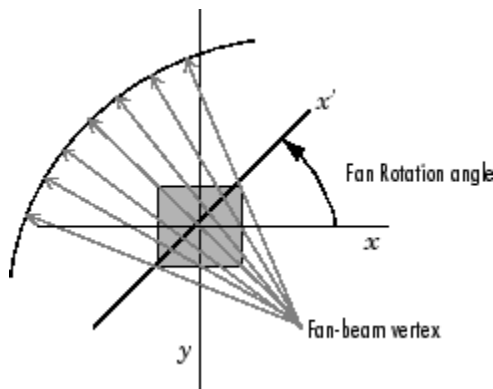
The number of rows in F is determined by the number of sensors. `fanbeam` determines the number of sensors by calculating how many beams are required to cover the entire image for any rotation angle.

For information about how to specify the rotation increment and sensor spacing, see the documentation for the `FanRotationIncrement` and `FanSensorSpacing` parameters, below.

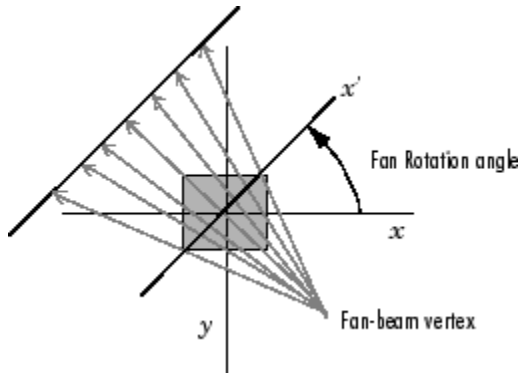
$F = \text{fanbeam}(\dots, \text{param1}, \text{val1}, \text{param1}, \text{val2}, \dots)$ specifies parameters, listed below, that control various aspects of the fan-beam projections. Parameter names can be abbreviated, and case does not matter.

'`FanRotationIncrement`' -- Positive real scalar specifying the increment of the rotation angle of the fan-beam projections. Measured in degrees. Default value is 1.

'`FanSensorGeometry`' -- Positioning of sensors, specified as the value '`arc`' or '`line`'. In the '`arc`' geometry, sensors are spaced equally along a circular arc, as shown below. This is the default value.



In '`line`' geometry, sensors are spaced equally along a line, as shown below.



'FanSensorSpacing' -- Positive real scalar specifying the spacing of the fan-beam sensors. Interpretation of the value depends on the setting of 'FanSensorGeometry'. If 'FanSensorGeometry' is set to 'arc' (the default), the value defines the angular spacing in degrees. Default value is 1. If 'FanSensorGeometry' is 'line', the value specifies the linear spacing. Default value is 1.

Note This linear spacing is measured on the x' axis. The x' axis for each column, `col`, of `F` is oriented at `fan_rotation_angles(col)` degrees counterclockwise from the x -axis. The origin of both axes is the center pixel of the image.

`[F, fan_sensor_positions, fan_rotation_angles] = fanbeam(...)` returns the location of fan-beam sensors in `fan_sensor_positions` and the rotation angles where the fan-beam projections are calculated in `fan_rotation_angles`.

If 'FanSensorGeometry' is 'arc' (the default), `fan_sensor_positions` contains the fan-beam spread angles. If 'FanSensorGeometry' is 'line', `fan_sensor_positions` contains the fan-beam sensor positions along the x' axis. See 'FanSensorSpacing' for more information.

Class Support

`I` can be logical or numeric. All other numeric inputs and outputs can be double. None of the inputs can be sparse.

Examples

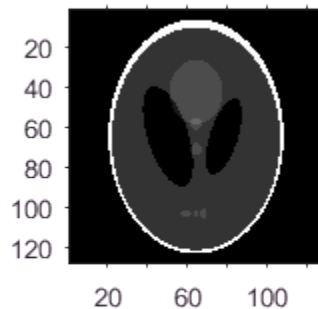
Compute Fan-beam Projections for Rotation Angles Over Entire Image

Set the IPT preference to make the axes visible.

```
iptsetpref('ImshowAxesVisible','on')
```

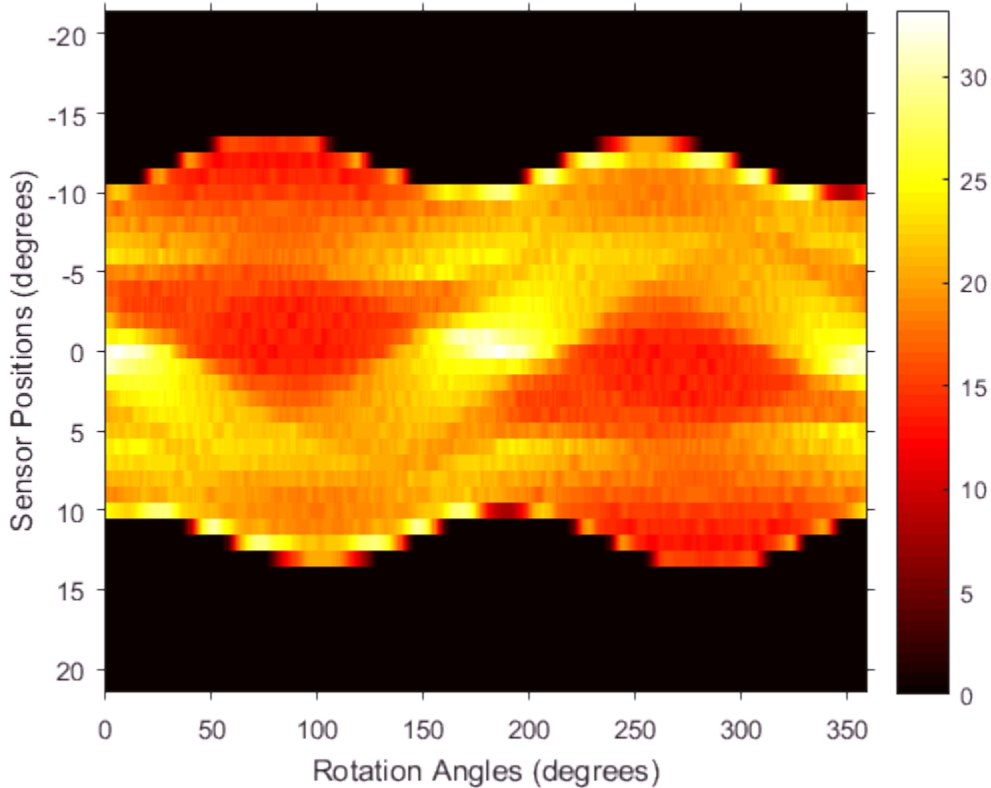
Create a sample image and display it.

```
ph = phantom(128);
imshow(ph)
```



Calculate the fanbeam projections and display them.

```
[F,Fpos,Fangles] = fanbeam(ph,250);
figure
imshow(F,[], 'XData',Fangles,'YData',Fpos,...
        'InitialMagnification','fit')
axis normal
xlabel('Rotation Angles (degrees)')
ylabel('Sensor Positions (degrees)')
colormap(gca,hot), colorbar
```



Compute Radon and Fan-beam Projections and Compare Results

Compute fan-beam projections for 'arc' geometry.

```
I = ones(100);
D = 200;
dtheta = 45;
[Farc,FposArcDeg,Fangles] = fanbeam(I,D,...
    'FanSensorGeometry','arc',...
    'FanRotationIncrement',dtheta);
```

Convert angular positions to linear distance along x-prime axis.

```
FposArc = D*tan(FposArcDeg*pi/180);
```

Compute fan-beam projections for 'line' geometry.

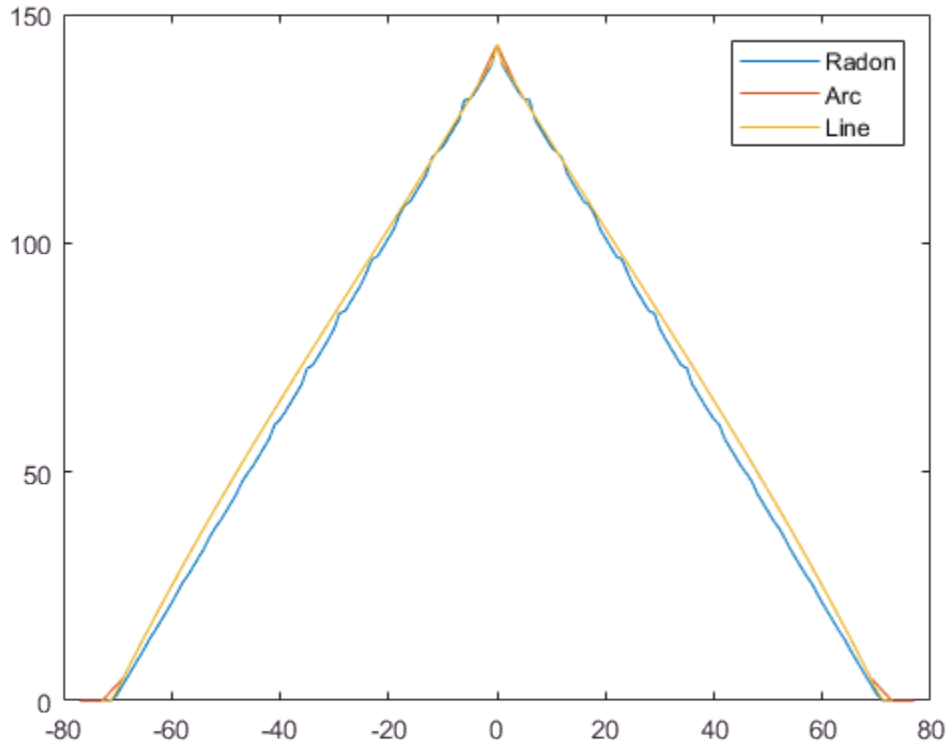
```
[Fline,FposLine] = fanbeam(I,D,...  
    'FanSensorGeometry','line',...  
    'FanRotationIncrement',dtheta);
```

Compute the corresponding Radon transform.

```
[R,Rpos]=radon(I,Fangles);
```

Display the three projections at one particular rotation angle. Note the three are very similar. Differences are due to the geometry of the sampling, and the numerical approximations used in the calculations.

```
figure  
idx = find(Fangles==45);  
plot(Rpos,R(:,idx),...  
    FposArc,Farc(:,idx),...  
    FposLine,Fline(:,idx))  
legend('Radon','Arc','Line')
```



Tips

As a guideline, try making D a few pixels larger than half the image diagonal dimension, calculated as follows

```
sqrt(size(I,1)^2 + size(I,2)^2)
```

The values returned in F are a numerical approximation of the fan-beam projections. The algorithm depends on the Radon transform, interpolated to the fan-beam geometry. The results vary depending on the parameters used. You can expect more accurate results

when the image is larger, D is larger, and for points closer to the middle of the image, away from the edges.

References

- [1] Kak, A.C., & Slaney, M., *Principles of Computerized Tomographic Imaging*, IEEE Press, NY, 1988, pp. 92-93.

See Also

fan2para | iradon | ifanbeam | para2fan | phantom | radon

Introduced before R2006a

fibermetric

Enhance elongated or tubular structures in image

Syntax

```
B = fibermetric(A)
B = fibermetric(A,thickness)
B = fibermetric( ____,Name,Value)
```

Description

`B = fibermetric(A)` enhances elongated or tubular structures in intensity image `A` using Hessian-based multiscale filtering. The image returned, `B`, contains the maximum response of the filter at a thickness that approximately matches the size of the tubular structure in the image.

`B = fibermetric(A,thickness)` enhances elongated or tubular structures in intensity image `A`, where `thickness` specifies the thickness of the tubular structures.

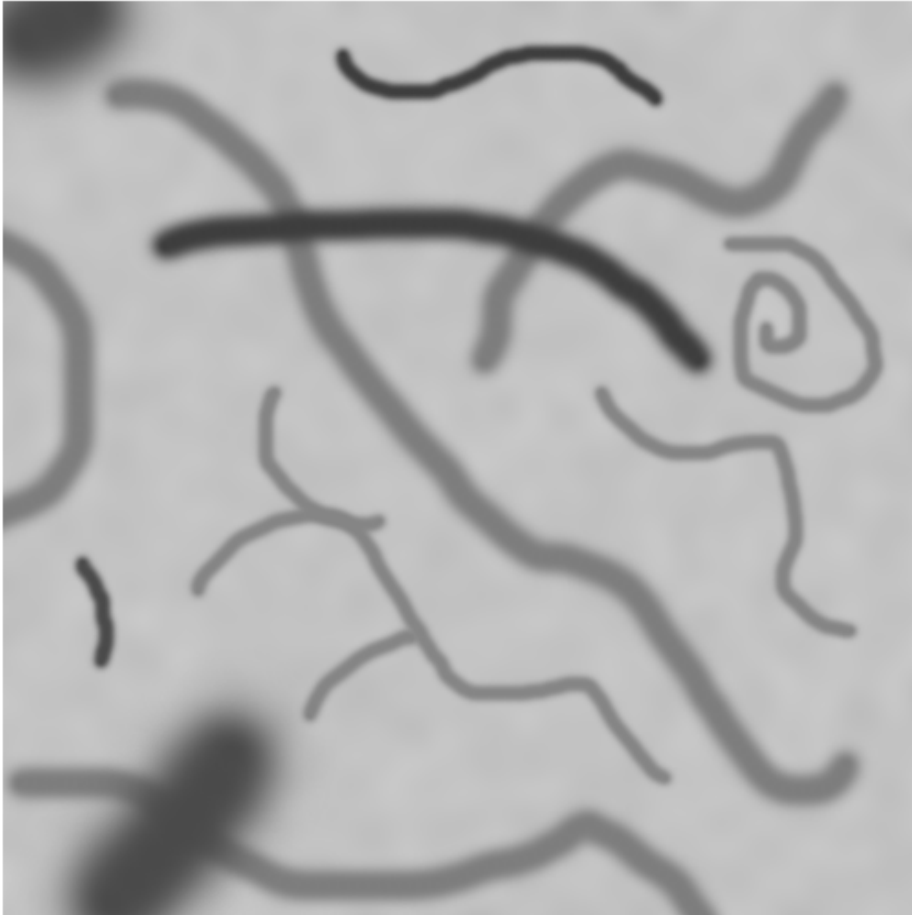
`B = fibermetric(____,Name,Value)` enhances the tubular structures in the image using name-value pairs to control different aspects of the filtering algorithm.

Examples

Find Threads Approximately Seven Pixels Thick

Read an image into the workspace that contains tubular structures of varying thicknesses, and display it.

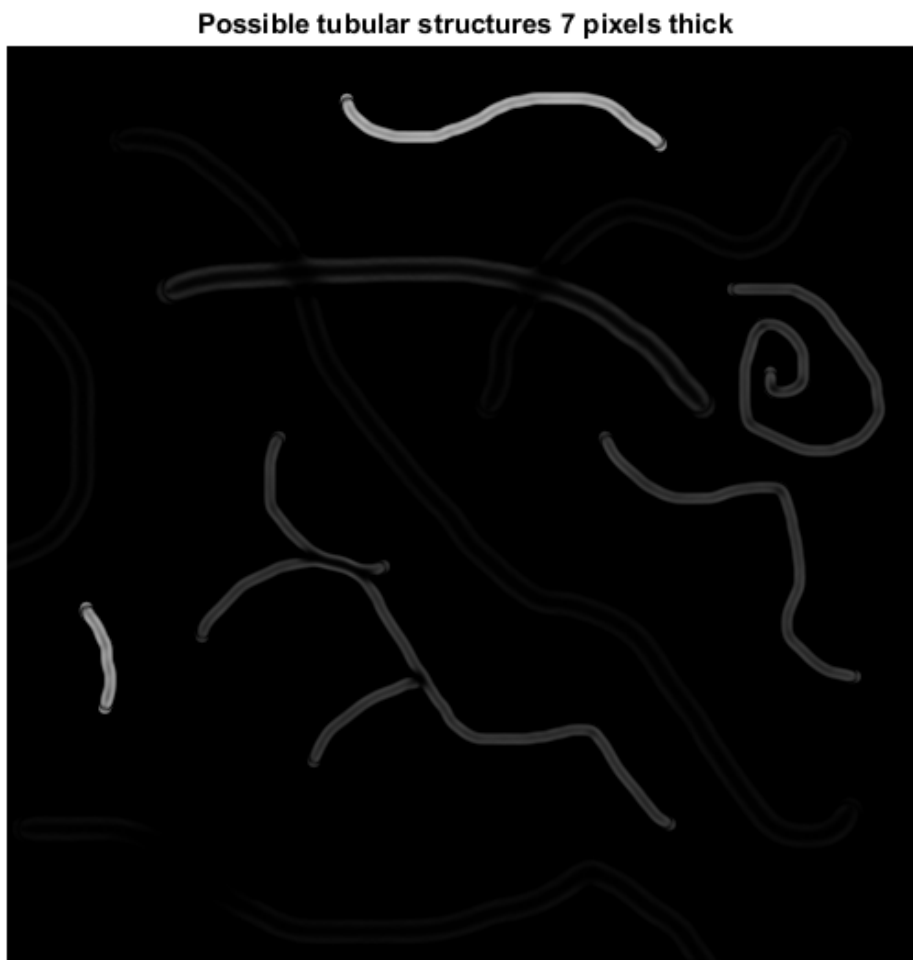
```
A = imread('threads.png');
imshow(A)
```

Create an enhanced version of the image that highlights threads that are seven pixels thick, and display it.

```
B = fibermetric(A, 7, 'ObjectPolarity', 'dark', 'StructureSensitivity', 7);  
figure;
```

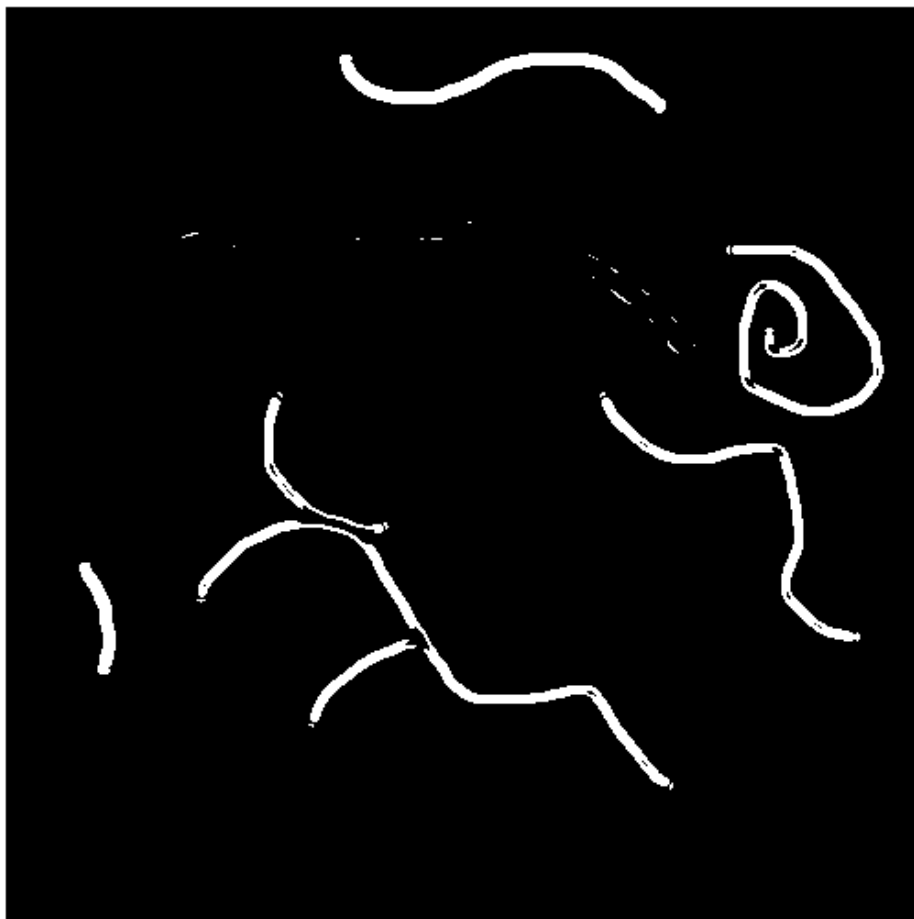
```
imshow(B);  
title('Possible tubular structures 7 pixels thick')
```



Threshold the enhanced image to create a binary mask image containing the threads with the specified thickness.

```
C = B > 0.15;  
figure;  
imshow(C);  
title('Thresholded result')
```

Thresholded result



Input Arguments

A — 2-D grayscale image
nonsparse numeric array

2-D grayscale image, specified as a nonsparse numeric array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

thickness — Thickness of tubular structures

vector | scalar

Thickness of tubular structures, specified as a scalar or vector, measured in pixels. Specify a value on the order of the width of the tubular structures in the image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `B = fibermetric(A, 'StructureSensitivity', 15)`

'StructureSensitivity' — Threshold for differentiating the tubular structure from the background

half the maximum of the Hessian norm of the image (default) | numeric scalar

Threshold for differentiating the tubular structure from the background, specified as the comma-separated pair consisting of `'StructureSensitivity'` and a numeric scalar. The value depends on the grayscale range of the image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'ObjectPolarity' — Polarity of the tubular structures with the background

'bright' (default) | 'dark'

Polarity of the tubular structures with the background, specified as the comma-separated pair consisting of `'ObjectPolarity'` and one of the following character vectors.

Value	Description
'bright'	Structure is brighter than the background.
'dark'	Structure is darker than the background.

Data Types: char

Output Arguments

B — Output image

numeric array

Output image, returned as a numeric array the same size as the input image of class `single`.

Tips

- The `fibermetric` function does not perform segmentation. The function enhances an image to highlight structures and is typically used as a preprocessing step for segmentation.

References

- [1] Frangi, Alejandro F., et al. *Multiscale vessel enhancement filtering*. Medical Image Computing and Computer-Assisted Intervention—MICCAI'98. Springer Berlin Heidelberg, 1998. 130-137.

See Also

See Also

`edge` | `imgradient`

Introduced in R2007a

findbounds

Find output bounds for spatial transformation

Syntax

```
outbounds = findbounds(tform,inbounds)
```

Description

`outbounds = findbounds(tform,inbounds)` estimates the output bounds corresponding to a given spatial transformation and a set of input bounds. `tform` is a spatial transformation structure. `inbounds` is a 2-by-`num_dims` matrix that specifies the lower and upper bounds of the output image. `outbounds` is an estimate of the smallest rectangular region completely containing the transformed rectangle represented by the input bounds, and has the same form as `inbounds`. Since `outbounds` is only an estimate, it might not completely contain the transformed input rectangle.

Examples

Calculate Boundaries of Transformed Output Image

Read an image into the workspace and display it.

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



Create a spatial transformation structure that stretches an image.

```
T = maketform('affine',[.5 0 0; .5 2 0; 0 0 1]);
```

Calculate the boundaries of the output image, given the size of the input image and the spatial transformation. The dimensions of the input image are 256-by-256. The boundaries of the output image reflect the transformation: 256-by-512.

```
outb = findbounds(T,[0 0;256 256])
```

```
outb =
```

```
    0    0  
  256  512
```

Apply the transformation and display the image.

```
transformedI = imtransform(I,T);  
figure
```


`imshow(transformedI)`



Input Arguments

tform — Spatial transformation

structure

Spatial transformation, specified as a structure (`tform`).

Data Types: `struct`

inbounds — Bounds for each dimension of the input image

2-by-`num_dims` matrix

Bounds for each dimension of the input image, specified as a 2-by-`num_dims` matrix. The first row of `inbounds` specifies the lower bounds for each dimension, and the second row specifies the upper bounds. `num_dims` has to be consistent with the `ndims_in` field of `tform`.

Example: `outb = findbounds(T,[0 0;256 256])` where input image is 256-by-256.

Data Types: `double`

Output Arguments

outbounds — Bounds for each dimension of the output image

2-by-`num_dims` matrix of class `double`

Bounds for each dimension of the output image (output space bounding box), returned as a 2-by-`num_dims` matrix of class `double`.

Algorithms

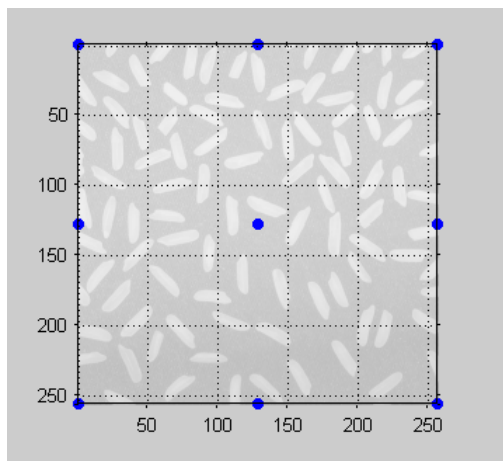
- 1 `findbounds` first creates a grid of input-space points. These points are at the center, corners, and middle of each edge in the image.

```
I = imread('rice.png');  
h = imshow(I);  
set(h, 'AlphaData',0.3);  
axis on, grid on
```

```

in_points = [ ...
    0.5000    0.5000
    0.5000   256.5000
    256.5000    0.5000
    256.5000   256.5000
    0.5000   128.5000
    128.5000    0.5000
    128.5000   128.5000
    128.5000   256.5000
    256.5000   128.5000];
hold on
plot(in_points(:,1),in_points(:,2),'.','MarkerSize',18)
hold off

```



Grid of Input-Space Points

- Next, `findbounds` transforms the grid of input-space points to output space. If `tform` contains a forward transformation (a nonempty `forward_fcn` field), then `findbounds` transforms the input-space points using `tformfwd`. For example:

```

tform = maketform('affine', ...
    [1.1067 -0.2341 0; 0.5872 1.1769 0; 1000 -300 1]);
out_points = tformfwd(tform, in_points)

out_points =

    1.0e+03 *

```

1.0008	-0.2995
1.1512	0.0018
1.2842	-0.3595
1.4345	-0.0582
1.0760	-0.1489
1.1425	-0.3295
1.2177	-0.1789
1.2928	-0.0282

If `tform` does not contain a forward transformation, then `findbounds` estimates the output bounds using the Nelder-Mead optimization function `fminsearch`.

- 3 Finally, `findbounds` computes the bounding box of the transformed grid of points.

See Also

See Also

`tformarray` | `tformfwd` | `tforminv`

Introduced before R2006a

fitgeotrans

Fit geometric transformation to control point pairs

Syntax

```
tform = fitgeotrans(movingPoints, fixedPoints, transformationType)
tform = fitgeotrans(movingPoints, fixedPoints, 'polynomial', degree)
tform = fitgeotrans(movingPoints, fixedPoints, 'pwl')
tform = fitgeotrans(movingPoints, fixedPoints, 'lwm', n)
```

Description

`tform = fitgeotrans(movingPoints, fixedPoints, transformationType)` takes the pairs of control points, `movingPoints` and `fixedPoints`, and uses them to infer the geometric transformation specified by `transformationType`.

`tform = fitgeotrans(movingPoints, fixedPoints, 'polynomial', degree)` fits a `PolynomialTransformation2D` object to control point pairs `movingPoints` and `fixedPoints`. Specify the degree of the polynomial transformation `degree`, which can be 2, 3, or 4.

`tform = fitgeotrans(movingPoints, fixedPoints, 'pwl')` fits a `PiecewiseLinearTransformation2D` object to control point pairs `movingPoints` and `fixedPoints`. This transformation maps control points by breaking up the plane into local piecewise-linear regions. A different affine transformation maps control points in each local region.

`tform = fitgeotrans(movingPoints, fixedPoints, 'lwm', n)` fits a `LocalWeightedMeanTransformation2D` object to control point pairs `movingPoints` and `fixedPoints`. The local weighted mean transformation creates a mapping, by inferring a polynomial at each control point using neighboring control points. The mapping at any location depends on a weighted average of these polynomials. The `n` closest points are used to infer a second degree polynomial transformation for each control point pair.

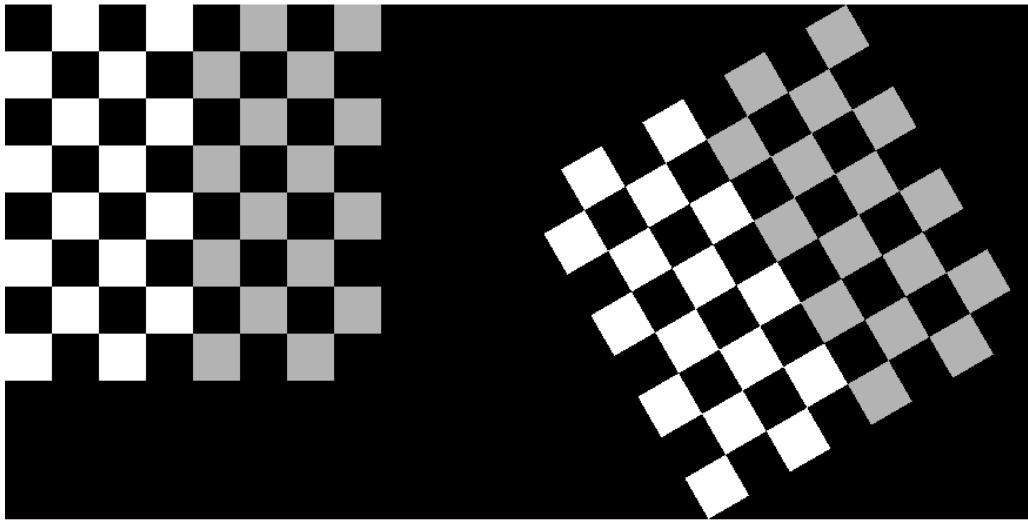
Examples

Create Geometric Transformation for Image Alignment

This example shows how to create a geometric transformation that can be used to align two images.

Create a checkerboard image and rotate it to create a misaligned image.

```
I = checkerboard(40);  
J = imrotate(I,30);  
imshowpair(I,J,'montage')
```



Define some matching control points on the fixed image (the checkerboard) and moving image (the rotated checkerboard). You can define points interactively using the Control Point Selection tool.

```
fixedPoints = [41 41; 281 161];  
movingPoints = [56 175; 324 160];
```

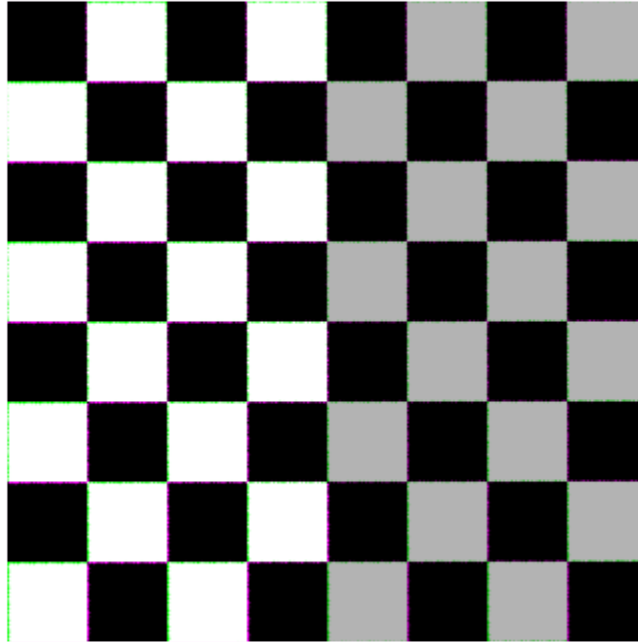
Create a geometric transformation that can be used to align the two images, returned as an `affine2d` geometric transformation object.

```
tform = fitgeotrans(movingPoints, fixedPoints, 'NonreflectiveSimilarity')
```

```
tform =  
  affine2d with properties:  
      T: [3×3 double]  
  Dimensionality: 2
```

Use the `tform` estimate to resample the rotated image to register it with the fixed image. The regions of color (green and magenta) in the false color overlay image indicate error in the registration. This error comes from a lack of precise correspondence in the control points.

```
Jregistered = imwarp(J, tform, 'OutputView', imref2d(size(I)));  
figure  
imshowpair(I, Jregistered)
```



Recover angle and scale of the transformation by checking how a unit vector parallel to the x-axis is rotated and stretched.

```
u = [0 1];  
v = [0 0];  
[x, y] = transformPointsForward(tform, u, v);  
dx = x(2) - x(1);  
dy = y(2) - y(1);  
angle = (180/pi) * atan2(dy, dx)  
  
angle = 29.7686  
  
scale = 1 / sqrt(dx^2 + dy^2)  
  
scale = 1.0003
```


Input Arguments

movingPoints — *x*- and *y*-coordinates of control points in the image you want to transform
m-by-2 double matrix

x- and *y*-coordinates of control points in the image you want to transform, specified as an *m*-by-2 double matrix.

Example: `movingPoints = [11 11; 41 71];`

Data Types: `double` | `single`

fixedPoints — *x*- and *y*-coordinates of control points in the fixed image
m-by-2 double matrix

x- and *y*- coordinates of control points in the fixed image, specified as an *m*-by-2 double matrix.

Example: `fixedPoints = [14 44; 70 81];`

Data Types: `double` | `single`

transformationType — Type of transformation
'nonreflectivesimilarity' | 'similarity' | 'affine' | 'projective'

Type of transformation, specified as one of the following: 'nonreflectivesimilarity', 'similarity', 'affine', or 'projective'. For more information, see “Transformation Types” on page 1-428.

Example: `tform = fitgeotrans(movingPoints, fixedPoints, 'nonreflectivesimilarity');`

Data Types: `char`

degree — Degree of the polynomial
2 | 3 | 4

Degree of the polynomial, specified as the integer 2, 3, or 4.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

n — Number of points to use in local weighted mean calculation

numeric value

Number of points to use in local weighted mean calculation, specified as a numeric value. *n* can be as small as 6, but making *n* small risks generating ill-conditioned polynomials

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

tform — Transformation

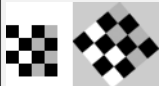
transformation object

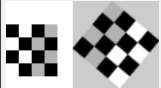





Transformation, returned as a transformation object. The type of object depends on the transformation type. For example, if you specify the transformation type 'affine', *tform* is an `affine2d` object. If you specify 'pwl', *tform* is an `image.geotrans.PiecewiseLinearTransformation2d` object.

Definitions

Transformation Types

The table lists all the transformation types supported by `fitgeotrans` 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	

Transformation Type	Description	Minimum Number of Control Point Pairs	Example
'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)	

References

- [1] Goshtasby, Ardeshir, "Piecewise linear mapping functions for image registration," *Pattern Recognition*, Vol. 19, 1986, pp. 459-466.
- [2] Goshtasby, Ardeshir, "Image registration by local approximation methods," *Image and Vision Computing*, Vol. 6, 1988, pp. 255-261.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, the `transformationType` argument must be a compile-time constant and only the following transformation types are supported: `'nonreflectivesimilarity'`, `'similarity'`, `'affine'`, and `'projective'`.

See Also

See Also

Functions

`cpselect` | `imwarp`

Classes

`affine2d` | `projective2d` | `images.geotrans.PiecewiseLinearTransformation2d`
| `images.geotrans.PolynomialTransformation2d` |
`images.geotrans.LocalWeightedMeanTransformation2d`

Topics

“Matrix Representation of Geometric Transformations”

Introduced in R2013b

fliptform

Flip input and output roles of spatial transformation structure

Syntax

```
tflip = fliptform(T)
```

Description

`tflip = fliptform(T)` creates a new spatial transformation structure (`tform`) by flipping the roles of the inputs and outputs in an existing `tform` struct.

Examples

Flip Spatial Transformation Structure

Create a spatial transformation structure (`tform`).

```
T = maketform('affine', [.5 0 0; .5 2 0; 0 0 1])
```

```
T =
```

```
struct with fields:
```

```
    ndims_in: 2
    ndims_out: 2
    forward_fcn: @fwd_affine
    inverse_fcn: @inv_affine
    tdata: [1x1 struct]
```

Create a new spatial transformation structure (`tform`) by flipping the roles of the inputs and outputs.

```
T2 = fliptform(T)
```

```
T2 =
```

```
struct with fields:

    ndims_in: 2
    ndims_out: 2
    forward_fcn: @inv_affine
    inverse_fcn: @fwd_affine
    tdata: [1×1 struct]
```

After flipping the spatial transformation structures, the following statements are equivalent.

```
x = tformfwd([-3 7],T)
x = tforminv([-3 7],T2)
```

```
x =

     2     14
```

```
x =

     2     14
```

Input Arguments

T — Spatial transformation

structure

Spatial transformation, specified as a structure (`tform`).

Data Types: `struct`

Output Arguments

tf1ip — Flipped spatial transformation

structure

Flipped spatial transformation, returned as a structure (`tform`).

See Also

See Also

tformfwd | tforminv

Introduced before R2006a

freqz2

2-D frequency response

Syntax

```
[H, f1, f2] = freqz2(h, n1, n2)
[H, f1, f2] = freqz2(h, [n2 n1])
[H, f1, f2] = freqz2(h)
[H, f1, f2] = freqz2(h, f1, f2)
[...] = freqz2(h, ..., [dx dy])
[...] = freqz2(h, ..., dx)
freqz2(...)
```

Description

`[H, f1, f2] = freqz2(h, n1, n2)` returns `H`, the `n2`-by-`n1` frequency response of `h`, and the frequency vectors `f1` (of length `n1`) and `f2` (of length `n2`). `h` is a two-dimensional FIR filter, in the form of a computational molecule. `f1` and `f2` are returned as normalized frequencies in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or π radians.

`[H, f1, f2] = freqz2(h, [n2 n1])` returns the same result returned by `[H, f1, f2] = freqz2(h, n1, n2)`.

`[H, f1, f2] = freqz2(h)` uses `[n2 n1] = [64 64]`.

`[H, f1, f2] = freqz2(h, f1, f2)` returns the frequency response for the FIR filter `h` at frequency values in `f1` and `f2`. These frequency values must be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or π radians.

`[...] = freqz2(h, ..., [dx dy])` uses `[dx dy]` to override the intersample spacing in `h`. `dx` determines the spacing for the x dimension and `dy` determines the spacing for the y dimension. The default spacing is 0.5, which corresponds to a sampling frequency of 2.0.

`[...] = freqz2(h, ..., dx)` uses `dx` to determine the intersample spacing in both dimensions.

`freqz2(...)` produces a mesh plot of the two-dimensional magnitude frequency response when no output arguments are specified.

Class Support

The input matrix `h` can be of class `double` or of any integer class. All other inputs to `freqz2` must be of class `double`. All outputs are of class `double`.

Examples

View Frequency Response of Filter

This example shows how to create a two-dimensional filter using `fwind1` and how to view the filter's frequency response using `freqz2`.

Create an ideal frequency response.

```
Hd = zeros(16,16);  
Hd(5:12,5:12) = 1;  
Hd(7:10,7:10) = 0;
```

Create a 1-D window. This example uses a Bartlett window of length 16.

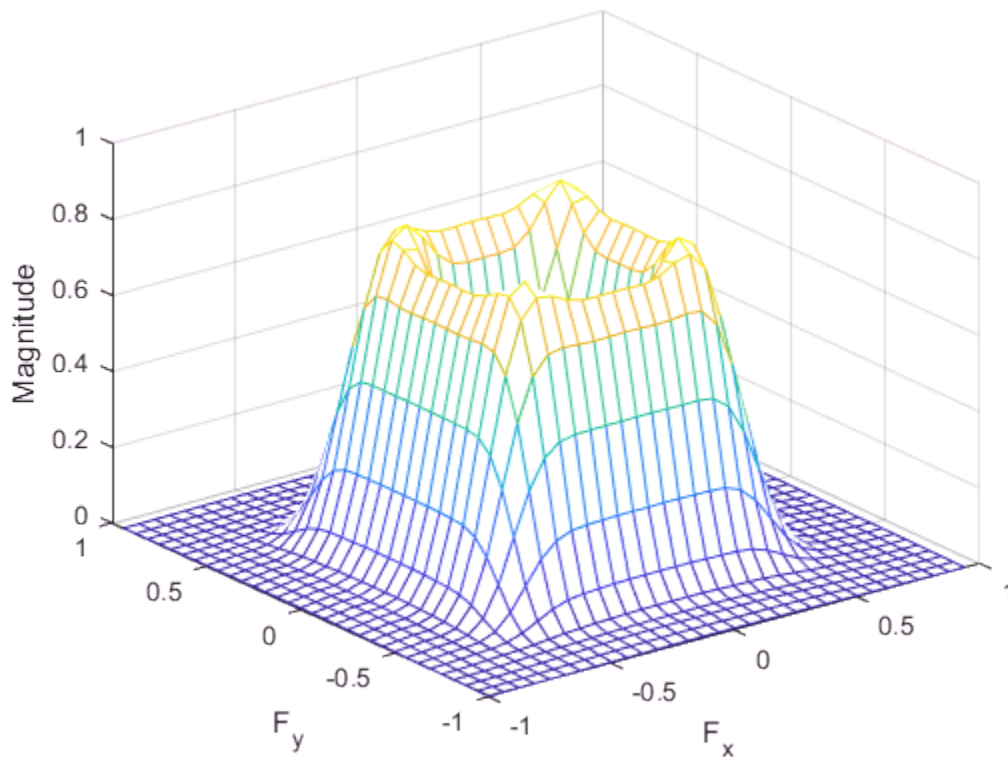
```
w = [0:2:16 16:-2:0]/16;
```

Create the 16-by-16 filter using `fwind1` and the 1-D window. This filter gives the closest match to the ideal frequency response.

```
h = fwind1(Hd,w);
```

Display the actual frequency response of the filter.

```
colormap(parula(64))  
freqz2(h,[32 32]);  
axis ([-1 1 -1 1 0 1])
```



See Also

freqz

Introduced before R2006a

fsamp2

2-D FIR filter using frequency sampling

Syntax

```
h = fsamp2(Hd)
h = fsamp2(f1, f2, Hd, [m n])
```

Description

`h = fsamp2(Hd)` designs a two-dimensional FIR filter with frequency response `Hd`, and returns the filter coefficients in matrix `h`. (`fsamp2` returns `h` as a computational molecule, which is the appropriate form to use with `filter2`.) The filter `h` has a frequency response that passes through points in `Hd`. If `Hd` is `m`-by-`n`, then `h` is also `m`-by-`n`.

`fsamp2` designs two-dimensional FIR filters based on a desired two-dimensional frequency response sampled at points on the Cartesian plane. `Hd` is a matrix containing the desired frequency response sampled at equally spaced points between -1.0 and 1.0 along the x and y frequency axes, where 1.0 corresponds to half the sampling frequency, or π radians.

$$H_d(f_1, f_2) = H_d(\omega_1, \omega_2) \Big|_{\omega_1 = \pi f_1, \omega_2 = \pi f_2}$$

For accurate results, use frequency points returned by `freqspace` to create `Hd`.

`h = fsamp2(f1, f2, Hd, [m n])` produces an `m`-by-`n` FIR filter by matching the filter response at the points in the vectors `f1` and `f2`. The frequency vectors `f1` and `f2` are in normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians. The resulting filter fits the desired response as closely as possible in the least squares sense. For best results, there must be at least `m*n` desired frequency points. `fsamp2` issues a warning if you specify fewer than `m*n` points.

Class Support

The input matrix `Hd` can be of class `double` or of any integer class. All other inputs to `fsamp2` must be of class `double`. All outputs are of class `double`.

Examples

Create 2-D FIR Filter using Frequency Sampling

This example shows how to create a two-dimensional bandpass filter using `fsamp2`.

Create the frequency range vectors `f1` and `f2` using `freqspace`. These vectors have length 21.

```
[f1,f2] = freqspace(21,'meshgrid');
```

Compute the distance of each position from the center frequency.

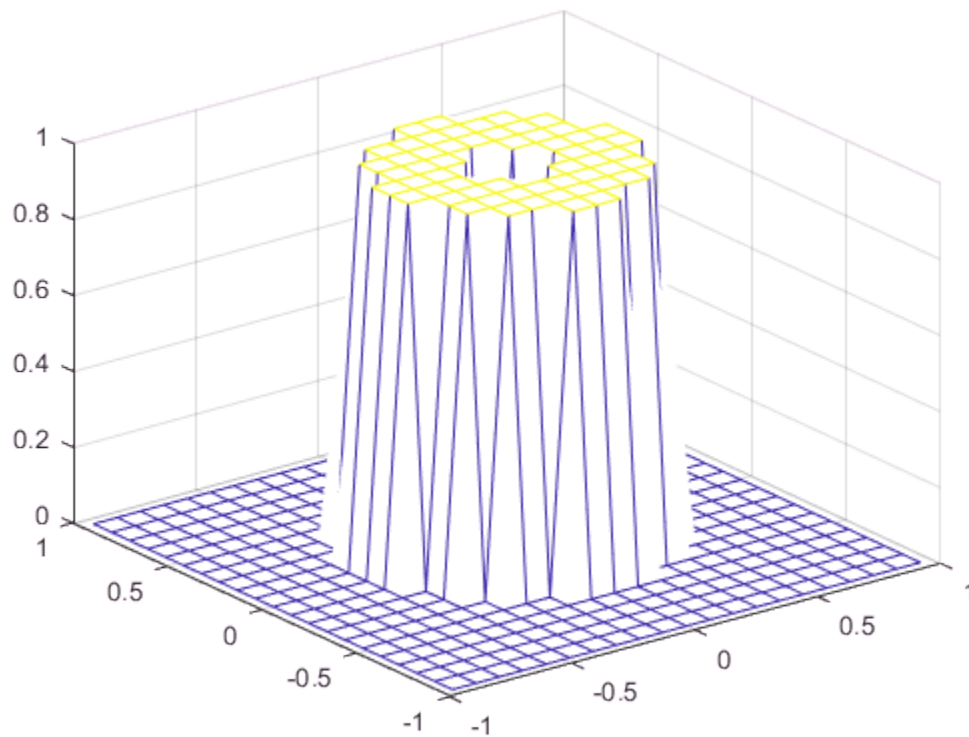
```
r = sqrt(f1.^2 + f2.^2);
```

Create a matrix `Hd` that contains the desired bandpass response. In this example, the desired passband is between 0.1 and 0.5 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians).

```
Hd = ones(21);  
Hd((r<0.1)|(r>0.5)) = 0;
```

Display the ideal bandpass response.

```
colormap(parula(64))  
mesh(f1,f2,Hd)
```

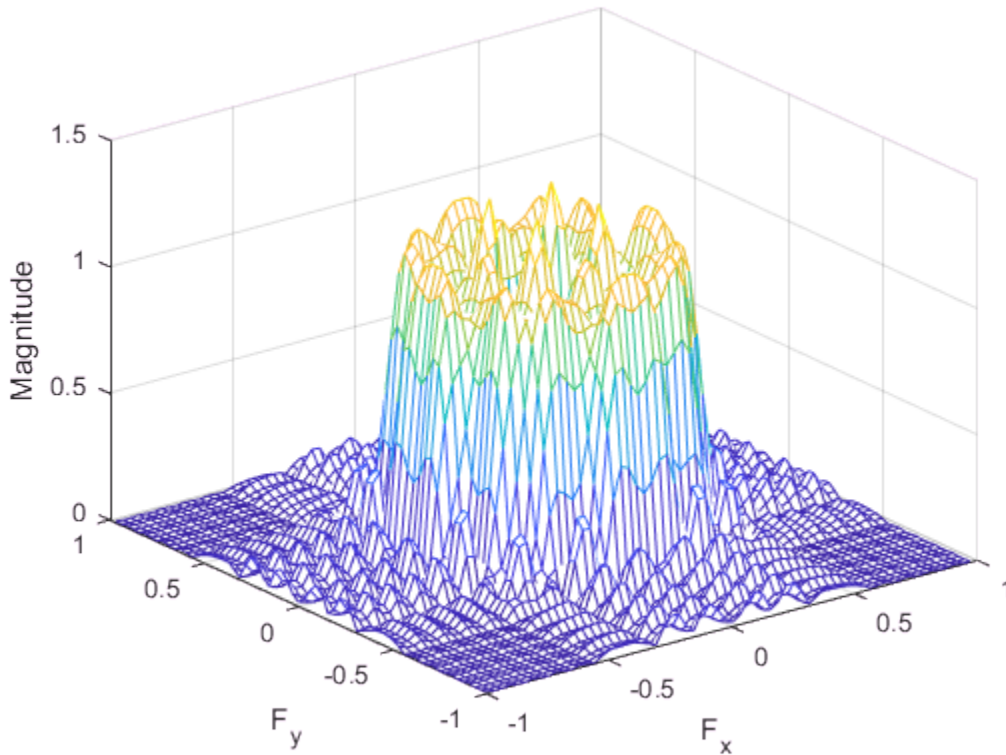


Using frequency sampling, design the filter that best produces this frequency response.

```
h = fsamp2(Hd);
```

Display the actual frequency response of this filter.

```
freqz2(h)
```



Algorithms

`fsamp2` computes the filter h by taking the inverse discrete Fourier transform of the desired frequency response. If the desired frequency response is real and symmetric (zero phase), the resulting filter is also zero phase.

References

- [1] Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, pp. 213-217.

See Also

`conv2` | `filter2` | `freqspace` | `ftrans2` | `fwind1` | `fwind2`

Introduced before R2006a

fspecial

Create predefined 2-D filter

Syntax

```
h = fspecial(type)
h = fspecial(type, parameters)
```

Description

`h = fspecial(type)` creates a two-dimensional filter `h` of the specified `type`. `fspecial` returns `h` as a correlation kernel, which is the appropriate form to use with `imfilter`. `type` can have any of the following values:

Value	Description
'average'	Averaging filter
'disk'	Circular averaging filter (pillbox)
'gaussian'	Gaussian lowpass filter
'laplacian'	Approximates the two-dimensional Laplacian operator
'log'	Laplacian of Gaussian filter
'motion'	Approximates the linear motion of a camera
'prewitt'	Prewitt horizontal edge-emphasizing filter
'sobel'	Sobel horizontal edge-emphasizing filter

`h = fspecial(type, parameters)` accepts the filter specified by `type` plus additional modifying `parameters` particular to the type of filter chosen. If you omit these arguments, `fspecial` uses default values for the `parameters`.

The following list shows the syntax for each filter type. Where applicable, additional parameters are also shown.

- `h = fspecial('average', hsize)` returns an averaging filter `h` of size `hsize`. The argument `hsize` can be a vector specifying the number of rows and columns in `h`,

or it can be a scalar, in which case `h` is a square matrix. The default value for `hsize` is `[3 3]`.

- `h = fspecial('disk', radius)` returns a circular averaging filter (pillbox) within the square matrix of size `2*radius+1`. The default `radius` is 5.
- `h = fspecial('gaussian', hsize, sigma)` returns a rotationally symmetric Gaussian lowpass filter of size `hsize` with standard deviation `sigma` (positive). `hsize` can be a vector specifying the number of rows and columns in `h`, or it can be a scalar, in which case `h` is a square matrix. The default value for `hsize` is `[3 3]`; the default value for `sigma` is 0.5. Not recommended. Use `imgaussfilt` or `imgaussfilt3` instead.
- `h = fspecial('laplacian', alpha)` returns a 3-by-3 filter approximating the shape of the two-dimensional Laplacian operator. The parameter `alpha` controls the shape of the Laplacian and must be in the range 0.0 to 1.0. The default value for `alpha` is 0.2.
- `h = fspecial('log', hsize, sigma)` returns a rotationally symmetric Laplacian of Gaussian filter of size `hsize` with standard deviation `sigma` (positive). `hsize` can be a vector specifying the number of rows and columns in `h`, or it can be a scalar, in which case `h` is a square matrix. The default value for `hsize` is `[5 5]` and 0.5 for `sigma`.
- `h = fspecial('motion', len, theta)` returns a filter to approximate, once convolved with an image, the linear motion of a camera by `len` pixels, with an angle of `theta` degrees in a counterclockwise direction. The filter becomes a vector for horizontal and vertical motions. The default `len` is 9 and the default `theta` is 0, which corresponds to a horizontal motion of nine pixels.

To compute the filter coefficients, `h`, for 'motion':

- 1 Construct an ideal line segment with the desired length and angle, centered at the center coefficient of `h`.
 - 2 For each coefficient location `(i, j)`, compute the nearest distance between that location and the ideal line segment.
 - 3 `h = max(1 - nearest_distance, 0);`
 - 4 Normalize `h`: `h = h / (sum(h(:)))`
- `h = fspecial('prewitt')` returns the 3-by-3 filter `h` (shown below) that emphasizes horizontal edges by approximating a vertical gradient. If you need to emphasize vertical edges, transpose the filter `h'`.

```
[ 1  1  1
```

```
0 0 0
-1 -1 -1 ]
```

To find vertical edges, or for x -derivatives, use h' .

- `h = fspecial('sobel')` returns a 3-by-3 filter h (shown below) that emphasizes horizontal edges using the smoothing effect by approximating a vertical gradient. If you need to emphasize vertical edges, transpose the filter h' .

```
[ 1 2 1
  0 0 0
 -1 -2 -1 ]
```

Class Support

h is of class `double`.

Examples

Create Various Filters and Filter an Image

Read image and display it.

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



Create a motion filter and use it to blur the image. Display the blurred image.

```
H = fspecial('motion',20,45);  
MotionBlur = imfilter(I,H,'replicate');  
imshow(MotionBlur);
```



Create a disk filter and use it to blur the image. Display the blurred image.

```
H = fspecial('disk',10);  
blurred = imfilter(I,H,'replicate');  
imshow(blurred);
```



Algorithms

fspecial creates Gaussian filters using

$$h_g(n_1, n_2) = e^{-\frac{(n_1^2 + n_2^2)}{2\sigma^2}}$$

$$h(n_1, n_2) = \frac{h_g(n_1, n_2)}{\sum_{n_1} \sum_{n_2} h_g}$$

fspecial creates Laplacian filters using

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

$$\nabla^2 = \frac{4}{(\alpha + 1)} \begin{bmatrix} \frac{\alpha}{4} & \frac{1-\alpha}{4} & \frac{\alpha}{4} \\ \frac{1-\alpha}{4} & -1 & \frac{1-\alpha}{4} \\ \frac{\alpha}{4} & \frac{1-\alpha}{4} & \frac{\alpha}{4} \end{bmatrix}$$

`fspecial` creates Laplacian of Gaussian (LoG) filters using

$$h_g(n_1, n_2) = e^{-\frac{(n_1^2 + n_2^2)}{2\sigma^2}}$$

$$h(n_1, n_2) = \frac{(n_1^2 + n_2^2 - 2\sigma^2)h_g(n_1, n_2)}{2\pi\sigma^6 \sum_{n_1} \sum_{n_2} h_g}$$

`fspecial` creates averaging filters using

`ones(n(1), n(2)) / (n(1) * n(2))`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, all inputs must be constants at compilation time.

See Also

conv2 | edge | filter2 | fsamp2 | fwind1 | fwind2 | imfilter | imsharpen | del2

Introduced before R2006a

ftrans2

2-D FIR filter using frequency transformation

Syntax

```
h = ftrans2(b, t)
h = ftrans2(b)
```

Description

`h = ftrans2(b, t)` produces the two-dimensional FIR filter `h` that corresponds to the one-dimensional FIR filter `b` using the transform `t`. (`ftrans2` returns `h` as a computational molecule, which is the appropriate form to use with `filter2`.) `b` must be a one-dimensional, Type I (even symmetric, odd-length) filter such as can be returned by `fir1`, `fir2`, or `firpm` in the Signal Processing Toolbox software. The transform matrix `t` contains coefficients that define the frequency transformation to use. If `t` is `m`-by-`n` and `b` has length `Q`, then `h` is size $((m-1)*(Q-1)/2+1)$ -by- $((n-1)*(Q-1)/2+1)$.

`h = ftrans2(b)` uses the McClellan transform matrix `t`.

```
t = [1 2 1; 2 -4 2; 1 2 1]/8;
```

All inputs and outputs should be of class `double`.

Examples

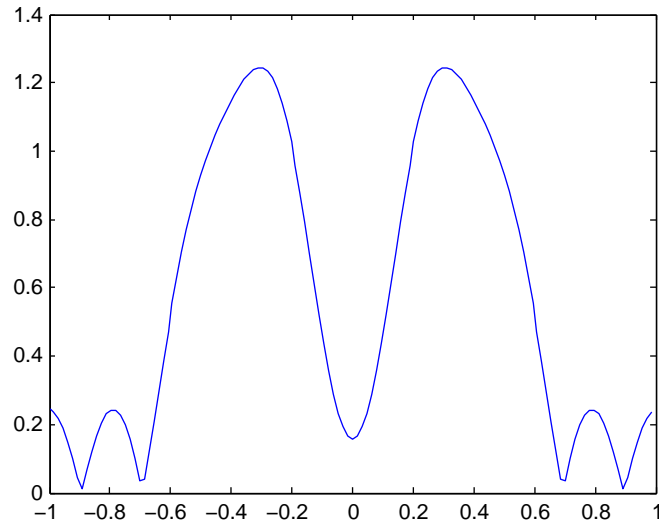
Use `ftrans2` to design an approximately circularly symmetric two-dimensional bandpass filter with passband between 0.1 and 0.6 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians):

- 1 Since `ftrans2` transforms a one-dimensional FIR filter to create a two-dimensional filter, first design a one-dimensional FIR bandpass filter using the Signal Processing Toolbox function `firpm`.

```
colormap(jet(64))
```

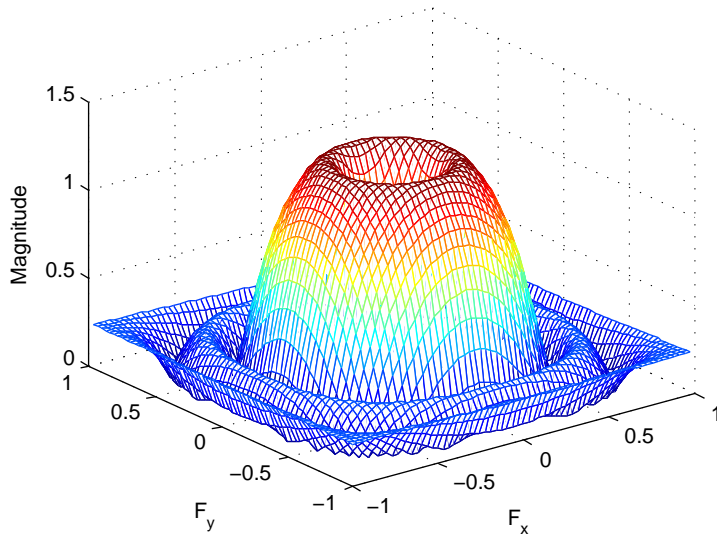


```
b = firpm(10,[0 0.05 0.15 0.55 0.65 1],[0 0 1 1 0 0]);  
[H,w] = freqz(b,1,128,'whole');  
plot(w/pi-1,fftshift(abs(H)))
```



- 2** Use `ftrans2` with the default McClellan transformation to create the desired approximately circularly symmetric filter.

```
h = ftrans2(b);  
freqz2(h)
```



Tips

The transformation below defines the frequency response of the two-dimensional filter returned by `ftrans2`.

$$H(\omega_1, \omega_2) = B(\omega) \Big|_{\cos \omega = T(\omega_1, \omega_2)},$$

where $B(\omega)$ is the Fourier transform of the one-dimensional filter \mathbf{b} :

$$B(\omega) = \sum_{n=-N}^N b(n)e^{-j\omega n}$$

and $T(\omega_1, \omega_2)$ is the Fourier transform of the transformation matrix \mathbf{t} :

$$T(\omega_1, \omega_2) = \sum_{n_2} \sum_{n_1} t(n_1, n_2)e^{-j\omega_1 n_1} e^{-j\omega_2 n_2}.$$

The returned filter h is the inverse Fourier transform of $H(\omega_1, \omega_2)$:

$$h(n_1, n_2) = \frac{1}{(2\pi)^2} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} H(\omega_1, \omega_2) e^{j\omega_1 n_1} e^{j\omega_2 n_2} d\omega_1 d\omega_2.$$

References

- [1] Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, pp. 218-237.

See Also

`conv2` | `filter2` | `fsamp2` | `fwind1` | `fwind2`

Introduced before R2006a

fwind1

2-D FIR filter using 1-D window method

Syntax

```
h = fwind1(Hd, win)
h = fwind1(Hd, win1, win2)
h = fwind1(f1, f2, Hd, ...)
```

Description

`fwind1` designs two-dimensional FIR filters using the window method. `fwind1` uses a one-dimensional window specification to design a two-dimensional FIR filter based on the desired frequency response `Hd`. `fwind1` works with one-dimensional windows only; use `fwind2` to work with two-dimensional windows.

`h = fwind1(Hd, win)` designs a two-dimensional FIR filter `h` with frequency response `Hd`. (`fwind1` returns `h` as a computational molecule, which is the appropriate form to use with `filter2`.) `fwind1` uses the one-dimensional window `win` to form an approximately circularly symmetric two-dimensional window using Huang's method. You can specify `win` using windows from the Signal Processing Toolbox software, such as `boxcar`, `hamming`, `hanning`, `bartlett`, `blackman`, `kaiser`, or `chebwin`. If `length(win)` is `n`, then `h` is `n`-by-`n`.

`Hd` is a matrix containing the desired frequency response sampled at equally spaced points between `-1.0` and `1.0` (in normalized frequency, where `1.0` corresponds to half the sampling frequency, or π radians) along the *x* and *y* frequency axes. For accurate results, use frequency points returned by `freqspace` to create `Hd`. (See the entry for `freqspace` for more information.)

`h = fwind1(Hd, win1, win2)` uses the two one-dimensional windows `win1` and `win2` to create a separable two-dimensional window. If `length(win1)` is `n` and `length(win2)` is `m`, then `h` is `m`-by-`n`.

`h = fwind1(f1, f2, Hd, ...)` lets you specify the desired frequency response `Hd` at arbitrary frequencies (`f1` and `f2`) along the *x*- and *y*-axes. The frequency vectors `f1`

and `f2` should be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or π radians. The length of the windows controls the size of the resulting filter, as above.

Class Support

The input matrix `Hd` can be of class `double` or of any integer class. All other inputs to `fwind1` must be of class `double`. All outputs are of class `double`.

Examples

Create 2-D FIR Filter using 1-D Window Method

This example shows how to design an approximately circularly symmetric two-dimensional bandpass filter using a 1-D window method.

Create the frequency range vectors `f1` and `f2` using `freqspace`. These vectors have length 21.

```
[f1,f2] = freqspace(21,'meshgrid');
```

Compute the distance of each position from the center frequency.

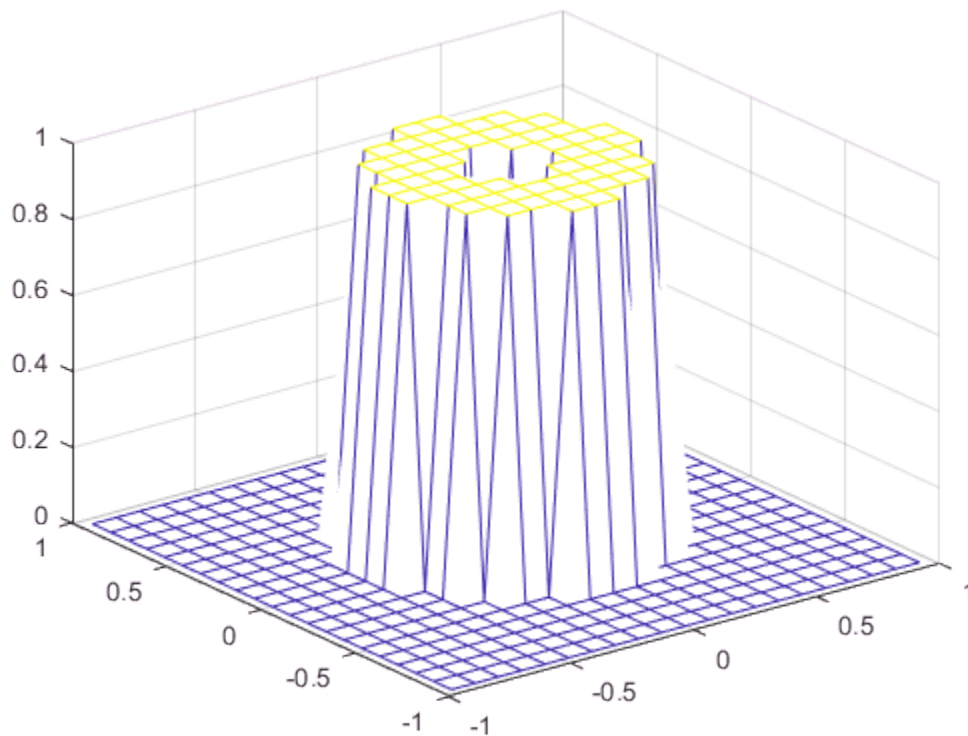
```
r = sqrt(f1.^2 + f2.^2);
```

Create a matrix `Hd` that contains the desired bandpass response. In this example, the desired passband is between 0.1 and 0.5 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians).

```
Hd = ones(21);
Hd((r<0.1)|(r>0.5)) = 0;
```

Display the ideal bandpass response.

```
colormap(parula(64))
mesh(f1,f2,Hd)
```

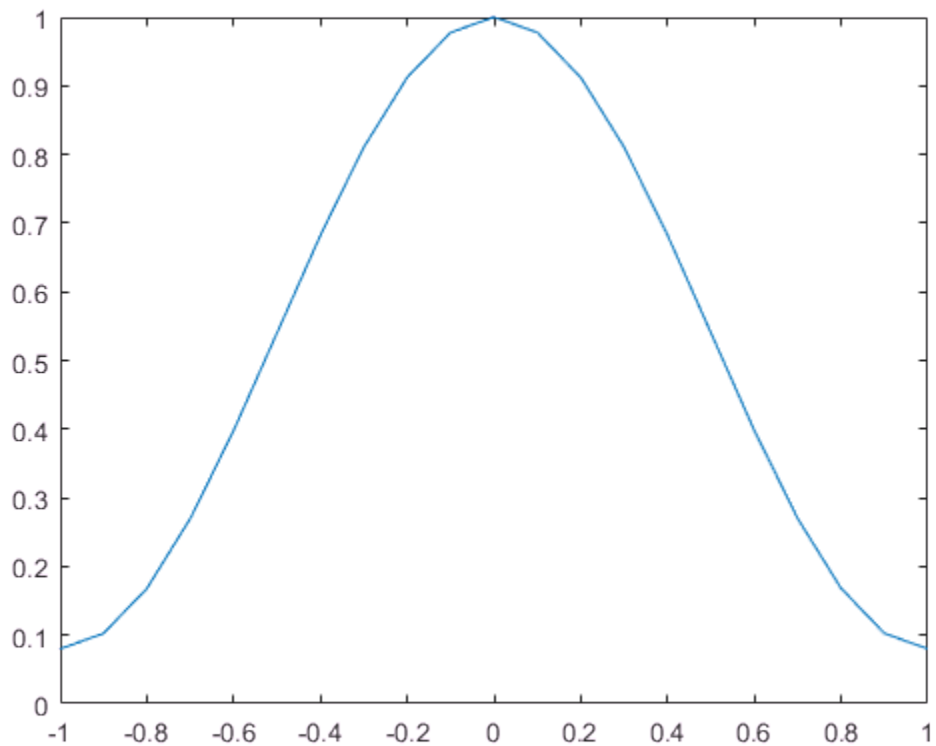


Design the 1-D window. This example uses a Hamming window of length 21.

```
win = 0.54 - 0.46*cos(2*pi*(0:20)/20);
```

Plot the 1-D window.

```
figure  
plot(linspace(-1,1,21),win);
```

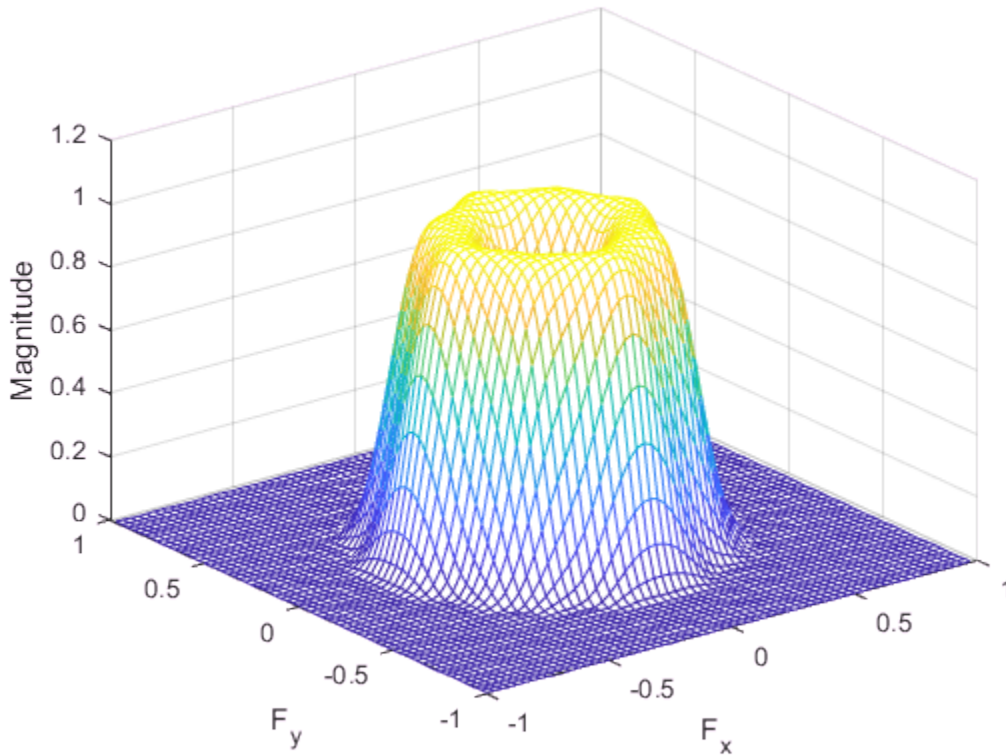


Using the 1-D window, design the filter that best produces this frequency response

```
h = fwind1(Hd,win);
```

Display the actual frequency response of this filter.

```
freqz2(h)
```



Algorithms

`fwind1` takes a one-dimensional window specification and forms an approximately circularly symmetric two-dimensional window using Huang's method,

$$w(n_1, n_2) = w(t) \Big|_{t=\sqrt{n_1^2+n_2^2}},$$

where $w(t)$ is the one-dimensional window and $w(n_1, n_2)$ is the resulting two-dimensional window.

Given two windows, `fwind1` forms a separable two-dimensional window:

$$w(n_1, n_2) = w_1(n_1)w_2(n_2).$$

`fwind1` calls `fwind2` with `Hd` and the two-dimensional window. `fwind2` computes `h` using an inverse Fourier transform and multiplication by the two-dimensional window:

$$h_d(n_1, n_2) = \frac{1}{(2\pi)^2} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} H_d(\omega_1, \omega_2) e^{j\omega_1 n_1} e^{j\omega_2 n_2} d\omega_1 d\omega_2$$

$$h(n_1, n_2) = h_d(n_1, n_2)w(n_2, n_2).$$

References

- [1] Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990.

See Also

`conv2` | `filter2` | `fsamp2` | `freqspace` | `ftrans2` | `fwind2`

Introduced before R2006a

fwind2

2-D FIR filter using 2-D window method

Syntax

```
h = fwind2(Hd, win)
h = fwind2(f1, f2, Hd, win)
```

Description

Use `fwind2` to design two-dimensional FIR filters using the window method. `fwind2` uses a two-dimensional window specification to design a two-dimensional FIR filter based on the desired frequency response `Hd`. `fwind2` works with two-dimensional windows; use `fwind1` to work with one-dimensional windows.

`h = fwind2(Hd, win)` produces the two-dimensional FIR filter `h` using an inverse Fourier transform of the desired frequency response `Hd` and multiplication by the window `win`. `Hd` is a matrix containing the desired frequency response at equally spaced points in the Cartesian plane. `fwind2` returns `h` as a computational molecule, which is the appropriate form to use with `filter2`. `h` is the same size as `win`.

For accurate results, use frequency points returned by `freqspace` to create `Hd`. (See the entry for `freqspace` for more information.)

`h = fwind2(f1, f2, Hd, win)` lets you specify the desired frequency response `Hd` at arbitrary frequencies (`f1` and `f2`) along the *x*- and *y*-axes. The frequency vectors `f1` and `f2` should be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or π radians. `h` is the same size as `win`.

Class Support

The input matrix `Hd` can be of class `double` or of any integer class. All other inputs to `fwind2` must be of class `double`. All outputs are of class `double`.

Examples

Create 2-D FIR Filter using 2-D Window Method

This example shows how to design an approximately circularly symmetric two-dimensional bandpass filter using a 2-D window method.

Create the frequency range vectors `f1` and `f2` using `freqspace`. These vectors have length 21.

```
[f1,f2] = freqspace(21, 'meshgrid');
```

Compute the distance of each position from the center frequency.

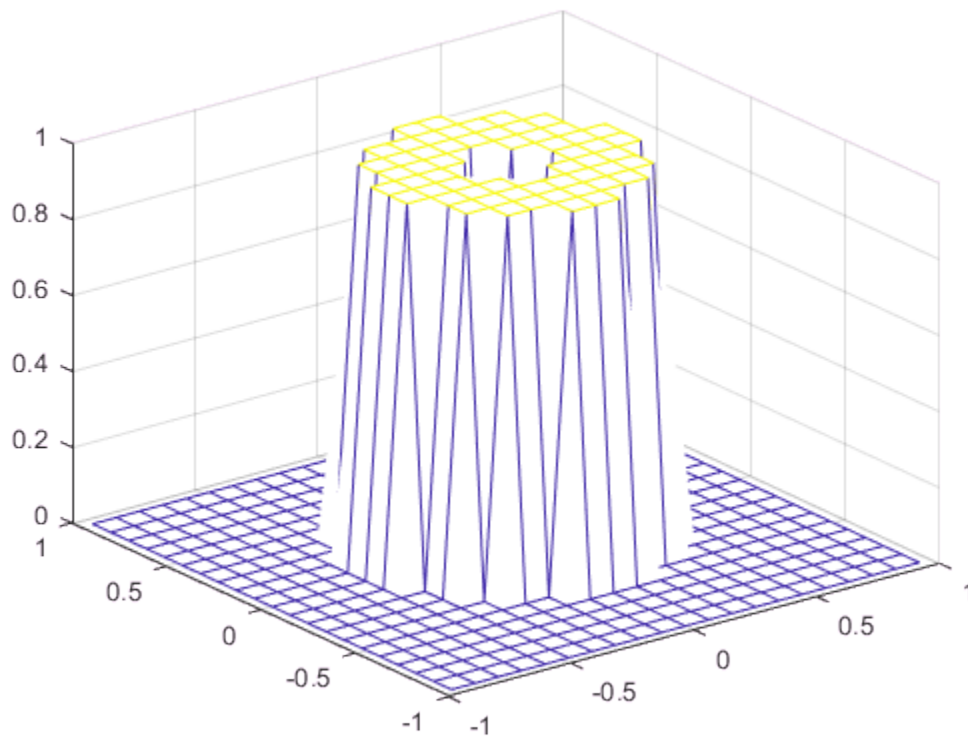
```
r = sqrt(f1.^2 + f2.^2);
```

Create a matrix `Hd` that contains the desired bandpass response. In this example, the desired passband is between 0.1 and 0.5 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or π radians).

```
Hd = ones(21);  
Hd((r<0.1)|(r>0.5)) = 0;
```

Display the ideal bandpass response.

```
colormap(parula(64))  
mesh(f1,f2,Hd)
```

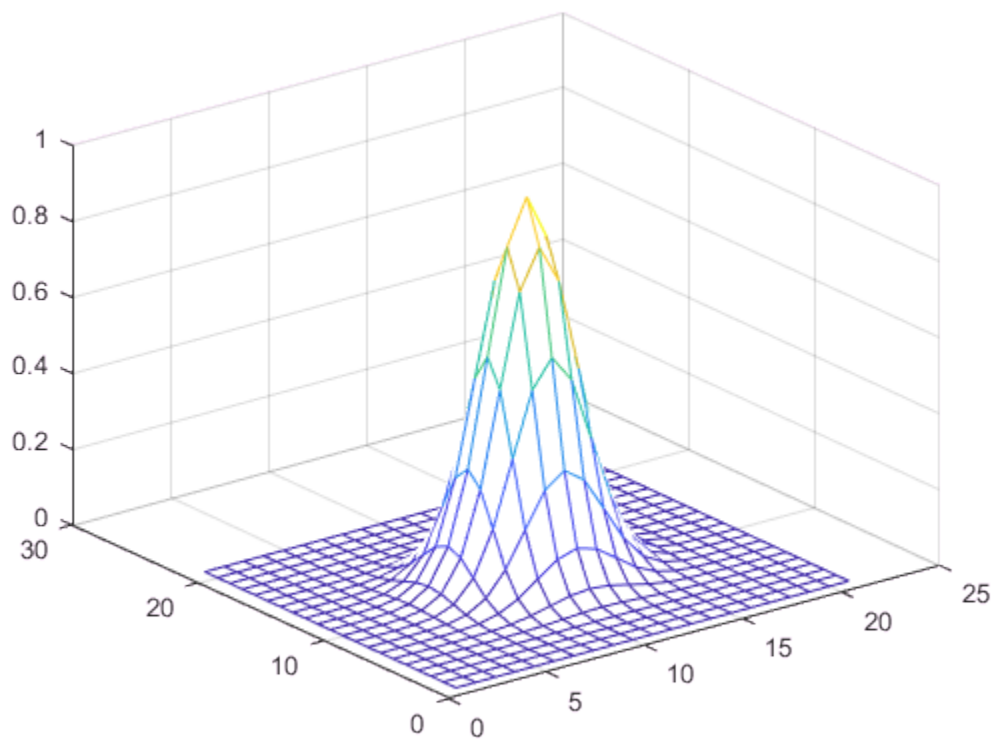


Create a 2-D Gaussian window using `fspecial`. Normalize the window.

```
win = fspecial('gaussian',21,2);  
win = win ./ max(win(:));
```

Display the window.

```
mesh(win)
```

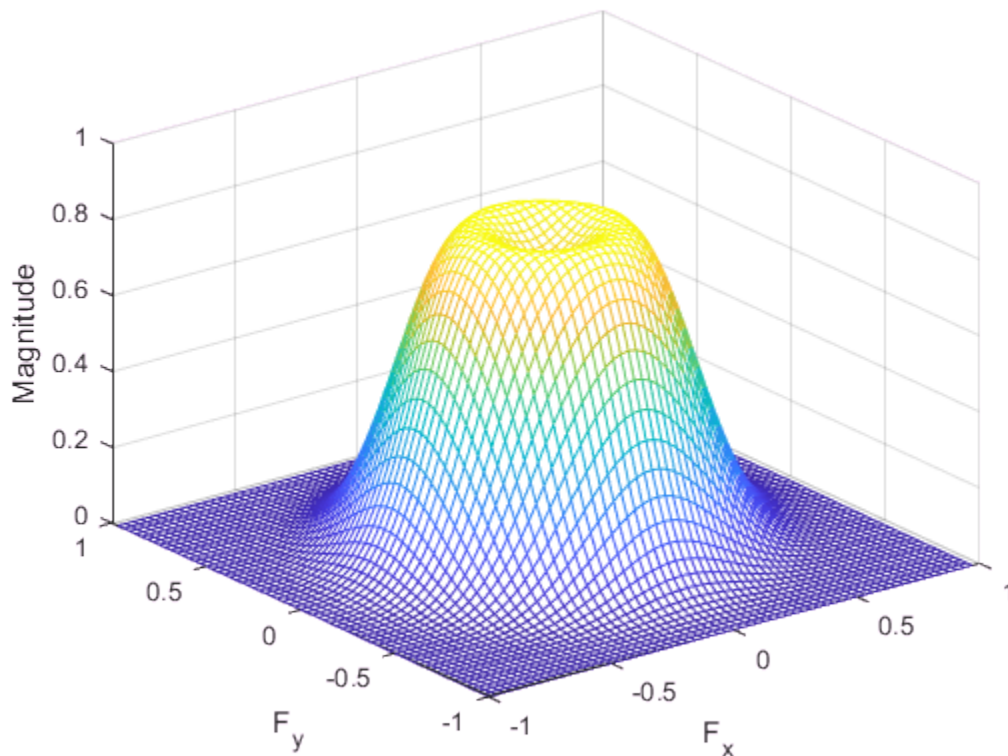


Using the 2-D window, design the filter that best produces the desired frequency response

```
h = fwind2(Hd,win);
```

Display the actual frequency response of this filter.

```
freqz2(h)
```



Algorithms

`fwind2` computes `h` using an inverse Fourier transform and multiplication by the two-dimensional window `win`.

$$h_d(n_1, n_2) = \frac{1}{(2\pi)^2} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} H_d(\omega_1, \omega_2) e^{j\omega_1 n_1} e^{j\omega_2 n_2} d\omega_1 d\omega_2$$

$$h(n_1, n_2) = h_d(n_1, n_2)w(n_1, n_2)$$

References

- [1] Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, pp. 202-213.

See Also

`conv2` | `filter2` | `fsamp2` | `freqspace` | `ftrans2` | `fwind1`

Introduced before R2006a

gabor

Create Gabor filter or Gabor filter bank

Syntax

```
g = gabor(wavelength,orientation)
g = gabor(____,Name,Value,____)
```

Description

`g = gabor(wavelength,orientation)` creates a Gabor filter with the specified `wavelength` (in pixels/cycle) and `orientation` (in degrees). If you specify `wavelength` or `orientation` as vectors, `gabor` returns an array of `gabor` objects, called a *filter bank*, that contain all the unique combinations of `wavelength` and `orientation`. For example, if `wavelength` is a vector of length 2 and `orientation` is a vector of length 3, then the output array `g` is a vector of length 6. To apply the Gabor filters to an image, use the `imgaborfilt` function.

`g = gabor(____,Name,Value,____)` creates an array of Gabor filters using name-value pairs to control aspects of Gabor filter design. If you specify a vector of values, the output array `g` contains all the unique combinations of the input values.

Examples

Construct Gabor Filter Array and Apply to Input Image

Create a sample image of a checkerboard.

```
A = checkerboard(20);
```

Create an array of Gabor filters.

```
wavelength = 20;
orientation = [0 45 90 135];
g = gabor(wavelength,orientation);
```

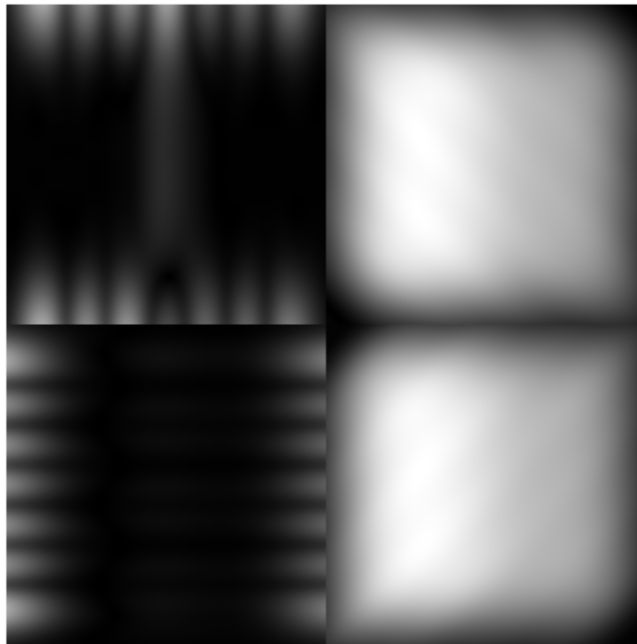
Apply the filters to the checkerboard image.


```
outMag = imgaborfilt(A,g);
```

Display the results.

```
outSize = size(outMag);  
outMag = reshape(outMag,[outSize(1:2),1,outSize(3)]);  
figure, montage(outMag,'DisplayRange',[1]);  
title('Montage of gabor magnitude output images.');
```

Montage of gabor magnitude output images.



Construct Gabor Filter Array and Visualize Wavelength and Orientation

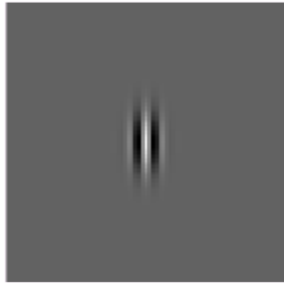
Create array of Gabor filters.

```
g = gabor([5 10],[0 90]);
```

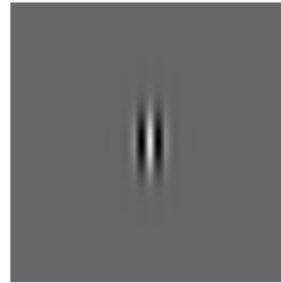
Visualize the real part of the spatial convolution kernel of each Gabor filter in the array.

```
figure;  
subplot(2,2,1)  
for p = 1:length(g)  
    subplot(2,2,p);  
    imshow(real(g(p).SpatialKernel),[]);  
    lambda = g(p).Wavelength;  
    theta = g(p).Orientation;  
    title(sprintf('Re[h(x,y)], \lambda = %d, \theta = %d',lambda,theta));  
end
```

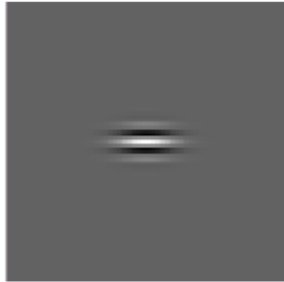
Re[h(x,y)], $\lambda = 5, \theta = 0$



Re[h(x,y)], $\lambda = 10, \theta = 0$



Re[h(x,y)], $\lambda = 5, \theta = 90$



Re[h(x,y)], $\lambda = 10, \theta = 90$



- “Texture Segmentation Using Gabor Filters”

Input Arguments

wavelength — Wavelength of sinusoid

numeric scalar in the range [2, Inf)

Wavelength of sinusoid, specified as a numeric scalar or vector, in pixels/cycle.

Example: `g = gabor(4,90);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

orientation — Orientation of filter in degrees

numeric scalar in the range [0 180]

Orientation of filter in degrees, specified as a numeric scalar in the range [0 180], where the orientation is defined as the normal direction to the sinusoidal plane wave.

Example: `g = gabor(4,90);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `g = gabor(4,90, 'SpatialFrequencyBandwidth', 1.5);`

'SpatialFrequencyBandwidth' — Define spatial-frequency bandwidth

1.0 (default) | numeric vector

A numeric vector that defines the spatial-frequency bandwidth in units of Octaves. The spatial-frequency bandwidth determines the cutoff of the filter response as frequency content in the input image varies from the preferred frequency, $1/\text{LAMBDA}$. Typical values for spatial-frequency bandwidth are in the range [0.5 2.5].

Example: `g = gabor(4,90, 'SpatialFrequencyBandwidth', 1.5);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'SpatialAspectRatio' — Aspect ratio of Gaussian in spatial domain

0.5 (default) | numeric scalar

Aspect ratio of Gaussian in spatial domain, specified as a numeric vector that defines the ratio of the semi-major and semi-minor axes of the Gaussian envelope: **semi-minor/semi-major**. This parameter controls the ellipticity of the Gaussian envelope. Typical values for spatial aspect ratio are in the range [0.23 0.92].

Example: `g = gabor(4,90,'SpatialAspectRatio',0.75);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

g — Gabor filter array

Array of `gabor` objects

Gabor filter array, returned as an array of `gabor` objects.

See Also

See Also

`imgaborfilt`

Topics

“Texture Segmentation Using Gabor Filters”

Introduced in R2015b

getheight

Height of structuring element

Note: `getheight` will be removed in a future release. See `strel` for the current list of methods.

Syntax

```
H = getheight(SE)
```

Description

`H = getheight(SE)` returns an array the same size as `getnhood(SE)` containing the height associated with each of the structuring element neighbors. `H` is all zeros for a flat structuring element.

Class Support

`SE` is a STREL object. `H` is of class `double`.

Examples

```
se = strel(ones(3,3),magic(3));  
getheight(se)
```

Introduced before R2006a

getimage

Image data from axes

Syntax

```
A = getimage(h)
[x, y, A] = getimage(h)
[... , A, flag] = getimage(h)
[...] = getimage
```

Description

`A = getimage(h)` returns the first image data contained in the graphics object `h`. `h` can be a figure, axes, or image. `A` is identical to the image `CData`; it contains the same values and is of the same class (`uint8`, `uint16`, `double`, or `logical`) as the image `CData`. If `h` is not an image or does not contain an image, `A` is empty.

`[x, y, A] = getimage(h)` returns the image `XData` in `x` and the `YData` in `y`. `XData` and `YData` are two-element vectors that indicate the range of the x -axis and y -axis.

`[... , A, flag] = getimage(h)` returns an integer `flag` that indicates the type of image `h` contains. This table summarizes the possible values for `flag`.

Flag	Type of Image
0	Not an image; <code>A</code> is returned as an empty matrix
1	Indexed image
2	Intensity image with values in standard range ([0,1] for <code>single</code> and <code>double</code> arrays, [0,255] for <code>uint8</code> arrays, [0,65535] for <code>uint16</code> arrays)
3	Intensity data, but not in standard range
4	RGB image
5	Binary image

`[...] = getimage` returns information for the current axes object. It is equivalent to `[...] = getimage(gca)`.

Class Support

The output array **A** is of the same class as the image **CData**. All other inputs and outputs are of class **double**.

Note

For **int16** and **single** images, the image data returned by **getimage** is of class **double**, not **int16** or **single**. This is because the **getimage** function gets the data from the image object's **CData** property and image objects store **int16** and **single** image data as class **double**.

For example, create an image object of class **int16**. If you retrieve the **CData** from the object and check its class, it returns **double**.

```
h = imshow(ones(10,'int16'));  
class(get(h,'CData'))
```

Therefore, if you get the image data using the **getimage** function, the data it returns is also of class **double**. The **flag** return value is set to **3**.

```
[img,flag] = getimage(h);  
class(img)
```

The same is true for an image of class **single**. Getting the **CData** directly from the image object or by using **getimage**, the class of the returned data is **double**.

```
h = imshow(ones(10,'single'));  
class(get(h,'CData'))  
[img,flag] = getimage(h);  
class(img)
```

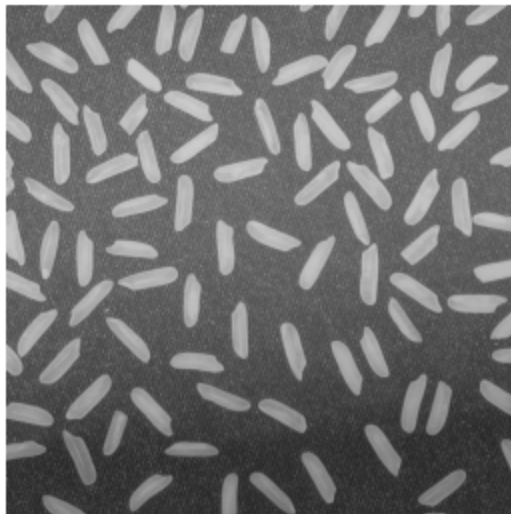
For images of class **single**, the **flag** return value is set to **2** because **single** and **double** share the same dynamic range.

Examples

Import Data into Workspace from Image Displayed in Figure or App

Display image directly from a file using **imshow** and create a variable in the workspace that contains the image data.

```
imshow rice.png  
I = getimage;
```



Display image directly from a file using the Image Viewer app (`imtool`) and create a variable in the workspace that contains the image data.

```
h = imtool('cameraman.tif');  
I = getimage(imgca);
```




See Also

`imshow` | `imshow`

Introduced before R2006a

getimagemodel

Image model object from image object

Syntax

```
imgmodel = getimagemodel(himage)
```

Description

`imgmodel = getimagemodel(himage)` returns the image model object associated with `himage`. `himage` must be a handle to an image object or an array of handles to image objects.

The return value `imgmodel` is an image model object. If `himage` is an array of handles to image objects, `imgmodel` is an array of image models.

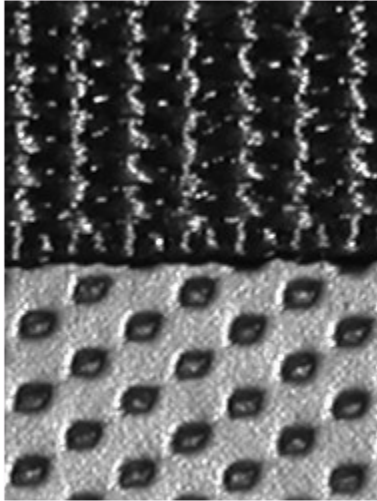
If `himage` does not have an associated image model object, `getimagemodel` creates one.

Examples

Retrieve `imagemodel` Object Associated with Image

Read an image into the workspace.

```
h = imshow('bag.png');
```



Retrieve the image model associated with this image.

```
imgmodel = getimagemodel(h)
```

```
imgmodel =
```

IMAGEMODEL object accessing an image with these properties:

```
ClassType: 'uint8'  
DisplayRange: [0 255]  
ImageHeight: 250  
ImageType: 'intensity'  
ImageWidth: 189  
MinIntensity: 0  
MaxIntensity: 255
```

See Also

`imagemodel`

Introduced before R2006a

getline

Select polyline with mouse

Syntax

```
[x, y] = getline(fig)
[x, y] = getline(ax)
[x, y] = getline
[x, y] = getline(..., 'closed')
```

Description

`[x, y] = getline(fig)` lets you select a polyline in the current axes of figure `fig` using the mouse. Coordinates of the polyline are returned in `X` and `Y`. Use normal button clicks to add points to the polyline. A shift-, right-, or double-click adds a final point and ends the polyline selection. Pressing **Return** or **Enter** ends the polyline selection without adding a final point. Pressing **Backspace** or **Delete** removes the previously selected point from the polyline.

`[x, y] = getline(ax)` lets you select a polyline in the axes specified by the handle `ax`.

`[x, y] = getline` is the same as `[x,y] = getline(gcf)`.

`[x, y] = getline(..., 'closed')` animates and returns a closed polygon.

See Also

getpts | getrect

Introduced before R2006a

getneighbors

Structuring element neighbor locations and heights

Note: `getneighbors` will be removed in a future release. See `strel` for the current list of methods.

Syntax

```
[offsets, heights] = getneighbors(SE)
```

Description

`[offsets, heights] = getneighbors(SE)` returns the relative locations and corresponding heights for each of the neighbors in the structuring element object `SE`.

`offsets` is a P-by-N array where P is the number of neighbors in the structuring element and N is the dimensionality of the structuring element. Each row of `offsets` contains the location of the corresponding neighbor, relative to the center of the structuring element.

`heights` is a P-element column vector containing the height of each structuring element neighbor.

Class Support

`SE` is a STREL object. The return values `offsets` and `heights` are arrays of double-precision values.

Examples

```
se = strel([1 0 1],[5 0 -5])
```

```
[offsets,heights] = getneighbors(se)
se =
Nonflat STREL object containing 2 neighbors.
```

```
Neighborhood:
```

```
  1    0    1
```

```
Height:
```

```
  5    0   -5
```

```
offsets =
```

```
  0   -1
```

```
  0    1
```

```
heights =
```

```
  5   -5
```

Introduced before R2006a

getnhood

Structuring element neighborhood

Note: `getnhood` will be removed in a future release. See `strel` for the current list of methods.

Syntax

```
NHOOD = getnhood(SE)
```

Description

`NHOOD = getnhood(SE)` returns the neighborhood associated with the structuring element `SE`.

Class Support

`SE` is a STREL object. `NHOOD` is a `logical` array.

Examples

```
se = strel(eye(5));  
NHOOD = getnhood(se)
```

Introduced before R2006a

getpts

Specify points with mouse

Syntax

```
[x, y] = getpts(fig)
[x, y] = getpts(ax)
[x, y] = getpts
```

Description

`[x, y] = getpts(fig)` lets you choose a set of points in the current axes of figure `fig` using the mouse. Coordinates of the selected points are returned in `X` and `Y`.

Use normal button clicks to add points. A shift-, right-, or double-click adds a final point and ends the selection. Pressing **Return** or **Enter** ends the selection without adding a final point. Pressing **Backspace** or **Delete** removes the previously selected point.

`[x, y] = getpts(ax)` lets you choose points in the axes specified by the handle `ax`.

`[x, y] = getpts` is the same as `[x,y] = getpts(gcf)`.

Examples

Select Points in Image Interactively

Display an image using `imshow`.

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

Choose points interactively in the displayed image using the mouse. Double-click to complete your selection. When you are done, `getpts` returns the position of your points.

```
pts = getpts
```

See Also

`getline` | `getrect`

Introduced before R2006a

getrangefromclass

Default display range of image based on its class

Syntax

```
range = getrangefromclass(I)
```

Description

`range = getrangefromclass(I)` returns the default display range of the image `I`, based on its class type. The function returns `range`, a two-element vector specifying the display range in the form `[min max]`.

Class Support

`I` can be `uint8`, `uint16`, `int16`, `logical`, `single`, or `double`. `range` is of class `double`.

Note

For `single` and `double` data, `getrangefromclass` returns the range `[0 1]` to be consistent with the way `double` and `single` images are interpreted in MATLAB. For integer data, `getrangefromclass` returns the default display range of the class. For example, if the class is `uint8`, the dynamic range is `[0 255]`.

Examples

Get Default Display Range of Image

Read 16-bit DICOM image into the workspace.

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

Get the display range from the image.

```
r = getrangefromclass(CT)
```

```
r =
```

```
    -32768    32767
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.

See Also

`intmin` | `intmax`

Introduced before R2006a

getrect

Specify rectangle with mouse

Syntax

```
rect = getrect
rect = getrect(fig)
rect = getrect(ax)
```

Description

`rect = getrect` lets you select a rectangle in the current axes using the mouse. Use the mouse to click and drag the desired rectangle. `rect` is a four-element vector with the form `[xmin ymin width height]`. To constrain the rectangle to be a square, use a shift- or right-click to begin the drag.

`rect = getrect(fig)` lets you select a rectangle in the current axes of figure `fig` using the mouse.

`rect = getrect(ax)` lets you select a rectangle in the axes specified by the handle `ax`.

Examples

Select Rectangle in Image Interactively

Display an image using `imshow`.

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

Choose points interactively in the displayed image using the mouse. When you are done, `getrect` returns the size and position of your rectangle.

```
rect = getrect
```

See Also

`getline` | `getpts`

Introduced before R2006a

getsequence

Sequence of decomposed structuring elements

Note: `getsequence` will be removed in a future release. See `strel` for the current list of methods.

Syntax

`SEQ = getsequence(SE)`

Description

`SEQ = getsequence(SE)` returns the array of structuring elements `SEQ`, containing the individual structuring elements that form the decomposition of `SE`. `SE` can be an array of structuring elements. `SEQ` is equivalent to `SE`, but the elements of `SEQ` have no decomposition.

Class Support

`SE` and `SEQ` are arrays of `STREL` objects.

Examples

The `strel` function uses decomposition for square structuring elements larger than 3-by-3. Use `getsequence` to extract the decomposed structuring elements.

```
se = strel('square',5)
se =
Flat STREL object containing 25 neighbors.
Decomposition: 2 STREL objects containing a total of 10 neighbors
```

```
Neighborhood:
  1   1   1   1   1
```

```
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
```

```
seq = getsequence(se)
seq =
2x1 array of STREL objects
```

Use `imdilate` with the `'full'` option to see that dilating sequentially with the decomposed structuring elements really does form a 5-by-5 square:

```
imdilate(1,seq,'full')
```

Introduced before R2006a

gradientweight

Calculate weights for image pixels based on image gradient

Syntax

```
W = gradientweight(I)
W = gradientweight(I,sigma)
W = gradientweight(___,Name,Value)
```

Description

`W = gradientweight(I)` calculates the pixel weight for each pixel in image `I` based on the gradient magnitude at that pixel, and returns the weight array `W`. The weight of a pixel is inversely related to the gradient values at the pixel location. Pixels with small gradient magnitude (smooth regions) have a large weight and pixels with large gradient magnitude (such as on the edges) have a small weight.

`W = gradientweight(I,sigma)` uses `sigma` as the standard deviation for the Derivative of Gaussian that is used for computing the image gradient.

`W = gradientweight(___,Name,Value)` returns the weight array `W` using name-value pairs to control aspects of weight computation.

Examples

Segment Image Using Weights Derived from Image Gradient

This example segments an image using the Fast Marching Method based on the weights derived from the image gradient.

Read image and display it.

```
I = imread('coins.png');
imshow(I)
title('Original Image')
```

Original Image



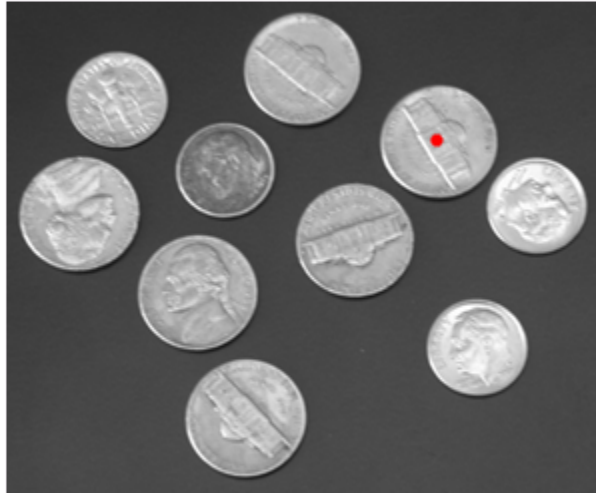
Compute weights based on image gradient.

```
sigma = 1.5;  
W = gradientweight(I, sigma, 'RolloffFactor', 3, 'WeightCutoff', 0.25);
```

Select a seed location.

```
R = 70; C = 216;  
hold on;  
plot(C, R, 'r.', 'LineWidth', 1.5, 'MarkerSize', 15);  
title('Original Image with Seed Location')
```

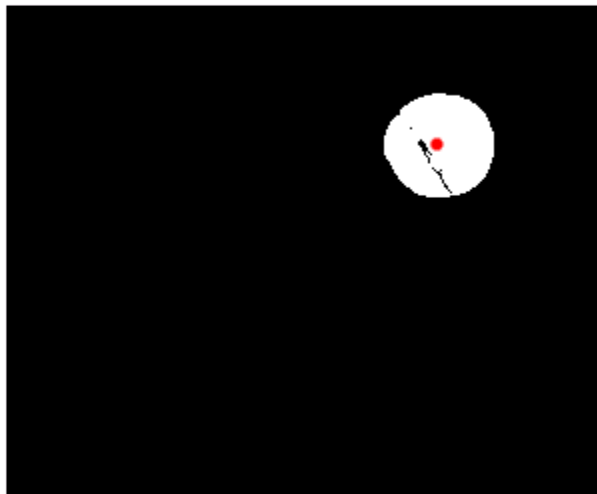
Original Image with Seed Location



Segment the image using the weight array.

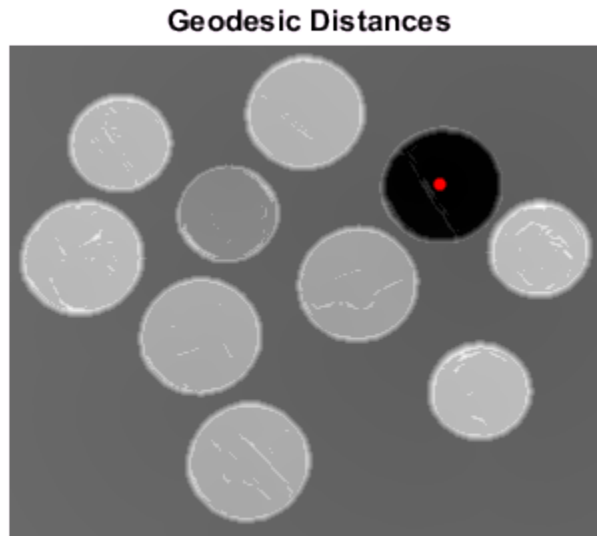
```
thresh = 0.1;  
[BW, D] = imsegfmm(W, C, R, thresh);  
figure, imshow(BW)  
title('Segmented Image')  
hold on;  
plot(C, R, 'r.', 'LineWidth', 1.5, 'MarkerSize', 15);
```

Segmented Image



Geodesic distance matrix D can be thresholded using different thresholds to get different segmentation results.

```
figure, imshow(D)
title('Geodesic Distances')
hold on;
plot(C, R, 'r.', 'LineWidth', 1.5, 'MarkerSize', 15);
```



Input Arguments

I — Input image

grayscale image

Input image, specified as a grayscale image. Must be nonsparse.

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

Data Types: `single` | `double` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`

sigma — Standard deviation for Derivative of Gaussian

1.5 (default) | positive scalar

Standard deviation for Derivative of Gaussian, specified as a positive scalar of class `double`.

Example: `sigma = 1.0; W = gradientweight(I, sigma)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `W = gradientweight(I, 1.5, 'RolloffFactor', 3, 'WeightCutoff', 0.25);`

'RolloffFactor' — Output weight roll-off factor

3 (default) | positive scalar

Output weight roll-off factor, specified as the comma-separated pair consisting of 'RolloffFactor' and a positive scalar of class `double`. Controls how fast weight values fall as a function of gradient magnitude. When viewed as a 2-D plot, pixel intensity values might vary gradually at the edges of regions, creating a gentle slope. In your segmented image, you might want the edge to be more well-defined. Using the roll-off factor, you control the slope of the weight value curve at points where intensity values start to change. If you specify a high value, the output weight values fall off sharply around the edges of smooth regions. If you specify a low value, the output weight has a more gradual fall-off around the edges. The suggested range for this parameter is [0.5 4].

Data Types: `double`

'WeightCutoff' — Threshold for weight values

0.25 (default) | positive scale in the range [1e-3 1]

Threshold for weight values, specified as the comma-separated pair consisting of 'WeightCutoff' and a positive scalar of class `double`. If you use this parameter to set a threshold on weight values, it suppresses any weight values less than the value you specify, setting these pixels to a small constant value (1e-3). This parameter can be useful in improving the accuracy of the output when you use the output weight array `W` as input to Fast Marching Method segmentation function, `imsegfmm`.

Data Types: `double`

Output Arguments

W – Weight array

numeric array

Weight array, returned as a numeric array. The weight array is the same size as the input image, **I**. The weight array is of class **double**, unless **I** is **single**, in which case it is of class **single**.

Tips

- `gradientweight` uses double-precision floating point operations for internal computations for all classes of **I**, except when **I** is of class **single**, in which case `gradientweight` uses single-precision floating point operations internally.

See Also

See Also

`graydiffweight` | `imseggmm`

Introduced in R2014b

gray2ind

Convert grayscale or binary image to indexed image

Syntax

```
[X, map] = gray2ind(I,n)  
[X, map] = gray2ind(BW,n)
```

Description

`[X, map] = gray2ind(I,n)` converts the grayscale image `I` to an indexed image `X`. `n` specifies the size of the colormap, `gray(n)`. `n` must be an integer between 1 and 65536. If `n` is omitted, it defaults to 64.

`[X, map] = gray2ind(BW,n)` converts the binary image `BW` to an indexed image `X`. `n` specifies the size of the colormap, `gray(n)`. If `n` is omitted, it defaults to 2.

`gray2ind` scales and then rounds the intensity image to produce an equivalent indexed image.

Class Support

The input image `I` can be `logical`, `uint8`, `uint16`, `int16`, `single`, or `double` and must be a real and nonsparse. The image `I` can have any dimension. The class of the output image `X` is `uint8` if the colormap length is less than or equal to 256; otherwise it is `uint16`.

Examples

Convert Grayscale Image to Indexed Image

Read grayscale image into the workspace.

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


Convert the image to an indexed image using `gray2ind`. This example creates an indexed image with 16 indices.

```
[X, map] = gray2ind(I, 16);
```

Display the indexed image.

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



See Also

`grayslice` | `ind2gray` | `mat2gray`

Introduced before R2006a

graycomatrix

Create gray-level co-occurrence matrix from image

Syntax

```
glcms = graycomatrix(I)
glcms = graycomatrix(I,Name,Value,...)
[glcms,SI] = graycomatrix(____)
```

Description

`glcms = graycomatrix(I)` creates a gray-level co-occurrence matrix (GLCM) from image `I`. Another name for a gray-level co-occurrence matrix is a *gray-level spatial dependence matrix*. Also, the word co-occurrence is frequently used in the literature without a hyphen, cooccurrence.

`graycomatrix` creates the GLCM by calculating how often a pixel with gray-level (grayscale intensity) value i occurs horizontally adjacent to a pixel with the value j . (You can specify other pixel spatial relationships using the 'Offsets' parameter.) Each element (i,j) in `glcm` specifies the number of times that the pixel with value i occurred horizontally adjacent to a pixel with value j .

`glcms = graycomatrix(I,Name,Value,...)` returns one or more gray-level co-occurrence matrices, depending on the values of the optional name/value pairs. Parameter names can be abbreviated, and case does not matter.

`[glcms,SI] = graycomatrix(____)` returns the scaled image, `SI`, used to calculate the gray-level co-occurrence matrix. The values in `SI` are between 1 and `NumLevels`.

Examples

Create Gray-Level Co-occurrence Matrix for Grayscale Image

Read a grayscale image into workspace.

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

Calculate the gray-level co-occurrence matrix (GLCM) for the grayscale image. By default, `graycomatrix` calculates the GLCM based on horizontal proximity of the pixels: [0 1]. That is the pixel next to the pixel of interest on the same row. This example specifies a different offset: two rows apart on the same column.

```
glcm = graycomatrix(I, 'Offset', [2 0])
```

```
glcm =
```

14205	2107	126	0	0	0	0
2242	14052	3555	400	0	0	0
191	3579	7341	1505	37	0	0
0	683	1446	7184	1368	0	0
0	7	116	1502	10256	1124	0
0	0	0	2	1153	1435	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Create Gray-Level Co-occurrence Matrix Returning Scaled Image

Create a simple 3-by-6 sample array.

```
I = [ 1 1 5 6 8 8; 2 3 5 7 0 2; 0 2 3 5 6 7]
```

```
I =
```

1	1	5	6	8	8
2	3	5	7	0	2
0	2	3	5	6	7

Calculate the gray-level co-occurrence matrix (GLCM) and return the scaled image used in the calculation. By specifying empty brackets for the `GrayLimits` parameter, the example uses the minimum and maximum grayscale vales in the input image as limits.

```
[glcm, SI] = graycomatrix(I, 'NumLevels', 9, 'GrayLimits', [])
```

```
glcm =
```

0	0	2	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0
0	0	0	2	0	0	0	0	0
0	0	0	0	0	2	0	0	0

```

0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 2 1 0
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 1

```

SI =

```

2 2 6 7 9 9
3 4 6 8 1 3
1 3 4 6 7 8

```

Calculate GLCMs using Four Different Offsets

Read grayscale image into the workspace.

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

Define four offsets.

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

Calculate the GLCMs, returning the scaled image as well.

```
[glcms, SI] = graycomatrix(I, 'Offset', offsets);
```

Note how the function returns an array of four GLCMs.

whos

Name	Size	Bytes	Class	Attributes
I	159x191	30369	uint8	
SI	159x191	242952	double	
glcms	8x8x4	2048	double	
offsets	4x2	64	double	

Calculate Symmetric GLCM for Grayscale Image

Read grayscale image into the workspace.

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

Calculate GLCM using the `Symmetric` option. The GLCM created when you set `Symmetric` to true is symmetric across its diagonal, and is equivalent to the GLCM described by Haralick (1973).

```
glcm = graycomatrix(I, 'Offset', [2 0], 'Symmetric', true)
```

```
glcm =
```

```

28410    4349    317      0      0      0      0
 4349   28104   7134   1083      7      0      0
  317    7134  14682   2951   153      0      0
   0    1083   2951  14368   2870      2      0
   0      7    153   2870  20512   2277      0
   0      0      0      2   2277   2870      0
   0      0      0      0      0      0      0
   0      0      0      0      0      0      0
```

Input Arguments

I — Input image

2-D, real, nonsparse, numeric or logical array

Input image, specified as a 2-D, real, nonsparse, numeric or logical array.

Example:

```
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 |
uint32 | uint64 | logical
```

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

'GrayLimits' — Range used scaling input image into gray levels

range specified by class (default) | two-element vector [low high]

Range used scaling input image into gray levels, specified as a two-element vector [low high]. If N is the number of gray levels (see parameter 'NumLevels') to use for scaling, the range [low high] is divided into N equal width bins and values in a bin get mapped to a single gray level. Grayscale values less than or equal to low are scaled to 1. Grayscale values greater than or equal to high are scaled to 'NumLevels'. If 'GrayLimits' is set to [], graycomatrix uses the minimum and maximum grayscale values in I as limits, [min(I(:)) max(I(:))], for example, [0 1] for class double and [-32768 32767] for class int16.

Example:

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

'NumLevels' — Number of gray levels

8 for numeric, 2 for binary (default) | integer

Number of gray levels, specified as an integer. For example, if NumLevels is 8, graycomatrix scales the values in I so they are integers between 1 and 8. The number of gray-levels determines the size of the gray-level co-occurrence matrix (glcm).

Example:

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

'Offset' — Distance between the pixel of interest and its neighbor

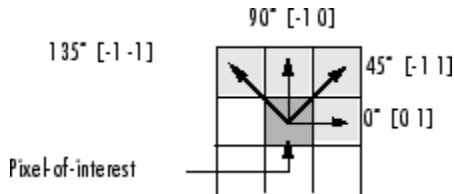
[0 1] (default) | p-by-2 array of integers

Distance between the pixel of interest and its neighbor, specified as a p-by-2 array of integers. Each row in the array is a two-element vector, [row_offset, col_offset], that specifies the relationship, or offset, of a pair of pixels. row_offset is the number of rows between the pixel-of-interest and its neighbor. col_offset is the number of columns between the pixel-of-interest and its neighbor. Because the offset is often expressed as an angle, the following table lists the offset values that specify common angles, given the pixel distance D.

Angle	Offset
0	[0 D]
45	[-D D]
90	[-D 0]

Angle	Offset
135	[-D -D]

The figure illustrates the array: `offset = [0 1; -1 1; -1 0; -1 -1]`



Example:

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

'Symmetric' — Consider ordering of values

`false` (default) | `true`

Consider ordering of values, specified as the Boolean value `true` or `false`. For example, when `'Symmetric'` is set to `true`, `graycomatrix` counts both 1,2 and 2,1 pairings when calculating the number of times the value 1 is adjacent to the value 2. When `'Symmetric'` is set to `false`, `graycomatrix` only counts 1,2 or 2,1, depending on the value of `'offset'`.

Example:

Data Types: `logical`

Output Arguments

glcms — Gray-level co-occurrence matrix (or matrices)

`double` array

Gray-level co-occurrence matrix (or matrices), returned as an `NumLevels`-by-`NumLevels`-by-`P` array of class `double`, where `P` is the number of offsets in `Offset`.

SI — Scaled image used in calculation of GLCM

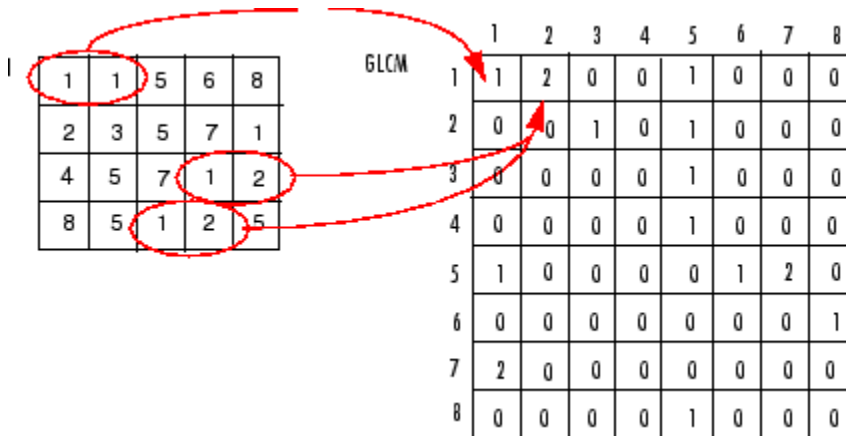
`double` matrix

Scaled image used in calculation of GLCM, returned as a **double** matrix the same size as the input image.

Algorithms

`graycomatrix` calculates the GLCM from a scaled version of the image. By default, if `I` is a binary image, `graycomatrix` scales the image to two gray-levels. If `I` is an intensity image, `graycomatrix` scales the image to eight gray-levels. You can specify the number of gray-levels `graycomatrix` uses to scale the image by using the `'NumLevels'` parameter, and the way that `graycomatrix` scales the values using the `'GrayLimits'` parameter.

The following figure shows how `graycomatrix` calculates several values in the GLCM of the 4-by-5 image `I`. Element (1,1) in the GLCM contains the value 1 because there is only one instance in the image where two, horizontally adjacent pixels have the values 1 and 1. Element (1,2) in the GLCM contains the value 2 because there are two instances in the image where two, horizontally adjacent pixels have the values 1 and 2. `graycomatrix` continues this processing to fill in all the values in the GLCM.



`graycomatrix` ignores pixel pairs if either of the pixels contains a NaN, replaces positive `Inf`s with the value `NumLevels`, and replaces negative `Inf`s with the value 1. `graycomatrix` ignores border pixels, if the corresponding neighbor pixel falls outside the image boundaries.

The GLCM created when 'Symmetric' is set to `true` is symmetric across its diagonal, and is equivalent to the GLCM described by Haralick (1973). The GLCM produced by the following syntax, with 'Symmetric' set to `true`

```
graycomatrix(I, 'offset', [0 1], 'Symmetric', true)
```

is equivalent to the sum of the two GLCMs produced by the following statements where 'Symmetric' is set to `false`.

```
graycomatrix(I, 'offset', [0 1], 'Symmetric', false)  
graycomatrix(I, 'offset', [0 -1], 'Symmetric', false)
```

References

- [1] Haralick, R.M., K. Shanmugan, and I. Dinstein, "Textural Features for Image Classification", IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-3, 1973, pp. 610-621.
- [2] Haralick, R.M., and L.G. Shapiro. Computer and Robot Vision: Vol. 1, Addison-Wesley, 1992, p. 459.

See Also

See Also

graycoprops

Introduced before R2006a

grayconnected

Select contiguous image region with similar gray values

Syntax

```
BW = grayconnected(I,row,column)
BW = grayconnected(I,row,column,tolerance)
```

Description

`BW = grayconnected(I,row,column)` finds connected regions of similar intensity in the grayscale image `I`. You specify the intensity value to use as a starting point, the *seed pixel*, by `row` and `column` indices. By default, `grayconnected` includes connected pixels with values in the range `[seedpixel-32,seedpixel+32]` for integer-valued images and within the range `[seedpixel-0.1,seedpixel+0.1]` for floating point images. `grayconnected` returns a binary mask image, `BW`, where all of the foreground pixels are 8-connected to the seed pixel at `(row, column)` by pixels of similar intensity.

`BW = grayconnected(I,row,column,tolerance)` finds connected regions of similar intensity in a grayscale image, where `tolerance` specifies the range of intensity values to include in the mask, as in `[(seedpixel-tolerance),(seedpixel+tolerance)]`.

Examples

Create Binary Mask from Connected Pixels

Create small sample image.

```
I = uint8([20 22 24 23 25 20 100
           21 10 12 13 12 30 6
           22 11 13 12 13 25 5
           23 13 13 13 13 20 5
           24 13 13 12 12 13 5
           25 26 5 28 29 50 6]);
```

Create mask image, specifying the seed location by row and column and the tolerance. Since the seed location specifies the pixel with the value 23 and the tolerance is 3, the range of grayscale values is [20,26].

```
seedrow = 4
seedrow = 4
seedcol = 1
seedcol = 1
tol = 3
tol = 3
BW = grayconnected(I,seedrow,seedcol,tol)
```

```
BW = 6x7 logical array
 1  1  1  1  1  1  0
 1  0  0  0  0  0  0
 1  0  0  0  0  0  0
 1  0  0  0  0  0  0
 1  0  0  0  0  0  0
 1  1  0  0  0  0  0
```

Input Arguments

I — Input grayscale image

real, nonsparse 2-D matrix

Input grayscale image, specified as a real, nonsparse, 2-D matrix.

Example: `BW = grayconnected(I,50,40);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

row — Row index of seed location

real, positive, scalar integer.

Row index of seed location, specified as a real, positive, scalar integer.

Example: `BW = grayconnected(I,50,40);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

column — Column index of seed location

real, positive, scalar integer

Column index of seed location, specified as a real, positive, scalar integer.

Example: `BW = grayconnected(I,50,40);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

tolerance — Range of intensity values to include in the mask

32 for integer valued images, 0.1 for floating point images (default) | numeric scalar

Range of intensity values to include in the mask, specified as a numeric scalar. The range is defined as $[(\text{seedvalue} - \text{tolerance}), (\text{seedvalue} + \text{tolerance})]$. By default, `grayconnected` includes connected pixels with values in the range $[\text{seedpixel} - 32, \text{seedpixel} + 32]$ for integer-valued images and within the range $[\text{seedpixel} - 0.1, \text{seedpixel} + 0.1]$ for floating point images.

Example: `BW = grayconnected(I,50,40,5);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output Arguments

BW — Mask binary image

logical array

Mask binary image, returned as a logical array where all of the foreground pixels are 8-connected to the seed pixel at `(row,column)` by pixels of similar intensity.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

See Also

See Also

`bwselect` | Image Segmenter | `imfill`

Introduced in R2015b

graycoprops

Properties of gray-level co-occurrence matrix

Syntax

```
stats = graycoprops(glcm,properties)
```

Description

`stats = graycoprops(glcm,properties)` calculates the statistics specified in `properties` from the gray-level co-occurrence matrix `glcm`. `glcm` is an m -by- n -by- p array of valid gray-level co-occurrence matrices. If `glcm` is an array of GLCMs, `stats` is an array of statistics for each `glcm`.

`graycoprops` normalizes the gray-level co-occurrence matrix (GLCM) so that the sum of its elements is equal to 1. Each element (r,c) in the normalized GLCM is the joint probability occurrence of pixel pairs with a defined spatial relationship having gray level values r and c in the image. `graycoprops` uses the normalized GLCM to calculate `properties`.

Examples

Calculate Statistics from Gray-level Co-occurrence Matrix

Create simple sample GLCM.

```
glcm = [0 1 2 3;1 1 2 3;1 0 2 0;0 0 0 3]
```

```
glcm =
```

```
    0     1     2     3
    1     1     2     3
    1     0     2     0
    0     0     0     3
```

Calculate statistical properties of the GLCM.

```
stats = graycoprops(glcm)

stats = struct with fields:
    Contrast: 2.8947
    Correlation: 0.0783
    Energy: 0.1191
    Homogeneity: 0.5658
```

Calculate Contrast and Homogeneity from Multiple GLCMs

Read grayscale image into the workspace.

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

Create two gray-level co-occurrence matrices (GLCM) from the image, specifying different offsets.

```
glcm = graycomatrix(I, 'Offset', [2 0; 0 2])
```

```
glcm =
```

```
(:,:,1) =
```

14205	2107	126	0	0	0	0
2242	14052	3555	400	0	0	0
191	3579	7341	1505	37	0	0
0	683	1446	7184	1368	0	0
0	7	116	1502	10256	1124	0
0	0	0	2	1153	1435	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

```
(:,:,2) =
```

13938	2615	204	4	0	0	0
2406	14062	3311	630	23	0	0
145	3184	7371	1650	133	0	0
2	371	1621	6905	1706	0	0
0	0	116	1477	9974	1173	0
0	0	0	1	1161	1417	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Get statistics on contrast and homogeneity of the image from the GLCMs.

```
stats = graycoprops(g1cm,{'contrast','homogeneity'})

stats = struct with fields:
    Contrast: [0.3420 0.3567]
    Homogeneity: [0.8567 0.8513]
```

Input Arguments

g1cm — Gray-level Co-occurrence Matrix

real, non-negative array of finite logical or numeric integers

Gray-level Co-occurrence Matrix, specified as a real, non-negative array of finite logical or numeric integers. Use the `graycomatrix` function to create a GLCM.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

properties — Statistical properties of the image derived from GLCM

'all' (default) | comma-separated list of property names | cell array of property names | space-separated list of property names

Statistical properties of the image derived from GLCM, specified as a comma-separated list of names, space-separated list of names, cell array of property names, or 'all'. You can specify any of the property names listed in this table. Property names can be abbreviated and are not case sensitive.

Property	Description	Formula
'Contrast'	Returns a measure of the intensity contrast between a pixel and its neighbor over the whole image. Range = $[0 \text{ (size(GLCM,1)-1)^2}]$ Contrast is 0 for a constant image. The property Contrast is also known as <i>variance</i> and <i>inertia</i> .	$\sum_{i,j} i-j ^2 p(i,j)$

Property	Description	Formula
'Correlation'	Returns a measure of how correlated a pixel is to its neighbor over the whole image. Range = [-1 1] Correlation is 1 or -1 for a perfectly positively or negatively correlated image. Correlation is NaN for a constant image.	$\sum_{i,j} \frac{(i - \mu_i)(j - \mu_j)p(i, j)}{\sigma_i \sigma_j}$
'Energy'	Returns the sum of squared elements in the GLCM. Range = [0 1] Energy is 1 for a constant image. The property Energy is also known as <i>uniformity</i> , <i>uniformity of energy</i> , and <i>angular second moment</i> .	$\sum_{i,j} p(i, j)^2$
'Homogeneity'	Returns a value that measures the closeness of the distribution of elements in the GLCM to the GLCM diagonal. Range = [0 1] Homogeneity is 1 for a diagonal GLCM.	$\sum_{i,j} \frac{p(i, j)}{1 + i - j }$

Data Types: char | cell

Output Arguments

stats — Statistics derived from the GLCM structure

Statistics derived from the GLCM, returned as a structure with fields that are specified by **properties**. Each field contains a 1-by-*p* array, where *p* is the number of gray-level co-occurrence matrices in GLCM. For example, if GLCM is an 8-by-8-by-3 array and **properties** is 'Energy', **stats** is a structure containing the field **Energy**, which contains a 1-by-3 array.

See Also

See Also

graycomatrix

Introduced before R2006a

graydiffweight

Calculate weights for image pixels based on grayscale intensity difference

Syntax

```
W = graydiffweight(I,refGrayVal)
W = graydiffweight(I,mask)
W = graydiffweight(I,C,R)
W = graydiffweight(V,C,R,P)
W = graydiffweight(___, Name,Value)
```

Description

`W = graydiffweight(I,refGrayVal)` computes the pixel weight for each pixel in the grayscale image `I`. The weight is the absolute value of the difference between the intensity of the pixel and the reference grayscale intensity specified by the scalar `refGrayVal`. Pick a reference grayscale intensity value that is representative of the object you want to segment. The weights are returned in the array `W`, which is the same size as input image `I`.

The weight of a pixel is inversely related to the absolute value of the grayscale intensity difference at the pixel location. If the difference is small (intensity value close to `refGrayVal`), the weight value is large. If the difference is large (intensity value very different from `refGrayVal`), the weight value is small.

`W = graydiffweight(I,mask)` computes the pixel weights, where the reference grayscale intensity value is the average of the intensity values of all the pixels in `I` that are marked as logical `true` in `mask`. Using the average of several pixels to calculate the reference grayscale intensity value can be more effective than using a single reference intensity value, as in the previous syntax.

`W = graydiffweight(I,C,R)` computes the pixel weights, where the reference grayscale intensity value is the average of the intensity values of the pixel locations specified by the vectors `C` and `R`. `C` and `R` contain the column and row indices of the pixel locations that must be valid pixel indices in `I`.

`W = graydiffweight(V,C,R,P)` computes the weights for each voxel in the volume `V`, specified by the vectors `C`, `R`, and `P`. `C`, `R`, and `P` contain the column, row, and plane indices of the voxel locations that must be valid voxel indices in `V`.

`W = graydiffweight(___, Name, Value)` returns the weight array `W` using name-value pairs to control aspects of weight computation.

Examples

Calculate Grayscale Intensity Difference Weights

This example segments an object in an image using Fast Marching Method using grayscale intensity difference weights calculated from the intensity values at the seed locations.

Read image and display it.

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

Original Image

Specify row and column index of pixels for use a reference grayscale intensity value.

```
seedpointR = 159;  
seedpointC = 67;
```

Calculate the grayscale intensity difference weight array for the image and display it. The example does log-scaling of W for better visualization.

```
W = graydiffweight(I, seedpointC, seedpointR, 'GrayDifferenceCutoff', 25);  
figure, imshow(log(W),[])
```



Segment the image using the grayscale intensity difference weight array. Specify the same seed point vectors you used to create the weight array.

```
thresh = 0.01;  
BW = imseghmm(W, seedpointC, seedpointR, thresh);  
figure, imshow(BW)  
title('Segmented Image')
```

Segmented Image

Input Arguments

I — Input image

grayscale image

Input image, specified as a grayscale image. Must be nonsparse.

Data Types: `single` | `double` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`

V — Input volume

3-D grayscale image

Input volume, specified as a 3-D grayscale image. Must be nonsparse.

Data Types: `single` | `double` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`

refGrayVal — Reference grayscale intensity value

scalar

Reference grayscale intensity value, specified as a scalar.

Data Types: `double`

mask — Reference grayscale intensity mask

logical array

Reference grayscale intensity mask, specified as a logical array, the same size as `I`.

Data Types: `logical`

C — Column index of reference pixel (or voxel)

numeric vector

Column index of reference pixel (or voxel), specified as a numeric (integer-valued) vector.

Data Types: `double`

R — Row index of reference pixel (or voxel)

numeric vector

Row index of reference pixel (or voxel), specified as a numeric (integer-valued) vector.

Data Types: `double`

P — Plane index of reference voxel

numeric vector

Plane index of reference voxel, specified as a numeric (integer-valued) vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `W = graydiffweight(I, seedpointC, seedpointR, 'GrayDifferenceCutoff', 25);`

'RolloffFactor' — Output weight roll-off factor

0.5 (default) | positive scalar

Output weight roll-off factor, specified as the comma-separated pair consisting of **'RolloffFactor'** and a positive scalar of class **double**. Controls how fast the output weight falls as the function of the absolute difference between an intensity value and the reference grayscale intensity. When viewed as a 2-D plot, pixel intensity values can vary gradually at the edges of regions, creating a gentle slope. In your segmented image, you might want the edge to be more well-defined. Using the roll-off factor, you control the slope of the weight value curve at points where intensity values start to change. If you specify a high value, the output weight values fall off sharply around the regions of change intensity. If you specify a low value, the output weight has a more gradual fall-off around the regions of changing intensity. The suggested range for this parameter is [0.5 4].

Data Types: **double****'GrayDifferenceCutoff' — Threshold for absolute grayscale intensity difference values**

Inf (default) | nonnegative scalar

Threshold for absolute grayscale intensity difference values, specified as the comma-separated pair consisting of **'GrayDifferenceCutoff'** and a nonnegative scalar of class **double**. When you put a threshold on intensity difference values, you strongly suppress output weight values greater than the cutoff value. **graydiffweight** assigns these pixels the smallest weight value. When the output weight array **W** is used for Fast Marching Method based segmentation (as input to **imsegfmm**), this parameter can be useful in improving the accuracy of the segmentation output. Default value of this parameter is **Inf**, which means that there is no hard cutoff.

Data Types: **double**

Output Arguments

W — Weight array

numeric array

Weight array, specified as numeric array the same size as **I**. **W** is of class **double**, unless **I** is of class **single**, in which case **W** is of class **single**.

See Also

See Also

`gradientweight` | `graydist` | `imsegfmm`

Introduced in R2014b

graydist

Gray-weighted distance transform of grayscale image

Syntax

```
T = graydist(A,mask)
T = graydist(A,C,R)
T = graydist(A,ind)
T = graydist(...,method)
```

Description

`T = graydist(A,mask)` computes the gray-weighted distance transform of the grayscale image `A`. Locations where `mask` is `true` are seed locations.

`T = graydist(A,C,R)` uses vectors `C` and `R` to specify the row and column coordinates of seed locations.

`T = graydist(A,ind)` specifies the linear indices of seed locations using the vector `ind`.

`T = graydist(...,method)` specifies an alternate distance metric. `method` determines the chamfer weights that are assigned to the local neighborhood during outward propagation. Each pixel's contribution to the geodesic time is based on the chamfer weight in a particular direction multiplied by the pixel intensity.

Input Arguments

A

Grayscale image.

mask

Logical image the same size as `A` that specifies seed locations.

C,R

Numeric vectors that contain the positive integer row and column coordinates of the seed locations. Coordinate values are valid C,R subscripts in A.

ind

Numeric vector of positive integer, linear indices of seed locations.

method

Type of distance metric. `method` can have any of these values.

Method	Description
'cityblock'	In 2-D, the cityblock distance between (x_1,y_1) and (x_2,y_2) is $ x_1 - x_2 + y_1 - y_2 $.
'chessboard'	The chessboard distance is $\max(x_1 - x_2 , y_1 - y_2)$.
'quasi-euclidean'	The quasi-Euclidean distance is $ x_1 - x_2 + (\sqrt{2} - 1) y_1 - y_2 $, $ x_1 - x_2 > y_1 - y_2 $ $(\sqrt{2} - 1) x_1 - x_2 + y_1 - y_2 $, otherwise.

Default: 'chessboard'

Output Arguments

T

Array the same size as A that specifies the gray-weighted distance transform. If the input numeric type of A is `double`, the output numeric type of T is `double`. If the input is any other numeric type, the output T is single.

Class Support

A can be numeric or logical, and it must be nonsparse. `mask` is a logical array of the same size as A. C, R, and `ind` are numeric vectors that contain positive integer values.

The output `T` is an array of the same size as `A`. If the input numeric type of `A` is `double`, the output `T` is `double`. If the input is any other numeric type, the output `T` is `single`.

Examples

Compute Minimum Path in Magic Square

Create a magic square. Matrices generated by the `magic` function have equal row, column, and diagonal sums. The minimum path between the upper-left and lower-right corner is along the diagonal.

```
A = magic(3)
```

```
A =
```

```

     8     1     6
     3     5     7
     4     9     2

```

Calculate the gray-weighted distance transform, specifying the upper left corner and the lower right corner of the square as seed locations.

```
T1 = graydist(A,1,1);
```

```
T2 = graydist(A,3,3);
```

Sum the two transforms to find the minimum path between the seed locations. As expected, there is a constant-value minimum path along the diagonal.

```
T = T1 + T2
```

```
T =
```

```

    10    11    17
    13    10    13
    17    17    10

```

Algorithms

`graydist` uses the geodesic time algorithm described in Soille, P., *Generalized geodesy via geodesic time*, Pattern Recognition Letters, vol.15, December 1994; pp. 1235–1240

The basic equation for geodesic time along a path is:

$$\tau_f(P) = \frac{f(p_o)}{2} + \frac{f(p_l)}{2} + \sum_{i=1}^{l-1} f(p_i)$$

See Also

`bwdist` | `bwdistgeodesic` | `watershed`

Introduced in R2011b

grayscale

Convert grayscale image to indexed image using multilevel thresholding

Syntax

```
X = grayscale(I, n)
```

Description

`X = grayscale(I, n)` thresholds the intensity image `I` returning an indexed image in `X`. `grayscale` uses the threshold values:

$$\frac{1}{n}, \frac{2}{n}, \dots, \frac{n-1}{n}$$

`X = grayscale(I, v)` thresholds the intensity image `I` using the values of `v`, where `v` is a vector of values between 0 and 1, returning an indexed image in `X`.

You can view the thresholded image using `imshow(X, map)` with a colormap of appropriate length.

Class Support

The input image `I` can be of class `uint8`, `uint16`, `int16`, `single`, or `double`, and must be nonsparse. Note that the threshold values are always between 0 and 1, even if `I` is of class `uint8` or `uint16`. In this case, each threshold value is multiplied by 255 or 65535 to determine the actual threshold to use.

The class of the output image `X` depends on the number of threshold values, as specified by `n` or `length(v)`. If the number of threshold values is less than 256, then `X` is of class `uint8`, and the values in `X` range from 0 to `n` or `length(v)`. If the number of threshold values is 256 or greater, `X` is of class `double`, and the values in `X` range from 1 to `n+1` or `length(v)+1`.

Examples

Convert Grayscale Image to Indexed Image Using Thresholding

Read grayscale image into the workspace.

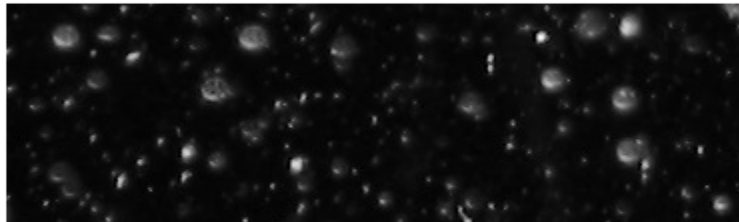
```
I = imread('snowflakes.png');
```

Threshold the intensity image, returning an indexed image.

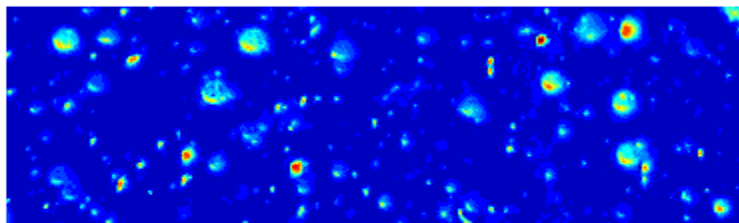
```
X = grayslice(I,16);
```

Display the original image and the indexed image, using one of the standard colormaps.

```
imshow(I)
```



```
figure  
imshow(X, jet(16))
```

See Also

`gray2ind`

Introduced before R2006a

graythresh

Global image threshold using Otsu's method

Syntax

```
level = graythresh(I)  
[level,EM] = graythresh(I)
```

Description

`level = graythresh(I)` computes a global threshold, `level`, that can be used to convert an intensity image to a binary image with `imbinarize`. The `graythresh` function uses Otsu's method, which chooses the threshold to minimize the intraclass variance of the black and white pixels [1].

Multidimensional arrays are converted automatically to 2-D arrays using `reshape`. The `graythresh` function ignores any nonzero imaginary part of `I`.

`[level,EM] = graythresh(I)` returns the effectiveness metric, `EM`, as the second output argument. The effectiveness metric is a value in the range [0, 1] that indicates the effectiveness of the thresholding of the input image. The lower bound is attainable only by images having a single gray level, and the upper bound is attainable only by two-valued images.

Examples

Convert Intensity Image to Binary Image Using Level Threshold

Read a grayscale image into the workspace.

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

Calculate a threshold using `graythresh`. The threshold is normalized to the range [0, 1].

```
level = graythresh(I)
```

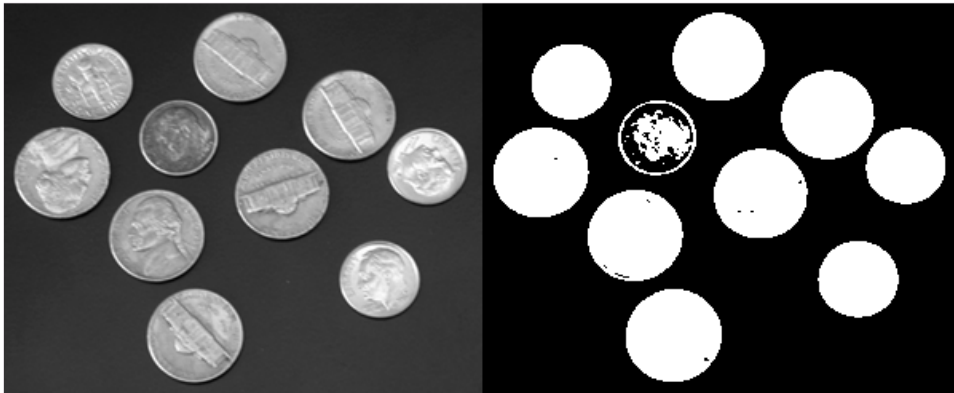
```
level = 0.4941
```

Convert the image into a binary image using the threshold.

```
BW = imbinarize(I,level);
```

Display the original image next to the binary image.

```
imshowpair(I,BW,'montage')
```



Input Arguments

I — Intensity image

nonsparse 2-D array

Intensity image, specified as a nonsparse 2-D array.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

Output Arguments

level — Global threshold

positive scalar

Global threshold, returned as a positive scalar. `level` is a normalized intensity value in the range [0, 1].

Data Types: `double`

EM — Effectiveness metric

positive scalar

Effectiveness metric, returned as a positive scalar.

Data Types: `double`

Tips

- By default, the function `imbinarize` creates a binary image using a threshold obtained using Otsu's method. This default threshold is identical to the threshold returned by `graythresh`. However, `imbinarize` only returns the binary image. If you want to know the level or the effectiveness metric, use `graythresh` before calling `imbinarize`.

References

- [1] Otsu, N., "A Threshold Selection Method from Gray-Level Histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 9, No. 1, 1979, pp. 62-66.

See Also

See Also

`imbinarize` | `imquantize` | `multithresh` | `rgb2ind`

Introduced before R2006a

hdrread

Read high dynamic range (HDR) image

Syntax

```
hdr = hdrread(filename)
```

Description

`hdr = hdrread(filename)` reads the high dynamic range (HDR) image from the file specified by `filename`. `hdr` is an m-by-n-by-3 RGB array in the range `[0, inf)` of type `single`. For scene-referred data sets, these values usually are scene illumination in radiance units. To display these images, use an appropriate tone-mapping operator.

Class Support

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

Examples

Read and Display High Dynamic Range Image

Read high dynamic range image into the workspace.

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

Convert the HDR image to a lower dynamic range, suitable for display.

```
rgb = tonemap(hdr);
```

Display the image.

```
imshow(rgb);
```



References

- [1] Larson, Greg W. “Radiance File Formats”<http://radsite.lbl.gov/radiance/refer/filefmts.pdf>

See Also

hdrwrite | makehdr | tonemap

Introduced in R2007b

hdrwrite

Write Radiance high dynamic range (HDR) image file

Syntax

```
hdrwrite(hdr, filename)
```

Description

`hdrwrite(hdr, filename)` creates a Radiance high dynamic range (HDR) image file from HDR, a single- or double-precision high dynamic range RGB image. The HDR file with the name `filename` uses run-length encoding to minimize file size.

Examples

Write High Dynamic Range Image to File

Read a high dynamic range image into the workspace.

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

Create a new HDR file, writing the high dynamic range data, `hdr`, to a file with a new `filename`.

```
hdrwrite(hdr, 'newHDRfile.hdr');
```

See Also

`hdrread` | `makehdr` | `tonemap`

Introduced in R2008a

histeq

Enhance contrast using histogram equalization

Syntax

```
J = histeq(I,hgram)
J = histeq(I,n)
[J,T] = histeq(I)

[gpuarrayJ,gpuarrayT] = histeq(gpuarrayI, ___ )

newmap = histeq(X,map)
newmap = histeq(X,map,hgram)
[newmap,T] = histeq(X, ___ )
```

Description

`J = histeq(I,hgram)` transforms the intensity image `I` so that the histogram of the output intensity image `J` with `length(hgram)` bins approximately matches the target histogram `hgram`.

`J = histeq(I,n)` transforms the intensity image `I`, returning in `J` an intensity image with `n` discrete gray levels. A roughly equal number of pixels is mapped to each of the `n` levels in `J`, so that the histogram of `J` is approximately flat. The histogram of `J` is flatter when `n` is much smaller than the number of discrete levels in `I`.

`[J,T] = histeq(I)` returns the grayscale transformation `T` that maps gray levels in the image `I` to gray levels in `J`.

`[gpuarrayJ,gpuarrayT] = histeq(gpuarrayI, ___)` performs the histogram equalization on a GPU. The input image and the output image are of type `gpuArray`. This syntax requires the Parallel Computing Toolbox.

`newmap = histeq(X,map)` transforms the values in the colormap so that the histogram of the gray component of the indexed image `X` is approximately flat. It returns the transformed colormap in `newmap`.

`newmap = histeq(X,map,hgram)` transforms the colormap associated with the indexed image `X` so that the histogram of the gray component of the indexed image

(X,newmap) approximately matches the target histogram hgram. The histeq function returns the transformed colormap in newmap. length(hgram) must be the same as size(map,1).

[newmap,T] = histeq(X, ___) returns the grayscale transformation T that maps the gray component of map to the gray component of newmap.

Examples

Enhance Contrast Using Histogram Equalization

Read an image into the workspace.

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

Enhance the contrast of an intensity image using histogram equalization.

```
J = histeq(I);
```

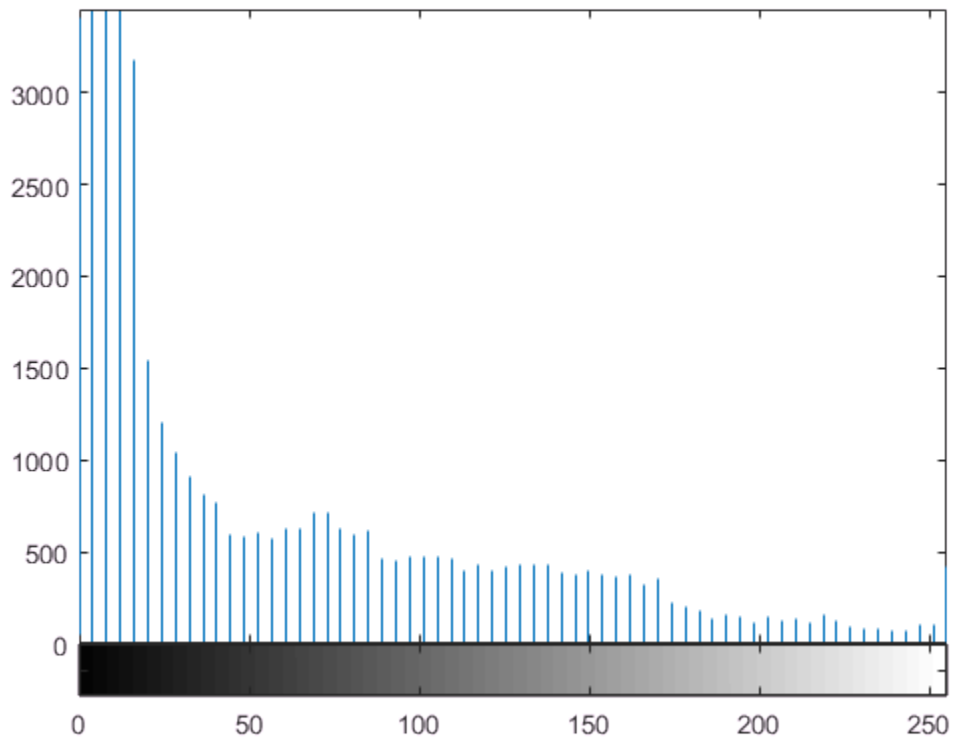
Display the original image and the adjusted image.

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



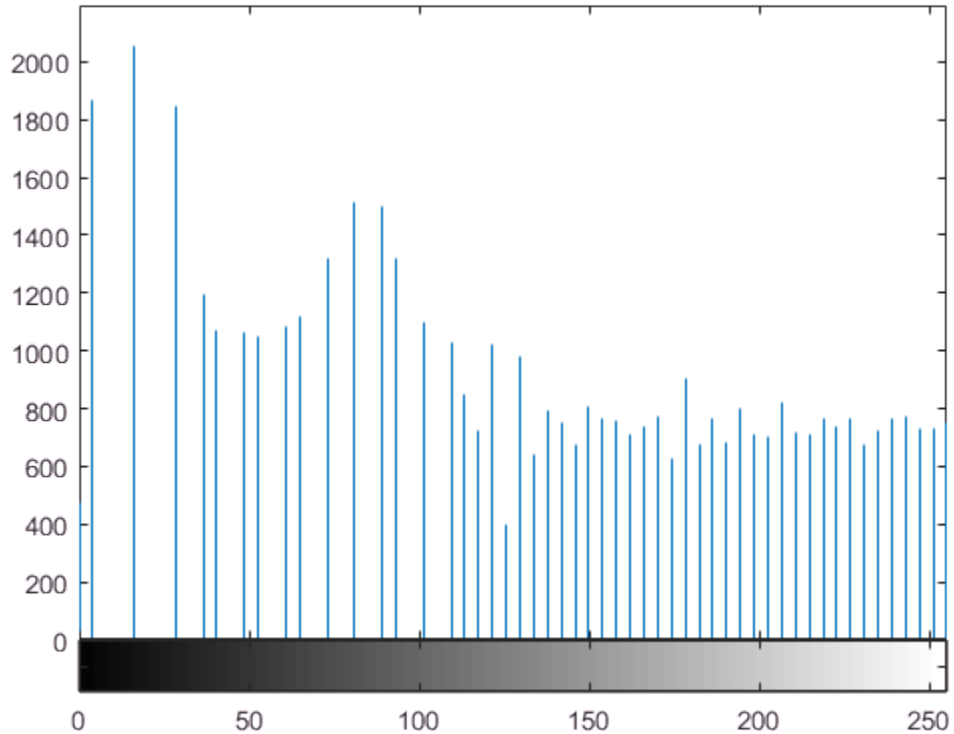
Display a histogram of the original image.

```
figure  
imhist(I,64)
```



Display a histogram of the processed image.

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



Enhance Contrast of Volumetric Image Using Histogram Equalization

Load a 3-D dataset.

```
load mrystack
```

Perform histogram equalization.

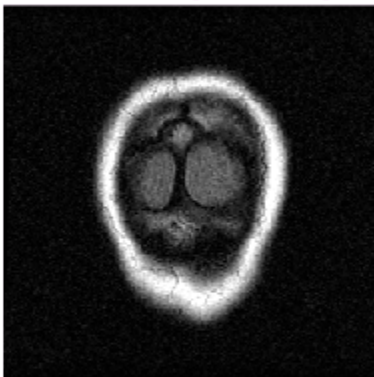
```
enhanced = histeq(mrystack);
```

Display the first slice of data for the original image and the contrast-enhanced image.

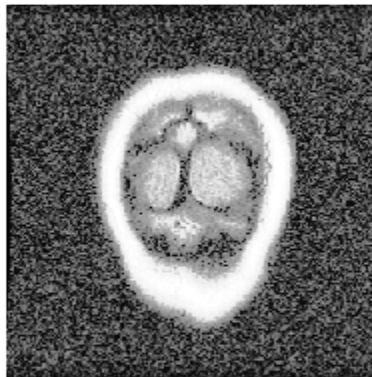
```
figure
```

```
subplot(1,2,1)
imshow(mrystack(:,:,1))
title('Slice of Original Image')
subplot(1,2,2)
imshow(enhanced(:,:,1))
title('Slice of Enhanced Image')
```

Slice of Original Image



Slice of Enhanced Image



Enhance Contrast Using Histogram Equalization on a GPU

This example performs the same histogram equalization on the GPU.

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

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

Input Arguments

I — Input intensity image

numeric array

Input intensity image, specified as a numeric array. **I** can be 2-D, 3-D, or *N*-D.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

hgram — Target histogram

numeric vector

Target histogram, specified as a numeric vector. **hgram** has equally spaced bins with intensity values in the appropriate range: `[0, 1]` for images of class `double` or `single`, `[0, 255]` for images of class `uint8`, `[0, 65535]` for images of class `uint16`, and `[-32768, 32767]` for images of class `int16`. `histeq` automatically scales **hgram** so that `sum(hgram)=numel(I)`. The histogram of **J** will better match **hgram** when `length(hgram)` is much smaller than the number of discrete levels in **I**.

Data Types: `single` | `double`

n — Number of discrete gray levels

64 (default) | scalar

Number of discrete gray levels, specified as a scalar.

Data Types: `single` | `double`

gpuarrayI — Input image when run on a GPU

gpuArray

Input image when run on a GPU, specified as a `gpuArray`.

X — Indexed Image

array of real numeric values

Indexed image, specified as an array of real numeric values. The values in *X* are an index into `map`. *X* can be 2-D, 3-D, or *N*-D.

Data Types: `single` | `double` | `uint8` | `uint16`

map — **Colormap**

n-by-3 array

Colormap, specified as an *n*-by-3 array. Each row specifies an RGB color value.

Data Types: `double`

Output Arguments

J — **Output intensity image**

numeric array

Output intensity image, returned as a numeric array of the same class as the input image *I*. *J* also has the same dimensions as *I*.

T — **Grayscale transformation**

numeric vector

Grayscale transformation, returned as a numeric vector. The transformation *T* maps gray levels in the image *I* to gray levels in *J*.

Data Types: `double`

gpuarrayJ — **Output image when run on a GPU**

`gpuArray`

Output image when run on a GPU, returned as a `gpuArray`.

gpuarrayT — **Grayscale transformation when run on a GPU**

`gpuArray`

Grayscale transformation when run on a GPU, returned as a `gpuArray`.

newmap — **Colormap**

n-by-3 array

Transformed colormap, specified as an *n*-by-3 array. Each row specifies an RGB color value.

Data Types: double

Algorithms

When you supply a desired histogram `hgram`, `histeq` chooses the grayscale transformation T to minimize

$$|c_1(T(k)) - c_0(k)|,$$

where c_0 is the cumulative histogram of `A`, c_1 is the cumulative sum of `hgram` for all intensities k . This minimization is subject to the constraints that T must be monotonic and $c_1(T(a))$ cannot overshoot $c_0(a)$ by more than half the distance between the histogram counts at a . `histeq` uses the transformation $b = T(a)$ to map the gray levels in `X` (or the colormap) to their new values.

If you do not specify `hgram`, `histeq` creates a flat `hgram`,

```
hgram = ones(1,n)*prod(size(A))/n;
```

and then applies the previous algorithm.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, `histeq` does not support indexed images.

See Also

See Also

brighten | gpuArray | imadjust | imhist

Introduced before R2006a

hough

Hough transform

Syntax

```
[H,theta,rho] = hough(BW)
[H,theta,rho] = hough(BW,Name,Value,...)
```

Description

`[H,theta,rho] = hough(BW)` computes the Standard Hough Transform (SHT) of the binary image `BW`. The `hough` function is designed to detect lines. The function uses the parametric representation of a line: $\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta)$. The function returns `rho`, the distance from the origin to the line along a vector perpendicular to the line, and `theta`, the angle in degrees between the x -axis and this vector. The function also returns the Standard Hough Transform, `H`, which is a parameter space matrix whose rows and columns correspond to `rho` and `theta` values respectively. For more information, see “Algorithms” on page 1-552.

`[H,theta,rho] = hough(BW,Name,Value,...)` computes the Standard Hough Transform (SHT) of the binary image `BW`, where named parameters affect the computation.

Examples

Compute and Display Hough Transform

Read image and convert to intensity image.

```
RGB = imread('gantrycrane.png');
I = rgb2gray(RGB);
```

Extract edges.

```
BW = edge(I, 'canny');
```

Calculate Hough transform.

```
[H,T,R] = hough(BW, 'RhoResolution',0.5, 'ThetaResolution',0.5);
```

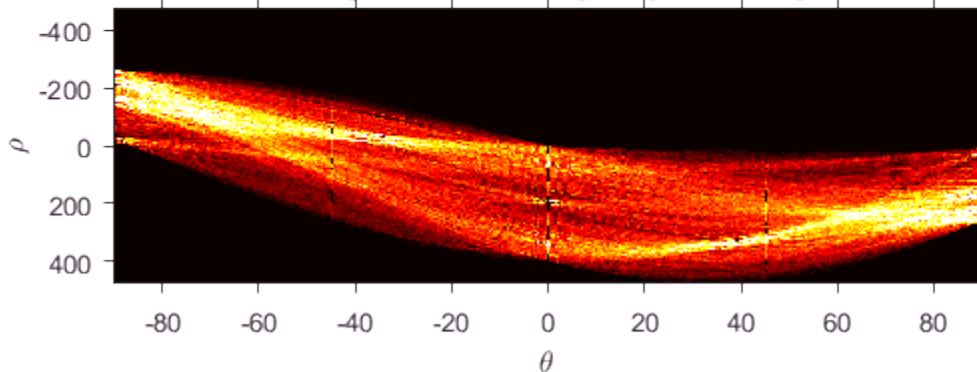
Display the original image and the Hough matrix.

```
subplot(2,1,1);
imshow(IMG);
title('gantrycrane.png');
subplot(2,1,2);
imshow(imadjust(mat2gray(H)), 'XData',T, 'YData',R, ...
       'InitialMagnification','fit');
title('Hough transform of gantrycrane.png');
xlabel('\theta'), ylabel('\rho');
axis on, axis normal, hold on;
colormap(gca,hot);
```

gantrycrane.png



Hough transform of gantrycrane.png



Compute Hough Transform Over Limited Theta Range

Read image and convert it to grayscale.

```
RGB = imread('gantrycrane.png');  
I = rgb2gray(RGB);
```

Extract edges.

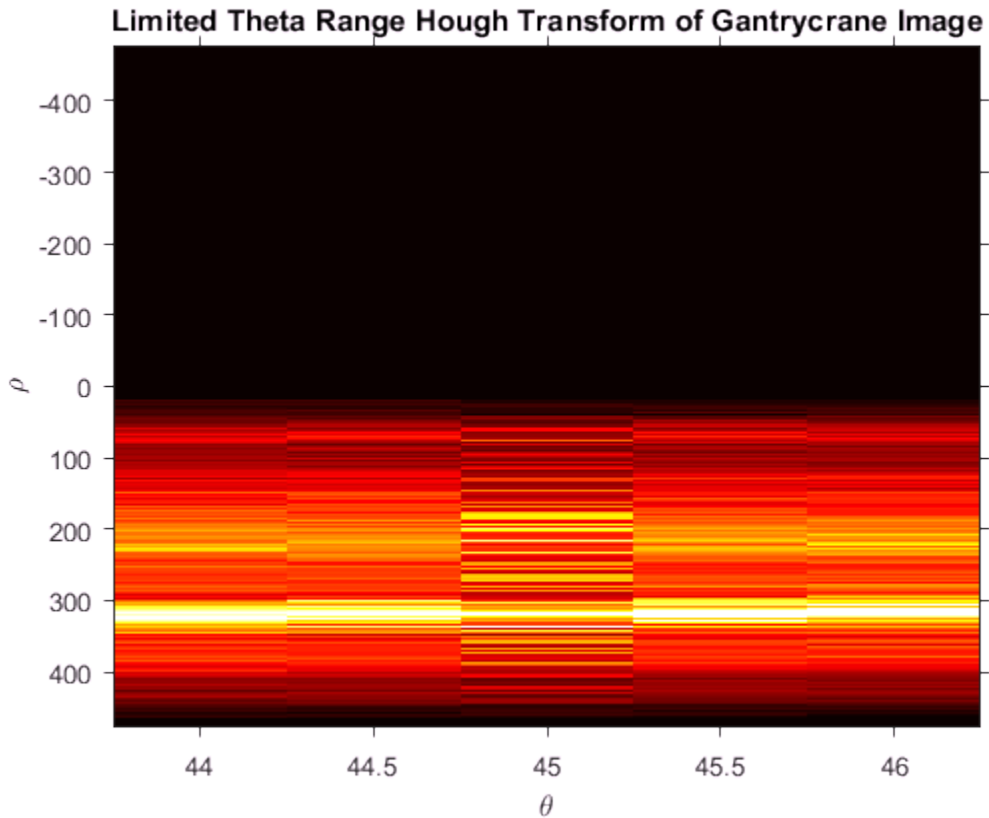
```
BW = edge(I, 'canny');
```

Calculate the Hough transform.

```
[H,T,R] = hough(BW, 'Theta', 44:0.5:46);
```

Display the Hough transform.

```
figure  
imshow(imadjust(mat2gray(H)), 'XData', T, 'YData', R, ...  
       'InitialMagnification', 'fit');  
title('Limited Theta Range Hough Transform of Gantrycrane Image');  
xlabel('\theta'), ylabel('\rho');  
axis on, axis normal;  
colormap(gca, hot)
```



- “Detect Lines in Images Using Hough”

Input Arguments

BW — Binary image

real, 2-D, nonsparse logical or numeric array

Binary image, specified as a real, 2-D, nonsparse logical or numeric array.

Example: `[H,T,R] = hough(BW);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[H,T,R] = hough(BW, 'RhoResolution',0.5, 'Theta',0.5);`

'RhoResolution' — Spacing of Hough transform bins along the *rho* axis

1 (default) | real, numeric scalar between 0 and `norm(size(BW))`, exclusive

Spacing of Hough transform bins along the *rho* axis, specified as the comma-separated pair consisting of 'RhoResolution' and a real, numeric scalar between 0 and `norm(size(BW))`, exclusive.

Data Types: `double`

'Theta' — *Theta* value for the corresponding column of the output matrix H

-90:89 (default) | real, numeric vector

Theta value for the corresponding column of the output matrix H, specified as the comma-separated pair consisting of 'Theta' and a real, numeric vector within the range [-90, 90).

Data Types: `double`

Output Arguments

H — Hough transform matrix

numeric array

Hough transform matrix, returned as a numeric array, *nrho*-by-*ntheta* in size. The rows and columns correspond to `rho` and `theta` values. For more information, see “Algorithms” on page 1-552.

theta — Angle in degrees between the x-axis and the *rho* vector

numeric array

Angle in degrees between the x -axis and the rho vector, returned as a numeric array of class `double`. For more information, see “Algorithms” on page 1-552.

rho — Distance from the origin to the line along a vector perpendicular to the line

numeric array

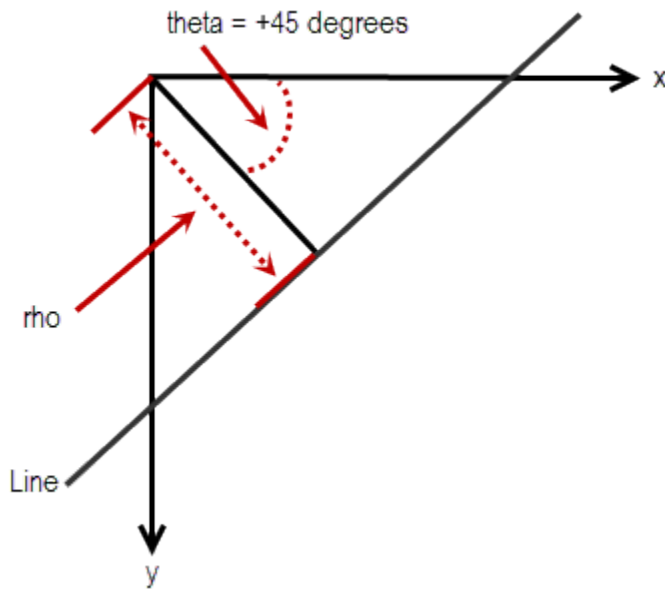
Distance from the origin to the line along a vector perpendicular to the line, returned as a numeric array of class `double`. For more information, see “Algorithms” on page 1-552.

Algorithms

The Standard Hough Transform (SHT) uses the parametric representation of a line:

$$rho = x \cdot \cos(theta) + y \cdot \sin(theta)$$

The variable rho is the distance from the origin to the line along a vector perpendicular to the line. $theta$ is the angle of the perpendicular projection from the origin to the line measured in degrees clockwise from the positive x -axis. The range of $theta$ is $-90^\circ \leq \theta < 90^\circ$. The angle of the line itself is $\theta + 90^\circ$, also measured clockwise with respect to the positive x -axis.



The SHT is a parameter space matrix whose rows and columns correspond to *rho* and *theta* values respectively. The elements in the SHT represent accumulator cells. Initially, the value in each cell is zero. Then, for every non-background point in the image, *rho* is calculated for every *theta*. *rho* is rounded off to the nearest allowed row in SHT. That accumulator cell is incremented. At the end of this procedure, a value of *Q* in *SHT(r,c)* means that *Q* points in the *xy*-plane lie on the line specified by *theta(c)* and *rho(r)*. Peak values in the SHT represent potential lines in the input image.

The Hough transform matrix, *H*, is *nrho-by-ntheta* where:

```
nrho = 2*(ceil(D/RhoResolution)) + 1, and
D = sqrt((numRowsInBW - 1)^2 + (numColsInBW - 1)^2).
rho values range from -diagonal to diagonal, where
diagonal = RhoResolution*ceil(D/RhoResolution).

ntheta = length(theta)
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- The optional parameters 'Theta' and 'RhoResolution' must be compile-time string constants.
- The optional Theta vector must have a bounded size.

See Also

See Also

houghlines | houghpeaks

Topics

“Detect Lines in Images Using Hough”

Introduced before R2006a

houghlines

Extract line segments based on Hough transform

Syntax

```
lines = houghlines(BW,theta,rho,peaks)
lines = houghlines( ___,Name,Value,...)
```

Description

`lines = houghlines(BW,theta,rho,peaks)` extracts line segments in the image `BW` associated with particular bins in a Hough transform. `theta` and `rho` are vectors returned by function `hough`. `peaks` is a matrix returned by the `houghpeaks` function that contains the row and column coordinates of the Hough transform bins to use in searching for line segments. The return value `lines` is a structure array whose length equals the number of merged line segments found.

`lines = houghlines(___,Name,Value,...)` extracts line segments in the image `BW`, where named parameters affect the operation.

Examples

Find Line Segments and Highlight longest segment

Read image into workspace.

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

Rotate the image.

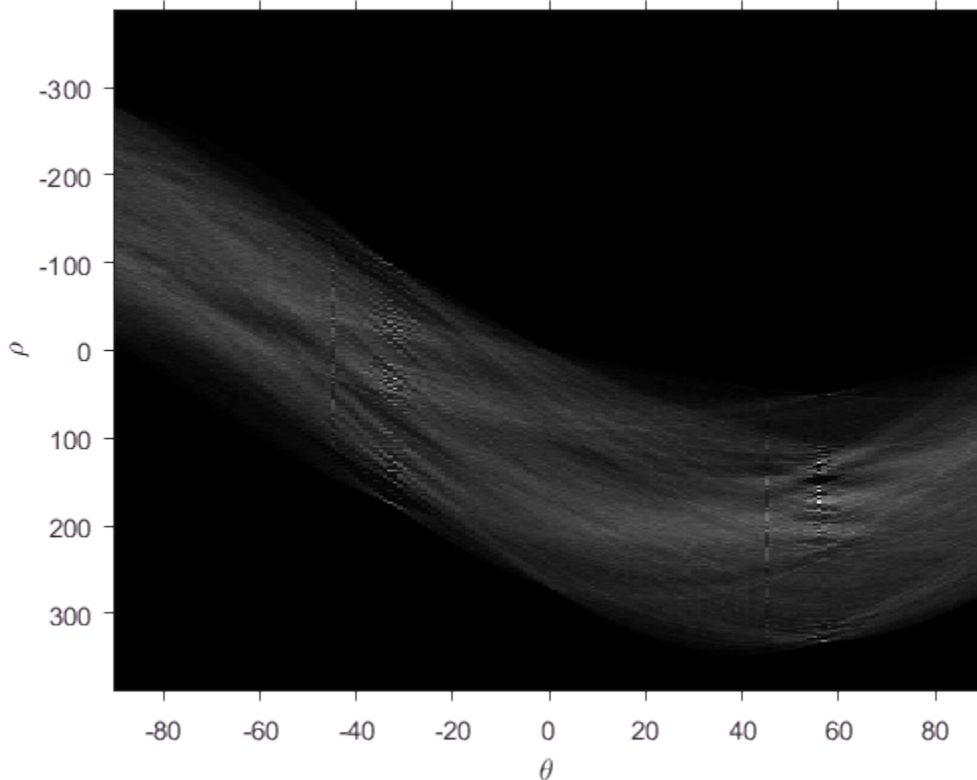
```
rotI = imrotate(I,33,'crop');
```

Create a binary image.

```
BW = edge(rotI,'canny');
```

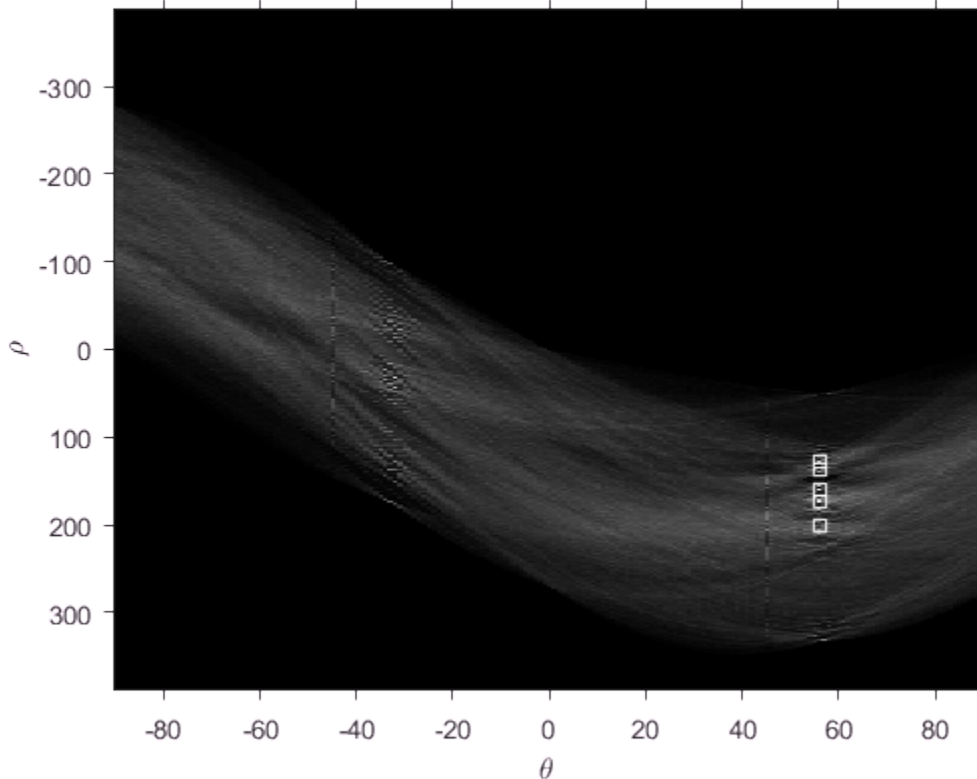
Create the Hough transform using the binary image.

```
[H,T,R] = hough(BW);
imshow(H,[], 'XData',T, 'YData',R,...
        'InitialMagnification','fit');
xlabel('\theta'), ylabel('\rho');
axis on, axis normal, hold on;
```



Find peaks in the Hough transform of the image.

```
P = houghpeaks(H,5, 'threshold',ceil(0.3*max(H(:))));
x = T(P(:,2)); y = R(P(:,1));
plot(x,y, 's', 'color', 'white');
```



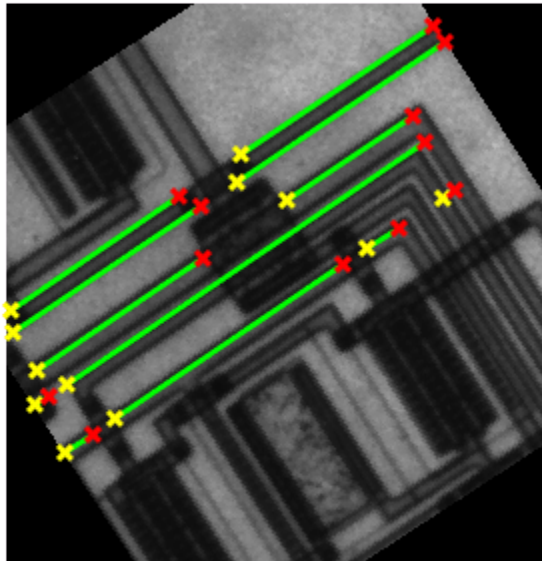
Find lines and plot them.

```
lines = houghlines(BW,T,R,P, 'FillGap',5, 'MinLength',7);
figure, imshow(rotI), hold on
max_len = 0;
for k = 1:length(lines)
    xy = [lines(k).point1; lines(k).point2];
    plot(xy(:,1),xy(:,2), 'LineWidth',2, 'Color', 'green');

    % 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');

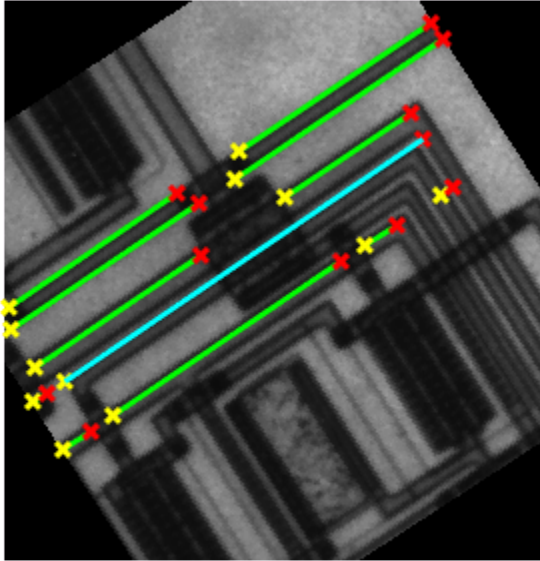
    % Determine the endpoints of the longest line segment
```

```
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 by coloring it cyan.

```
plot(xy_long(:,1),xy_long(:,2), 'LineWidth',2,'Color','cyan');
```



Input Arguments

BW — Binary image

real, 2-D, nonsparse logical or numeric array

Binary image, specified as a real, 2-D, nonsparse logical or numeric array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

theta — Line rotation angle in radians

real, 2-D, nonsparse numeric array

Line rotation angle in radians, specified as a real, 2-D, nonsparse logical or numeric array.

Data Types: double

rho — Distance from the coordinate origin

real, 2-D, nonsparse logical or numeric array

Distance from the coordinate origin, specified as a real, 2-D, nonsparse logical or numeric array. The coordinate origin is the top-left corner of the image (0,0).

Data Types: double

peaks — Row and column coordinates of Hough transform bins

real, nonsparse numeric matrix

Row and column coordinates of Hough transform bins, specified as a real, nonsparse numeric array.

Data Types: double

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `lines = houghlines(BW,T,R,P,'FillGap',5,'MinLength',7);`

'FillGap' — Distance between two line segments associated with the same Hough transform bin

20 (default) | positive real scalar

Distance between two line segments associated with the same Hough transform bin, specified as a positive real scalar. When the distance between the line segments is less than the value specified, the `houghlines` function merges the line segments into a single line segment.

Data Types: double

'MinLength' — Minimum line length

40 (default) | positive real scalar

Minimum line length, specified as a positive real scalar. `houghlines` discards lines that are shorter than the value specified.

Data Types: double

Output Arguments

lines — Lines found

structure array

Lines found, returned as a structure array whose length equals the number of merged line segments found. Each element of the structure array has these fields:

Field	Description
point1	Two element vector [X Y] specifying the coordinates of the end-point of the line segment
point2	Two element vector [X Y] specifying the coordinates of the end-point of the line segment
theta	Angle in degrees of the Hough transform bin
rho	rho axis position of the Hough transform bin

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- The optional parameter names 'FillGap' and 'MinLength' must be compile-time constants. Their associated values need not be compile-time constants.

See Also

See Also

hough | houghpeaks

Introduced before R2006a

houghpeaks

Identify peaks in Hough transform

Syntax

```
peaks = houghpeaks(H,numpeaks)
peaks = houghpeaks( ____,Name,Value,...)
```

Description

`peaks = houghpeaks(H,numpeaks)` locates peaks in the Hough transform matrix, `H`, generated by the `hough` function. `numpeaks` specifies the maximum number of peaks to identify. The function returns `peaks` a matrix that holds the row and column coordinates of the peaks.

`peaks = houghpeaks(____,Name,Value,...)` locates peaks in the Hough transform matrix, where named parameters control aspects of the operation.

Examples

Locate and Display Peaks in Hough Transform of Rotated Image

Read image into workspace.

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

Create binary image.

```
BW = edge(imrotate(I,50,'crop'),'canny');
```

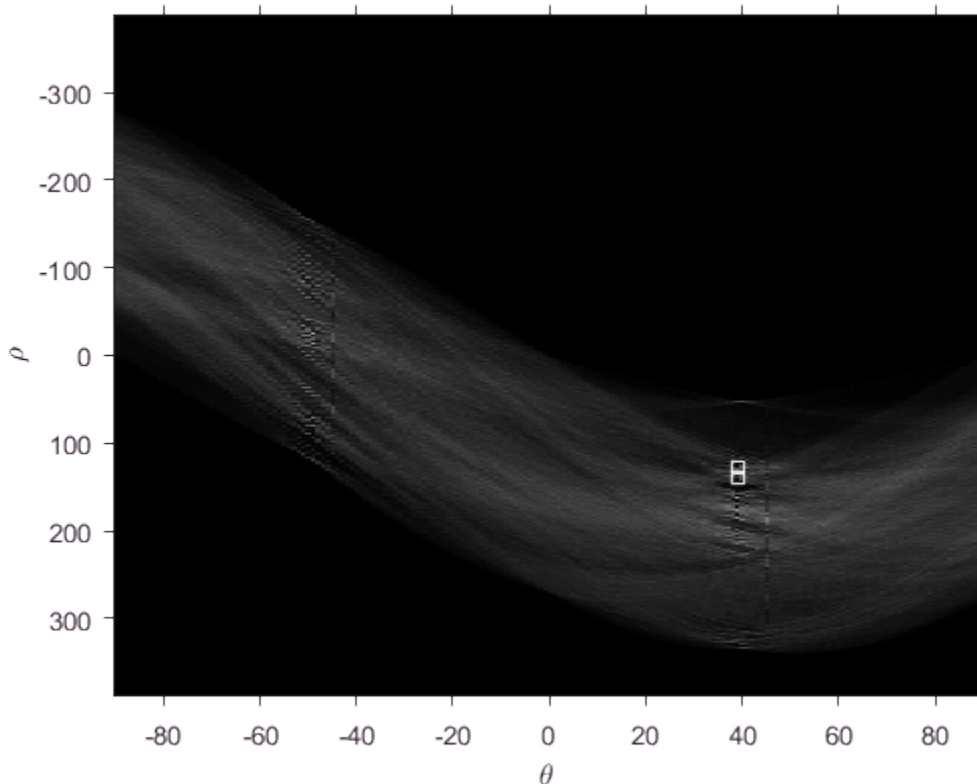
Create Hough transform of image.

```
[H,T,R] = hough(BW);
```

Find peaks in the Hough transform of the image and plot them.

```
P = houghpeaks(H,2);
imshow(H,[],'XData',T,'YData',R,'InitialMagnification','fit');
```

```
xlabel('\theta'), ylabel('\rho');  
axis on, axis normal, hold on;  
plot(T(P(:,2)),R(P(:,1)),'s','color','white');
```



Input Arguments

H — Hough transform matrix
numeric array

Hough transform matrix, specified as a numeric array of class `double`. The rows and columns correspond to `rho` and `theta` values. Use the `hough` function to create a Hough transform matrix.

Data Types: double

numpeaks — Maximum number of peaks to identify

1 (default) | positive integer scalar

Maximum number of peaks to identify, specified as a numeric scalar.

Data Types: double

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `P = houghpeaks(H,2,'Threshold',15);`

'Threshold' — Minimum value to be considered a peak

$0.5 * \max(H(:))$ (default) | nonnegative numeric scalar

Minimum value to be considered a peak, specified as a nonnegative numeric scalar. The value can be any value between 0 and Inf.

Data Types: double

'NHoodSize' — Size of suppression neighborhood

smallest odd values greater than or equal to $\text{size}(H)/50$ (default) | two-element vector of positive odd integers

Size of suppression neighborhood, specified as a two-element vector of positive odd integers. The *suppression neighborhood* is the neighborhood around each peak that is set to zero after the peak is identified.

Data Types: double

Output Arguments

peaks — Row and column coordinates of peaks found

Q -by-2 matrix

Row and column coordinates of peaks found, returned as a Q -by-2 matrix, where the value Q can range from 0 to `numpeaks`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- The optional parameter names `'Threshold'` and `'NHoodSize'` must be compile-time constants. Their associated values need not be compile-time constants.

See Also

See Also

`hough` | `houghlines`

Introduced before R2006a

iccfind

Search for ICC profiles

Syntax

```
P = iccfind(directory)
[P, descriptions] = iccfind(directory)
[...] = iccfind(directory, pattern)
```

Description

`P = iccfind(directory)` searches for all of the International Color Consortium (ICC) profiles in the directory specified by `directory`. The function returns `P`, a cell array of structures containing profile information.

`[P, descriptions] = iccfind(directory)` searches for all of the ICC profiles in the specified directory and returns `P`, a cell array of structures containing profile information, and `descriptions`, a cell array of character vectors, where each character vector describes the corresponding profile in `P`. Each character vector is the value of the `Description.String` field in the profile information structure.

`[...] = iccfind(directory, pattern)` returns all of the ICC profiles in the specified directory with the given `pattern` in their `Description.String` fields. `iccfind` performs case-insensitive pattern matching.

Note To improve performance, `iccfind` caches copies of the ICC profiles in memory. Adding or modifying profiles might not change the results of `iccfind`. To clear the cache, use the `clear functions` command.

Examples

Find International Color Consortium Profiles

Get all the International Color Consortium (ICC) profiles in the default system directory where profiles are stored.

```
profiles = iccfind(iccroot)
```

```
profiles =
```

```
2×1 cell array
```

```
 [1×1 struct]  
 [1×1 struct]
```

Get a listing of all the ICC profiles with descriptions.

```
[profiles, descriptions ] = iccfind(iccroot)
```

```
profiles =
```

```
2×1 cell array
```

```
 [1×1 struct]  
 [1×1 struct]
```

```
descriptions =
```

```
2×1 cell array
```

```
 'Agfa : Swop Standard      '  
 'sRGB IEC61966-2.1'
```

Find profiles whose descriptions contain the character vector 'rgb'.

```
[profiles, descriptions] = iccfind(iccroot, 'rgb')
```

```
profiles =
```

```
cell
```

```
 [1×1 struct]
```

```
descriptions =
```

```
cell
```

'sRGB IEC61966-2.1'

See Also

iccread | iccroot | iccwrite

Introduced before R2006a

iccread

Read ICC profile

Syntax

```
P = iccread(filename)
```

Description

`P = iccread(filename)` reads the International Color Consortium (ICC) color profile information from the file specified by `filename`. The file can be either an ICC profile file or a TIFF file containing an embedded ICC profile. To determine if a TIFF file contains an embedded ICC profile, use the `imfinfo` function to get information about the file and look for the `ICCProfileOffset` field. `iccread` looks for the file in the current directory, a directory on the MATLAB path, or in the directory returned by `iccroot`, in that order.

`iccread` returns the profile information in the structure `P`, a 1-by-1 structure array whose fields contain the data structures (called tags) defined in the ICC specification. `iccread` can read profiles that conform with either Version 2 (ICC.1:2001-04) or Version 4 (ICC.1:2001-12) of the ICC specification. For more information about ICC profiles, visit the ICC web site, www.color.org.

ICC profiles provide color management systems with the information necessary to convert color data between native device color spaces and device independent color spaces, called the Profile Connection Space (PCS). You can use the profile as the source or destination profile with the `makecform` function to compute color space transformations.

The number of fields in `P` depends on the profile class and the choices made by the profile creator. `iccread` returns all the tags for a given profile, both public and private. Private tags and certain public tags are left as encoded `uint8` data. The following table lists fields that are found in any profile structure generated by `iccread`, in the order they appear in the structure.

Field	Data Type	Description
Header	1-by-1 struct array	Profile header fields

Field	Data Type	Description
TagTable	n-by-3 cell array	Profile tag table
Copyright	Character vector	Profile copyright notice
Description	1-by-1 struct array	The <code>String</code> field in this structure contains a character vector describing the profile.
MediaWhitepoint	double array	XYZ tristimulus values of the device's media white point
PrivateTags	m-by-2 cell array	Contents of all the private tags or tags not defined in the ICC specifications. The tag signatures are in the first column, and the contents of the tags are in the second column. Note that <code>iccread</code> leaves the contents of these tags in unsigned 8-bit encoding.
Filename	Character vector	Name of the file containing the profile

Additionally, `P` might contain one or more of the following transforms:

- Three-component, matrix-based transform: A simple transform that is often used to transform between the RGB and XYZ color spaces. If this transform is present, `P` contains a field called `MatTRC`.
- N-component LUT-based transform: A transform that is used for transforming between color spaces that have a more complex relationship. This type of transform is found in any of the following fields in `P`:

AToB0	BToA0	Preview0
AToB1	BToA1	Preview1
AToB2	BToA2	Preview2
AToB3	BToA3	Gamut

Examples

Read ICC Profile for Typical PC Computer Monitor

Read the International Color Consortium (ICC) profile that describes a typical PC computer monitor.

```
P = iccread('sRGB.icm')
```

```
P = struct with fields:
```

```
    Header: [1×1 struct]
    TagTable: {17×3 cell}
    Copyright: 'Copyright (c) 1999 Hewlett-Packard Company'
    Description: [1×1 struct]
    MediaWhitePoint: [0.9505 1 1.0891]
    MediaBlackPoint: [0 0 0]
    DeviceMfgDesc: [1×1 struct]
    DeviceModelDesc: [1×1 struct]
    ViewingCondDesc: [1×1 struct]
    ViewingConditions: [1×1 struct]
        Luminance: [76.0365 80 87.1246]
    Measurement: [1×1 struct]
    Technology: 'Cathode Ray Tube Display'
    MatTRC: [1×1 struct]
    PrivateTags: {}
    Filename: 'sRGB.icm'
```

Determine the source color space. The profile header provides general information about the profile, such as its class, color space, and PCS.

```
P.Header.ColorSpace
```

```
ans =
'RGB'
```

See Also

`applycform` | `iccfind` | `iccroot` | `iccwrite` | `isicc` | `makecform`

Introduced before R2006a

iccroot

Find system default ICC profile repository

Syntax

```
rootdir = iccroot
```

Description

`rootdir = iccroot` returns the system directory containing ICC profiles. Additional profiles can be stored in other directories, but this is the default location used by the color management system.

Note Only Windows and Mac OS X platforms are supported.

Examples

View All Profiles in ICC Root Folder

View all the International Color Consortium (ICC) profiles in the default system ICC profile repository.

```
iccfind(iccroot)
```

```
ans =
```

```
    2×1 cell array
```

```
    [1×1 struct]
```

```
    [1×1 struct]
```

See Also

`iccfind` | `iccread` | `iccwrite`

Introduced before R2006a

iccwrite

Write ICC color profile to disk file

Syntax

```
P_new = iccwrite(P, filename)
```

Description

`P_new = iccwrite(P, filename)` writes the International Color Consortium (ICC) color profile data in structure `P` to the file specified by `filename`.

`P` is a structure representing an ICC profile in the data format returned by `iccread` and used by `makecform` and `applycform` to compute color-space transformations. `P` must contain all the tags and fields required by the ICC profile specification. Some fields may be inconsistent, however, because of interactive changes to the structure. For instance, the tag table may not be correct because tags may have been added, deleted, or modified since the tag table was constructed. `iccwrite` makes any necessary corrections to the profile structure before writing it to the file and returns this corrected structure in `P_new`.

Note Because some applications use the `Description.String` field in the ICC profile to present choices to users, the ICC recommends modifying the profile description in the ICC profile data before writing the data to a file. Each profile should have a unique profile description. For more information, see the example.

`iccwrite` can write the color profile data using either Version 2 (ICC.1:2001-04) or Version 4 (ICC.1:2001-12) of the ICC specification, depending on the value of the `Version` field in the file profile header. If any required fields are missing, `iccwrite` errors. For more information about ICC profiles, visit the ICC web site, www.color.org.

Note `iccwrite` does not perform automatic conversions from one version of the ICC specification to another. Such conversions have to be done manually, by adding fields or modifying fields. Use `isicc` to validate a profile.

Examples

Read a profile into the MATLAB workspace and export the profile data to a new file. The example changes the profile description in the profile data before writing the data to a file.

```
P = iccread('monitor.icm');  
  
P.Description.String  
  
ans =  
  
sgC4_050102_d50.pf  
  
P.Description.String = 'my new description';  
  
pmon = iccwrite(P, 'monitor2.icm');
```

See Also

[applycform](#) | [isicc](#) | [iccread](#) | [makecform](#)

Introduced before R2006a

idct2

2-D inverse discrete cosine transform

Syntax

```
B = idct2(A)
B = idct2(A,m,n)
B = idct2(A,[m n])
```

Description

`B = idct2(A)` returns the two-dimensional inverse discrete cosine transform (DCT) of `A`.

`B = idct2(A,m,n)` pads `A` with 0's to size `m`-by-`n` before transforming. If `[m n] < size(A)`, `idct2` crops `A` before transforming.

`B = idct2(A,[m n])` same as above.

For any `A`, `idct2(dct2(A))` equals `A` to within roundoff error.

Class Support

The input matrix `A` can be of class `double` or of any numeric class. The output matrix `B` is of class `double`.

Examples

Remove High Frequencies in Image using DCT

This example shows how to remove high frequencies from an image using the two-dimensional discrete cosine transfer (DCT).

Read an image into the workspace, then convert the image to grayscale.

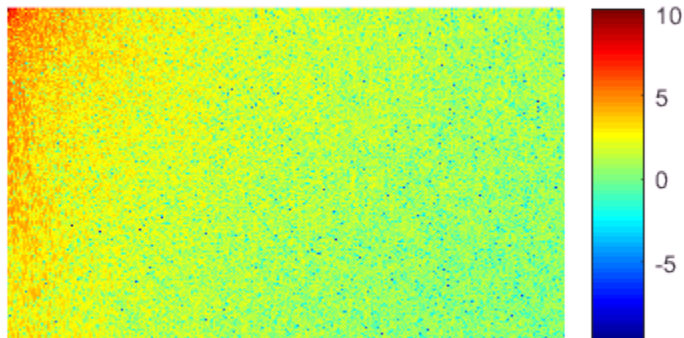
```
RGB = imread('autumn.tif');  
I = rgb2gray(RGB);
```

Perform a 2-D DCT of the grayscale image using the `dct2` function.

```
J = dct2(I);
```

Display the transformed image using a logarithmic scale. Notice that most of the energy is in the upper left corner.

```
figure  
imshow(log(abs(J)),[])  
colormap(gca, jet(64))  
colorbar
```



Set values less than magnitude 10 in the DCT matrix to zero.

```
J(abs(J) < 10) = 0;
```

Reconstruct the image using the inverse DCT function `idct2`.

```
K = idct2(J);
```


Display the original grayscale image alongside the processed image.

```
figure
imshowpair(I,K,'montage')
title('Original Grayscale Image (Left) and Processed Image (Right)');
```

Original Grayscale Image (Left) and Processed Image (Right)



Algorithms

idct2 computes the two-dimensional inverse DCT using:

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

where

$$\alpha_p = \begin{cases} \frac{1}{\sqrt{M}}, & p=0 \\ \sqrt{\frac{2}{M}}, & 1 \leq p \leq M-1 \end{cases}$$

and

$$\alpha_q = \begin{cases} \frac{1}{\sqrt{N}}, & q = 0 \\ \sqrt{\frac{2}{N}}, & 1 \leq q \leq N-1 \end{cases} .$$

References

- [1] Jain, A. K., *Fundamentals of Digital Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1989, pp. 150-153.
- [2] Pennebaker, W. B., and J. L. Mitchell, *JPEG: Still Image Data Compression Standard*, New York, Van Nostrand Reinhold, 1993.

See Also

dct2 | dctmtx | fft2 | ifft2

Introduced before R2006a

ifanbeam

Inverse fan-beam transform

Syntax

```
I = ifanbeam(F,D)
I = ifanbeam(...,param1,val1,param2,val2,...)
[I,H] = ifanbeam(...)
```

Description

`I = ifanbeam(F,D)` reconstructs the image `I` from projection data in the two-dimensional array `F`. Each column of `F` contains fan-beam projection data at one rotation angle. `ifanbeam` assumes that the center of rotation is the center point of the projections, which is defined as `ceil(size(F,1)/2)`.

The fan-beam spread angles are assumed to be the same increments as the input rotation angles split equally on either side of zero. The input rotation angles are assumed to be stepped in equal increments to cover `[0:359]` degrees.

`D` is the distance from the fan-beam vertex to the center of rotation.

`I = ifanbeam(...,param1,val1,param2,val2,...)` specifies parameters that control various aspects of the `ifanbeam` reconstruction, described in the following table. Parameter names can be abbreviated, and case does not matter.

Parameter	Description
'FanCoverage'	Range through which the beams are rotated, specified as one of the following: 'cycle' — Rotate through the full range <code>[0,360)</code> . This is the default. 'minimal' — Rotate the minimum range necessary to represent the object.

Parameter	Description
'FanRotationIncrement'	Positive real scalar specifying the increment of the rotation angle of the fan-beam projections, measured in degrees. See <code>fanbeam</code> for details.
'FanSensorGeometry'	Positioning of sensors, specified as one of the following: 'arc' — Sensors are spaced equally along a circular arc at distance D from the center of rotation. Default value is 'arc' 'line' — Sensors are spaced equally along a line, the closest point of which is distance D from the center of rotation. See <code>fanbeam</code> for details.
'FanSensorSpacing'	Positive real scalar specifying the spacing of the fan-beam sensors. Interpretation of the value depends on the setting of 'FanSensorGeometry'. If 'FanSensorGeometry' is set to 'arc' (the default), the value defines the angular spacing in degrees. Default value is 1. If 'FanSensorGeometry' is 'line', the value specifies the linear spacing. Default value is 1. See <code>fanbeam</code> for details.
'Filter'	Name of filter, specified as a character vector. See <code>iradon</code> for details.
'FrequencyScaling'	Scalar in the range (0,1] that modifies the filter by rescaling its frequency axis. See <code>iradon</code> for details.

Parameter	Description
'Interpolation'	Type of interpolation used between the parallel-beam and fan-beam data, specified as one of the following: 'nearest' — Nearest-neighbor 'linear' — Linear (the default) 'spline' — Piecewise cubic spline 'pchip' — Piecewise cubic Hermite (PCHIP) 'v5cubic' — The cubic interpolation from MATLAB 5
'OutputSize'	Positive scalar specifying the number of rows and columns in the reconstructed image. If 'OutputSize' is not specified, ifanbeam determines the size automatically. If you specify 'OutputSize', ifanbeam reconstructs a smaller or larger portion of the image, but does not change the scaling of the data. <hr/> Note: If the projections were calculated with the fanbeam function, the reconstructed image might not be the same size as the original image.

[I,H] = ifanbeam(...) returns the frequency response of the filter in the vector H.

Notes

ifanbeam converts the fan-beam data to parallel beam projections and then uses the filtered back projection algorithm to perform the inverse Radon transform. The filter is designed directly in the frequency domain and then multiplied by the FFT of the projections. The projections are zero-padded to a power of 2 before filtering to prevent spatial domain aliasing and to speed up the FFT.

Class Support

The input arguments, `F` and `D`, can be `double` or `single`. All other numeric input arguments must be `double`. The output arguments are `double`.

Examples

Recreate Image from Fan-beam Transformation

Create a sample image. The `phantom` function creates a phantom head image.

```
ph = phantom(128);
```

Create a fan-beam transformation of the phantom head image.

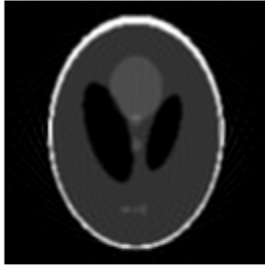
```
d = 100;  
F = fanbeam(ph,d);
```

Reconstitute the phantom head image from the fan-beam representation. Display the original image and the reconstituted image.

```
I = ifanbeam(F,d);  
imshow(ph)
```



```
figure  
imshow(I);
```



Generate Fan-beam with Fancoverage Set to Minimal

Create a sample image. The phantom function creates a phantom head image.

```
ph = phantom(128);
```

Create a radon transformation of the image.

```
P = radon(ph);
```

Convert the transformation from parallel beam projection to fan-beam projection.

```
[F,obeta,otheta] = para2fan(P,100,...
    'FanSensorSpacing',0.5,...
    'FanCoverage','minimal',...
    'FanRotationIncrement',1);
```

Reconstitute the image from fan-beam data.

```
phReconstructed = ifanbeam(F,100,...
    'FanSensorSpacing',0.5,...
    'Filter','Shepp-Logan',...
    'OutputSize',128,...
    'FanCoverage','minimal',...
    'FanRotationIncrement',1);
```

Display the original and the transformed image.

`imshow(ph)`



`figure`
`imshow(phReconstructed)`



References

- [1] Kak, A. C., and M. Slaney, *Principles of Computerized Tomographic Imaging*, New York, NY, IEEE Press, 1988.

See Also

fan2para | iradon | fanbeam | para2fan | phantom | radon

Introduced before R2006a

im2bw

Convert image to binary image, based on threshold

Compatibility

`im2bw` is not recommended. Use `imbinarize` instead.

Syntax

```
BW = im2bw(I, level)
BW = im2bw(X, map, level)
BW = im2bw(RGB, level)
```

Description

`BW = im2bw(I, level)` converts the grayscale image `I` to a binary image. The output image `BW` replaces all pixels in the input image with luminance greater than `level` with the value 1 (white) and replaces all other pixels with the value 0 (black). Specify `level` in the range [0,1]. This range is relative to the signal levels possible for the image's class. Therefore, a `level` value of 0.5 is midway between black and white, regardless of class. To compute the `level` argument, you can use the function `graythresh`. If you do not specify `level`, `im2bw` uses the value 0.5.

`BW = im2bw(X, map, level)` converts the indexed image `X` with colormap `map` to a binary image.

`BW = im2bw(RGB, level)` converts the truecolor image `RGB` to a binary image.

If the input image is not a grayscale image, `im2bw` converts the input image to grayscale, and then converts this grayscale image to binary by thresholding.

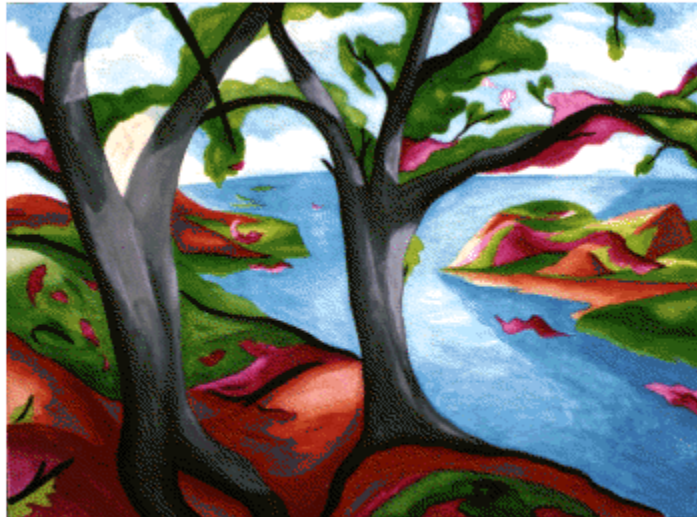
Class Support

The input image can be of class `uint8`, `uint16`, `single`, `int16`, or `double`, and must be nonsparse. The output image `BW` is of class `logical`. `I` and `X` must be 2-D. RGB images are M-by-N-by-3.

Examples

Convert an Indexed Image To a Binary Image

```
load trees
BW = im2bw(X,map,0.4);
imshow(X,map), figure, imshow(BW)
```





See Also

[graythresh](#) | [ind2gray](#) | [rgb2gray](#)

Introduced before R2006a

im2col

Rearrange image blocks into columns

Syntax

```
B = im2col(A,[m n],block_type)
B = im2col(A,'indexed',...)
```

Description

`B = im2col(A,[m n],block_type)` rearranges image blocks into columns. `block_type` is a character vector that can have one of the following values. The default value is enclosed in braces (`{}`).

Value	Description
'distinct'	Rearranges each <i>distinct</i> m -by- n block in the image A into a column of B . <code>im2col</code> pads A with 0's, if necessary, so its size is an integer multiple of m -by- n . If $A = [A_{11} \ A_{12}; \ A_{21} \ A_{22}]$, where each A_{ij} is m -by- n , then $B = [A_{11}(:) \ A_{12}(:) \ A_{21}(:) \ A_{22}(:)]$.
{'sliding'}	Converts each <i>sliding</i> m -by- n block of A into a column of B , with no zero padding. B has $m*n$ rows and contains as many columns as there are m -by- n neighborhoods of A . If the size of A is $[mm \ nn]$, then the size of B is $(m*n)$ -by- $((mm-m+1)*(nn-n+1))$.

For the sliding block case, each column of B contains the neighborhoods of A reshaped as `NHOOD(:)` where `NHOOD` is a matrix containing an m -by- n neighborhood of A . `im2col` orders the columns of B so that they can be reshaped to form a matrix in the normal way. For Examples, suppose you use a function, such as `sum(B)`, that returns a scalar for each column of B . You can directly store the result in a matrix of size $(mm-m+1)$ -by- $(nn-n+1)$, using these calls.

```
B = im2col(A,[m n],'sliding');
C = reshape(sum(B),mm-m+1,nn-n+1);
```

`B = im2col(A,'indexed',...)` processes A as an indexed image, padding with 0's if the class of A is `uint8` or `uint16`, or 1's if the class of A is `double`.

Class Support

The input image **A** can be numeric or logical. The output matrix **B** is of the same class as the input image.

Examples

Calculate Local Mean Using [2 2] Neighborhood

Create a matrix.

```
A = reshape(linspace(0,1,16),[4 4])'
```

A =

0	0.0667	0.1333	0.2000
0.2667	0.3333	0.4000	0.4667
0.5333	0.6000	0.6667	0.7333
0.8000	0.8667	0.9333	1.0000

Rearrange the values into a column-wise arrangement.

```
B = im2col(A,[2 2])
```

B =

0	0.2667	0.5333	0.0667	0.3333	0.6000	0.1333	0.4000	0.6667	0.9333
0.2667	0.5333	0.8000	0.3333	0.6000	0.8667	0.4000	0.6667	0.2000	0.7333
0.0667	0.3333	0.6000	0.1333	0.4000	0.6667	0.2000	0.4667	0.8000	0.9333
0.3333	0.6000	0.8667	0.4000	0.6667	0.9333	0.4667	0.7333	1.0000	0.0000

Calculate the mean.

```
M = mean(B)
```

M =

0.1667	0.4333	0.7000	0.2333	0.5000	0.7667	0.3000	0.5667	0.8000	0.9333
--------	--------	--------	--------	--------	--------	--------	--------	--------	--------

Rearrange the values back into their original, row-wise orientation.

```
newA = col2im(M,[1 1],[3 3])
```

```
newA =
```

```
  0.1667  0.2333  0.3000  
  0.4333  0.5000  0.5667  
  0.7000  0.7667  0.8333
```

See Also

[blockproc](#) | [col2im](#) | [colfilt](#) | [nlfilter](#)

Introduced before R2006a

im2int16

Convert image to 16-bit signed integers

Syntax

```
I2 = im2int16(I)
RGB2 = im2int16(RGB)
I = im2int16(BW)
gpuarrayB = im2int16(gpuarrayA, ___ )
```

Description

`I2 = im2int16(I)` converts the intensity image `I` to `int16`, rescaling the data if necessary. If the input image is of class `int16`, the output image is identical to it.

`RGB2 = im2int16(RGB)` converts the truecolor image `RGB` to `int16`, rescaling the data if necessary.

`I = im2int16(BW)` converts the binary image `BW` to an `int16` intensity image, changing false-valued elements to -32768 and true-valued elements to 32767.

`gpuarrayB = im2int16(gpuarrayA, ___)` performs the conversion on a GPU. The input image and output image are `gpuArrays`. This syntax requires the Parallel Computing Toolbox.

Class Support

Intensity and truecolor images can be `uint8`, `uint16`, `int16`, `single`, `double`, or `logical`. Binary images must be `logical`. The output image is `int16`.

Intensity and truecolor `gpuArray` images can be `uint8`, `uint16`, `int16`, `logical`, `single`, or `double`. Binary `gpuArray` images must be `logical`. The output `gpuArray` image is `int16`.

Examples

Convert Array from double to int16

Create an array of class `double`.

```
I = reshape(linspace(0,1,20),[5 4])
```

I =

0	0.2632	0.5263	0.7895
0.0526	0.3158	0.5789	0.8421
0.1053	0.3684	0.6316	0.8947
0.1579	0.4211	0.6842	0.9474
0.2105	0.4737	0.7368	1.0000

Convert the array to class `int16`.

```
I2 = im2int16(I)
```

I2 = 5×4 *int16 matrix*

-32768	-15522	1724	18970
-29319	-12073	5173	22419
-25870	-8624	8623	25869
-22420	-5174	12072	29318
-18971	-1725	15521	32767

Convert Array from double to int16 on a GPU

Create array of class `double`.

```
I1 = gpuArray(reshape(linspace(0,1,20),[5 4]))
```

Convert array to `uint8`.

```
I2 = im2int16(I1)
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

See Also

`im2double` | `im2single` | `im2uint8` | `im2uint16` | `int16` | `gpuArray`

Introduced before R2006a

im2java2d

Convert image to Java buffered image

Syntax

```
jimage = im2java2d(I)  
jimage = im2java2d(X,MAP)
```

Description

`jimage = im2java2d(I)` converts the image `I` to an instance of the Java image class `java.awt.image.BufferedImage`. The image `I` can be an intensity (grayscale), RGB, or binary image.

`jimage = im2java2d(X,MAP)` converts the indexed image `X` with colormap `MAP` to an instance of the Java class `java.awt.image.BufferedImage`.

Note The `im2java2d` function works with the Java 2D API. The `im2java` function works with the Java Abstract Windowing Toolkit (AWT).

Class Support

Intensity, indexed, and RGB input images can be of class `uint8`, `uint16`, or `double`. Binary input images must be of class `logical`.

Examples

Read an image into the MATLAB workspace and then use `im2java2d` to convert it into an instance of the Java class `java.awt.image.BufferedImage`.

```
I = imread('moon.tif');  
javaImage = im2java2d(I);  
frame = javax.swing.JFrame;
```

```
icon = javax.swing.ImageIcon(javaImage);  
label = javax.swing.JLabel(icon);  
frame.getContentPane().add(label);  
frame.pack  
frame.show
```

Introduced before R2006a

im2single

Convert image to single precision

Syntax

```
I2 = im2single(I)
RGB2 = im2single(RGB)
I = im2single(BW)
X2 = im2single(X, 'indexed')
gpuarrayB = im2single(gpuarrayA, ___)
```

Description

`I2 = im2single(I)` converts the intensity image `I` to single, rescaling the data if necessary. If the input image is of class `single`, the output image is identical to it.

`RGB2 = im2single(RGB)` converts the truecolor image `RGB` to single, rescaling the data if necessary.

`I = im2single(BW)` converts the binary image `BW` to a single-precision intensity image.

`X2 = im2single(X, 'indexed')` converts the indexed image `X` to single precision, offsetting the data if necessary.

`gpuarrayB = im2single(gpuarrayA, ___)` performs the conversion on a GPU. The input image and the output image are `gpuArrays`. This syntax requires the Parallel Computing Toolbox.

Class Support

Intensity and truecolor images can be `uint8`, `uint16`, `int16`, `logical`, `single`, or `double`. Indexed images can be `uint8`, `uint16`, `double` or `logical`. Binary input images must be `logical`. The output image is `single`.

If input `gpuArray` `gpuarrayA` is an intensity and truecolor image, it can be `uint8`, `uint16`, `int16`, `logical`, `single`, or `double`. If `gpuarrayA` is an indexed image, it

can be `uint8`, `uint16`, `double` or `logical`. If `gpuarrayA` is a binary image, it must be `logical`. The output `gpuArray` image is `single`.

Examples

Convert Array to Class Single

This example shows how to convert an array of class `uint8` into class `single` .

Create a numeric array of class `uint8` .

```
I = reshape(uint8(linspace(1,255,25)),[5 5])
```

```
I = 5×5 uint8 matrix
```

```
    1    54   107   160   213
   12    65   117   170   223
   22    75   128   181   234
   33    86   139   192   244
   43    96   149   202   255
```

Convert the array to class `single` .

```
I2 = im2single(I)
```

```
I2 = 5×5 single matrix
```

```
    0.0039    0.2118    0.4196    0.6275    0.8353
    0.0471    0.2549    0.4588    0.6667    0.8745
    0.0863    0.2941    0.5020    0.7098    0.9176
    0.1294    0.3373    0.5451    0.7529    0.9569
    0.1686    0.3765    0.5843    0.7922    1.0000
```

Convert Array to Class Single on GPU

Create an array of class `uint8` on the GPU by creating a `gpuArray` object.

```
I = gpuArray(reshape(uint8(linspace(1,255,25)),[5 5]))
```

```
I =
```

```
1   54  107  160  213
12  65  117  170  223
22  75  128  181  234
33  86  139  192  244
43  96  149  202  255
```

Convert the array from class `uint8` to class `single` on the GPU. You can pass `im2single` a `gpuArray` object.

```
I2 = im2single(I)
```

```
I2 =
```

```
0.0039    0.2118    0.4196    0.6275    0.8353
0.0471    0.2549    0.4588    0.6667    0.8745
0.0863    0.2941    0.5020    0.7098    0.9176
0.1294    0.3373    0.5451    0.7529    0.9569
0.1686    0.3765    0.5843    0.7922    1.0000
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.

See Also

`im2double` | `im2int16` | `im2uint8` | `im2uint16` | `single` | `gpuArray`

Introduced before R2006a

im2uint16

Convert image to 16-bit unsigned integers

Syntax

```
I2 = im2uint16(I)
RGB2 = im2uint16(RGB)
I = im2uint16(BW)
X2 = im2uint16(X, 'indexed')
gpuarrayB = im2uint16(gpuarrayA, ___)
```

Description

`I2 = im2uint16(I)` converts the intensity image `I` to `uint16`, rescaling the data if necessary. If the input image is of class `uint16`, the output image is identical to it.

`RGB2 = im2uint16(RGB)` converts the truecolor image `RGB` to `uint16`, rescaling the data if necessary.

`I = im2uint16(BW)` converts the binary image `BW` to a `uint16` intensity image, changing 1-valued elements to 65535.

`X2 = im2uint16(X, 'indexed')` converts the indexed image `X` to `uint16`, offsetting the data if necessary. If `X` is of class `double`, `max(X(:))` must be 65536 or less.

`gpuarrayB = im2uint16(gpuarrayA, ___)` performs the conversion on a GPU. The input image and the output image are `gpuArrays`. This syntax requires the Parallel Computing Toolbox.

Class Support

Intensity and truecolor images can be `uint8`, `uint16`, `double`, `logical`, `single`, or `int16`. Indexed images can be `uint8`, `uint16`, `double`, or `logical`. Binary input images must be `logical`. The output image is `uint16`.

If the input `gpuArray` `gpuarrayA` is an intensity or truecolor image, it can be `uint8`, `uint16`, `int16`, `logical`, `single`, or `double`. If `gpuarrayA` is an indexed image, it can be `uint8`, `uint16`, `double` or `logical`. If `gpuarrayA` is a binary image, it must be `logical`. The output `gpuArray` image is `uint16`.

Examples

Convert Array from double to uint16

Create an array of class `double`.

```
I = reshape(linspace(0,1,20),[5 4])
```

```
I =
```

```

    0    0.2632    0.5263    0.7895
  0.0526    0.3158    0.5789    0.8421
  0.1053    0.3684    0.6316    0.8947
  0.1579    0.4211    0.6842    0.9474
  0.2105    0.4737    0.7368    1.0000
```

Convert the array to class `uint16`.

```
I2 = im2uint16(I)
```

```
I2 = 5×4 uint16 matrix
```

```

    0   17246   34492   51738
  3449   20695   37941   55187
  6898   24144   41391   58637
 10348   27594   44840   62086
 13797   31043   48289   65535
```

Convert Array from double to uint16 on a GPU

Create array of class `double`.

```
I1 = gpuArray(reshape(linspace(0,1,20),[5 4]))
```

Convert array to `uint16`.

```
I2 = im2uint16(I1)
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

See Also

`im2uint8` | `uint8` | `uint16` | `double` | `im2double` | `imapprox` | `gpuArray`

Introduced before R2006a

im2uint8

Convert image to 8-bit unsigned integers

Syntax

```
I2 = im2uint8(I)
RGB2 = im2uint8(RGB)
I = im2uint8(BW)
X2 = im2uint8(X, 'indexed')
gpuarrayB = im2uint8(gpuarrayA, ___ )
```

Description

`I2 = im2uint8(I)` converts the grayscale image `I` to `uint8`. If the input image is of class `uint8`, the output image is identical to the input image. If the input image is not `uint8`, `im2uint8` returns the equivalent image of class `uint8`, rescaling or offsetting the data as necessary.

`RGB2 = im2uint8(RGB)` converts the truecolor image `RGB` to `uint8`, rescaling the data if necessary.

`I = im2uint8(BW)` converts the binary image `BW` to a `uint8` grayscale image, changing 1-valued elements to 255.

`X2 = im2uint8(X, 'indexed')` converts the indexed image `X` to `uint8`, offsetting the data if necessary. Note that it is not always possible to convert an indexed image to `uint8`. If `X` is of class `double`, the maximum value of `X` must be 256 or less. If `X` is of class `uint16`, the maximum value of `X` must be 255 or less.

`gpuarrayB = im2uint8(gpuarrayA, ___)` performs the conversion on a GPU. The input image, `gpuarrayA`, can be a grayscale, truecolor, binary, or indexed `gpuArray` image. The output image is a `gpuArray`. This syntax requires the Parallel Computing Toolbox.

Examples

Convert uint16 Array to uint8 Array

Create an array of class `uint16`.

```
I = reshape(uint16(linspace(0,65535,25)),[5 5])
```

```
I = 5×5 uint16 matrix
```

```
      0   13653   27306   40959   54613
    2731   16384   30037   43690   57343
    5461   19114   32768   46421   60074
    8192   21845   35498   49151   62804
   10923   24576   38229   51882   65535
```

Convert the array to class `uint8`.

```
I2 = im2uint8(I)
```

```
I2 = 5×5 uint8 matrix
```

```
      0    53   106   159   213
     11    64   117   170   223
     21    74   128   181   234
     32    85   138   191   244
     43    96   149   202   255
```

Convert uint16 Array to uint8 on a GPU

Create array of class `uint16`.

```
I1 = gpuArray(reshape(uint16(linspace(0,65535,25)),[5 5]))
```

Convert array to `uint8`.

```
I2 = im2uint8(I1);
```

Input Arguments

I — Input grayscale image

real, nonsparse, numeric array

Input grayscale image, specified as a real, nonsparse, numeric array.

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

Data Types: `single` | `double` | `int16` | `uint8`

RGB — Input truecolor image

real, nonsparse, numeric array

Truecolor image, specified as a real, nonsparse, numeric array.

Example: `RGB = imread('peppers.png');`

Data Types: `single` | `double` | `int16` | `uint8`

BW — Binary image

real, nonsparse, logical array

Binary image, specified as a real, nonsparse, logical array.

Example: `BW = imread('text.png');`

Data Types: `logical`

X — Indexed image

real, nonsparse, numeric array

Indexed image, specified as a real, nonsparse, numeric array.

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

Data Types: `double` | `uint8` | `uint16`

gpuarrayA — Input image

`gpuArray`

Input image, specified as a `gpuArray`.

Example: `I = gpuArray(imread('cameraman.tif'));`

Output Arguments

I2 — Grayscale image

`uint8` array

Grayscale image, returned as a `uint8` array.

RGB2 — Truecolor image

numeric array

Truecolor image, returned as a `uint8` array.

X2 — Output indexed image

numeric array

Output indexed image, returned as a `uint8` numeric array.

gpuarrayB — Output image

`gpuArray`

Output image, returned as a `gpuArray`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

See Also

See Also

`gpuArray` | `im2double` | `im2int16` | `im2single` | `im2uint16` | `uint8`

Introduced before R2006a

imabsdiff

Absolute difference of two images

Syntax

```
Z = imabsdiff(X,Y)
gpuarrayZ = imabsdiff(gpuarrayX,gpuarrayY)
```

Description

`Z = imabsdiff(X,Y)` subtracts each element in array `Y` from the corresponding element in array `X` and returns the absolute difference in the corresponding element of the output array `Z`.

`gpuarrayZ = imabsdiff(gpuarrayX,gpuarrayY)` performs the computation on a GPU, if at least one of the inputs is a `gpuArray`. The output image is a `gpuArray`. This syntax requires the Parallel Computing Toolbox.

Examples

Display Absolute Difference between Filtered image and Original

Read image into workspace.

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

Filter the image.

```
J = uint8(filter2(fspecial('gaussian'), I));
```

Calculate the absolute difference of the two images.

```
K = imabsdiff(I,J);
```

Display the absolute difference image.

```
figure
```



```
imshow(K, [])
```



Display Absolute Difference Between Filtered Image and Original on GPU

Read image and convert it to a GPUarray.

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

Filter the image, performing the operation on a GPU.

```
J = imfilter(I, fspecial('gaussian'));
```

Calculate the absolute difference between the filtered image and original image.

```
K = imabsdiff(I,J);
```

Display the absolute difference image.

```
figure
```

```
imshow(K, [])
```

Input Arguments

X — Input image

real, nonsparse numeric array

Input image, specified as a real, nonsparse numeric array. X must be the same size and class as Y.

If X is of class **double**, use the expression `abs(X-Y)` instead of this function. If X is of class **logical**, use the expression `XOR(X,Y)` instead of this function.

Example: `Z = imabsdiff(X,Y);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

Y — Input image

real, nonsparse numeric array

Input image, specified as a real, nonsparse numeric array. Y must be the same size and class as X.

If Y is of class **double**, use the expression `abs(X-Y)` instead of this function. If Y is of class **logical**, use the expression `XOR(X,Y)` instead of this function.

Example: `Z = imabsdiff(X,Y);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

gpuarrayX — Input image

GPUarray

Input image, specified as a GPUarray.

Example: `gpuarrayZ = imabsdiff(gpuarrayX,gpuarrayY);`

gpuarrayY — Input image

GPUarray

Input image, specified as a GPUarray.

Example: `gpuarrayZ = imabsdiff(gpuarrayX,gpuarrayY);`

Output Arguments

Z — Difference image

real, nonsparse, numeric array

Difference image, returned as a real, nonsparse, numeric array. **Z** has the same class and size as **X** and **Y**. If **X** and **Y** are integer arrays, **imabsdiff** truncates elements in the output that exceed the range of the integer type.

gpuarrayZ — Difference image

gpuArray

Difference image, returned as a **gpuArray**. **gpuarrayZ** has the same class and size as **gpuarrayX** and **gpuarrayY**.

Tips

- When **X** and **Y** are of class **uint8**, **int16**, or **single**, **imabsdiff** might take advantage of hardware optimization to run faster.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.

See Also

See Also

`gpuArray` | `imadd` | `imcomplement` | `imdivide` | `imlincomb` | `immultiply` | `imsubtract`

Introduced before R2006a

imadd

Add two images or add constant to image

Syntax

```
Z = imadd(X,Y)
```

Description

`Z = imadd(X,Y)` adds each element in array `X` with the corresponding element in array `Y` and returns the sum in the corresponding element of the output array `Z`. `X` and `Y` are real, nonsparse numeric arrays with the same size and class, or `Y` is a scalar double. `Z` has the same size and class as `X`, unless `X` is logical, in which case `Z` is double.

If `X` and `Y` are integer arrays, elements in the output that exceed the range of the integer type are truncated, and fractional values are rounded.

Examples

Add Two uint8 Arrays

This example shows how to add two `uint8` arrays with truncation for values that exceed 255.

```
X = uint8([ 255 0 75; 44 225 100]);  
Y = uint8([ 50 50 50; 50 50 50 ]);  
Z = imadd(X,Y)
```

```
Z = 2×3 uint8 matrix
```

```
    255     50    125  
     94    255    150
```

Add Two Images and Specify Output Class

Read two grayscale `uint8` images into the workspace.

```
I = imread('rice.png');  
J = imread('cameraman.tif');
```

Add the images. Specify the output as type `uint16` to avoid truncating the result.

```
K = imadd(I,J,'uint16');
```

Display the result.

```
imshow(K,[])
```



Add a Constant to an Image

Read an image into the workspace.

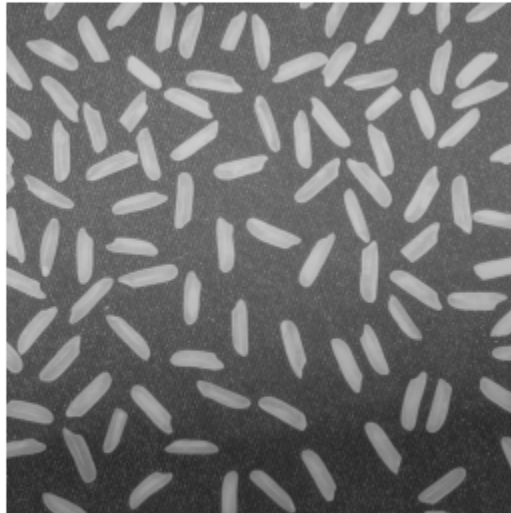
```
I = imread('rice.png');
```

Add a constant to the image.

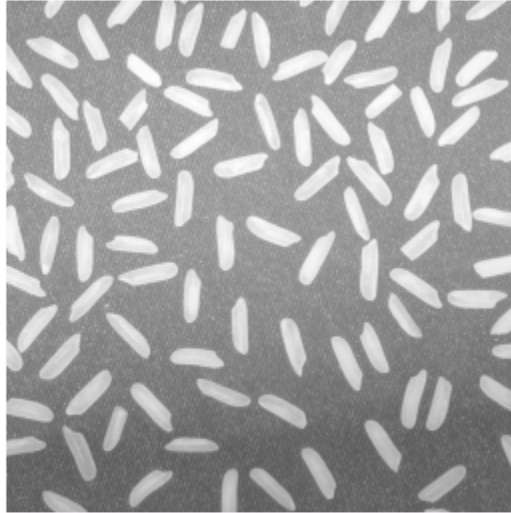
```
J = imadd(I,50);
```

Display the original image and the result.

```
imshow(I)
```



```
figure  
imshow(J)
```



See Also

`imabsdiff` | `imcomplement` | `imdivide` | `imlincomb` | `immultiply` | `imsubtract`

Introduced before R2006a

imadjust

Adjust image intensity values or colormap

Syntax

```
J = imadjust(I)
J = imadjust(I,[low_in; high_in],[low_out; high_out])
J = imadjust(I,[low_in; high_in],[low_out; high_out],gamma)
newmap = imadjust(map,[low_in; high_in],[low_out; high_out],gamma)
RGB2 = imadjust(RGB1, ___ )
gpuarrayB = imadjust(gpuarrayA, ___ )
```

Description

`J = imadjust(I)` maps the intensity values in grayscale image `I` to new values in `J` such that 1% of data is saturated at low and high intensities of `I`. This increases the contrast of the output image `J`. This syntax is equivalent to `imadjust(I,stretchlim(I))`.

`J = imadjust(I,[low_in; high_in],[low_out; high_out])` maps the values in `I` to new values in `J` such that values between `low_in` and `high_in` map to values between `low_out` and `high_out`.

Note If `high_out` is less than `low_out`, `imadjust` reverses the output image, as in a photographic negative.

`J = imadjust(I,[low_in; high_in],[low_out; high_out],gamma)` maps the values in `I` to new values in `J`, where `gamma` specifies the shape of the curve describing the relationship between the values in `I` and `J`.

`newmap = imadjust(map,[low_in; high_in],[low_out; high_out],gamma)` transforms the m -by-3 array colormap associated with an indexed image. `low_in`, `high_in`, `low_out`, and `high_out` must be 1-by-3 vectors. `gamma` can be a 1-by-3 vector that specifies a unique gamma value for each channel or a scalar that specifies the value used for all three channels. The rescaled colormap `newmap` is the same size as `map`.

`RGB2 = imadjust(RGB1, ___)` performs the adjustment on each plane (red, green, and blue) of the RGB image `RGB1`. If `low_in`, `high_in`, `low_out`, `high_out`, and `gamma` are scalars, `imadjust` applies the same mapping to the red, green, and blue components of the image. To specify unique mappings for each color component of the image, specify `low_in`, `high_in`, `low_out`, `high_out`, and `gamma` as 1-by-3 vectors.

`gpuarrayB = imadjust(gpuarrayA, ___)` performs the contrast adjustment on a GPU. The input gpuArray `gpuarrayA` is an intensity image, RGB image, or a colormap. The output gpuArray `gpuarrayB` is the same as the input image. This syntax requires the Parallel Computing Toolbox.

Examples

Adjust Contrast of Grayscale Image

Read a low-contrast grayscale image into the workspace and display it.

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



Adjust the contrast of the image so that 1% of the data is saturated at low and high intensities, and display it.

```
J = imadjust(I);  
figure  
imshow(J)
```



Adjust Contrast of Grayscale Image on GPU

Read an image into a `gpuArray` and then pass the `gpuArray` to `imadjust`.

```
I = gpuArray(imread('pout.tif'));  
figure  
imshow(I)
```

```
J = imadjust(I);  
figure  
imshow(J)
```

Adjust Contrast of Grayscale Image Specifying Contrast Limits

Read a low-contrast grayscale image into the workspace and display it.

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



Adjust the contrast of the image, specifying contrast limits.

```
K = imadjust(I,[0.3 0.7],[1]);  
figure  
imshow(K)
```



Adjust Contrast of Grayscale Image Specifying Contrast Limits on GPU

Read an image into a `gpuArray` and then pass the `gpuArray` to `imadjust`.

```
I = gpuArray(imread('pout.tif'));  
figure  
imshow(I)
```

```
K = imadjust(I,[0.3 0.7],[0 1]);  
figure  
imshow(K)
```

Adjust Contrast of RGB Image

Read an RGB image into the workspace and display it.

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



Adjust the contrast of the RGB image, specifying contrast limits.

```
RGB2 = imadjust(RGB,[.2 .3 0; .6 .7 1],[1]);  
figure  
imshow(RGB2)
```



Adjust Contrast of RGB Image on GPU

Read an RGB image into a `gpuArray` and then pass the `gpuArray` to `imadjust`, specifying contrast limits for the input image.

```
RGB = gpuArray(imread('football.jpg'));  
RGB2 = imadjust(RGB,[.2 .3 0; .6 .7 1],[1]);  
figure  
imshow(RGB)  
figure  
imshow(RGB2)
```

Standard Deviation Based Image Stretching

Read image into the workspace.

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


Calculate standard deviations from mean for stretching.

```
n = 2;  
Idouble = im2double(I);  
avg = mean2(Idouble);  
sigma = std2(Idouble);
```

Adjust the contrast based on standard deviation.

```
J = imadjust(I,[avg-n*sigma avg+n*sigma],[,]);
```

Display original image and adjusted image.

```
figure  
imshow(I)  
figure  
imshow(J)
```





Input Arguments

I — Image to be adjusted

grayscale image

Image to be adjusted, specified as a grayscale image.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `uint32`

[low_in; high_in] — Contrast limits for the input image

`[0 1]` (default) | two-element numeric vector with values between 0 and 1 | 1-by-3 element vector

Contrast limits for input image, specified as a two-element numeric vector with values between 0 and 1. Values below `low_in` and above `high_in` are clipped; that is, values

below `low_in` map to `low_out`, and those above `high_in` map to `high_out`. If you specify an empty matrix (`[]`), `imadjust` uses the default limits `[0 1]`.

If the input image is a colormap, must be 1-by-3 vectors.

Data Types: `single` | `double`

[low_out; high_out] — Contrast limits for the output image

`[0 1]` (default) | two-element numeric vector with values between 0 and 1

Contrast limits for the output image, specified as a two-element numeric vector with values between 0 and 1. Values below `low_in` and above `high_in` are clipped; that is, values below `low_in` map to `low_out`, and those above `high_in` map to `high_out`. If you specify an empty matrix (`[]`), `imadjust` uses the default limits `[0 1]`.

Data Types: `single` | `double`

gamma — Shape of the curve describing relationship of input and output values

1 (default) | numeric value of class `double`

Shape of curve describing relationship of input and output values, specified as a numeric value. If `gamma` is less than 1, `imadjust` weights the mapping toward higher (brighter) output values. If `gamma` is greater than 1, `imadjust` weights the mapping toward lower (darker) output values. If you omit the argument, `gamma` defaults to 1 (linear mapping).

Data Types: `double`

map — Colormap to be adjusted

m-by-3 array

Colormap to be adjusted, specified as an *m*-by-3 array.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `uint32`

RGB1 — Color image to be adjusted

RGB image

Color image to be adjusted, specified as an RGB image.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `uint32`

gpuarrayA — Image to be adjusted on a GPU

`gpuArray`

Image to be adjusted on a GPU, specified as an `gpuArray` containing a grayscale image, an RGB image, or a colormap.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `uint32`

Output Arguments

J — Adjusted image

grayscale image

Adjusted image, returned as a grayscale image. **J** has the same class as the input image.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `uint32`

newmap — Adjusted colormap

m-by-3 array

Adjusted colormap, returned as an *m*-by-3 array, of the same class as the input colormap.

Data Types: `single` | `double`

RGB2 — Adjusted color image

RGB image

Adjusted color image, returned as an RGB image. **RGB2** has the same class as the input image.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `uint32`

gpuarrayB — Adjusted image or colormap on a GPU

`gpuArray`

Adjusted image or colormap on a GPU, returned as a `gpuArray`. The `gpuArray` contains a grayscale or an RGB image, or a colormap.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, `imadjust` does not support indexed images.

See Also

See Also

`brighten` | `gpuArray` | `histeq` | `stretchlim`

Introduced before R2006a

ImageAdapter class

Interface for image I/O

Description

`ImageAdapter`, an abstract class, specifies the Image Processing Toolbox interface for region-based reading and writing of image files. You can use classes that inherit from the `ImageAdapter` interface with the `blockproc` function for file-based processing of arbitrary image file formats.

Construction

`adapter = ClassName(...)` handles object initialization, manages file opening or creation, and sets the initial values of class properties. The class constructor can take any number of arguments.

Properties

Colormap

Specifies a colormap. Use the `Colormap` property when working with indexed images.

Data Type: 2-D, real, M-by-3 matrix

Default: Empty (`[]`), indicating either a grayscale, logical, or truecolor image

ImageSize

Holds the size of the entire image. When you construct a new class that inherits from `ImageAdapter`, set the `ImageSize` property in your class constructor.

Data Type: 2- or 3-element vector specified as `[rows cols]` or `[rows cols bands]`, where `rows` indicates height and `cols` indicates width

Methods

Classes that inherit from `ImageAdapter` must implement the `readRegion` and `close` methods to support basic region-based reading of images. The `writeRegion` method allows for incremental, region-based writing of images and is optional. Image Adapter classes that do not implement the `writeRegion` method are read-only.

<code>close</code>	Close <code>ImageAdapter</code> object
<code>readRegion</code>	Read region of image
<code>writeRegion</code>	Write block of data to region of image

See Also

`blockproc`

Topics

Computing Statistics for Large Images

“Abstract Classes” (MATLAB)

“Read and Write Data in Unsupported Formats Using `blockproc`”

close

Class: ImageAdapter

Close ImageAdapter object

Syntax

```
adapter.close
```

Description

`adapter.close` closes the `ImageAdapter` object and performs any necessary clean-up, such as closing file handles. When you construct a class that inherits from the `ImageAdapter` class, implement this method.

readRegion

Class: ImageAdapter

Read region of image

Syntax

```
data = adapter.readRegion(region_start, region_size)
```

Description

`data = adapter.readRegion(region_start, region_size)` reads a region of the image. `region_start` and `region_size` define a rectangular region in the image. `region_start`, a two-element vector, specifies the [row col] of the first pixel (minimum-row, minimum-column) of the region. `region_size`, a two-element vector, specifies the size of the requested region in [rows cols]. When you construct a class that inherits from the `ImageAdapter` class, implement this method.

writeRegion

Class: ImageAdapter

Write block of data to region of image

Syntax

```
adapter.writeRegion(region_start, region_data)
```

Description

`adapter.writeRegion(region_start, region_data)` writes a contiguous block of data to a region of the image. The method writes the block of data specified by the `region_data` argument. The two-element vector, `region_start`, specifies the [row col] location of the first pixel (minimum-row, minimum-column) of the target region in the image. When you construct a class that inherits from the `ImageAdapter` class, implement this method if you need to write data.

imageinfo

Image Information tool

Syntax

```
imageinfo
imageinfo(h)
imageinfo(filename)
imageinfo(info)
imageinfo(himage, filename)
imageinfo(himage, info)
hfig = imageinfo(...)
```

Description

`imageinfo` creates an Image Information tool associated with the image in the current figure. The tool displays information about the basic attributes of the target image in a separate figure. `imageinfo` gets information about image attributes by querying the image object's `CData`.

The following table lists the basic image information included in the Image Information tool display. Note that the tool contains either four or six fields, depending on the type of image.

Attribute Name	Value
Width (columns)	Number of columns in the image
Height (rows)	Number of rows in the image
Class	Data type used by the image, such as <code>uint8</code> .
	Note: For <code>single</code> or <code>int16</code> images, <code>imageinfo</code> returns a class value of <code>double</code> , because image objects convert the <code>CData</code> of these images to <code>double</code> .

Attribute Name	Value
Image type	One of the image types identified by the Image Processing Toolbox software: 'intensity' 'truecolor', 'binary', or 'indexed'.
Minimum intensity or index	For grayscale images, this value represents the lowest intensity value of any pixel. For indexed images, this value represents the lowest index value into a color map. Not included for 'binary' or 'truecolor' images.
Maximum intensity or index	For grayscale images, this value represents the highest intensity value of any pixel. For indexed images, this value represents the highest index value into a color map. Not included for 'binary' or 'truecolor' images.

`imageinfo(h)` creates an Image Information tool associated with `h`, where `h` is a handle to a figure, axes, or image object.

`imageinfo(filename)` creates an Image Information tool containing image metadata from the graphics file `filename`. The image does not have to be displayed in a figure window. `filename` can be any file type that has been registered with an information function in the file formats registry, `imformats`, so its information can be read by `imfinfo`. `filename` can also be a DICOM, NITF, Interfile, or Analyze file.

`imageinfo(info)` creates an Image Information tool containing the image metadata in the structure `info`. `info` is a structure returned by the functions `imfinfo`, `dicominfo`, `nitfinfo` `interfileinfo`, or `analyze75info`. `info` can also be a user-created structure.

`imageinfo(himage, filename)` creates an Image Information tool containing information about the basic attributes of the image specified by the handle `himage` and the image metadata from the graphics file `filename`.

`imageinfo(himage, info)` creates an Image Information tool containing information about the basic attributes of the image specified by the handle `himage` and the image metadata in the structure `info`.

`hfig = imageinfo(...)` returns a handle to the Image Information tool figure.

Examples

```
imageinfo('peppers.png')  
  
h = imshow('bag.png');  
info = imfinfo('bag.png');  
imageinfo(h,info);  
  
imshow('canoe.tif');  
imageinfo;
```

See Also

[analyze75info](#) | [dicominfo](#) | [imattributes](#) | [imfinfo](#) | [imformats](#) | [imtool](#) | [interfileinfo](#) | [nitfinfo](#)

Introduced before R2006a

imagemodel

Image Model object

Syntax

```
imgmodel = imagemodel(himage)
```

Description

`imgmodel = imagemodel(himage)` creates an image model object associated with a target image. The target image `himage` is a handle to an image object or an array of handles to image objects. `imagemodel` returns an image model object or, if `himage` is an array of image objects, an array of image model objects. `imagemodel` works by querying the image object's `CData`.

API Functions

An image model object stores information about an image such as class, type, display range, width, height, minimum intensity value, and maximum intensity value.

The image model object supports methods that you can use to access this information, get information about the pixels in an image, and perform special text formatting. Brief descriptions of these methods follow.

Methods

`imagemodel` supports the following methods. Type `methods imagemodel` to see a list of methods, or type `help imagemodel/methodname` for more information about a specific method.

getClassType — Return class of image

`imageclass = getClassType(imgmodel)` returns the class associated with the `imagemodel`, `imgmodel`. The return value, `imageclass` is a character vector, such as

'uint8', specifying the class of the image object's CData. `imagemodel` is expected to contain only one `imagemodel` object.

getDisplayRange — Return display range of intensity image

`disp_range = getDisplayRange(imgmodel)`, where `imgmodel` is a valid image model and `disp_range` is an array of doubles such as `[0 255]`, returns a double array containing the minimum and maximum values of the display range for an intensity image. For image types other than intensity, the value returned is an empty array.

getImageHeight — Return number of rows

`height = getImageHeight(imgmodel)`, where `imgmodel` is a valid image model and `height` is a double scalar, returns a double scalar containing the number of rows.

getImageType — Return image type

`str = getImageType(imgmodel)` returns the type of image associated with the `imagemodel, imgmodel`. The return value, `str`, is one of the following: 'intensity', 'truecolor', 'binary', or 'indexed'.

getImageWidth — Return number of columns

`width = getImageWidth(imgmodel)`, where `imgmodel` is a valid image model and `width` is a double scalar, returns a double scalar containing the number of columns.

getMinIntensity — Return minimum value in image

`minval = getMinIntensity(imgmodel)`, where `imgmodel` is a valid image model and `minval` is a numeric value, returns the minimum value in the image calculated as `min(Image(:))`. For an intensity image, the value returned is the minimum intensity. For an indexed image, the value returned is the minimum index. For any other image type, the value returned is an empty array. The class of `minval` depends on the class of the target image.

getMaxIntensity — Return maximum value in image

`maxval = getMaxIntensity(imgmodel)`, where `imgmodel` is a valid image model and `maxval` is a numeric value, returns the maximum value in the image calculated as `max(Image(:))`. For an intensity image, the value returned is the maximum intensity. For an indexed image, the value returned is the maximum index. For any other image type, the value returned is an empty array. The class of `maxval` depends on the class of the target image.

getNumberFormatFcn — Return handle to function that converts numeric value into character vector

`fun = getNumberFormatFcn(imgmodel)` returns the handle to a function that converts a numeric value into a character vector, where `imgmodel` is a valid image model. For example, `str = fun(getPixelValue(imgmodel, 100, 100))` converts the numeric return value of the `getPixelValue` method into a character vector.

getPixelInfoString — Return value of specific pixel as character vector

`str = getPixelInfoString(imgmodel, row, column)` returns a character vector containing the value of the pixel at the location specified by `row` and `column`, where `str` is a character array, `imgmodel` is a valid image model and `row` and `column` are numeric scalar values. For example, for an RGB image, the method returns a character vector such as `'[66 35 60]'`.

getPixelRegionFormatFcn — Return handle to function that formats value of pixel into character vector

`fun = getPixelRegionFormatFcn(imgmodel)` returns the value of the pixel as a specially formatted character vector, where `imgmodel` is a valid image model and `fun` is a handle to a function that accepts the location (`row`, `column`) of a pixel in the target image. For example, when used with an RGB image, this function returns a character vector of the form `'R:000 G:000 B:000'` where 000 is the actual pixel value.

```
str = fun(100,100)
```

getPixelValue — Return value of specific pixel as numeric array

`val = getPixelValue(imgmodel, row, column)`, where `imgmodel` is a valid image model and `row` and `column` are numeric scalar values, returns the value of the pixel at the location specified by `row` and `column` as a numeric array. The class of `val` depends on the class of the target image.

getDefaultPixelInfoString — Return pixel information type as character vector

`str = getDefaultPixelInfoString(imgmodel)` returns a character vector indicating the pixel information type, where `imgmodel` is a valid image model. This character vector can be used in place of actual pixel information values. Depending on the image type, `str` can be the value `'Intensity'`, `'[R G B]'`, `'BW'`, or `'<Index> [R G B]'`.

getDefaultPixelRegionString — Return type of information displayed in Pixel Region tool

`str = getDefaultPixelRegionString(imgmodel)` returns a character vector indicating the type of information displayed in the Pixel Region tool for each image type, where `imgmodel` is a valid image model. This character vector can be used in place of actual pixel values. Depending on the image type, `str` can be `'000'`, `'R:000 G:000 B:000']`, `'0'`, or `'<000> R:0.00 G:0.00 B:0.00'`.

getScreenPixelRGBValue — Return screen display value of specific pixel

`val = getScreenPixelRGBValue(imgmodel, row, col)` returns the screen display value of the pixel at the location specified by `row` and `col` as a double array. `imgmodel` is a valid image model, `row` and `col` are numeric scalar values, and `val` is an array of doubles, such as `[0.2 0.5 0.3]`.

Examples

Create an Image Model from Image Objects

Create an image model associated with a single image object.

```
h = imshow('peppers.png');
```



```
im = imagemodel(h)
```

```
im =
```

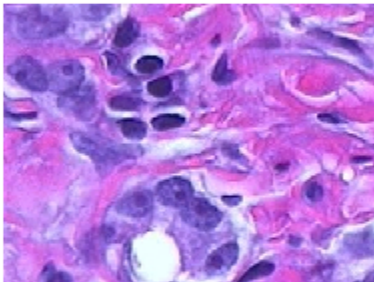
```
IMAGEMODEL object accessing an image with these properties:
```

```
    ClassType: 'uint8'  
    DisplayRange: []  
    ImageHeight: 384  
    ImageType: 'truecolor'  
    ImageWidth: 512  
    MinIntensity: []
```

```
MaxIntensity: []
```

Create an image model for an array of image object handles.

```
figure  
subplot(1,2,1)  
h1 = imshow('hestain.png');  
subplot(1,2,2)  
h2 = imshow('coins.png');
```



```
im = imagemodel([h1 h2])
```

`im =`

`1x2 array of IMAGEMODEL objects.`

See Also

`getimagemodel`

Introduced before R2006a

imapplymatrix

Linear combination of color channels

Syntax

```
Y = imapplymatrix(M,X)
Y = imapplymatrix(M,X,C)
Y = imapplymatrix(...,output_type)
```

Description

`Y = imapplymatrix(M,X)` computes the linear combination of the rows of `M` with the color channels of `X`.

`Y = imapplymatrix(M,X,C)` computes the linear combination of the rows of `M` with the color channels of `X`, adding the corresponding constant value from `C` to each combination.

`Y = imapplymatrix(...,output_type)` returns the result of the linear combination in an array of type `output_type`.

Input Arguments

M — Weighting coefficients for each color channel

Numeric array of size q -by- p

Weighting coefficients for each color channel, specified as a numeric array. If `X` is size m -by- n -by- p , `M` must be size q -by- p , where q is in the range $[1,p]$. `M` must be type `double`.

Data Types: `double`

X — Input image

Numeric array of size m -by- n -by- p

Input image, specified as a numeric array of size m -by- n -by- p .

C — Constant to add to each channel

Numeric vector of length q

Constant to add to each channel during the linear combination, specified as a numeric vector of length q , where q is number of rows in M .

Data Types: `double`

output_type — Output data type

'double' | 'single' | 'uint8' | 'uint16' | 'uint32' | 'int8' | 'int16' | 'int32'

Output data type, specified as one of the following character vectors: 'double', 'single', 'uint8', 'uint16', 'uint32', 'int8', 'int16', or 'int32'.

Output Arguments

Y — Output image

Output image, comprised of the linear combination of the rows of M with the color channels of X . If `output_type` is not specified, the data type of Y is the same as the data type of X .

Examples

Compute Linear Combination of Color Channels

This example shows how to create a grayscale image by computing the linear combination of three colors channels.

Read a truecolor image into the workspace.

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

Create a coefficient matrix

```
M = [0.30, 0.59, 0.11];
```

Compute the linear combination of the RGB channels using the coefficient matrix.

```
gray = imapplymatrix(M, RGB);
```

Display the original image and the grayscale conversion.

```
imshowpair(RGB,gray,'montage')
```



See Also

See Also

`imlincomb` | `immultiply`

Introduced in R2011b

imattributes

Information about image attributes

Syntax

```
attrs = imattributes
attrs = imattributes(himage)
attrs = imattributes(imgmodel)
```

Description

`attrs = imattributes` returns information about an image in the current figure. If the current figure does not contain an image, `imattributes` returns an empty array.

`attrs = imattributes(himage)` returns information about the image specified by `himage`, a handle to an image object. `imattributes` gets the image attributes by querying the image object's `CData`.

`imattributes` returns image attribute information in `attrs`, a 4-by-2 or 6-by-2 cell array, depending on the image type. The first column of the cell array contains the name of the attribute. The second column contains the value of the attribute. Both attribute names and values are character vectors. The following table lists these attributes in the order they appear in the cell array.

Attribute Name	Value
'Width (columns)'	Number of columns in the image
'Height (rows)'	Number of rows in the image
'Class'	Data type used by the image, such as <code>uint8</code> .
	Note: For <code>single</code> or <code>int16</code> images, <code>imageinfo</code> returns a class value of <code>double</code> , because image objects convert <code>CData</code> of these classes to <code>double</code> .

Attribute Name	Value
'Image type'	One of the image types identified by the Image Processing Toolbox software: 'intensity', 'truecolor', 'binary', or 'indexed'.
'Minimum intensity'	For intensity images, this value represents the lowest intensity value of any pixel. For indexed images, this value represents the lowest index value into a color map. Not included for 'binary' or 'truecolor' images.
'Maximum intensity'	For intensity images, this value represents the highest intensity value of any pixel. For indexed images, this value represents the highest index value into a color map. Not included for 'binary' or 'truecolor' images.

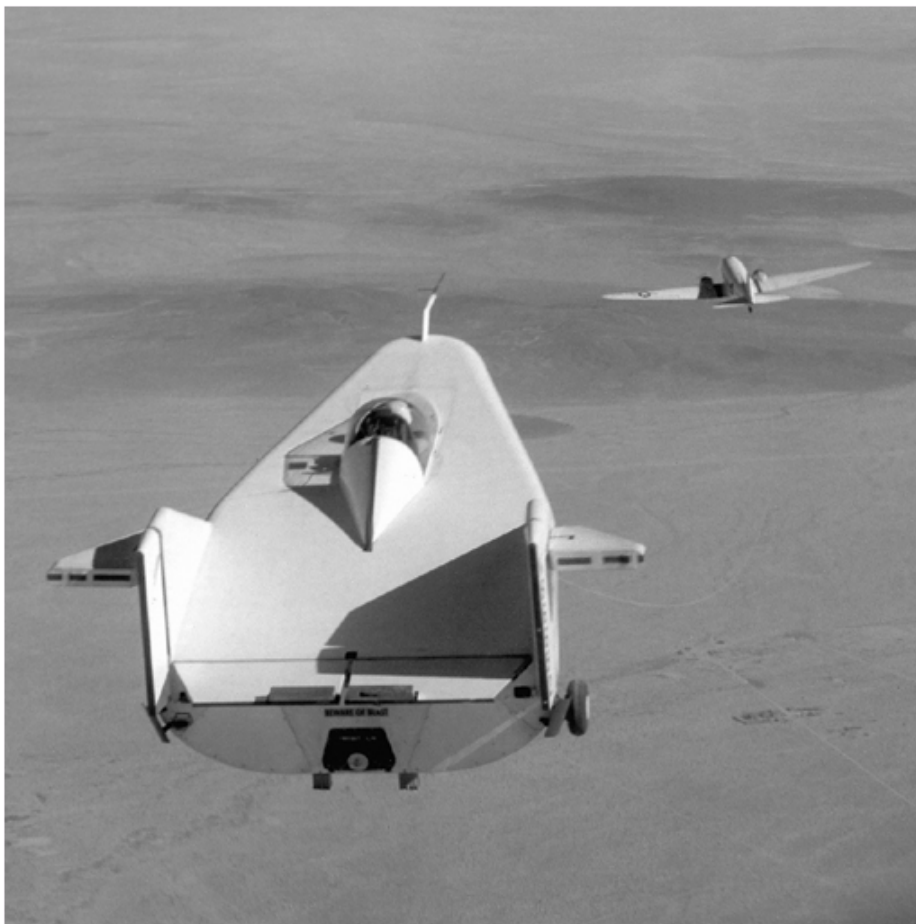
`attrs = imattributes(imgmodel)` returns information about the image represented by the image model object, `imgmodel`.

Examples

Retrieve Attributes of Grayscale Image

Read a grayscale image into the workspace.

```
h = imshow('liftingbody.png');
```



Get the image attributes.

```
attrs = imattributes(h)
```

```
attrs = 6x2 cell array  
    'Width (columns)'    '512'
```

```
'Height (rows)'      '512'  
'Class'              'uint8'  
'Image type'         'intensity'  
'Minimum intensity'  '0'  
'Maximum intensity'  '255'
```

Retrieve Attributes of Truecolor Image

```
h = imshow('gantrycrane.png');
```



```
im = imagemodel(h);  
attrs = imattributes(im)  
  
attrs = 4x2 cell array  
    'Width (columns)'      '400'  
    'Height (rows)'       '264'  
    'Class'                'uint8'
```

'Image type'

'truecolor'

See Also

imagemodel

Introduced before R2006a

imbinarize

Binarize image by thresholding

Syntax

```
BW = imbinarize(I)
BW = imbinarize(I,method)
BW = imbinarize(I,T)
BW = imbinarize(I,'adaptive',Name,Value)
```

Description

`BW = imbinarize(I)` creates a binary image from image `I` by replacing all values above a globally determined threshold with 1s and setting all other values to 0s. By default, `imbinarize` uses Otsu's method, which chooses the threshold value to minimize the intraclass variance of the thresholded black and white pixels. `imbinarize` uses a 256-bin image histogram to compute Otsu's threshold. To use a different histogram, see `otsuthresh`. `BW` is the output binary image.

`BW = imbinarize(I,method)` creates a binary image from image `I` using the thresholding method specified by `method`.

`BW = imbinarize(I,T)` creates a binary image from image `I` using the threshold value `T`.

`BW = imbinarize(I,'adaptive',Name,Value)` creates a binary image from image `I` using name-value pairs to control aspects of adaptive thresholding.

Examples

Binarize Image Using Global Threshold

Read grayscale image into the workspace.

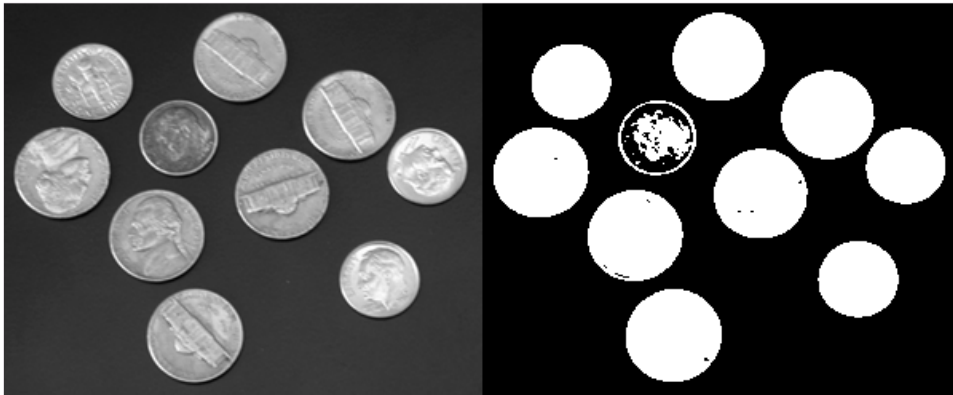
```
I = imread('coins.png');
```

Convert the image into a binary image.

```
BW = imbinarize(I);
```

Display the original image next to the binary version.

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



Binarize Image Using Locally Adaptive Thresholding

Read grayscale image into workspace.

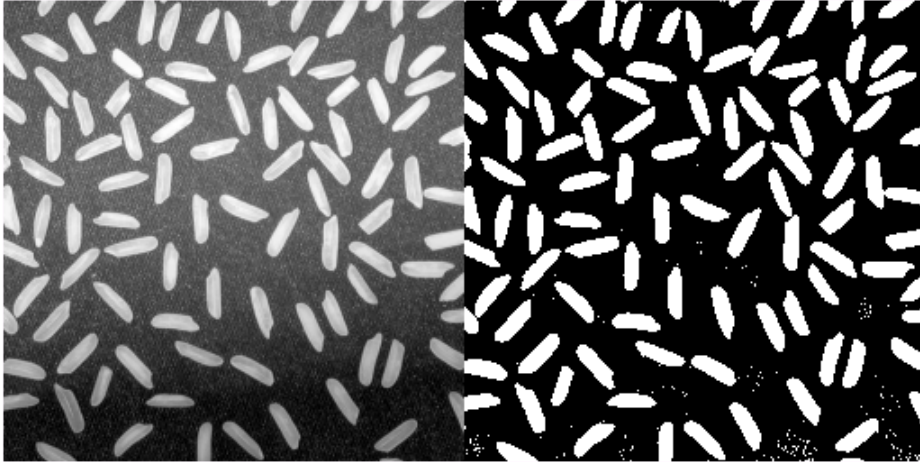
```
I = imread('rice.png');
```

Convert grayscale image to binary image.

```
BW = imbinarize(I, 'adaptive');
```

Display original image along side binary version.

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

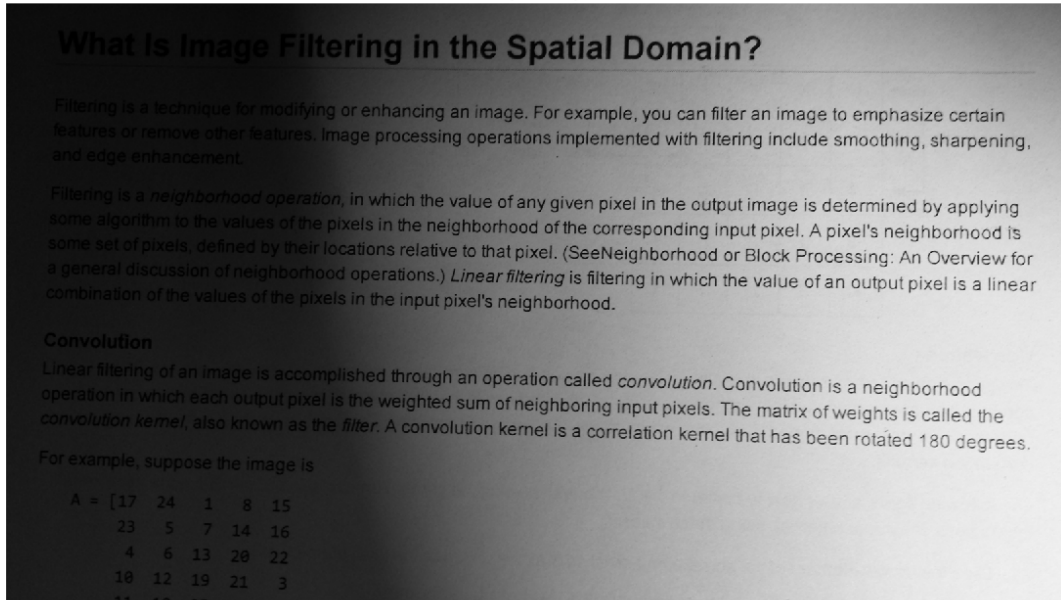


Binarize Images with Darker Foreground Than Background.

Read a grayscale image into the workspace and display it.

```
I = imread('printedtext.png');  
figure  
imshow(I)  
title('Original Image')
```

Original Image



Convert the image to a binary image using adaptive thresholding. Use the `ForegroundPolarity` parameter to indicate that the foreground is darker than the background.

```
BW = imbinarize(I, 'adaptive', 'ForegroundPolarity', 'dark', 'Sensitivity', 0.4);
```

Display the binary version of the image.

```
figure
imshow(BW)
title('Binary Version of Image')
```


Binary Version of Image

What Is Image Filtering in the Spatial Domain?

Filtering is a technique for modifying or enhancing an image. For example, you can filter an image to emphasize certain features or remove other features. Image processing operations implemented with filtering include smoothing, sharpening, and edge enhancement.

Filtering is a *neighborhood operation*, in which the value of any given pixel in the output image is determined by applying some algorithm to the values of the pixels in the neighborhood of the corresponding input pixel. A pixel's neighborhood is some set of pixels, defined by their locations relative to that pixel. (See Neighborhood or Block Processing: An Overview for a general discussion of neighborhood operations.) *Linear filtering* is filtering in which the value of an output pixel is a linear combination of the values of the pixels in the input pixel's neighborhood.

Convolution

Linear filtering of an image is accomplished through an operation called *convolution*. Convolution is a neighborhood operation in which each output pixel is the weighted sum of neighboring input pixels. The matrix of weights is called the *convolution kernel*, also known as the *filter*. A convolution kernel is a correlation kernel that has been rotated 180 degrees.

For example, suppose the image is

```
A = [17 24 1 8 15
      23 5 7 14 16
      4 6 13 20 22
      10 12 19 21 3
      11 10 11 12 13]
```

Input Arguments

I — Input image

real, nonsparse, 2-D matrix

Input image, specified as a real, nonsparse, 2-D matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `uint64`

method — Method used to binarize image

'global' (default) | 'adaptive'

Method used to binarize image, specified as one of the following values (names can be abbreviated).

Values	Meaning
'global'	Calculate global image threshold using Otsu's method. See <code>graythresh</code> for more information about Otsu's method.
'adaptive'	Calculate locally adaptive image threshold chosen using local first-order image statistics around each pixel. See <code>adaptthresh</code> for details. If the image contains <code>Infs</code> or <code>NaNs</code> , the behavior of <code>imbinarize</code> for the 'adaptive' method is undefined. Propagation of <code>Infs</code> or <code>NaNs</code> may not be localized to the neighborhood around <code>Inf</code> and <code>NaN</code> pixels.

Data Types: `char`

T — Threshold

scalar luminance value or matrix of luminance values

Threshold, specified as a scalar luminance value or as a matrix of luminance values. If `T` is a scalar luminance value, `imbinarize` interprets it as a global image threshold. If `T` is a matrix of luminance values, `imbinarize` interprets it as a locally adaptive threshold. `T` must have a value between 0 and 1. If `T` is a matrix, it must be of the same size as `I`. Use the functions `graythresh`, `otsuthresh`, or `adaptthresh` to compute `T`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `BW = imbinarize(I, 'adaptive', 'Sensitivity', 0.4);`

'Sensitivity' — Sensitivity factor for adaptive thresholding

0.50 (default) | value in the range [0 1]

Sensitivity factor for adaptive thresholding, specified as a value in the range [0 1]. A high sensitivity value leads to thresholding more pixels as foreground, at the risk of including some background pixels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'ForegroundPolarity' — Determine which pixels are considered foreground pixels
'bright' (default) | 'dark'

Determine which pixels are considered foreground pixels, specified as either of the following values:

Values	Meaning
'bright'	The foreground is brighter than the background. This is the default.
'dark'	The foreground is darker than the background

Data Types: char

Output Arguments

BW — Output binary image

logical matrix

Output binary image, returned as a logical matrix the same size as I.

Tips

- To produce a binary image from an indexed image, first convert the image to a grayscale intensity image using `ind2gray`.
- To produce a binary image from an RGB image, first convert the image to a grayscale intensity image using `rgb2gray`.

Algorithms

The 'adaptive' method binarizes the image using a locally adaptive threshold. `imbinarize` computes a threshold for each pixel using the local mean intensity around the neighborhood of the pixel. (This technique is also called Bradley's method.) The 'adaptive' method also uses a neighborhood size of approximately 1/8th of the size of the image (computed as $2 * \text{floor}(\text{size}(I) / 16) + 1$). To use a different first order local statistic or a different neighborhood size, see `adaptthresh`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, all character vector input arguments must be compile-time constants.

See Also

See Also

`adaptthresh` | `graythresh` | `Image Segmenter` | `otsuthresh`

Introduced in R2016a

imbothat

Bottom-hat filtering

Syntax

```
IM2 = imbothat(IM,SE)  
IM2 = imbothat(IM,NHOOD)  
gpuarrayIM2 = imbothat(gpuarrayIM, ___ )
```

Description

`IM2 = imbothat(IM,SE)` performs morphological bottom-hat filtering on the grayscale or binary input image, `IM`, returning the filtered image, `IM2`. `SE` is a structuring element returned by the `strel` function. `SE` must be a single structuring element object, not an array containing multiple structuring element objects.

`IM2 = imbothat(IM,NHOOD)` performs morphological bottom-hat filtering where `NHOOD` is an array of 0's and 1's that specifies the size and shape of the structuring element. This is equivalent to `imbothat(IM,strel(NHOOD))`.

`gpuarrayIM2 = imbothat(gpuarrayIM, ___)` performs operation on a graphics processing unit (GPU), where `gpuarrayIM` is a `gpuArray` of class `uint8` or `logical`. This syntax requires the Parallel Computing Toolbox.

Class Support

`IM` can be numeric or logical and must be nonsparse. If the input is binary (logical), then the structuring element must be flat.

`gpuarrayIM` must be a `gpuArray` of type `uint8` or `logical`. When used with a `gpuarray`, the structuring element must be flat and two-dimensional.

The output has the same class as the input.

Examples

Enhance Contrast Using Bottom-hat and Top-hat Filtering

Read image into the workspace and display it.

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



Create a disk-shaped structuring element.

```
se = strel('disk',3);
```

Add the original image `I` to the top-hat filtered image, and then subtract the bottom-hat filtered image.

```
J = imsubtract(imadd(I,imtophat(I,se)),imbothat(I,se));  
figure  
imshow(J)
```



Enhance Contrast using Bottom Hat Filtering on a GPU

Read the image into a `gpuArray`.

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

Create a disk-shaped structuring element, needed for morphological processing.

```
se = strel('disk',3);
```

Add the original image `I` to the top-hat filtered image, and then subtract the bottom-hat filtered image.

```
contrastFiltered = ...  
    (original+imtophat(original,se))-imbothat(original,se);
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the input image, `IM`, must be 2-D or 3-D and the structuring element argument `SE` must be a single element—arrays of structuring elements are not supported. To obtain the same result as that obtained using an array of structuring elements, call the function sequentially.

See Also

`imtophat` | `strel` | `offsetstrel` | `gpuArray`

Introduced before R2006a

imboxfilt

2-D box filtering of images

Syntax

```
B = imboxfilt(A)
B = imboxfilt(A,filterSize)
B = imboxfilt( ____,Name,Value)
```

Description

`B = imboxfilt(A)` filters image `A` with a 2-D, 3-by-3 box filter. A box filter is also called a mean filter.

`B = imboxfilt(A,filterSize)` filters image `A` with a 2-D box filter with size specified by `filterSize`.

`B = imboxfilt(____,Name,Value)` filters image `A` with a 2-D box filter where Name-Value pairs control aspects of the filtering.

Examples

Compute Mean Filter Over Specified Neighborhood

Read image into the workspace.

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

Perform the mean filtering using an 11-by-11 filter.

```
localMean = imboxfilt(A,11);
```

Display the original image and the filtered image, side-by-side.

```
imshowpair(A,localMean,'montage')
```



Compute Local Area Sums Over Specified Neighborhood

Read image into the workspace.

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

Change the data type of the image to `double` to avoid integer overflow.

```
A = double(A);
```

Filter image, calculating local area sums, using a 15-by-15 box filter. To calculate local area sums, rather than the mean, set the `NormalizationFactor` parameter to 1.

```
localSums = imboxfilt(A, 15, 'NormalizationFactor',1);
```

Display the original image and the filtered image, side-by-side.

```
imshowpair(A,localSums,'montage')
```



Input Arguments

A — Image to be filtered

real, nonsparse array of any dimension

Image to be filtered, specified as a real, nonsparse array of any dimension.

If **A** contains `Inf`s or `NaN`s, the behavior of `imboxfilt` is undefined. This can happen when integral image based filtering is used. To restrict the propagation of `Inf`s and `NaN`s in the output, consider using `imfilter` instead.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

filterSize — Size of box filter

3-by-3 (default) | scalar or 2-element vector of positive, odd integers

Size of box filter, specified as a scalar or 2-element vector of positive, odd integers. If `filterSize` is scalar, the box filter is square.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

```
Example: A = imread('cameraman.tif'); B =
imboxfilt(A,5,'Padding','circular');
```

'Padding' — Padding pattern

'replicate' (default) | 'circular' | 'symmetric' | numeric scalar

Padding pattern, specified as one of the following values or a numeric scalar. If you specify a scalar value, input image pixels outside the bounds of the image are implicitly assumed to have the scalar value.

Value	Description
'circular'	Input image values outside the bounds of the image are computed by implicitly assuming the input image is periodic.
'replicate'	Input image values outside the bounds of the image are assumed equal to the nearest image border value.
'symmetric'	Input image values outside the bounds of the image are computed by mirror-reflecting the array across the array border.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char`

'NormalizationFactor' — Normalization factor applied to box filter

$1/\text{filterSize}.^2$, if scalar, and $1/\text{prod}(\text{filterSize})$, if vector (default) | numeric scalar

Normalization factor applied to box filter, specified as a numeric scalar.

The default 'NormalizationFactor' has the effect of a mean filter—the pixels in the output image are the local means of the image over the neighborhood determined

by `filterSize`. To get local area sums, set `'NormalizationFactor'` to 1. To avoid overflow in such circumstances, consider using double precision images by converting the input image to class `double`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

B — Filtered image

real, nonsparse matrix

Filtered image, returned as a real, nonsparse matrix, the same size as the input image.

Algorithms

`imboxfilt` performs filtering using either convolution-based filtering or integral image filtering, using an internal heuristic to determine which filtering approach to use.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, all character vector input arguments must be compile-time constants.

See Also

See Also

`imboxfilt3` | `imfilter` | `integralBoxFilter`

Introduced in R2015b

imboxfilt3

3-D box filtering of 3-D images

Syntax

```
B = imboxfilt3(A)
B = imboxfilt3(A,filterSize)
B = imboxfilt3( ___,Name,Value)
```

Description

`B = imboxfilt3(A)` filters the 3-D image `A` with a 3-D box filter, 3-by-3-by-3 in size.

`B = imboxfilt3(A,filterSize)` filters 3-D image `A` with a 3-D box filter with size specified by `filterSize`.

`B = imboxfilt3(___,Name,Value)` filters 3-D image `A` where Name-Value pairs control aspects of the filtering.

Examples

Compute Mean Filter in MRI Volume

Load 3-D image data into the workspace.

```
volData = load('mri');
vol = squeeze(volData.D);
```

Filter the image with a 3-D box filter.

```
localMean = imboxfilt3(vol,[5 5 3]);
```

Input Arguments

A — Image to be filtered

real, nonsparse 3-D array

Image to be filtered, specified as a real, nonsparse 3-D array.

If *A* contains `Infs` or `NaNs`, the behavior of `imboxfilt3` is undefined. This can happen when integral image based filtering is used. To restrict the propagation of `Infs` and `NaNs` in the output, consider using `imfilter` instead.

Example: `B = imboxfilt3(A);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

filterSize — Size of box filter

(default) | scalar or 3-element vector of positive, odd integers

Size of box filter, specified as a scalar or 3-element vector of positive, odd integers. If `filterSize` is scalar, the filter is a cube.

Example: `B = imboxfilt3(A,5);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `B = imboxfilt3(A,5,'padding','circular');`

'Padding' — Padding pattern

'replicate' (default) | 'circular' | 'symmetric' | numeric scalar

Padding pattern, specified as one of the following values or a numeric scalar. If you specify a scalar value, input image pixels outside the bounds of the image are implicitly assumed to have the scalar value.

Value	Description
'circular'	Input image values outside the bounds of the image are computed by implicitly assuming the input image is periodic.
'replicate'	Input image values outside the bounds of the image are assumed equal to the nearest image border value.

Value	Description
'symmetric'	Input image values outside the bounds of the image are computed by mirror-reflecting the array across the array border.

Example: `B = imboxfilt3(A,5,'padding','circular');`

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

'NormalizationFactor' — Normalization factor applied to box filter

`1/filterSize.^3`, if scalar, and `1/prod(filterSize)`, if vector (default) | numeric scalar

Normalization factor applied to box filter, specified as a numeric scalar.

The default 'NormalizationFactor' has the effect of a mean filter—the pixels in the output image are the local means of the image. To get local area sums, set 'NormalizationFactor' to 1. To avoid overflow in such circumstances, consider using double precision images by converting the input image to class `double`.

Example: `B = imboxfilt3(A,5,'NormalizationFactor',1);`

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

Output Arguments

B — Filtered image

real, nonsparse 3-D array

Filtered image, returned as a real, nonsparse 3-D array.

Algorithms

`imboxfilt` performs filtering using either convolution-based filtering or integral image filtering, using an internal heuristic to determine which filtering approach to use.

See Also

See Also

`imboxfilt` | `imfilter` | `integralBoxFilter3`

Introduced in R2015b

imclearborder

Suppress light structures connected to image border

Syntax

```
IM2 = imclearborder(IM)
IM2 = imclearborder(IM,conn)
```

Description

`IM2 = imclearborder(IM)` suppresses structures that are lighter than their surroundings and that are connected to the image border. Use this function to clear the image border. `IM` can be a grayscale or binary image. For grayscale images, `imclearborder` tends to reduce the overall intensity level in addition to suppressing border structures. The output image, `IM2`, is grayscale or binary, depending on the input. The default connectivity is 8 for two dimensions, 26 for three dimensions, and `conndef(ndims(BW), 'maximal')` for higher dimensions.

`IM2 = imclearborder(IM,conn)` specifies the desired connectivity.

Examples

Impact of Connectivity on Clearing the Border

Create a simple binary image.

```
BW = [0  0  0  0  0  0  0  0  0
      0  0  0  0  0  0  0  0  0
      0  0  0  0  0  0  0  0  0
      1  0  0  1  1  1  0  0  0
      0  1  0  1  1  1  0  0  0
      0  0  0  1  1  1  0  0  0
      0  0  0  0  0  0  0  0  0
      0  0  0  0  0  0  0  0  0
      0  0  0  0  0  0  0  0  0];
```

Clear pixels on the border of the image using 4-connectivity. Note that `imclearborder` does not clear the pixel at (5,2) because, with 4-connectivity, it is not considered connected to the border pixel at (4,1).

```
BWc1 = imclearborder(BW,4)
```

```
BWc1 =
```

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

Now clear pixels on the border of the image using 8-connectivity. `imclearborder` clears the pixel at (5,2) because, with 8-connectivity, it is considered connected to the border pixel (4,1).

```
BWc2 = imclearborder(BW,8)
```

```
BWc2 =
```

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

Input Arguments

IM — Grayscale or binary image

real, nonsparse, numeric or logical array

Grayscale or binary image, specified as a real, nonsparse, numeric or logical array.

Example: `I = imread('pout.tif'); I2 = imclearborder(I);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

conn — Connectivity

8 (for 2-D) (default) | 4 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix

Connectivity, specified as one of the values in this table.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can also be defined in a more general way for any dimension by using for `conn` a 3-by-3-by- ... -by-3 matrix of 0s and 1s. The 1-valued elements define neighborhood locations relative to the center element of `conn`. Note that `conn` must be symmetric around its center element.

Note A pixel on the edge of the input image might not be considered to be a border pixel if you specify a nondefault connectivity. For example, if `conn = [0 0 0; 1 1 1; 0 0 0]`, elements on the first and last row are not considered to be border pixels because, according to that connectivity definition, they are not connected to the region outside the image.

Example: `I2 = imclearborder(I,4);`

Data Types: `double` | `logical`

Output Arguments

IM2 — Grayscale or binary image

real, nonsparse, numeric or logical array

Grayscale or binary image, returned as a real, nonsparse, numeric or logical array, depending on the input image you specify.

Algorithms

`imclearborder` uses morphological reconstruction where:

- Mask image is the input image.
- Marker image is zero everywhere except along the border, where it equals the mask image.

References

- [1] Soille, P., *Morphological Image Analysis: Principles and Applications*, Springer, 1999, pp. 164-165.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- Supports only up to 3-D inputs.

- The optional second input argument, `conn`, must be a compile-time constant.

See Also

See Also

`conndef`

Introduced before R2006a

imclose

Morphologically close image

Syntax

```
IM2 = imclose(IM,SE)
IM2 = imclose(IM,NHOOD)
gpuarrayIM2 = imclose(gpuarrayIM, ___ )
```

Description

`IM2 = imclose(IM,SE)` performs morphological closing on the grayscale or binary image `IM`, returning the closed image, `IM2`. The structuring element, `SE`, must be a single structuring element object, as opposed to an array of objects. The morphological close operation is a dilation followed by an erosion, using the same structuring element for both operations.

`IM2 = imclose(IM,NHOOD)` performs closing with the structuring element `strel(NHOOD)`, where `NHOOD` is an array of 0's and 1's that specifies the structuring element neighborhood.

`gpuarrayIM2 = imclose(gpuarrayIM, ___)` performs the operation on a graphics processing unit (GPU), where `gpuarrayIM` is a `gpuArray` containing the grayscale or binary image. `gpuarrayIM2` is a `gpuArray` of the same class as the input image. This syntax requires the Parallel Computing Toolbox.

Class Support

`IM` can be any numeric or logical class and any dimension, and must be nonsparse. If `IM` is logical, then `SE` must be flat.

`gpuarrayIM` must be a `gpuArray` of type `uint8` or `logical`. When used with a `gpuarray`, the structuring element must be flat and two-dimensional.

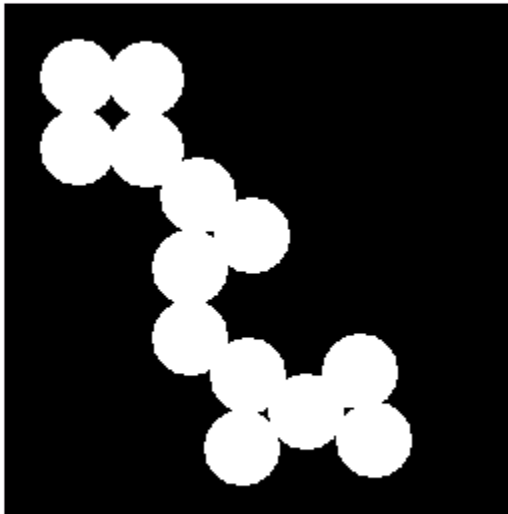
The output has the same class as the input.

Examples

Use Morphological Closing to Fill Gaps in an Image

Read a binary image into the workspace and display it.

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



Create a disk-shaped structuring element. Use a disk structuring element to preserve the circular nature of the object. Specify a radius of 10 pixels so that the largest gap gets filled.

```
se = strel('disk',10);
```

Perform a morphological close operation on the image.

```
closeBW = imclose(originalBW,se);
```

```
figure, imshow(closeBW)
```



Use Morphological Closing to Fill Gaps in an Image on a GPU

Use morphological closing to join the circles in an image together by filling in the gaps between them and by smoothing their outer edges.

Read the image into the MATLAB workspace and view it.

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

Create a disk-shaped structuring element. Use a disk structuring element to preserve the circular nature of the object. Specify a radius of 10 pixels so that the largest gap gets filled.

```
se = strel('disk',10);
```

Perform a morphological close operation on the image on a GPU.

```
closeBW = imclose(gpuArray(originalBW),se);  
figure, imshow(closeBW)
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the input image, **IM**, must be 2-D or 3-D and the structuring element argument **SE** must be a single element—arrays of structuring elements are not supported. To obtain the same result as that obtained using an array of structuring elements, call the function sequentially.

See Also

`imdilate` | `imopen` | `imerode` | `gpuArray` | `strel` | `offsetstrel`

Introduced before R2006a

imcolormaptool

Choose Colormap tool

Syntax

```
imcolormaptool  
imcolormaptool(hclientfig)  
hfig = imcolormaptool(...)
```

Description

The Choose Colormap tool is an interactive colormap selection tool that allows you to change the colormap of the target (current) figure by selecting a colormap from a list of MATLAB colormap functions or workspace variables, or by entering a custom MATLAB expression.

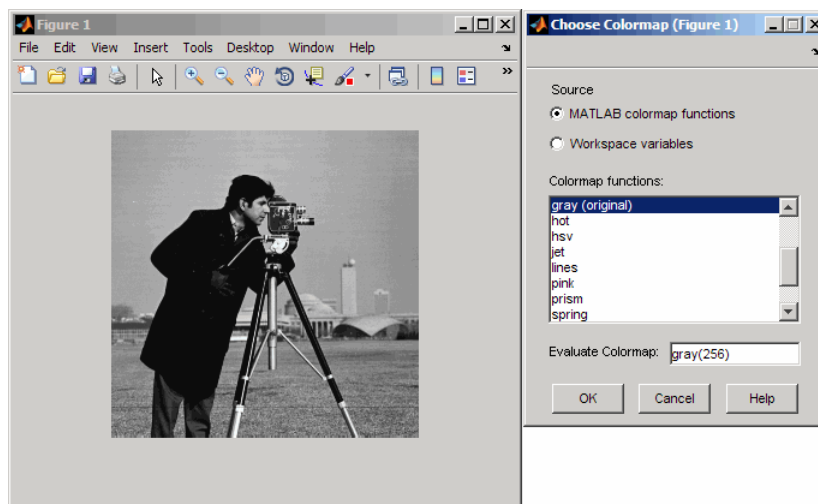
`imcolormaptool` launches the Choose Colormap tool in a separate figure, which is associated with the target figure.

`imcolormaptool(hclientfig)` launches the Choose Colormap tool using `hclientfig` as the target figure. `hclientfig` must contain either a grayscale or an indexed image.

`hfig = imcolormaptool(...)` returns a handle to the Choose Colormap tool, `hfig`.

Examples

```
h = figure;  
imshow('cameraman.tif');  
imcolormaptool(h);
```



Choose Colormap Tool

See Also

[colormap](#) | [imshow](#) | [imtool](#)

Introduced in R2009a

imcomplement

Complement image

Syntax

```
IM2 = imcomplement(IM)
gpuarrayIM2 = imcomplement(gpuarrayIM)
```

Description

`IM2 = imcomplement(IM)` computes the complement of the image `IM`. `IM` can be a binary, grayscale, or RGB image. `IM2` has the same class and size as `IM`.

In the complement of a binary image, zeros become ones and ones become zeros; black and white are reversed. In the complement of an intensity or RGB image, each pixel value is subtracted from the maximum pixel value supported by the class (or 1.0 for double-precision images) and the difference is used as the pixel value in the output image. In the output image, dark areas become lighter and light areas become darker.

`gpuarrayIM2 = imcomplement(gpuarrayIM)` computes the complement of the image on a GPU. The input image `gpuarrayIM` and the return values are `gpuArrays`. `gpuarrayIM2` is a `gpuArray` with the same underlying class and size as `gpuarrayIM`. This syntax requires the Parallel Computing Toolbox.

Examples

Create the Complement of a uint8 Array

```
X = uint8([ 255 10 75; 44 225 100]);
X2 = imcomplement(X)
```

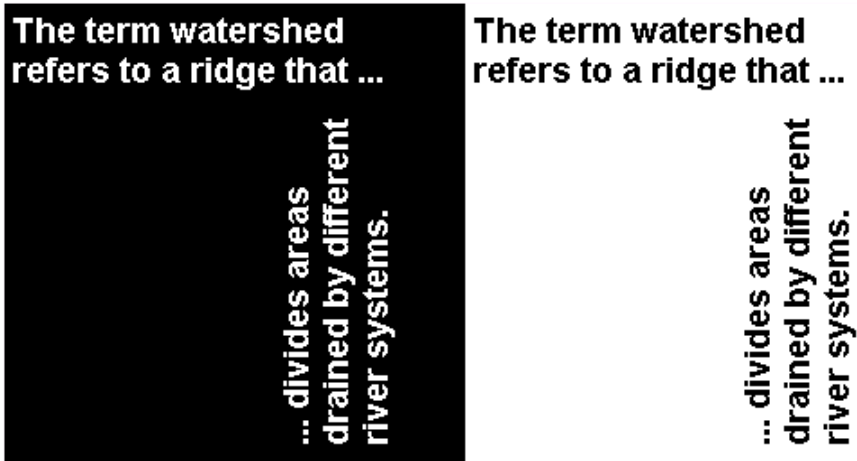
```
X2 = 2×3 uint8 matrix
```

```
    0    245    180
```

211 30 155

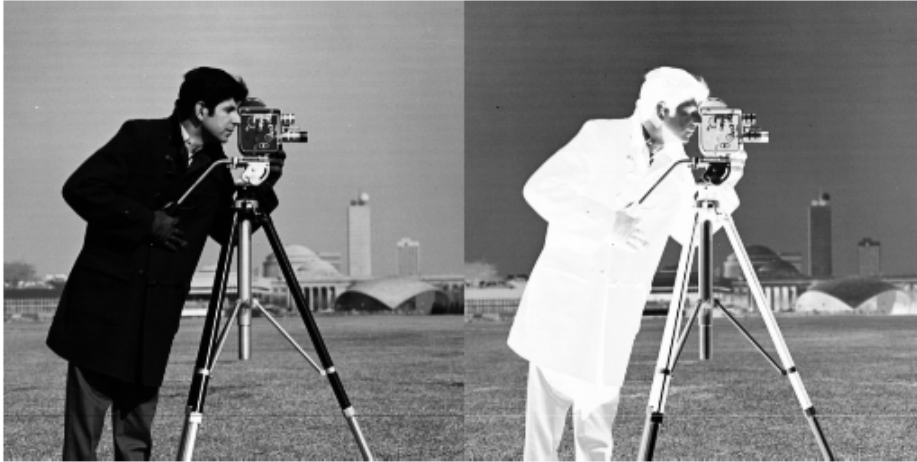
Reverse Black and White in a Binary Image

```
bw = imread('text.png');  
bw2 = imcomplement(bw);  
imshowpair(bw,bw2,'montage')
```



Create the Complement of an Intensity Image

```
I = imread('cameraman.tif');  
J = imcomplement(I);  
imshowpair(I,J,'montage')
```



Create the complement of an intensity image on a GPU

```
I = imread('glass.png');  
J = imcomplement(I);  
imshow(I), figure, imshow(J)
```

Tips

- If **IM** is a grayscale or RGB image of class **double**, you can use the expression **1 - IM** instead of this function. If **IM** is a binary image, you can use the expression **~IM** instead of this function.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- `imcomplement` does not support `int64` and `uint64` data types.

See Also

`imabsdiff` | `imadd` | `imdivide` | `imlincomb` | `immultiply` | `imsubtract`

Introduced before R2006a

imcontour

Create contour plot of image data

Syntax

```
imcontour(I)  
imcontour(I,n)  
imcontour(I,v)  
imcontour(x,y,...)  
imcontour(...,LineStyle)  
[C,handle] = imcontour(...)
```

Description

`imcontour(I)` draws a contour plot of the grayscale image `I`, automatically setting up the axes so their orientation and aspect ratio match the image.

`imcontour(I,n)` draws a contour plot of the grayscale image `I`, automatically setting up the axes so their orientation and aspect ratio match the image. `n` is the number of equally spaced contour levels in the plot; if you omit the argument, the number of levels and the values of the levels are chosen automatically.

`imcontour(I,v)` draws a contour plot of `I` with contour lines at the data values specified in vector `v`. The number of contour levels is equal to `length(v)`.

`imcontour(x,y,...)` uses the vectors `x` and `y` to specify the *x*- and *y*-axis limits.

`imcontour(...,LineStyle)` draws the contours using the line type and color specified by `LineStyle`. Marker symbols are ignored.

`[C,handle] = imcontour(...)` returns the contour matrix `C` and a handle to an `hggroup` object.

Class Support

The input image can be of class `uint8`, `uint16`, `int16`, `single`, `double`, or `logical`.

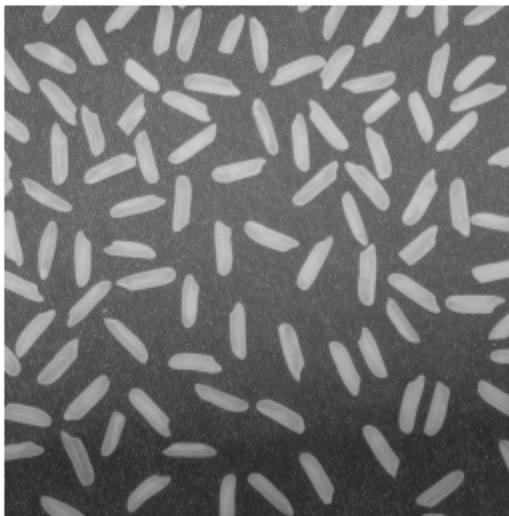
Examples

Create Contour Plot of Image Data

This example shows how to create a contour plot of an image.

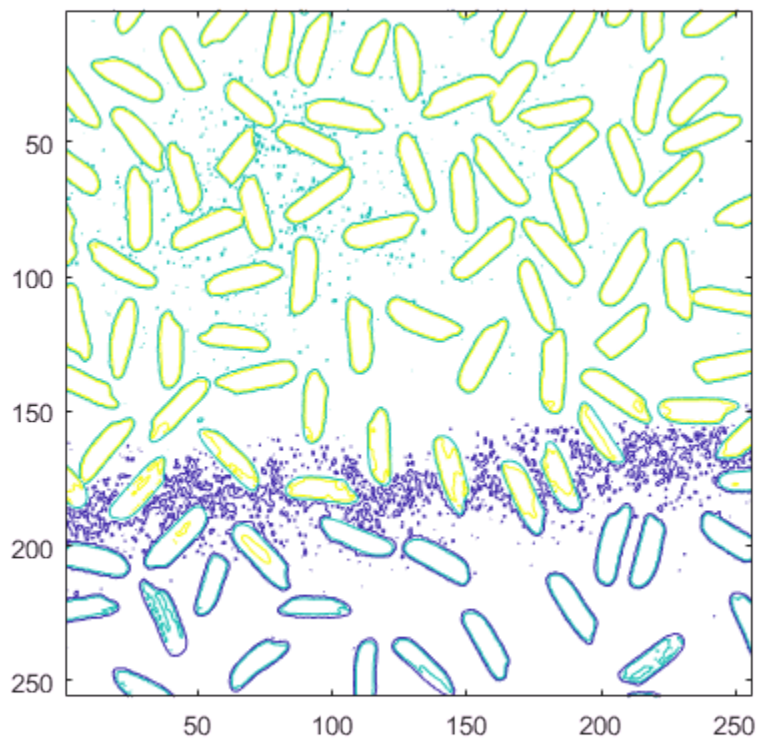
Read grayscale image and display it. The example uses an example image of grains of rice.

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



Create a contour plot of the image using `imcontour`.

```
figure;  
imcontour(I,3)
```



See Also

[clabel](#) | [contour](#) | [LineSpec](#)

Introduced before R2006a

imcontrast

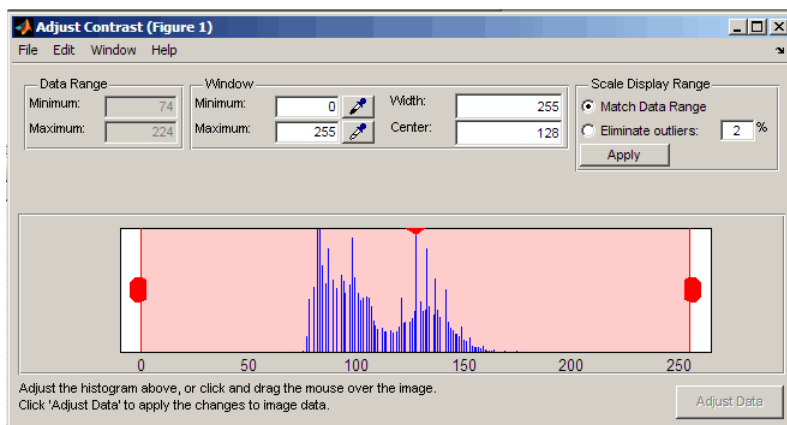
Adjust Contrast tool

Syntax

```
imcontrast
imcontrast(h)
hfigure = imcontrast(...)
```

Description

`imcontrast` creates an Adjust Contrast tool in a separate figure that is associated with the grayscale image in the current figure, called the target image. The Adjust Contrast tool is an interactive contrast and brightness adjustment tool, shown in the following figure, that you can use to adjust the black-to-white mapping used to display the image. When you use the tool, `imcontrast` adjusts the contrast of the displayed image by modifying the axes `CLim` property. To modify the actual pixel values in the target image, click the **Adjust Data** button. (This button is unavailable until you make a change to the contrast of the image.) For more information about using the tool, see “Tips” on page 1-696.



Note The Adjust Contrast tool can handle grayscale images of class `double` and `single` with data ranges beyond the default display range, which is `[0 1]`. For these images, `imcontrast` sets the histogram limits to fit the image data range, with padding at the upper and lower bounds.

`imcontrast(h)` creates the Adjust Contrast tool associated with the image specified by the handle `h`. `h` can be a handle to a figure, axes, uipanel, or image object. If `h` is an axes or figure handle, `imcontrast` uses the first image returned by `findobj(H, 'Type', 'image')`.

`hfigure = imcontrast(...)` returns a handle to the Adjust Contrast tool figure.

Examples

```
imshow('pout.tif')
imcontrast(gca)
```

Tips

The Adjust Contrast tool presents a scaled histogram of pixel values (overly represented pixel values are truncated for clarity). Dragging on the left red bar in the histogram display changes the minimum value. The minimum value (and any value less than the minimum) displays as black. Dragging on the right red bar in the histogram changes the maximum value. The maximum value (and any value greater than the maximum) displays as white. Values in between the red bars display as intermediate shades of gray.

Together the minimum and maximum values create a "window". Stretching the window reduces contrast. Shrinking the window increases contrast. Changing the center of the window changes the brightness of the image. It is possible to manually enter the minimum, maximum, width, and center values for the window. Changing one value automatically updates the other values and the image.

For more information about using the Adjust Contrast tool, see “Adjust Image Contrast In Image Viewer App”.

Window/Level Interactivity

Clicking and dragging the mouse within the target image interactively changes the image's window values. Dragging the mouse horizontally from left to right changes the window width (i.e., contrast). Dragging the mouse vertically up and down changes the window center (i.e., brightness). Holding down the **Ctrl** key before clicking and dragging the mouse accelerates the rate of change; holding down the **Shift** key before clicking and dragging the mouse slows the rate of change. Keys must be pressed before clicking and dragging.

See Also

`imadjust` | `imtool` | `stretchlim`

Introduced before R2006a

imcrop

Crop image

Syntax

```
I2 = imcrop
I2 = imcrop(I)
X2 = imcrop(X,map)
___ = imcrop(obj)

I2 = imcrop(I,rect)
X2 = imcrop(X,map,rect)
___ = imcrop(XData,YData,___)
[___,rect2] = imcrop(___)
[XData2,YData2,___] = imcrop(___)
```

Description

`I2 = imcrop` creates an interactive image cropping tool associated with the image displayed in the current figure, called the target image. The Crop Image tool is a moveable, resizable rectangle that you can position over the image and perform the crop operation interactively using the mouse. For more information about using the Crop Image tool, see “Interactive Behavior” on page 1-707. `imcrop` returns the cropped image, `I2`. With this syntax and the other interactive syntaxes, the Crop Image tool blocks the MATLAB command line until you complete the operation.

`I2 = imcrop(I)` displays the image `I` in a figure window and creates a cropping tool associated with that image. `I` can be a grayscale image, a truecolor image, or a logical array. `imcrop` returns the cropped image, `I2`.

`X2 = imcrop(X,map)` displays the indexed image `X` in a figure using the colormap `map`, and creates a cropping tool associated with that image. After you crop the image, `imcrop` returns the cropped indexed image, `X2`.

`___ = imcrop(obj)` creates a cropping tool associated with the object `obj`. `obj` can be an image, axes, uipanel, or figure. If `obj` is an axes, uipanel, or figure, the cropping tool acts on the first image found in the container object.

`I2 = imcrop(I,rect)` crops the image `I`. `rect` is a four-element position vector of the form `[xmin ymin width height]` that specifies the size and position of the crop rectangle. `imcrop` returns the cropped image, `I2`.

`X2 = imcrop(X,map,rect)` crops the indexed image `X`. `map` specifies the colormap used with `X`. `rect` is a four-element position vector `[xmin ymin width height]` that specifies the size and position of the cropping rectangle. `imcrop` returns the cropped indexed image, `X2`.

`___ = imcrop(XData,YData, ___)` specifies a nondefault spatial coordinate system for the target image.

`[___,rect2] = imcrop(___)` returns the cropping rectangle in `rect2`, a four-element position vector, in addition to the cropped image.

`[XData2,YData2, ___] = imcrop(___)` returns two-element vectors that specify the `XData` and `YData` of the target image.

Examples


Crop Image Using Crop Image Interactive Tool

Read image into the workspace.

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

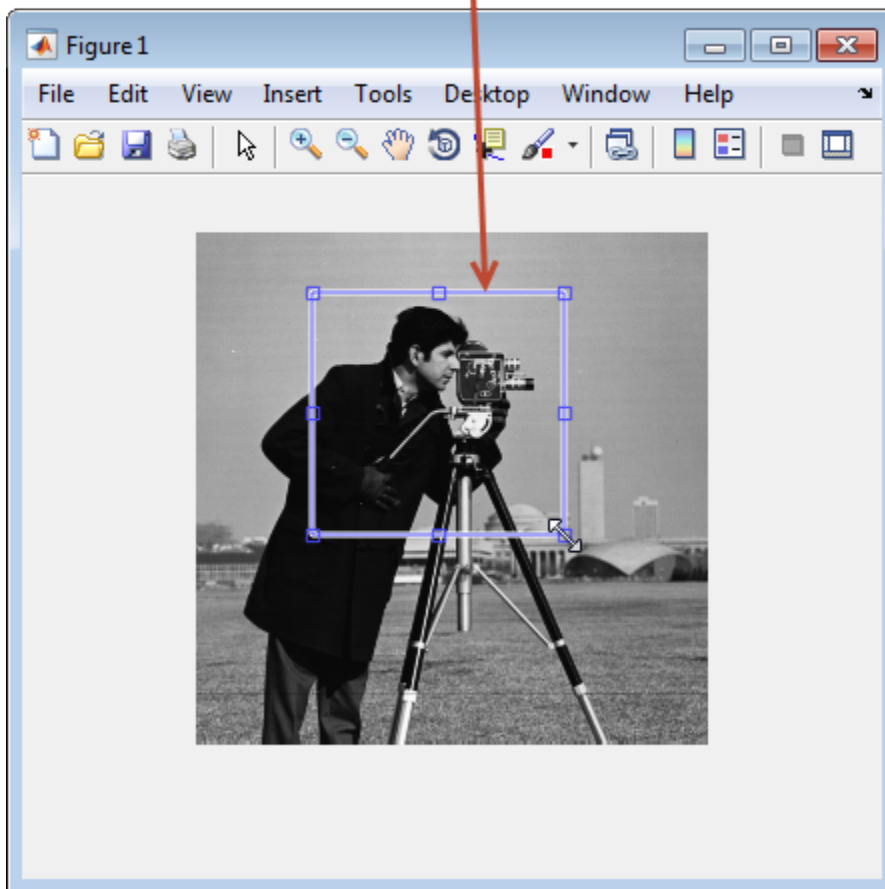
Open Crop Image tool associated with this image. Specify a variable in which to store the cropped image. The example includes the optional return value `rect` in which `imcrop` returns the four-element position vector of the rectangle you draw.

```
[I2, rect] = imcrop(I);
```

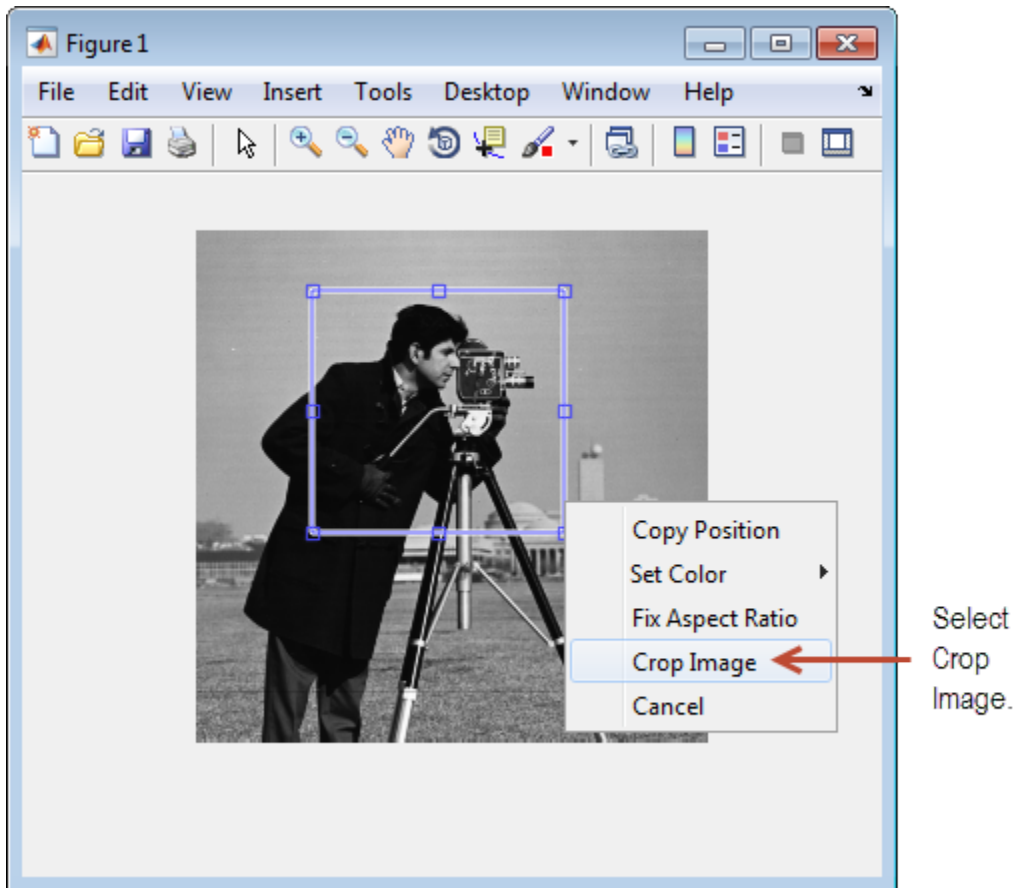
When you move the cursor over the image, it changes to a cross-hairs . The Crop Image tool blocks the MATLAB command line until you complete the operation.

Using the mouse, draw a rectangle over the portion of the image that you want to crop.

Draw crop rectangle.



Perform the crop operation by double-clicking in the crop rectangle or selecting Crop Image on the context menu.



The Crop Image tool returns the cropped area in the return variable, `I2`. The variable `rect` is the four-element position vector describing the crop rectangle you specified.

`whos`

Name	Size	Bytes	Class	Attributes
<code>I</code>	256x256	65536	uint8	
<code>I2</code>	121x126	15246	uint8	

rect 1x4 32 double

Crop Image By Specifying Crop Rectangle

Read image into the workspace.

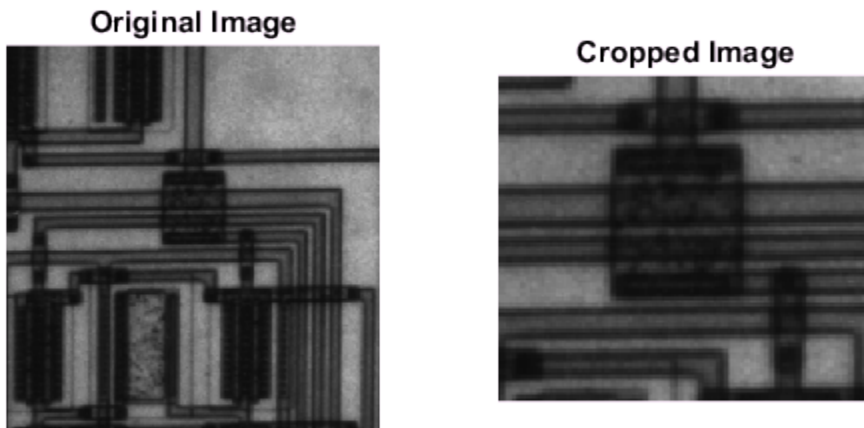
```
I = imread('circuit.tif');
```

Crop image, specifying crop rectangle.

```
I2 = imcrop(I,[75 68 130 112]);
```

Display original image and cropped image.

```
subplot(1,2,1)
imshow(I)
title('Original Image')
subplot(1,2,2)
imshow(I2)
title('Cropped Image')
```



Crop Indexed Image Specifying Crop Rectangle

Load indexed image with its associated map into the workspace.

```
load trees
```

Crop indexed image, specifying crop rectangle.

```
X2 = imcrop(X,map,[30 30 50 75]);
```

Display original image and cropped image.

```
subplot(1,2,1)
```

```
imshow(X,map)
title('Original Image')
subplot(1,2,2)
imshow(X2,map)
title('Cropped Image')
```

Original Image



Cropped Image



Input Arguments

I — Image to be cropped
real, nonsparse, numeric array

Image to be cropped, specified as a real, nonsparse, numeric array. With the syntaxes where you specify `rect` as an input argument, the input image can be logical or numeric, and must be real and nonsparse. With the other syntaxes, `imcrop` calls `imshow` and accepts whatever image classes `imshow` accepts. `I` can be logical, `uint8`, `uint16`, `int16`, `single`, or `double`. A truecolor image can be `uint8`, `int16`, `uint16`, `single`, or `double`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

X — Indexed image to be cropped

real, nonsparse numeric array

Indexed image to be cropped, specified as a real, nonsparse numeric array. With the syntaxes where you specify `rect` as an input argument, the input image can be logical or numeric, and must be real and nonsparse. With the other syntaxes, `imcrop` calls `imshow` and accepts any image class that `imshow` accepts. For an indexed image, `X` can be logical, `uint8`, `uint16`, `single`, or `double`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

map — Colormap associated with indexed image

m-by-3 numeric array

Colormap associated with indexed image, specified as an *m*-by-3 numeric array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

rect — Size and position of the crop rectangle

four-element position vector

Size and position of the crop rectangle, specified as a four-element position vector of the form `[xmin ymin width height]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

obj — Object containing the image to be cropped

image, axes, `uipanel`, or figure object

Object containing the image to be cropped, specified as an image, axes, `uipanel`, or figure object.

Data Types: `double`

XData — X limits of nondefault coordinate system

two-element vector

X limits of nondefault coordinate system, specified as a two-element vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

YData — Y limits of nondefault coordinate system

two-element vector

Y limits of nondefault coordinate system, specified as a two-element vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

I2 — Cropped image

real, nonsparse, numeric array

Cropped image, returned as a real, nonsparse, numeric array.

If you specify an input image, the output image has the same class as the input image. If you do not specify an input image, the output image generally has the same class as the input image. However, if the input image is `int16` or `single`, the output image is `double`.

X2 — Cropped indexed image

real, nonsparse, numeric array

Cropped image, returned as a real, nonsparse, numeric array.

If you specified an input image, the output image has the same class as the input image. If you do not specify an input image, the output image generally has the same class as the input image. However, if the input image is `int16` or `single`, the output image is `double`.

XData2 — X limits of nondefault coordinate system

two-element vector

X limits of nondefault coordinate system, returned as a two-element vector.

YData2 — Y limits of nondefault coordinate system

two-element vector

Y limits of nondefault coordinate system, returned as a two-element vector.


rect2 — Size and position of the crop rectangle

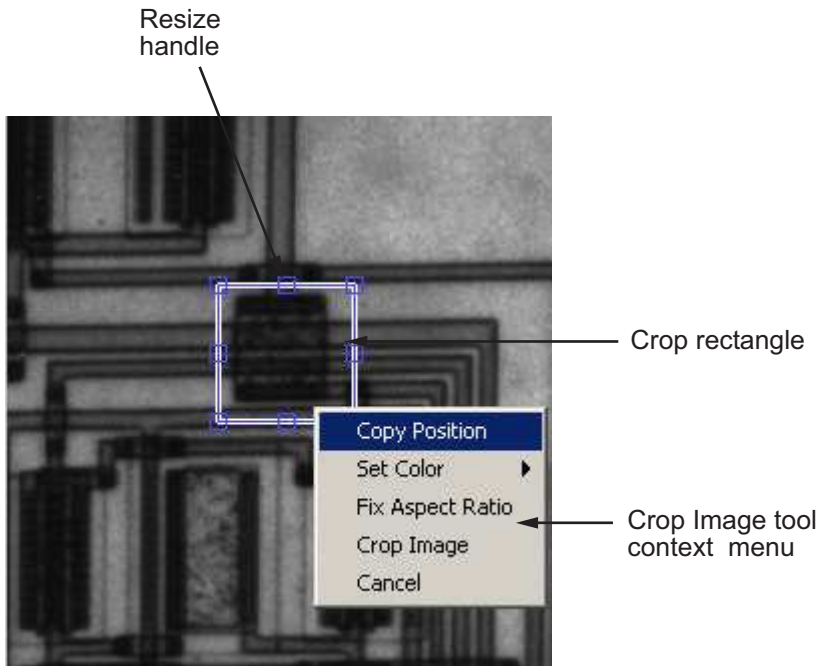
four-element position vector



Size and position of the crop rectangle, returned as a four-element position vector of the form [xmin ymin width height].

Additional Capabilities

Interactive Behavior

When the Crop Image tool is active in a figure, the pointer changes to cross hairs  when you move it over the target image. Using the mouse, you specify the crop rectangle by clicking and dragging the mouse. You can move or resize the crop rectangle using the mouse. When you are finished sizing and positioning the crop rectangle, create the cropped image by double-clicking the left mouse button. You can also choose **Crop Image** from the context menu. `imcrop` returns the cropped image **I2**. The following figure illustrates the Crop Image tool with the context menu displayed. For more information about the interactive capabilities of the tool, see the table that follows.



Interactive Behavior	Description
Deleting the Crop Image tool.	Press Backspace , Escape or Delete , or right-click inside the crop rectangle and select Cancel from the context menu. Note: If you delete the ROI, the function returns empty values.
Resizing the Crop Image tool.	Select any of the resize handles on the crop rectangle. The pointer changes to a double-headed arrow  . Click and drag the mouse to resize the crop rectangle.
Moving the Crop Image tool.	Move the pointer inside the boundary of the crop rectangle. The pointer changes to a fleur shape  . Click and drag the mouse to move the rectangle over the image.
Changing the color used to display the crop rectangle.	Right-click inside the boundary of the crop rectangle and select Set Color from the context menu.

Interactive Behavior	Description
Cropping the image.	Double-click the left mouse button or right-click inside the boundary of the crop rectangle and select Crop Image from the context menu.
Retrieving the coordinates of the crop rectangle.	Right-click inside the boundary of the crop rectangle and select Copy Position from the context menu. <code>imcrop</code> copies a four-element position vector (<code>[xmin ymin width height]</code>) to the clipboard.

Tips

- Because `rect` is specified in terms of spatial coordinates, the `width` and `height` elements of `rect` do not always correspond exactly with the size of the output image. For example, suppose `rect` is `[20 20 40 30]`, using the default spatial coordinate system. The upper left corner of the specified rectangle is the center of the pixel (20,20). The lower right corner of the rectangle is the center of the pixel (50,60). The resulting output image is 31-by-41, not 30-by-40, because it includes all pixels in the input image that are completely *or partially* enclosed by the rectangle.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- The interactive syntaxes are not supported, including:
 - `I2 = imcrop`
 - `I2 = imcrop(I)`
 - `X2 = imcrop(X,map)`

- `I2 = imcrop(h)`
- Indexed images are not supported, including the noninteractive syntax `X2 = imcrop(X,map,rect);`

See Also

See Also

`imrect` | `zoom`

Introduced before R2006a

imdilate

Dilate image

Syntax

```
IM2 = imdilate(IM,SE)
IM2 = imdilate(IM,NHOOD)
IM2 = imdilate( ____,PACKOPT)
IM2 = imdilate( ____,SHAPE)
gpuarrayIM2 = imdilate(gpuarrayIM, ____)
```

Description

`IM2 = imdilate(IM,SE)` dilates the grayscale, binary, or packed binary image `IM`, returning the dilated image, `IM2`. The argument `SE` is a structuring element object, or array of structuring element objects, returned by the `strel` or `offsetstrel` function.

If `IM` is logical, the structuring element must be flat and `imdilate` performs binary dilation. Otherwise, `imdilate` performs grayscale dilation. If `SE` is an array of structuring element objects, `imdilate` performs multiple dilations of the input image, using each structuring element in succession.

`IM2 = imdilate(IM,NHOOD)` dilates the image `IM`, where `NHOOD` is a matrix of 0's and 1's that specifies the structuring element neighborhood. This is equivalent to the syntax `imdilate(IM,strel(NHOOD))`. The `imdilate` function determines the center element of the neighborhood by `floor((size(NHOOD)+1)/2)`.

`IM2 = imdilate(____,PACKOPT)` specifies whether `IM` is a packed binary image. `PACKOPT` can have either of the following values. Default value is enclosed in braces (`{}`).

Value	Description
'ispacked'	<code>IM</code> is treated as a packed binary image as produced by <code>bwpack</code> . <code>IM</code> must be a 2-D <code>uint32</code> array and <code>SE</code> must be a flat 2-D structuring element. If the value of <code>PACKOPT</code> is 'ispacked', <code>PADOPT</code> must be 'same'.

Value	Description
{ 'notpacked' }	IM is treated as a normal array.

IM2 = imdilate(____, SHAPE) specifies the size of the output image. SHAPE can have either of the following values. Default value is enclosed in braces ({}).

Value	Description
{ 'same' }	Make the output image the same size as the input image. If the value of PACKOPT is 'ispacked', SHAPE must be 'same'.
'full'	Compute the full dilation.

gpuarrayIM2 = imdilate(gpuarrayIM, ____) performs the operation on a graphics processing unit (GPU), where gpuarrayIM is a gpuArray that contains a grayscale or binary image. gpuarrayIM2 is a gpuArray of the same class as the input image. Note that the PACKOPT syntax is not supported on a GPU. This syntax requires the Parallel Computing Toolbox.

Class Support

IM can be logical or numeric and must be real and nonsparse. It can have any dimension. If IM is logical, SE must be flat. The output has the same class as the input. If the input is packed binary, then the output is also packed binary.

gpuarrayIM must be a gpuArray of type uint8 or logical. When used with a gpuarray, the structuring element must be flat and two-dimensional. The output has the same class as the input.

Examples

Dilate Image with Vertical Line Structuring Element

Read a binary image into the workspace.

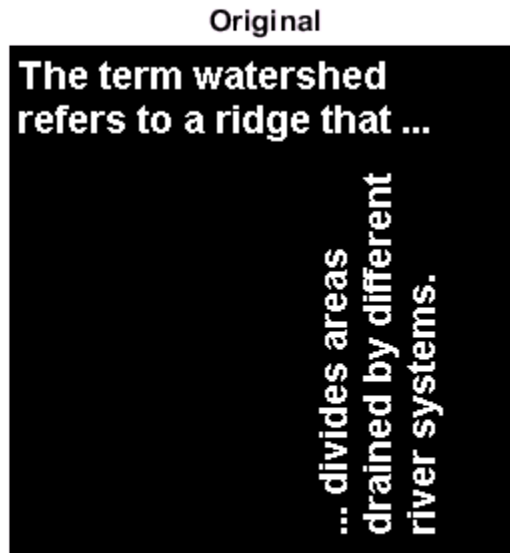
```
BW = imread('text.png');
```

Create a vertical line shaped structuring element.

```
se = strel('line',11,90);
```

Dilate the image with a vertical line structuring element and compare the results.

```
BW2 = imdilate(BW,se);  
imshow(BW), title('Original')
```



```
figure, imshow(BW2), title('Dilated')
```



Dilate Grayscale Image with Rolling Ball

Read grayscale image into the workspace.

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

Create a nonflat ball-shaped structuring element.

```
se = offsetstrel('ball',5,5);
```

Dilate the image.

```
dilatedI = imdilate(originalI,se);
```

Display the original image and the dilated image.

```
figure  
imshow(originalI)
```




```
figure  
imshow(dilatedI)
```



Determine Domain of Composition of Structuring Elements

Create two flat, line-shaped structuring elements, one at 0 degrees and the other at 90 degrees.

```
se1 = strel('line',3,0)
```

```
se1 =  
strel is a line shaped structuring element with properties:
```

```
    Neighborhood: [1 1 1]  
    Dimensionality: 2
```

```
se2 = strel('line',3,90)
```

```
se2 =
```

strel is a line shaped structuring element with properties:

```
Neighborhood: [3×1 logical]
Dimensionality: 2
```

Dilate the scalar value 1 with both structuring elements in sequence, using the 'full' option.

```
composition = imdilate(1,[se1 se2], 'full')
```

```
composition =
```

```
  1    1    1
  1    1    1
  1    1    1
```

Dilate Points in 3D Space Using Spherical Structuring Elements

Create a logical 3D volume with two points.

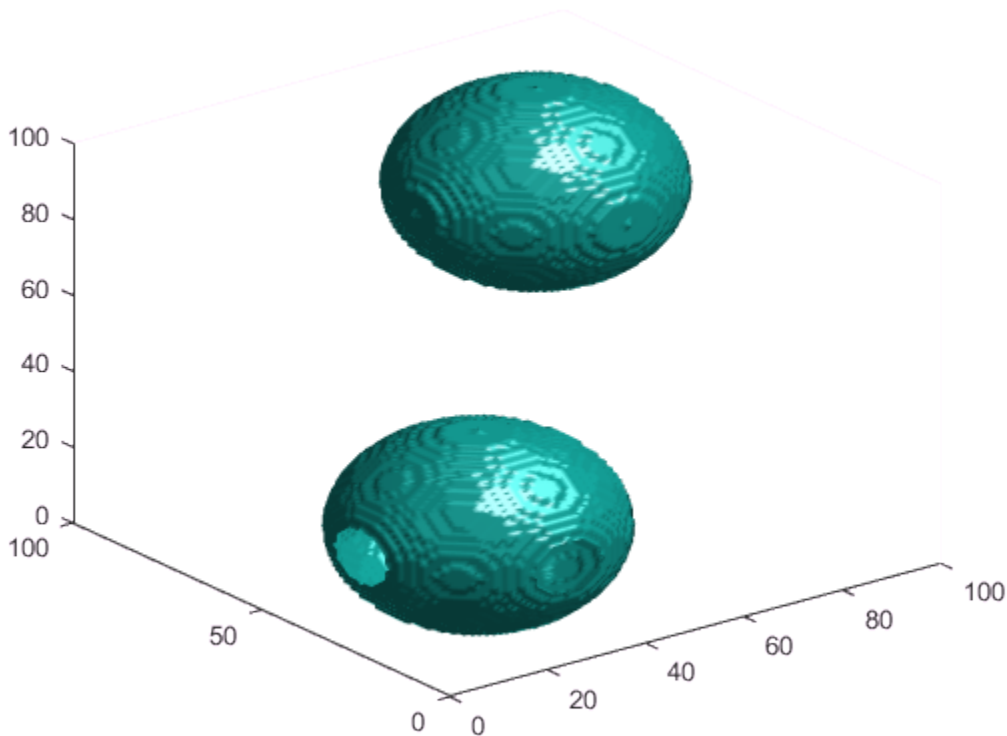
```
BW = false(100,100,100);
BW(25,25,25) = true;
BW(75,75,75) = true;
```

Dilate the 3D volume using a spherical structuring element.

```
se = strel('sphere',25);
dilatedBW = imdilate(BW,se);
```

Visualize the dilated image volume.

```
figure
isosurface(dilatedBW)
```



Dilate a Binary Image with a Vertical Line Structuring Element on a GPU

Read a binary image.

```
originalBW = imread('text.png');
```

Create a structuring element.

```
se = strel('line',11,90);
```

Dilate the image with a vertical line structuring element and compare the results. Note how the example passes the image to the `gpuArray` function as it passes it to `imdilate`.

```
dilatedBW = imdilate(gpuArray(originalBW),se);
```

```
figure, imshow(originalBW), figure, imshow(dilatedBW)
```

Dilate a Grayscale Image with a Disk Structuring Element on a GPU

Read a grayscale image.

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

Create a structuring element.

```
se = strel('disk',5);
```

Dilate a grayscale image with a rolling ball structuring element.

```
dilatedI = imdilate(gpuArray(originalI),se);
figure, imshow(originalI), figure, imshow(dilatedI)
```

Definitions

Binary Dilation

The *binary dilation* of A by B , denoted $A \oplus B$, is defined as the set operation:

$$A \oplus B = \left\{ z \mid \left(\hat{B} \right)_z \cap A \neq \emptyset \right\},$$

where \hat{B} is the reflection of the structuring element B . In other words, it is the set of pixel locations z , where the reflected structuring element overlaps with foreground pixels in A when translated to z . Note that some people use a definition of dilation in which the structuring element is not reflected.

For more information about binary dilation, see [1].

Gray-Scale Dilation

In the general form of *gray-scale dilation*, the structuring element has a height. The gray-scale dilation of $A(x,y)$ by $B(x,y)$ is defined as:

$$(A \oplus B)(x, y) = \max\{A(x - x', y - y') + B(x', y') \mid (x', y') \in D_B\},$$

where D_B is the domain of the structuring element B and $A(x, y)$ is assumed to be $-\infty$ outside the domain of the image. To create a structuring element with nonzero height values, use the syntax `strel(nhood, height)`, where `height` gives the height values and `nhood` corresponds to the structuring element domain, D_B .

Most commonly, gray-scale dilation is performed with a flat structuring element ($B(x, y) = 0$). Gray-scale dilation using such a structuring element is equivalent to a local-maximum operator:

$$(A \oplus B)(x, y) = \max\{A(x - x', y - y') \mid (x', y') \in D_B\}.$$

All of the `strel` syntaxes except for `strel(nhood, height)`, `strel('arbitrary', nhood, height)`, and `strel('ball', ...)` produce flat structuring elements.

Algorithms

`imdilate` automatically takes advantage of the decomposition of a structuring element object (if it exists). Also, when performing binary dilation with a structuring element object that has a decomposition, `imdilate` automatically uses binary image packing to speed up the dilation.

Dilation using bit packing is described in [3].

References

- [1] Gonzalez, R. C., R. E. Woods, and S. L. Eddins, *Digital Image Processing Using MATLAB*, Gatesmark Publishing, 2009.
- [2] Haralick, R. M., and L. G. Shapiro, *Computer and Robot Vision*, Vol. I, Addison-Wesley, 1992, pp. 158-205.
- [3] van den Boomgard, R, and R. van Balen, "Methods for Fast Morphological Image Transforms Using Bitmapped Images," *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, Vol. 54, Number 3, pp. 254-258, May 1992.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- The input image, **IM**, must be 2-D or 3-D.
- The structuring element argument **SE** must be a single element—arrays of structuring elements are not supported. To obtain the same result as that obtained using an array of structuring elements, call the function sequentially.
- When the target is **MATLAB Host Computer**, the **PACKOPT** and **SHAPE** arguments must be compile-time constants. When the target is any other platform, the **PACKOPT** syntax is not supported.

See Also

See Also

`strel` | `offsetstrel` | `bwpack` | `bwunpack` | `conv2` | `filter2` | `gpuArray` | `imclose` | `imerode` | `imopen`

Introduced before R2006a

imshow

Display Range tool

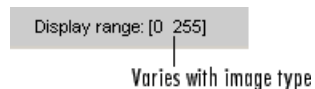
Syntax

```
imshow  
imshow(h)  
imshow(hparent,himage)  
hpanel = imshow(...)
```

Description

`imshow` creates a Display Range tool in the current figure. The Display Range tool shows the display range of the intensity image or images in the figure.

The tool is a uipanel object, positioned in the lower-right corner of the figure. It contains the label `Display range:` followed by the display range values for the image, as shown in the following figure.



For an indexed, truecolor, or binary image, the display range is not applicable and is set to empty (`[]`).

`imshow(h)` creates a Display Range tool in the figure specified by the handle `h`, where `h` is a handle to an image, axes, uipanel, or figure object. Axes, uipanel, or figure objects must contain at least one image object.

`imshow(hparent,himage)` creates a Display Range tool in `hparent` that shows the display range of `himage`. `himage` is a handle to an image or an array of image handles. `hparent` is a handle to the figure or uipanel object that contains the display range tool.

`hpanel = imshow(...)` returns a handle to the Display Range tool uipanel.

Note

The Display Range tool can work with multiple images in a figure. When the pointer is not in an image in a figure, the Display Range tool displays [black white].

Examples

Display an image and include the Display Range tool.

```
imshow('bag.png');  
imdisplayrange;
```

Import a 16-bit DICOM image and display it with its default range and scaled range in the same figure.

```
dcm = dicomread('CT-MONO2-16-ankle.dcm');  
subplot(1,2,1), imshow(dcm);  
subplot(1,2,2), imshow(dcm,[]);  
imdisplayrange;
```

See also

`imtool`

Introduced before R2006a

imdistline

Distance tool

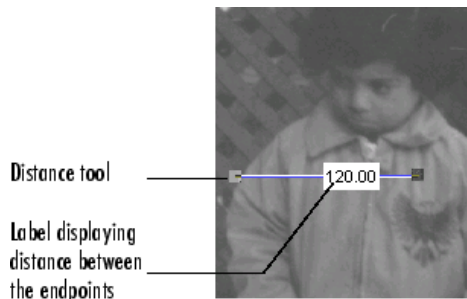
Syntax


```
h = imdistline
h = imdistline(hparent)
h = imdistline(..., x, y)
```


Description

`h = imdistline` creates a Distance tool on the current axes. The function returns `h`, a handle to an `imdistline` object.

The Distance tool is a draggable, resizable line, superimposed on an axes, that measures the distance between the two endpoints of the line. The Distance tool displays the distance in a text label superimposed over the line. The tools specifies the distance in data units determined by the `XData` and `YData` properties, which is pixels, by default. The following figure shows a Distance tool on an axes.



To move the Distance tool, position the pointer over the line, the shape changes to the fleur, . Click and drag the line using the mouse. To resize the Distance tool, move the pointer over either of the endpoints of the line, the shape changes to the pointing

finger, . Click and drag the endpoint of the line using the mouse. The line also supports a context menu that allows you to control various aspects of its functioning and appearance. See Context Menu for more information. Right-click the line to access the context menu.

`h = imdistline(hparent)` creates a draggable Distance tool on the object specified by `hparent`. `hparent` specifies the Distance tool's parent, which is typically an axes object, but can also be any other object that can be the parent of an `hgroup` object.

`h = imdistline(..., x, y)` creates a Distance tool with endpoints located at the locations specified by the vectors `x` and `y`, where `x = [x1 x2]` and `y = [y1 y2]`.

Context Menu

Distance Tool Behavior	Context Menu Item
Export endpoint and distance data to the workspace	Select Export to Workspace from the context menu.
Toggle the distance label on/off.	Select Show Distance Label from the context menu.
Specify horizontal and vertical drag constraints	Select Constrain Drag from the context menu.
Change the color used to display the line.	Select Set Color from the context menu.
Delete the Distance tool object	Select Delete from the context menu.

API Functions

The Distance tool contains a structure of function handles, called an API, that can be used to retrieve distance information and control other aspects of Distance tool behavior. To retrieve this structure from the Distance tool, use the `iptgetapi` function, where `h` is a handle to the Distance tool.

```
api = iptgetapi(h)
```

The following table lists the functions in the API, with their syntax and brief descriptions.

Function	Description
<code>getDistance</code>	Returns the distance between the endpoints of the Distance tool. <code>dist = getDistance()</code> The value returned is in data units determined by the <code>XData</code> and <code>YData</code> properties, which is pixels, by default.
<code>getAngleFromHorizontal</code>	Returns the angle in degrees between the line defined by the Distance tool and the horizontal axis. The angle returned is between 0 and 180 degrees. (For information about how this angle is calculated, see “Tips” on page 1-730.) <code>angle = getAngleFromHorizontal()</code>
<code>setLabelTextFormatter</code>	Sets the format used in displaying the distance label. <code>setLabelTextFormatter(str)</code> <code>str</code> is a character array specifying a format in the form expected by <code>sprintf</code> .
<code>getLabelTextFormatter</code>	Returns a character array specifying the format used to display the distance label. <code>str = getLabelTextFormatter()</code> <code>str</code> is in a format expected by <code>sprintf</code> .
<code>setLabelVisible</code>	Sets visibility of Distance tool text label. <code>setLabelVisible(h,TF)</code> <code>h</code> is the Distance tool. <code>TF</code> is a logical scalar. When the distance label is visible, <code>TF</code> is true. When the distance label is invisible, <code>TF</code> is false.
<code>getLabelVisible</code>	Gets visibility of Distance tool text label. <code>TF = getLabelVisible(h)</code>

Function	Description
	h is the Distance tool. TF is a logical scalar. When TF is true, the distance label is visible. When TF is false, the distance label is invisible.
setPosition	Sets the endpoint positions of the Distance tool. setPosition(X,Y) setPosition([X1 Y1; X2 Y2])
getPosition	Returns the endpoint positions of the Distance tool. pos = getPosition() pos is a 2-by-2 array [X1 Y1; X2 Y2].
delete	Deletes the Distance tool associated with the API. delete()
setColor	Sets the color used to draw the Distance tool. setColor(new_color) new_color can be a three-element vector specifying an RGB triplet, or the long or short names of a predefined color, such as 'white' or 'w'. For a complete list of these predefined colors and their short names, see ColorSpec.
getColor	Gets the color used to draw the ROI object h. color = getColor(h) color is a three-element vector that specifies an RGB triplet.

Function	Description
<p><code>addNewPositionCallback</code></p>	<p>Adds the function handle <code>fcn</code> to the list of new-position callback functions.</p> <pre>id = addNewPositionCallback(fcn)</pre> <p>Whenever the Distance tool changes its position, each function in the list is called with the following syntax.</p> <pre>fcn(pos)</pre> <p><code>pos</code> is a 2-by-2 array [X1 Y1; X2 Y2].</p> <p>The return value, <code>id</code>, is used only with <code>removeNewPositionCallback</code>.</p>
<p><code>removeNewPositionCallback</code></p>	<p>Removes the corresponding function from the new-position callback list.</p> <pre>removeNewPositionCallback(id)</pre> <p><code>id</code> is the identifier returned by <code>addNewPositionCallback</code>.</p>
<p><code>setPositionConstraintFcn</code></p>	<p>Sets the position constraint function to be the specified function handle, <code>fcn</code>. Use this function to control where the Distance tool can be moved and resized.</p> <pre>setPositionConstraintFcn(fcn)</pre> <p>Whenever the Distance tool is moved or resized because of a mouse drag, the constraint function is called using the following syntax.</p> <pre>constrained_position = fcn(new_position)</pre> <p><code>new_position</code> is a 2-by-2 array [X1 Y1; X2 Y2].</p>
<p><code>getPositionConstraintFcn</code></p>	<p>Returns the function handle of the current drag constraint function.</p> <pre>fcn = getDragConstraintFcn()</pre>

Examples

Example 1

Insert a Distance tool into an image. Use `makeConstrainToRectFcn` to specify a drag constraint function that prevents the Distance tool from being dragged outside the extent of the image. Right-click the Distance tool and explore the context menu options.

```
figure, imshow('pout.tif');
h = imdistline(gca);
api = iptgetapi(h);
fcn = makeConstrainToRectFcn('imline',...
    get(gca,'XLim'),get(gca,'YLim'));
api.setDragConstraintFcn(fcn);
```

Example 2

Position endpoints of the Distance tool at the specified locations.

```
close all, imshow('pout.tif');
h = imdistline(gca,[10 100],[10 100]);
```

Delete the Distance tool.

```
api = iptgetapi(h);
api.delete();
```

Example 3

Use the Distance tool with `XData` and `YData` of associated image in non-pixel units. This example requires the `boston.tif` image from the Mapping Toolbox software, which includes material copyrighted by GeoEye™, all rights reserved.

```
start_row = 1478;
end_row = 2246;
meters_per_pixel = 1;
rows = [start_row meters_per_pixel end_row];
start_col = 349;
end_col = 1117;
cols = [start_col meters_per_pixel end_col];
img = imread('boston.tif','PixelRegion',{rows,cols});
figure;
hImg = imshow(img);
title('1 meter per pixel');
```

```
% Specify initial position of distance tool on Harvard Bridge.
hline = imdistline(gca,[271 471],[108 650]);
api = iptgetapi(hline);
api.setLabelTextFormatter('%02.0f meters');

% Repeat process but work with a 2 meter per pixel sampled image. Verify
% that the same distance is obtained.
meters_per_pixel = 2;
rows = [start_row meters_per_pixel end_row];
cols = [start_col meters_per_pixel end_col];
img = imread('boston.tif','PixelRegion',{rows,cols});
figure;
hImg = imshow(img);
title('2 meters per pixel');

% Convert XData and YData to meters using conversion factor.
XDataInMeters = get(hImg,'XData')*meters_per_pixel;
YDataInMeters = get(hImg,'YData')*meters_per_pixel;

% Set XData and YData of image to reflect desired units.
set(hImg,'XData',XDataInMeters,'YData',YDataInMeters);
set(gca,'XLim',XDataInMeters,'YLim',YDataInMeters);

% Specify initial position of distance tool on Harvard Bridge.
hline = imdistline(gca,[271 471],[108 650]);
api = iptgetapi(hline);
api.setLabelTextFormatter('%02.0f meters');
```

Tips

If you use `imdistline` with an axes that contains an image object, and do not specify a drag constraint function, users can drag the line outside the extent of the image. When used with an axes created by the `plot` function, the axes limits automatically expand to accommodate the movement of the line.

To understand how `imdistline` calculates the angle returned by `getAngleToHorizontal`, draw an imaginary horizontal vector from the bottom endpoint of the distance line, extending to the right. The value returned by `getAngleToHorizontal` is the angle from this horizontal vector to the distance line, which can range from 0 to 180 degrees.

See Also

`iptgetapi` | `makeConstrainToRectFcn`

Introduced before R2006a

imdivide

Divide one image into another or divide image by constant

Syntax

```
Z = imdivide(X,Y)
```

Description

`Z = imdivide(X,Y)` divides each element in the array `X` by the corresponding element in array `Y` and returns the result in the corresponding element of the output array `Z`. `X` and `Y` are real, nonsparse numeric arrays with the same size and class, or `Y` can be a scalar `double`. `Z` has the same size and class as `X` and `Y`, unless `X` is logical, in which case `Z` is `X = uint8([255 0 75; 44 225 100]); Y = uint8([50 50 50; 50 50 50]); Z = imdivide(X,Y)double`.

If `X` is an integer array, elements in the output that exceed the range of integer type are truncated, and fractional values are rounded.

If `X` and `Y` are numeric arrays of the same size and class, you can use the expression `X./Y` instead of `imdivide`.

Examples

Divide Two uint8 Arrays

This example shows how to divide two `uint8` arrays.

```
X = uint8([ 255 0 75; 44 225 100]);  
Y = uint8([ 50 50 50; 50 50 50 ]);
```

Divide each element in `X` by the corresponding element in `Y`. Note that fractional values greater than or equal to 0.5 are rounded up to the nearest integer.

```
Z = imdivide(X,Y)
```

```
Z = 2×3 uint8 matrix
```

```
5   0   2  
1   5   2
```

Divide each element in Y by the corresponding element in X. Note that when dividing by zero, the output is truncated to the range of the integer type.

```
W = imdivide(Y,X)
```

```
W = 2×3 uint8 matrix
```

```
0   255   1  
1     0   1
```

Divide Image Background

Read a grayscale image into the workspace.

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

Estimate the background.

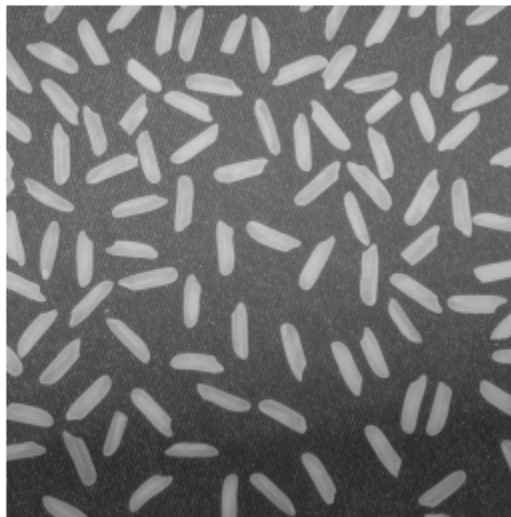
```
background = imopen(I, strel('disk', 15));
```

Divide out the background from the image.

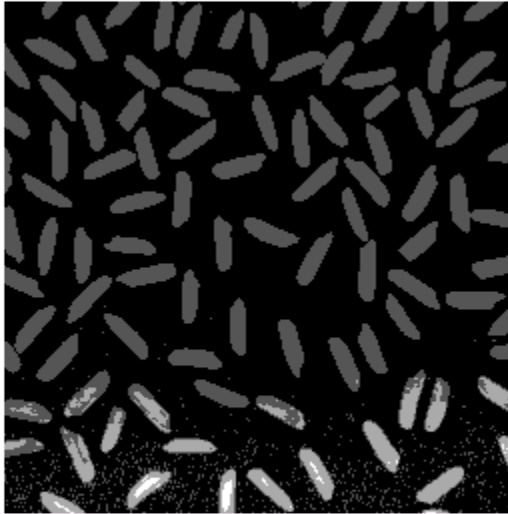
```
J = imdivide(I, background);
```

Display the original image and the processed image.

```
imshow(I)
```



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



Divide an Image by a Constant Factor

Read an image into the workspace.

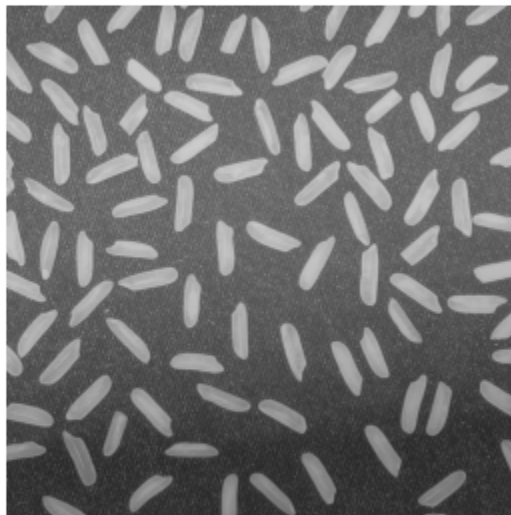
```
I = imread('rice.png');
```

Divide each value of the image by a constant factor of 2.

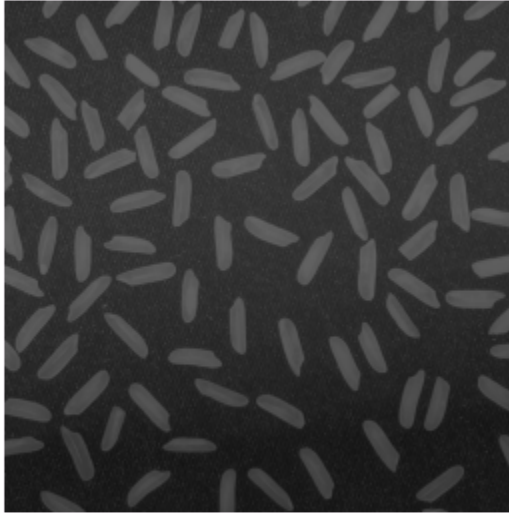
```
J = imdivide(I,2);
```

Display the original image and the processed image.

```
imshow(I)
```



```
figure  
imshow(J)
```



See Also

`imabsdiff` | `imadd` | `imcomplement` | `imlincomb` | `immultiply` | `imsubtract`

Introduced before R2006a

imellipse

Create draggable ellipse

Syntax

```
h = imellipse
h = imellipse(hparent)
h = imellipse(hparent, position)
h = imellipse(...,param1, val1, ...)
```

Description

`h = imellipse` begins interactive placement of an ellipse on the current axes. The function returns `h`, a handle to an `imellipse` object. The ellipse has a context menu associated with it that controls aspects of its appearance and behavior—see “Interactive Behavior” on page 1-738. Right-click on the line to access this context menu.

`h = imellipse(hparent)` begins interactive placement of an ellipse on the object specified by `hparent`. `hparent` specifies the HG parent of the ellipse graphics, which is typically an axes but can also be any other object that can be the parent of an `hggroup`.


`h = imellipse(hparent, position)` creates a draggable ellipse on the object specified by `hparent`. `position` is a four-element vector that specifies the initial location of the ellipse in terms of a bounding rectangle. `position` has the form `[xmin ymin width height]`.

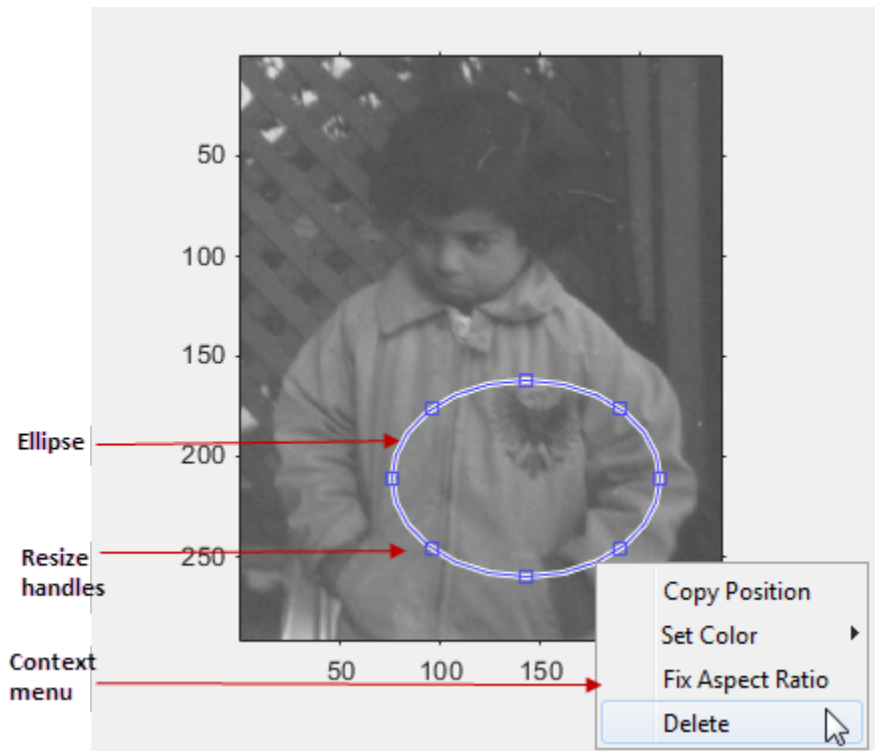
`h = imellipse(...,param1, val1, ...)` creates a draggable ellipse, specifying parameters and corresponding values that control the behavior of the ellipse. The following table lists the parameter available. Parameter names can be abbreviated, and case does not matter.

Parameter	Value
'PositionConstraintFcn'	Function handle that is called whenever the mouse is dragged. You can use this to control where the ellipse may be dragged. See the help for the



Parameter	Value
	setPositionConstraintFcn method for information about valid function handles.

Interactive Behavior

When you call `imellipse` with an interactive syntax, the pointer changes to a cross hairs  when over an image. Click and drag the mouse to specify the size and position of the ellipse. The ellipse also supports a context menu that you can use to control aspects of its appearance and behavior. The following figure illustrates the ellipse with its context menu.



The following table lists the interactive behavior supported by `imellipse`.

Interactive Behavior	Description
Moving the entire ellipse.	Move the pointer inside the ellipse. The pointer changes to a fleur shape  . Click and drag the mouse to move the ellipse.
Resizing the ellipse.	Move the pointer over a resizing handle on the ellipse. The pointer changes to a double-ended arrow shape  . Click and drag the mouse to resize the ellipse.
Changing the color used to display the ellipse.	Move the pointer inside the ellipse. Right-click and select Set Color from the context menu.
Retrieving the current position of the ellipse.	Move the pointer inside the ellipse. Right-click and select Copy Position from the context menu. <code>imellipse</code> copies a four-element position vector [xmin ymin width height] to the clipboard.
Preserving the current aspect ratio of the ellipse during resizing.	Move the pointer inside the ellipse. Right-click and select Fix Aspect Ratio from the context menu.
Deleting the ellipse	Move the pointer inside the ellipse. Right-click and select Delete from the context menu. To remove this option from the context menu, set the <code>Deletable</code> property to false: <code>h = imellipse(); h.Deletable = false;</code>

Methods

Each `imellipse` object supports a number of methods. Type methods `imellipse` to see a complete list.

addNewPositionCallback — Add new-position callback to ROI object

See `imroi` for information.

createMask — Create mask within image

See `imroi` for information.

delete — Delete ROI object

See `imroi` for information.

getColor — Get color used to draw ROI object

See `imroi` for information.

getPosition — Return current position of ellipse

See `imrect` for information.

getPositionConstraintFcn — Return function handle to current position constraint function

See `imroi` for information.

getVertices — Return vertices on perimeter of ellipse

`vert = getVertices(h)` returns a set of vertices which lie along the perimeter of the ellipse `h`. `vert` is a N-by-2 array.

removeNewPositionCallback — Remove new-position callback from ROI object.

See `imroi` for information.

resume — Resume execution of MATLAB command line

See `imroi` for information.

setColor — Set color used to draw ROI object

See `imroi` for information.

setConstrainedPosition — Set ROI object to new position

See `imroi` for information.

setFixedAspectRatioMode — Control whether aspect ratio preserved during resize

See `imrect` for information.

setPosition — Set ellipse to new position

See `imrect` for information.

setPositionConstraintFcn — Set position constraint function of ROI object.

See `imroi` for information.

setResizable — Set resize behavior of ellipse

See `imrect` for information.

wait — Block MATLAB command line until ROI creation is finished

`vert = wait(h)` blocks execution of the MATLAB command line until you finish positioning the ROI object `h`. You indicate completion by double-clicking on the ROI object. The returned vertices, `vert`, is of the form returned by the `getVertices` method.

Examples

Example 1

Create an ellipse, using callbacks to display the updated position in the title of the figure. The example illustrates using the `makeConstrainToRectFcn` to keep the ellipse inside the original `xlim` and `ylim` ranges.

```
figure, imshow('cameraman.tif');
h = imellipse(gca, [10 10 100 100]);
addNewPositionCallback(h,@(p) title(mat2str(p,3)));
fcn = makeConstrainToRectFcn('imellipse',get(gca,'XLim'),get(gca,'YLim'));
setPositionConstraintFcn(h,fcn);
```

Example 2

Interactively place an ellipse by clicking and dragging. Use `wait` to block the MATLAB command line. Double-click on the ellipse to resume execution of the MATLAB command line.

```
figure, imshow('pout.tif');
h = imellipse;
position = wait(h);
```

Tips

If you use `imellipse` with an axes that contains an image object, and do not specify a position constraint function, users can drag the ellipse outside the extent of the image and lose the ellipse. When used with an axes created by the `plot` function, the axes limits automatically expand to accommodate the movement of the ellipse.

See Also

`imfreehand` | `impoly` | `imrect` | `imroi` | `imline` | `impoint` | `iptgetapi` | `makeConstrainToRectFcn`

Introduced in R2007b

imerode

Erode image

Syntax

```
IM2 = imerode(IM,SE)
IM2 = imerode(IM,NHOOD)
IM2 = imerode( ____,PACKOPT,M)
IM2 = imerode( ____,SHAPE)
gpuarrayIM2 = imerode(gpuarrayIM, ____)
```

Description

`IM2 = imerode(IM,SE)` erodes the grayscale, binary, or packed binary image `IM`, returning the eroded image `IM2`. The argument `SE` is a structuring element object or array of structuring element objects returned by the `strel` or `offsetstrel` functions.

If `IM` is logical and the structuring element is flat, `imerode` performs binary erosion; otherwise it performs grayscale erosion. If `SE` is an array of structuring element objects, `imerode` performs multiple erosions of the input image, using each structuring element in `SE` in succession.

`IM2 = imerode(IM,NHOOD)` erodes the image `IM`, where `NHOOD` is an array of 0's and 1's that specifies the structuring element neighborhood. This is equivalent to the syntax `imerode(IM,strel(NHOOD))`. The `imerode` function determines the center element of the neighborhood by `floor((size(NHOOD)+1)/2)`.

`IM2 = imerode(____,PACKOPT,M)` specifies whether `IM` is a packed binary image and, if it is, provides the row dimension `M` of the original unpacked image. `PACKOPT` can have either of the following values. Default value is enclosed in braces (`{}`).

Value	Description
'ispacked'	<code>IM</code> is treated as a packed binary image as produced by <code>bwpack</code> . <code>IM</code> must be a 2-D <code>uint32</code> array and <code>SE</code> must be a flat 2-D structuring element.

Value	Description
{ 'notpacked' }	IM is treated as a normal array.

If `PACKOPT` is 'ispacked', you must specify a value for `M`.

`IM2 = imerode(____, SHAPE)` specifies the size of the output image. `SHAPE` can have either of the following values. Default value is enclosed in braces ({}).

Value	Description
{ 'same' }	Make the output image the same size as the input image. If the value of <code>PACKOPT</code> is 'ispacked', <code>SHAPE</code> must be 'same'.
'full'	Compute the full erosion.

`gpuarrayIM2 = imerode(gpuarrayIM, ____,)` performs the operation on a graphics processing unit (GPU). `gpuarrayIM` is a `gpuArray` that contains a grayscale or binary image. `gpuarrayIM2` is a `gpuArray` of the same class as the input image. Note that the `PACKOPT` syntax is not supported on a GPU. This syntax requires the Parallel Computing Toolbox.

Class Support

`IM` can be numeric or logical and it can be of any dimension. If `IM` is logical and the structuring element is flat, the output image is logical; otherwise the output image has the same class as the input. If the input is packed binary, then the output is also packed binary.

`gpuarrayIM` must be a `gpuArray` of type `uint8` or `logical`. When used with a `gpuarray`, the structuring element must be flat and two-dimensional. The output has the same class as the input.

Examples

Erode Binary Image with Line Structuring Element

Read binary image into the workspace.

```
originalBW = imread('text.png');
```

Create a flat, line-shaped structuring element.

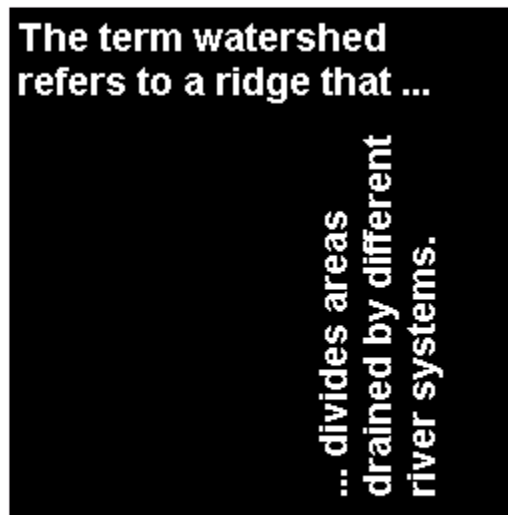
```
se = strel('line',11,90);
```

Erode the image with the structuring element.

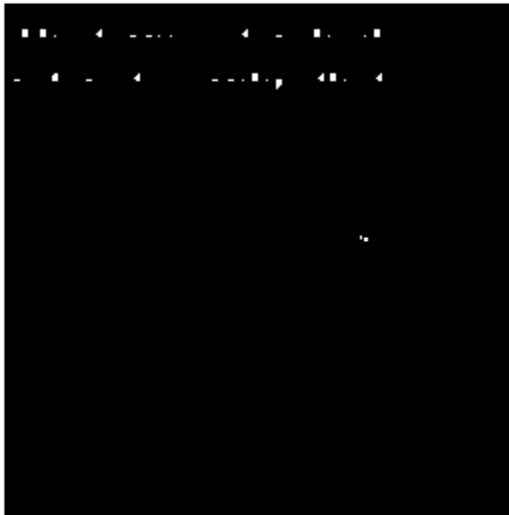
```
erodedBW = imerode(originalBW,se);
```

View the original image and the eroded image.

```
figure  
imshow(originalBW)
```



```
figure  
imshow(erodedBW)
```



Erode Grayscale Image with Rolling Ball

Read grayscale image into the workspace.

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

Create a nonflat offsetstrel object.

```
se = offsetstrel('ball',5,5);
```

Erode the image.

```
erodedI = imerode(originalI,se);
```

Display original image and eroded image.

```
figure  
imshow(originalI)
```




```
figure  
imshow(erodedI)
```



Erode Binary Image on GPU

Read binary image into the workspace.

```
originalBW = imread('text.png');
```

Create a structuring element.

```
se = strel('line',11,90);  
erodedBW = imerode(gpuArray(originalBW),se);  
figure, imshow(originalBW), figure, imshow(erodedBW)
```

Erode the image, creating a GPUarray.

```
erodedBW = imerode(gpuArray(originalBW),se);
```

Display the original image and the eroded image.

```
figure, imshow(originalBW), figure, imshow(erodedBW)
```

Erode Grayscale Image on GPU

Read grayscale image into the workspace.

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

Create a structuring element.

```
se = strel('disk',5);
```

Erode the image, creating a GPUarray.

```
erodedI = imerode(gpuArray(originalI),se);
```

Display the original image and the eroded image.

```
figure
imshow(originalI)
figure
imshow(erodedI)
```

Erode MRI Stack Volume Using Cubic Structuring Element

Create a binary volume.

```
load mrystack
BW = mrystack < 100;
```

Create a cubic structuring element.

```
se = strel('cube',3)
```

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

```
    Neighborhood: [3×3×3 logical]
    Dimensionality: 3
```

Erode the volume with a cubic structuring element.

```
erodedBW = imerode(BW, se);
```

Definitions

Binary Erosion

The *binary erosion* of A by B , denoted $A \ominus B$, is defined as the set operation $A \ominus B = \{z \mid (B_z \subseteq A)\}$. In other words, it is the set of pixel locations z , where the structuring element translated to location z overlaps only with foreground pixels in A .

For more information on binary erosion, see [1].

Gray-Scale Erosion

In the general form of *gray-scale erosion*, the structuring element has a height. The gray-scale erosion of $A(x, y)$ by $B(x, y)$ is defined as:

$$(A \ominus B)(x, y) = \min \{A(x + x', y + y') - B(x', y') \mid (x', y') \in D_B\},$$

where D_B is the domain of the structuring element B and $A(x, y)$ is assumed to be $+\infty$ outside the domain of the image. To create a structuring element with nonzero height values, use the syntax `strel(nhood, height)`, where `height` gives the height values and `nhood` corresponds to the structuring element domain, D_B .

Most commonly, gray-scale erosion is performed with a flat structuring element ($B(x, y) = 0$). Gray-scale erosion using such a structuring element is equivalent to a local-minimum operator:

$$(A \ominus B)(x, y) = \min \{A(x + x', y + y') \mid (x', y') \in D_B\}.$$

All of the `strel` syntaxes except for `strel(nhood, height)`, `strel('arbitrary', nhood, height)`, and `strel('ball', ...)` produce flat structuring elements.

Algorithms

`imerode` automatically takes advantage of the decomposition of a structuring element object (if a decomposition exists). Also, when performing binary dilation with a structuring element object that has a decomposition, `imerode` automatically uses binary image packing to speed up the dilation.

Erosion using bit packing is described in [3].

References

- [1] Gonzalez, R. C., R. E. Woods, and S. L. Eddins, *Digital Image Processing Using MATLAB*, Gatesmark Publishing, 2009.
- [2] Haralick, Robert M., and Linda G. Shapiro, *Computer and Robot Vision*, Vol. I, Addison-Wesley, 1992, pp. 158-205.
- [3] van den Boomgard, R, and R. van Balen, "Methods for Fast Morphological Image Transforms Using Bitmapped Images," *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, Vol. 54, Number 3, pp. 254-258, May 1992.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- The input image, `IM`, must be 2-D or 3-D.
- The structuring element argument `SE` must be a single element—arrays of structuring elements are not supported. To obtain the same result as that obtained using an array of structuring elements, call the function sequentially.
- When the target is `MATLAB Host Computer`, the `PACKOPT` and `SHAPE` arguments must be compile-time constants. When the target is any other platform, the `PACKOPT` syntax is not supported.

See Also

See Also

`strel` | `offsetstrel` | `bwpack` | `bwunpack` | `conv2` | `filter2` | `gpuArray` | `imclose` | `imdilate` | `imopen`

Introduced before R2006a

imextendedmax

Extended-maxima transform

Syntax

```
BW = imextendedmax(I,H)
BW = imextendedmax(I,H,conn)
```

Description

`BW = imextendedmax(I,H)` returns the extended-maxima transform for `I`, which is the regional maxima of the `H`-maxima transform. Regional maxima are connected components of pixels with a constant intensity value, and whose external boundary pixels all have a lower value. `H` is a nonnegative scalar. By default, `imextendedmax` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imextendedmax` uses `conndef(numel(size(I)), 'maximal')`.

`BW = imextendedmax(I,H,conn)` computes the extended-maxima transform, where `conn` specifies the connectivity.

Examples

Perform Extended-Maxima transform

Read image into workspace.

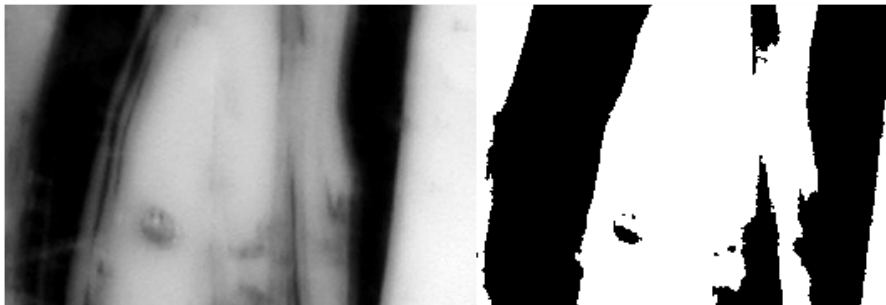
```
I = imread('glass.png');
```

Calculate the extended-maxima transform.

```
BW = imextendedmax(I,80);
```

Display original image and transformed image side-by-side.

```
imshowpair(I,BW,'montage')
```



Input Arguments

I — Input image

real, nonsparse numeric array of any dimension

Input image, specified as a real, nonsparse numeric array of any dimension.

```
Example: I = imread('glass.png'); BW = imextendedmax(I,80);
```

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

H — H-maxima transform

real, nonnegative scalar

H-maxima transform, specified as a real, nonnegative scalar.

```
Example: BW = imextendedmax(I,80);
```

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

conn — Connectivity

8 (default) | 4 | 6 | 18 | 26 | 3-by-3-by- ...-by-3 matrix of zeroes and ones

Connectivity, specified as a one of the scalar values in the following table. By default, `imextendedmax` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imextendedmax` uses `conndef(numel(size(I)), 'maximal')`. Connectivity can be defined in a more general way for any dimension by using for `conn` a 3-by-3-by- ...-by-3 matrix of 0s and 1s. The 1-valued elements define neighborhood locations relative to the center element of `conn`. Note that `conn` must be symmetric around its center element.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Example: `BW = imextendedmax(I,80,4);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

BW — Transformed image

logical array

Transformed image, returned as a logical array the same size as `I`.

References

- [1] Soille, P., *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 170-171.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the optional third input argument, **conn**, must be a compile-time constant.

See Also

See Also

`conndef` | `imextendedmin` | `imhmax` | `imreconstruct` | `imregionalmax`

Introduced before R2006a

imextendedmin

Extended-minima transform

Syntax

```
BW = imextendedmin(I,h)  
BW = imextendedmin(I,h,conn)
```

Description

`BW = imextendedmin(I,h)` computes the extended-minima transform, which is the regional minima of the H-minima transform. Regional minima are connected components of pixels with a constant intensity value, and whose external boundary pixels all have a higher value. `h` is a nonnegative scalar. By default, `imextendedmin` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imextendedmin` uses `conndef(numel(size(I)), 'maximal')`.

`BW = imextendedmin(I,h,conn)` computes the extended-minima transform, which is the regional minima of the H-minima transform. `h` is a nonnegative scalar.

Examples

Perform Extended-Minima transform

Read image into the workspace.

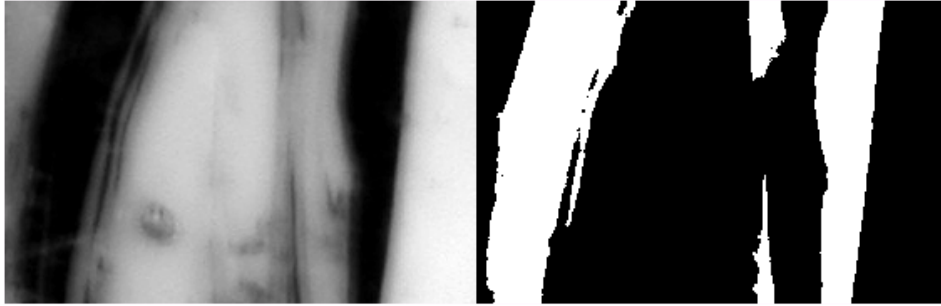
```
I = imread('glass.png');
```

Calculate the extended-minima transform.

```
BW = imextendedmin(I,50);
```

Display the original image and the transformation side-by-side.

```
imshowpair(I,BW,'montage');
```



Input Arguments

I — Input image

nonsparse numeric array of any dimension

Input array, specified as a nonsparse numeric array of any dimension.

Example: `I = imread('glass.png');` `BW = imextendedmax(I,80);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

h — h-maxima transform

(default) | nonnegative scalar

h-maxima transform, specified as a nonnegative scalar.

Example: `BW = imextendedmin(I,80);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

conn — Connectivity

8 (default) | 4 | 6 | 18 | 26 | 3-by-3-by- ...-by-3 matrix of zeroes and ones

Connectivity, specified as a one of the scalar values in the following table. By default, `imextendedmin` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imextendedmin` uses `conndef(numel(size(I)), 'maximal')`. Connectivity can be defined in a more general way for any dimension by using for `conn` a 3-by-3-by-...-by-3 matrix of 0s and 1s. The 1-valued elements define neighborhood locations relative to the center element of `conn`. Note that `conn` must be symmetric around its center element.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Example: `BW = imextendedmin(I,80,4);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

BW — Transformed image

logical array

Transformed image, returned as a logical array the same size as `I`.

References

- [1] Soille, P., *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 170-171.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the optional third input argument, **conn**, must be a compile-time constant.

See Also

See Also

`conndef` | `imextendedmax` | `imhmin` | `imreconstruct` | `imregionalmin`

Introduced before R2006a

imfill

Fill image regions and holes

Syntax

```
BW2= imfill(BW,locations)
BW2= imfill(BW,'holes')
I2= imfill(I)

BW2 = imfill(BW)
BW2 = imfill(BW,0,conn)
[BW2,locations_out] = imfill(BW)

BW2= imfill(BW,locations,conn)
BW2= imfill(BW,conn,'holes')
I2= imfill(I,conn)

gpuarrayB = imfill(gpuarrayA, ___ )
```

Description

`BW2= imfill(BW,locations)` performs a flood-fill operation on background pixels of the input binary image `BW`, starting from the points specified in `locations`. If `locations` is a P -by-1 vector, it contains the linear indices of the starting locations. If `locations` is a P -by-`ndims(BW)` matrix, each row contains the array indices of one of the starting locations.

`BW2= imfill(BW,'holes')` fills holes in the input binary image `BW`. In this syntax, a hole is a set of background pixels that cannot be reached by filling in the background from the edge of the image.

`I2= imfill(I)` fills holes in the grayscale image `I`. In this syntax, a hole is defined as an area of dark pixels surrounded by lighter pixels.

`BW2 = imfill(BW)` displays the binary image `BW` on the screen and lets you define the region to fill by selecting points interactively with the mouse. To use this syntax, `BW` must

be a 2-D image. Press **Backspace** or **Delete** to remove the previously selected point. Shift-click, right-click, or double-click to select a final point and start the fill operation. Press **Return** to finish the selection without adding a point.

`BW2 = imfill(BW,0,conn)` lets you override the default connectivity as you interactively specify locations.

`[BW2,locations_out] = imfill(BW)` returns the locations of points selected interactively in `locations_out`. The return value `locations_out` is a vector of linear indices into the input image. To use this syntax, `BW` must be a 2-D image.

`BW2= imfill(BW,locations,conn)` fills the area defined by `locations`, where `conn` specifies the connectivity.

`BW2= imfill(BW,conn, 'holes ')` fills holes in the binary image `BW`, where `conn` specifies the connectivity.

`I2= imfill(I,conn)` fills holes in the grayscale image `I`, where `conn` specifies the connectivity.

`gpuarrayB = imfill(gpuarrayA, ___)` performs the fill operation on a GPU. The input image and the return image are 2-D `gpuArrays`. Use of this syntax requires Parallel Computing Toolbox. When run on a GPU, `imfill` does not support interactive syntaxes, where you select locations using the mouse.

Examples

Fill Image from Specified Starting Point

```
BW1 = logical([1 0 0 0 0 0 0 0
               1 1 1 1 1 0 0 0
               1 0 0 0 1 0 1 0
               1 0 0 0 1 1 1 0
               1 1 1 1 0 1 1 1
               1 0 0 1 1 0 1 0
               1 0 0 0 1 0 1 0
               1 0 0 0 1 1 1 0]);
```

```
BW2 = imfill(BW1,[3 3],8)
```

```
BW2 = 8x8 logical array
```



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

Fill Holes in a Binary Image

Read image into workspace.

```
I = imread('coins.png');
figure
imshow(I)
title('Original Image')
```

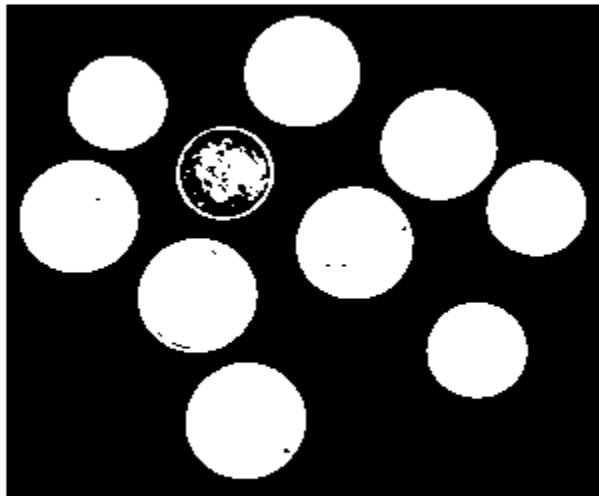
Original Image



Convert image to binary image.

```
BW = imbinarize(I);  
figure  
imshow(BW)  
title('Original Image Converted to Binary Image')
```

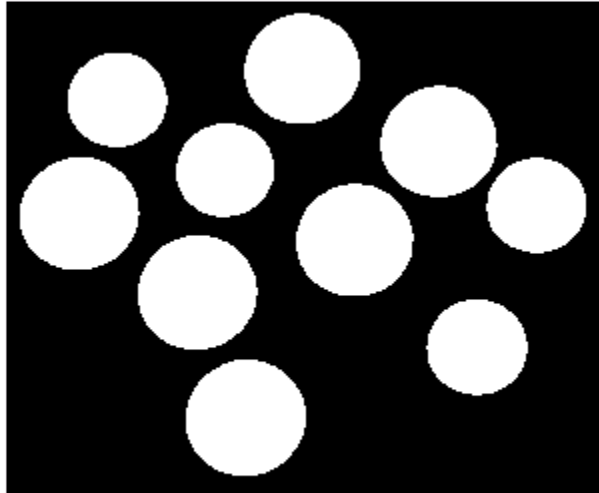
Original Image Converted to Binary Image



Fill holes in the binary image and display the result.

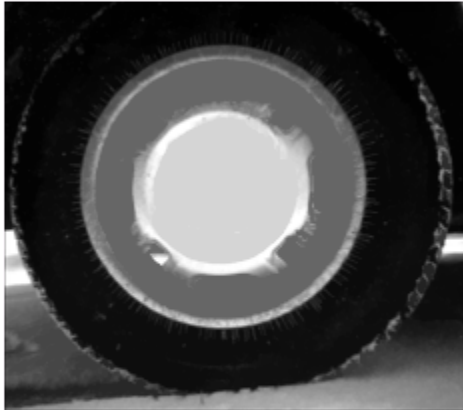
```
BW2 = imfill(BW, 'holes');  
figure  
imshow(BW2)  
title('Filled Image')
```

Filled Image



Fill Holes in a Grayscale Image

```
I = imread('tire.tif');  
I2 = imfill(I);  
figure, imshow(I), figure, imshow(I2)
```



Fill Operation on a GPU

Create a simple sample binary image.

```
BW1 = logical([1 0 0 0 0 0 0 0
               1 1 1 1 1 0 0 0
               1 0 0 0 1 0 1 0
               1 0 0 0 1 1 1 0
               1 1 1 1 0 1 1 1
               1 0 0 1 1 0 1 0
               1 0 0 0 1 0 1 0
               1 0 0 0 1 1 1 0])
```

BW1 =

```

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

Create a gpuArray.

```
BW1 = gpuArray(BW1);
```

Fill in the background of the image from a specified starting location.

```
BW2 = imfill(BW1,[3 3],8)
```

BW2 =

```

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

1 0 0 0 1 1 1 0

Input Arguments

BW — Input binary image

real, nonsparse, numeric or logical array

Input binary image, specified as a real, nonsparse, numeric or logical array of any dimension.

Example: `BW = imread('text.png');`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

locations — Linear indices identifying pixel locations

2-D, real, numeric vector or matrix of positive integers

Linear indices identifying pixel locations, specified as a 2-D, real, numeric vector or matrix of positive integers.

Example: `BW2 = imfill(BW,[3 3],8);`

Data Types: `double`

I — Input grayscale image

real, nonsparse numeric array

Input grayscale image, specified as a real, nonsparse, numeric array of any dimension.

Example: `I = imread('cameraman.tif');` `I2 = imfill(I);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

conn — Connectivity

4 (default) | 8 | 6 | 18 | 26 | 3-by-3-by- ...-by-3 matrix of zeroes and ones

Connectivity, specified as a one of the scalar values in the following table. By default, `imfill` uses 4-connected neighborhoods for 2-D images and 6-connected neighborhoods for 3-D inputs. For higher dimensions, `imfill` uses `conndef(ndims(I),'minimal')`. Connectivity can be defined in a more general way for any dimension by using for `conn` a 3-by-3-by- ...-by-3 matrix of 0s and 1s. The 1-valued elements define neighborhood

locations relative to the center element of `conn`. Note that `conn` must be symmetric around its center element.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Example:

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

gpuarrayA — Input image

`gpuArray`

Input image, specified as a `gpuArray`.

Output Arguments

BW2 — Filled image

logical array

Filled image, returned as an array the same class as the input image.

locations_out — Linear indices of pixel locations

numeric vector or matrix

Linear indices of pixel locations, returned as a numeric vector or matrix.

I2 — Filled grayscale image

numeric array

Filled grayscale image, returned as a numeric array.

gpuarrayB — Output image

gpuArray

Output image, returned as a gpuArray.

Algorithms

`imfill` uses an algorithm based on morphological reconstruction [1].

References

[1] Soille, P., *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 173-174.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- The optional input arguments, `conn` and `'holes'` must be a compile-time constants.
- `imfill` supports up to 3-D inputs only. (No N-D support.)
- The interactive syntax to select points, `imfill(BW,0,CONN)` is not supported.
- With the `locations` input argument, once you select a format at compile time, you cannot change it at run time. However, the number of points in `locations` can be varied at run time.

See Also

See Also

`bwselect` | `conndef` | `imreconstruct` | `regionfill`

Introduced before R2006a

imfilter

N-D filtering of multidimensional images

Syntax

```
B = imfilter(A,h)
gpuarrayB = imfilter(gpuArrayA,h)
___ = imfilter( ___,options,...)
```

Description

`B = imfilter(A,h)` filters the multidimensional array `A` with the multidimensional filter `h`. The array `A` can be `logical` or a nonsparse numeric array of any class and dimension. The result `B` has the same size and class as `A`.

`imfilter` computes each element of the output, `B`, using double-precision floating point. If `A` is an integer or logical array, `imfilter` truncates output elements that exceed the range of the given type, and rounds fractional values.

`gpuarrayB = imfilter(gpuArrayA,h)` performs the operation on a GPU. `gpuArrayA` is a `gpuArray` that contains a `logical` or a nonsparse numeric array of any class and dimension. When used with a `gpuArray`, `H` must be a vector or 2-D matrix. This syntax requires the Parallel Computing Toolbox.

`___ = imfilter(___,options,...)` performs multidimensional filtering according to the specified options.

Examples

Create Filter and Apply It

Read a color image into the workspace and display it.

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

```
imshow(originalRGB)
```



Create a motion-blur filter using the `fspecial` function.

```
h = fspecial('motion', 50, 45);
```

Apply the filter to the original image to create an image with motion blur. Note that `imfilter` is more memory efficient than some other filtering functions in that it outputs an array of the same data type as the input image array. In this example, the output is an array of `uint8`.

```
filteredRGB = imfilter(originalRGB, h);  
figure, imshow(filteredRGB)
```



Filter the image again, this time specifying the replicate boundary option.

```
boundaryReplicateRGB = imfilter(originalRGB, h, 'replicate');  
figure, imshow(boundaryReplicateRGB)
```



Create a Filter and Apply it on a GPU

Read a color image into the workspace as a `gpuArray` and view it.

```
originalRGB = gpuArray(imread('peppers.png'));  
imshow(originalRGB)
```



Original Image

Create a filter, `h`, that can be used to approximate linear camera motion.

```
h = fspecial('motion', 50, 45);
```

Apply the filter, using `imfilter`, to the image `originalRGB` to create a new image, `filteredRGB`. The image is returned as a `gpuArray`.

```
filteredRGB = imfilter(originalRGB, h);  
figure, imshow(filteredRGB)
```



Filtered Image

Note that `imfilter` is more memory efficient than some other filtering operations in that it outputs an array of the same data type as the input image array. In this example, the output is an array of `uint8`.

`whos`

Name	Size	Bytes	Class	Attributes
filteredRGB	384x512x3	108	gpuArray	
h	37x37	10952	double	
originalRGB	384x512x3	108	gpuArray	

Try the filtering operation again, this time specifying the `replicate` boundary option.

```
boundaryReplicateRGB = imfilter(originalRGB, h, 'replicate');
figure, imshow(boundaryReplicateRGB)
```

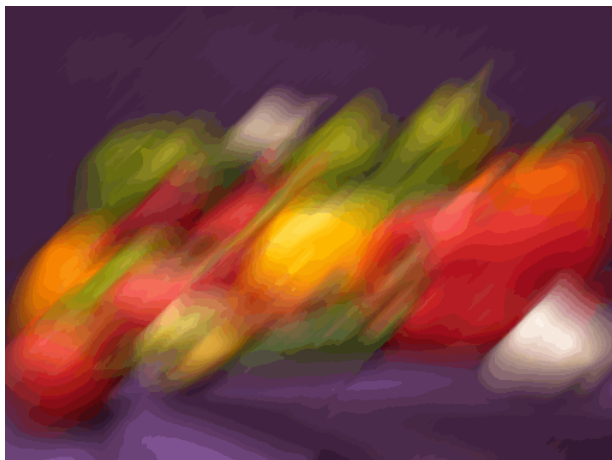


Image with Replicate Boundary

Input Arguments

A — Image to be filtered

nonsparse, numeric array of any class and dimension

Image to be filtered, specified as a nonsparse, numeric array of any class and dimension

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

h — Multidimensional filter

N-D array of doubles

Multidimensional filter, specified as an N-D array of doubles.

Data Types: `double`

gpuArrayA — Image to be filtered

gpuArray object

Image to be filtered, specified as a `gpuArray` object. When used with a `gpuArray`, `imfilter` computes `gpuarrayB`, using either single- or double-precision floating point,

depending on the data type of `gpuArrayA`. When `gpuArrayA` contains double-precision or `uint32` values, `imfilter` uses double-precision values. For all other data types, `imfilter` uses single-precision. If `gpuArrayA` is an integer or logical array, `imfilter` truncates output elements that exceed the range of the given type, and rounds off fractional values.

options — Options that control the filtering operation

character vector | numeric value

Options that control the filtering operation, specified as a character vector or numeric value. The following table lists all supported options.

Boundary Options

Option	Description
Boundary Options	
X	Input array values outside the bounds of the array are implicitly assumed to have the value X. When no boundary option is specified, the default is 0.
'symmetric'	Input array values outside the bounds of the array are computed by mirror-reflecting the array across the array border.
'replicate'	Input array values outside the bounds of the array are assumed to equal the nearest array border value.
'circular'	Input array values outside the bounds of the array are computed by implicitly assuming the input array is periodic.
Output Size	
'same'	The output array is the same size as the input array. This is the default behavior when no output size options are specified.
'full'	The output array is the full filtered result, and so is larger than the input array.
Correlation and Convolution Options	
'corr'	<code>imfilter</code> performs multidimensional filtering using correlation, which is the same way that <code>filter2</code> performs filtering. When no correlation or convolution option is specified, <code>imfilter</code> uses correlation.
'conv'	<code>imfilter</code> performs multidimensional filtering using convolution.

Output Arguments

B — Filtered image

numeric array the same size and class as input image

Filtered image, returned as an array the same size and class as the input image.

gpuarrayB — Filtered image

gpuArray

Filtered image, returned as a `gpuArray`, the same size and class as `gpuarrayA`

Tips

- This function may take advantage of hardware optimization for data types `uint8`, `uint16`, `int16`, `single`, and `double` to run faster.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the input image, `A`, must be 2-D or 3-D. The value of the input argument, `options`, must be a compile-time constant.

See Also

See Also

`conv2` | `convn` | `filter2` | `fspecial` | `gpuArray`

Introduced before R2006a

imfindcircles

Find circles using circular Hough transform

Syntax

```
centers = imfindcircles(A, radius)
[centers, radii] = imfindcircles(A, radiusRange)
[centers, radii, metric] = imfindcircles(A, radiusRange)
[centers, radii, metric] = imfindcircles( ____, Name, Value)
```

Description

`centers = imfindcircles(A, radius)` finds the circles in image `A` whose radii are approximately equal to `radius`. The output, `centers`, is a two-column matrix containing the x,y coordinates of the circles centers in the image.

`[centers, radii] = imfindcircles(A, radiusRange)` finds circles with radii in the range specified by `radiusRange`. The additional output argument, `radii`, contains the estimated radii corresponding to each circle center in `centers`.

`[centers, radii, metric] = imfindcircles(A, radiusRange)` also returns a column vector, `metric`, containing the magnitudes of the accumulator array peaks for each circle (in descending order). The rows of `centers` and `radii` correspond to the rows of `metric`.

`[centers, radii, metric] = imfindcircles(____, Name, Value)` specifies additional options with one or more `Name, Value` pair arguments, using any of the previous syntaxes.

Examples

Detect Five Strongest Circles in an Image

This example shows how to find all circles in an image, and how to retain and display the strongest circles.

Read a grayscale image into the workspace and display it.

```
A = imread('coins.png');  
imshow(A)
```



Find all the circles with radius r pixels in the range [15, 30].

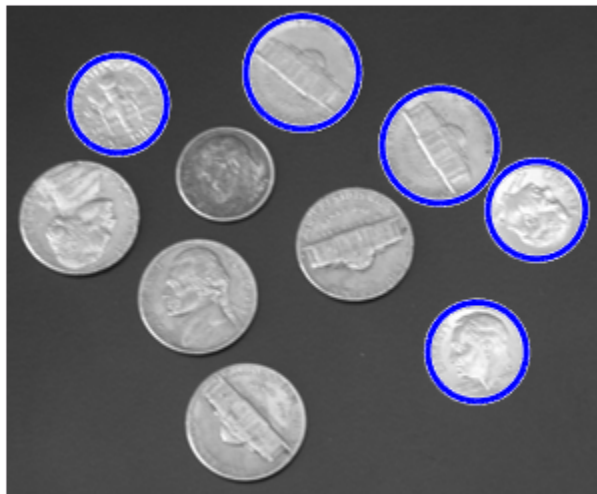
```
[centers, radii, metric] = imfindcircles(A,[15 30]);
```

Retain the five strongest circles according to the metric values.

```
centersStrong5 = centers(1:5,:);  
radiiStrong5 = radii(1:5);  
metricStrong5 = metric(1:5);
```

Draw the five strongest circle perimeters over the original image.

```
viscircles(centersStrong5, radiiStrong5, 'EdgeColor', 'b');
```

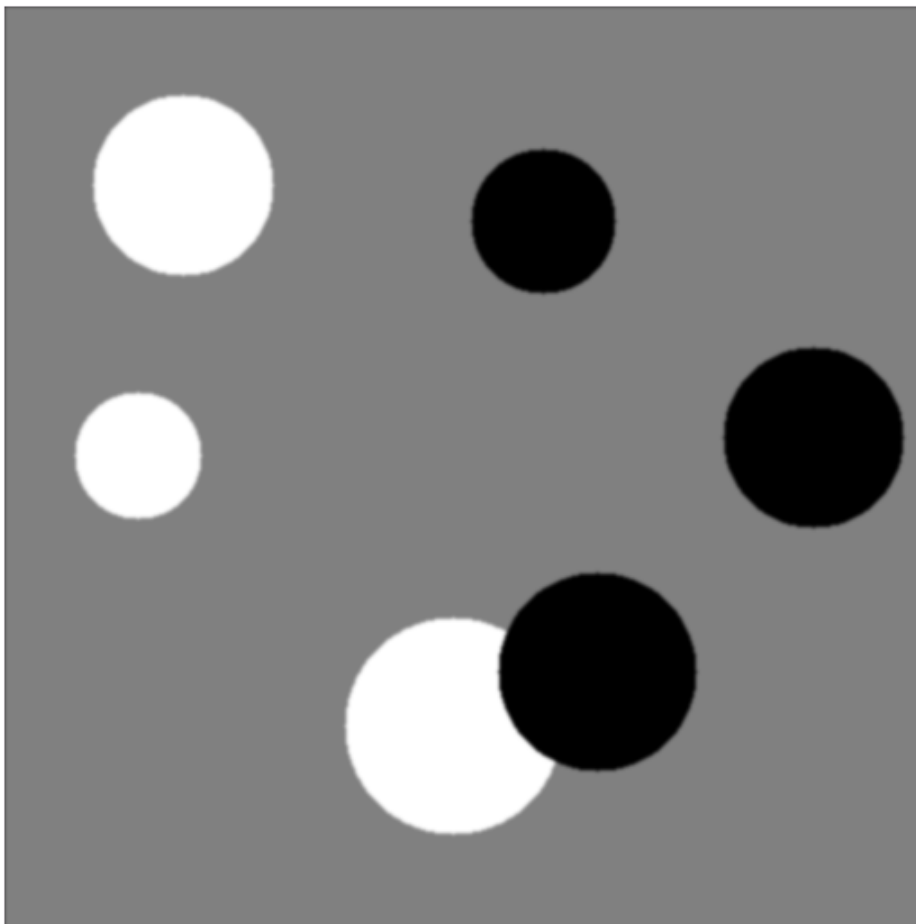


Draw Lines Around Bright and Dark Circles in Image

This example shows how to draw lines around both bright and dark circles in an image.

Read the image into the workspace and display it.

```
A = imread('circlesBrightDark.png');  
imshow(A)
```



Define the radius range.

```
Rmin = 30;  
Rmax = 65;
```

Find all the bright circles in the image within the radius range.

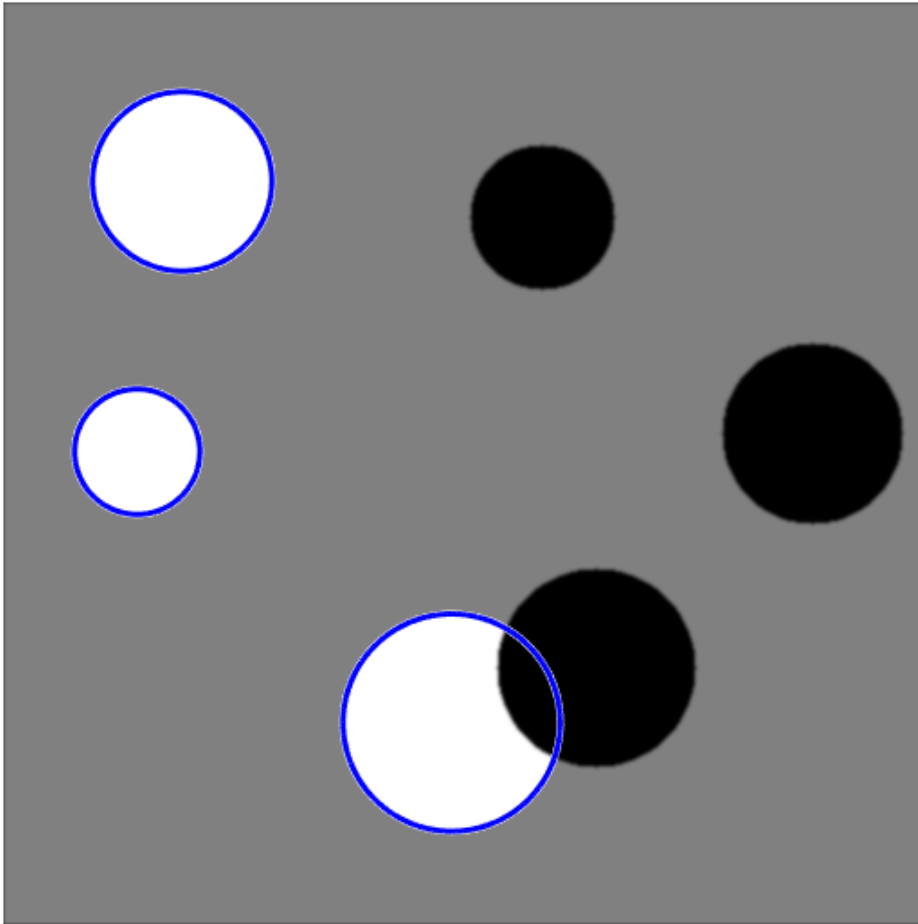
```
[centersBright, radiiBright] = imfindcircles(A,[Rmin Rmax], 'ObjectPolarity', 'bright');
```

Find all the dark circles in the image within the radius range.

```
[centersDark, radiiDark] = imfindcircles(A,[Rmin Rmax], 'ObjectPolarity', 'dark');
```

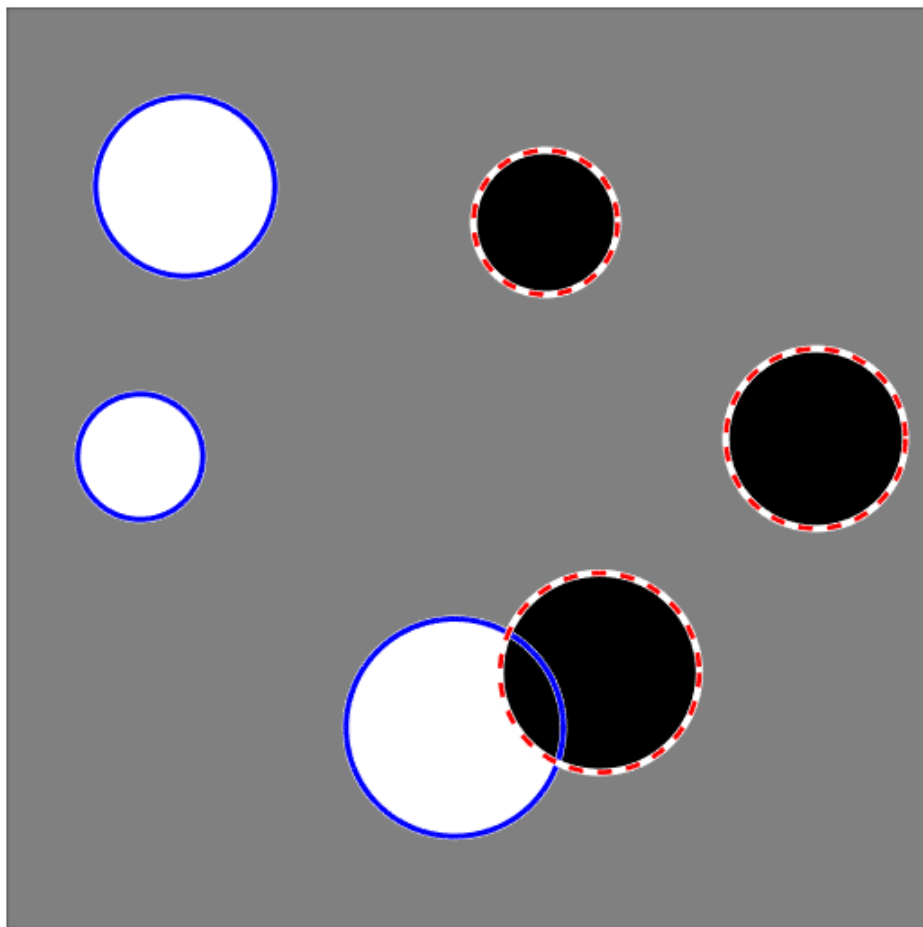
Draw blue lines around the edges of the bright circles.

```
viscircles(centersBright, radiiBright, 'Color', 'b');
```

Draw red dashed lines around the edges of the dark circles.

```
viscircles(centersDark, radiiDark, 'LineStyle', '--');
```



Input Arguments

A — Input image

grayscale image | truecolor image | binary image

Input image is the image in which to detect circular objects, specified as a grayscale, truecolor, or binary image.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

radius — Circle radius

scalar numeric

Circle radius is the approximate radius of the circular objects you want to detect, specified as a scalar of any numeric type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

radiusRange — Range of radii

two-element vector of integers

Range of radii for the circular objects you want to detect, specified as a two-element vector, `[rmin rmax]`, of integers of any numeric type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'ObjectPolarity', 'bright'` specifies bright circular objects on a dark background.

'ObjectPolarity' — Object polarity

`'bright'` (default) | `'dark'`

Object polarity indicates whether the circular objects are brighter or darker than the background, specified as the comma-separated pair consisting of `'ObjectPolarity'` and either of the values in the following table.

'bright' The circular objects are brighter than the background.

'dark' The circular objects are darker than the background.

'Method' — Computation method

'PhaseCode' (default) | 'TwoStage'

Computation method is the technique used to compute the accumulator array, specified as the comma-separated pair consisting of 'Method' and either of the values in the following table.

'PhaseCode' Atherton and Kerbyson's [1] phase coding method . This is the default.

'TwoStage' The method used in two-stage circular Hough transform [2], [3].

Example: 'Method', 'PhaseCode' specifies the Atherton and Kerbyson's phase coding method.

'Sensitivity' — Sensitivity factor

0.85 (default) | nonnegative scalar between 0 and 1

Sensitivity factor is the sensitivity for the circular Hough transform accumulator array, specified as the comma-separated pair consisting of 'Sensitivity' and a nonnegative scalar value in the range [0,1]. As you increase the sensitivity factor, `imfindcircles` detects more circular objects, including weak and partially obscured circles. Higher sensitivity values also increase the risk of false detection.

'EdgeThreshold' — Edge gradient threshold

nonnegative scalar between 0 and 1

Edge gradient threshold sets the gradient threshold for determining edge pixels in the image, specified as the comma-separated pair consisting of 'EdgeThreshold' and a nonnegative scalar value in the range [0,1]. Specify 0 to set the threshold to zero-gradient magnitude. Specify 1 to set the threshold to the maximum gradient magnitude. `imfindcircles` detects more circular objects (with both weak and strong edges) when you set the threshold to a lower value. It detects fewer circles with weak edges as you increase the value of the threshold. By default, `imfindcircles` chooses the edge gradient threshold automatically using the function `graythresh`.

Example: 'EdgeThreshold', 0.5 sets the edge gradient threshold to 0.5.

Output Arguments

centers — Coordinates of circle centers

two-column matrix

Coordinates of the circle centers, returned as a P -by-2 matrix containing the x -coordinates of the circle centers in the first column and the y -coordinates in the second column. The number of rows, P , is the number of circles detected. **centers** is sorted based on the strength of the circles.

radii — Estimated radii

column vector

The estimated radii for the circle centers, returned as a column vector. The radius value at **radii**(j) corresponds to the circle centered at **centers**(j ,:).

metric — Circle strengths

column vector

Circle strengths is the relative strengths for the circle centers, returned as a vector. The value at **metric**(j) corresponds to the circle with radius **radii**(j) centered at **centers**(j ,:).

Tips

- Specify a relatively small **radiusRange** for better accuracy. A good rule of thumb is to choose **radiusRange** such that $r_{\max} < 3 \cdot r_{\min}$ and $(r_{\max} - r_{\min}) < 100$.
- The accuracy of **imfindcircles** is limited when the value of **radius** (or **rmin**) is less than or equal to 5.
- The radius estimation step is typically faster if you use the (default) 'PhaseCode' method instead of 'TwoStage'.
- Both computation methods, 'PhaseCode' and 'TwoStage' are limited in their ability to detect concentric circles. The results for concentric circles can vary depending on the input image.
- **imfindcircles** does not find circles with centers outside the domain of the image.
- **imfindcircles** preprocesses binary (logical) images to improve the accuracy of the result. It converts truecolor images to grayscale using the function **rgb2gray** before processing them.

Algorithm

Function `imfindcircles` uses a Circular Hough Transform (CHT) based algorithm for finding circles in images. This approach is used because of its robustness in the presence of noise, occlusion and varying illumination.

The CHT is not a rigorously specified algorithm, rather there are a number of different approaches that can be taken in its implementation. However, by and large, there are three essential steps which are common to all.

1 Accumulator Array Computation.

Foreground pixels of high gradient are designated as being candidate pixels and are allowed to cast ‘votes’ in the accumulator array. In a classical CHT implementation, the candidate pixels vote in pattern around them that forms a full circle of a fixed radius. Figure 1a shows an example of a candidate pixel lying on an actual circle (solid circle) and the classical CHT voting pattern (dashed circles) for the candidate pixel.

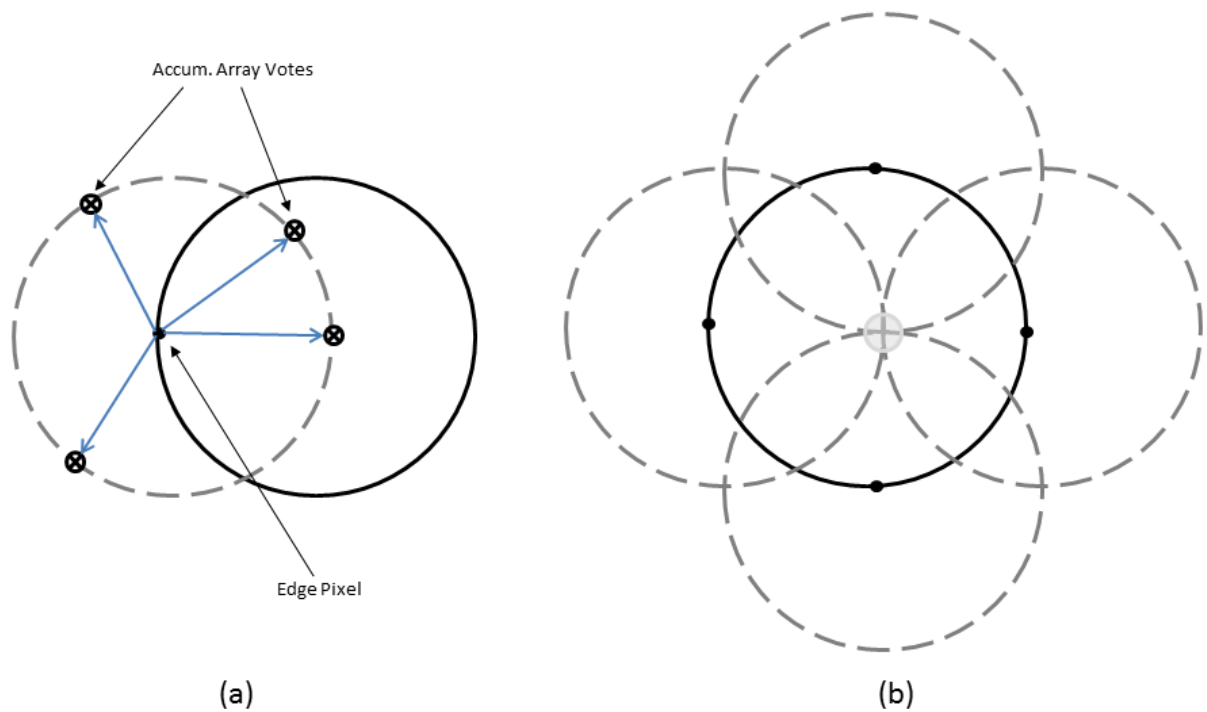


Figure 1: classical CHT voting pattern

2 Center Estimation

The votes of candidate pixels belonging to an image circle tend to accumulate at the accumulator array bin corresponding to the circle's center. Therefore, the circle centers are estimated by detecting the peaks in the accumulator array. Figure 1b shows an example of the candidate pixels (solid dots) lying on an actual circle (solid circle), and their voting patterns (dashed circles) which coincide at the center of the actual circle.

3 Radius Estimation

If the same accumulator array is used for more than one radius value, as is commonly done in CHT algorithms, radii of the detected circles have to be estimated as a separate step.

Function `imfindcircles` provides two algorithms for finding circles in images: Phase-Coding (default) and Two-Stage. Both share some common computational steps, but each has its own unique aspects as well.

The common computational features shared by both algorithms are as follow:

- Use of 2-D Accumulator Array:

The classical Hough Transform requires a 3-D array for storing votes for multiple radii, which results in large storage requirements and long processing times. Both the Phase-Coding and Two-Stage methods solve this problem by using a single 2-D accumulator array for all the radii. Although this approach requires an additional step of radius estimation, the overall computational load is typically lower, especially when working over large radius range. This is a widely adopted practice in modern CHT implementations.

- Use of Edge Pixels

Overall memory requirements and speed is strongly governed by the number of candidate pixels. To limit their number, the gradient magnitude of the input image is threshold so that only pixels of high gradient are included in tallying votes.

- Use of Edge Orientation Information:

Another way to optimize performance is to restrict the number of bins available to candidate pixels. This is accomplished by utilizing locally available edge information to only permit voting in a limited interval along direction of the gradient (Figure 2).

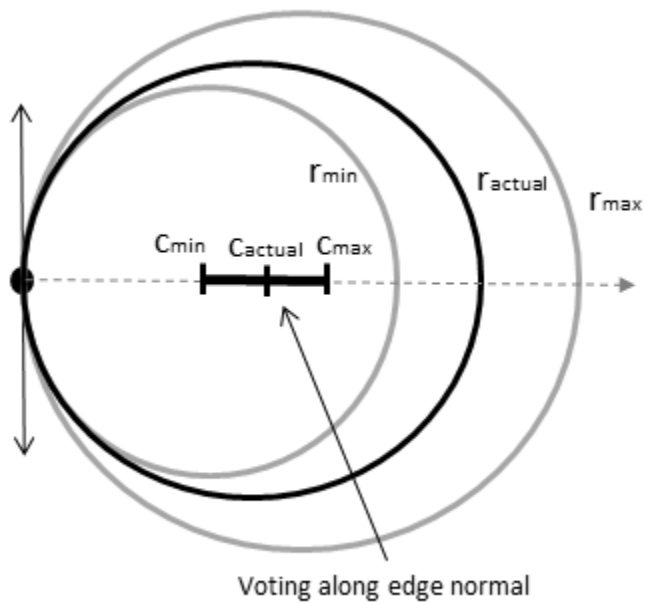


Figure 2: Voting mode: multiple radii, along direction of the gradient

r_{\min}	Minimum search radius
r_{\max}	Maximum search radius
r_{actual}	Radius of the circle that the candidate pixel belongs to
c_{\min}	Center of the circle of radius r_{\min}
c_{\max}	Center of the circle of radius r_{\max}
c_{actual}	Center of the circle of radius r_{actual}

The two CHT methods employed by function `imfindcircles` fundamentally differ in the manner by which the circle radii are computed.

- Two-Stage

Radii are explicitly estimated utilizing the estimated circle centers along with image information. The technique is based on computing radial histograms; see references 1 & 2 for a detailed explanation.

- Phase-Coding

The key idea in Phase Coding (see reference [1]) is the use of complex values in the accumulator array with the radius information encoded in the phase of the array entries. The votes cast by the edge pixels contain information not only about the possible center locations but also about the radius of the circle associated with the center location. Unlike the Two-Stage method where radius has to be estimated explicitly using radial histograms, in Phase Coding the radius can be estimated by simply decoding the phase information from the estimated center location in the accumulator array.

References

- [1] T.J Atherton, D.J. Kerbyson. "Size invariant circle detection." *Image and Vision Computing*. Volume 17, Number 11, 1999, pp. 795-803.
- [2] H.K Yuen, .J. Princen, J. Illingworth, and J. Kittler. "Comparative study of Hough transform methods for circle finding." *Image and Vision Computing*. Volume 8, Number 1, 1990, pp. 71–77.
- [3] E.R. Davies, *Machine Vision: Theory, Algorithms, Practicalities*. Chapter 10. 3rd Edition. Morgan Kauffman Publishers, 2005,

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic MATLAB Host Computer target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a

shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

- When generating code, all character vector input parameters and values must be a compile-time constant.

See Also

See Also

`hough` | `houghlines` | `houghpeaks` | `viscircles`

Introduced in R2012a

imfreehand

Create draggable freehand region

Syntax

```
h = imfreehand
h = imfreehand(hparent)
h = imfreehand(...,param1, val1,...)
```

Description

`h = imfreehand` begins interactive placement of a freehand region of interest on the current axes. The function returns `h`, a handle to an `imfreehand` object. A freehand region of interest can be dragged interactively using the mouse and supports a context menu that controls aspects of its appearance and behavior. See “Interactive Behavior” on page 1-799.


`h = imfreehand(hparent)` begins interactive placement of a freehand region of interest on the object specified by `hparent`. `hparent` specifies the HG parent of the freehand region graphics, which is typically an axes, but can also be any other object that can be the parent of an `hggroup`.

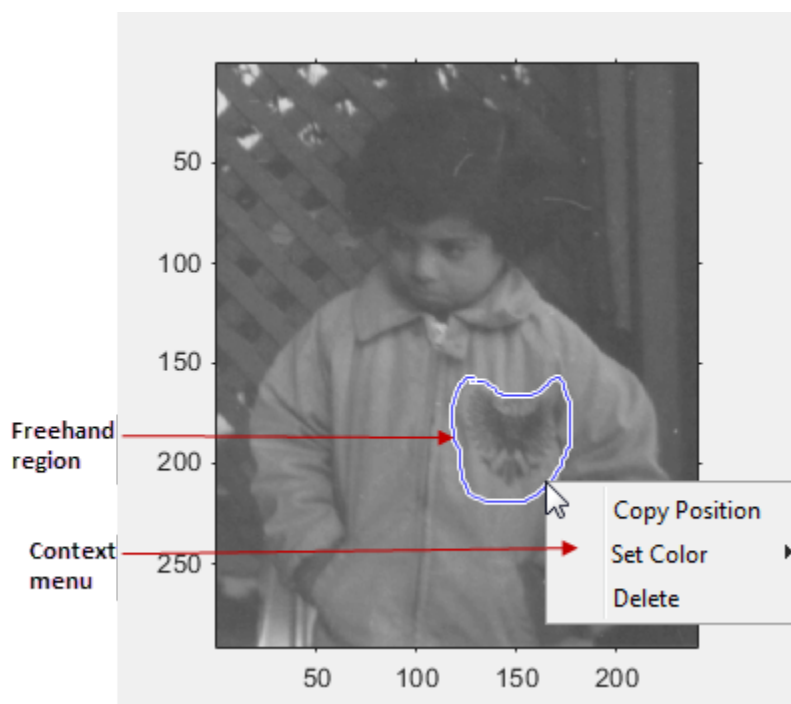
`h = imfreehand(...,param1, val1,...)` creates a freehand ROI, specifying parameters and corresponding values that control the behavior of the tool. The following table lists the parameters available. Parameter names can be abbreviated, and case does not matter.

Parameter	Description
'Closed'	Scalar logical that controls whether the freehand region is closed. When set to <code>true</code> (the default), <code>imfreehand</code> draws a straight line to connect the endpoints of the freehand line to create a closed region. If set to <code>false</code> , <code>imfreehand</code> leaves the region open.
'PositionConstraintFcn'	Function handle specifying the function that is called whenever the freehand region is dragged using the


Parameter	Description
	mouse. Use this parameter to control where the freehand region can be dragged. See the help for the setPositionConstraintFcn method for information about valid function handles.

Interactive Behavior

When you call `imfreehand` with an interactive syntax, the pointer changes to a cross hairs  when positioned over an image. Click and drag the mouse to draw the freehand region. By default, `imfreehand` draws a straight line connecting the last point you drew with the first point, but you can control this behavior using the 'Closed' parameter. The following figure illustrates a freehand region with its context menu.



The following table lists the interactive features supported by `imfreehand`.

Interactive Behavior	Description
Moving the region.	Move the pointer inside the freehand region. The pointer changes to a fleur shape  . Click and hold the left mouse button to move the region.
Changing the color used to draw the region.	Move the pointer inside the freehand region. Right-click and select Set Color from the context menu.
Retrieving the current position of the freehand region.	Move the pointer inside the freehand region. Right-click and select Copy Position from the context menu. <code>imfreehand</code> copies an <i>n</i> -by-2 array of coordinates on the boundary of the ROI to the clipboard.
Deleting the region	Move the pointer inside the region. Right-click and select Delete from the context menu. To remove this option from the context menu, set the <code>Deletable</code> property to false: <code>h = imfreehand(); h.Deletable = false;</code>

Methods

The `imfreehand` object supports the following methods. Type methods `imfreehand` to see a complete list of all methods.

addNewPositionCallback — Add new-position callback to ROI object

See `imroi` for information.

createMask — Create mask within image

See `imroi` for information.

delete — Delete ROI object

See `imroi` for information.

getColor — Get color used to draw ROI object

See `imroi` for information.

getPosition — Return current position of freehand region

`pos = getPosition(h)` returns the current position of the freehand region `h`. The returned position, `pos`, is an *N*-by-2 array `[X1 Y1; ...; XN YN]`.

getPositionConstraintFcn — Return function handle to current position constraint function

See `imroi` for information.

removeNewPositionCallback — Remove new-position callback from ROI object.

See `imroi` for information.

resume — Resume execution of MATLAB command line

See `imroi` for information.

setClosed — Set geometry of freehand region

`setClosed(h,TF)` sets the geometry of the freehand region `h`. `TF` is a logical scalar. `True` means that the freehand region is closed. `False` means that the freehand region is open.

setColor — Set color used to draw ROI object

See `imroi` for information.

setConstrainedPosition — Set ROI object to new position

See `imroi` for information.

setPositionConstraintFcn — Set position constraint function of ROI object.

See `imroi` for information.

wait — Block MATLAB command line until ROI creation is finished

See `imroi` for information.

Examples

Interactively place a closed freehand region of interest by clicking and dragging over an image.

```
figure, imshow('pout.tif');  
h = imfreehand(gca);
```

Interactively place a freehand region by clicking and dragging. Use the `wait` method to block the MATLAB command line. Double-click on the freehand region to resume execution of the MATLAB command line.

```
figure, imshow('pout.tif');  
h = imfreehand;  
position = wait(h);
```

Tips

- If you use `imfreehand` with an axes that contains an image object, and do not specify a position constraint function, users can drag the freehand region outside the extent of the image and lose the freehand region. When used with an axes created by the `plot` function, the axes limits automatically expand to accommodate the movement of the freehand region.
- To cancel the interactive placement, press the Esc key. `imfreehand` returns an empty object.

See Also

See Also

`imellipse` | `imline` | `impoint` | `impoly` | `imrect` | `iptgetapi` | `makeConstrainToRectFcn`

Introduced in R2007b

imfuse

Composite of two images

Syntax

```
C = imfuse(A,B)
[C RC] = imfuse(A,RA,B,RB)
C = imfuse( ____,method)
C = imfuse( ____,Name,Value)
```

Description

`C = imfuse(A,B)` creates a composite image from two images, `A` and `B`. If `A` and `B` are different sizes, `imfuse` pads the smaller dimensions with zeros so that both images are the same size before creating the composite. The output, `C`, is a numeric matrix containing a fused version of images `A` and `B`.

`[C RC] = imfuse(A,RA,B,RB)` creates a composite image from two images, `A` and `B`, using the spatial referencing information provided in `RA` and `RB`. The output `RC` defines the spatial referencing information for the output fused image `C`.

`C = imfuse(____,method)` uses the algorithm specified by `method`.

`C = imfuse(____,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments, using any of the previous syntaxes.

Examples

Create Blended Overlay of Two Images

Load an image into the workspace. Create a copy with a rotation offset applied.

```
A = imread('cameraman.tif');
B = imrotate(A,5,'bicubic','crop');
```

Create blended overlay image, scaling the intensities of A and B jointly as a single data set. View the fused image.

```
C = imfuse(A,B,'blend','Scaling','joint');  
imshow(C)
```



Save the resulting image as a .png file.

```
imwrite(C,'my_blend_overlay.png');
```

Create Overlay Image Using Color to Distinguish Areas of Similar Intensity

Load an image into the workspace. Create a copy and apply a rotation offset.

```
A = imread('cameraman.tif');  
B = imrotate(A,5,'bicubic','crop');
```

Create a blended overlay image, using red for image A, green for image B, and yellow for areas of similar intensity between the two images. Then, display the overlay image.

```
C = imfuse(A,B,'falsecolor','Scaling','joint','ColorChannels',[1 2 0]);  
imshow(C)
```



Save the resulting image as a .png file.

```
imwrite(C,'my_blend_red-green.png');
```

Create Overlay of Two Spatially Referenced Images

Load an image into the workspace and create a spatial referencing object associated with it.

```
A = dicomread('knee1.dcm');  
RA = imref2d(size(A));
```

Create a second image by resizing image A and create a spatial referencing object associated with that image.

```
B = imresize(A,2);  
RB = imref2d(size(B));
```

Set referencing object parameters to specify the limits of the coordinates in world coordinates.

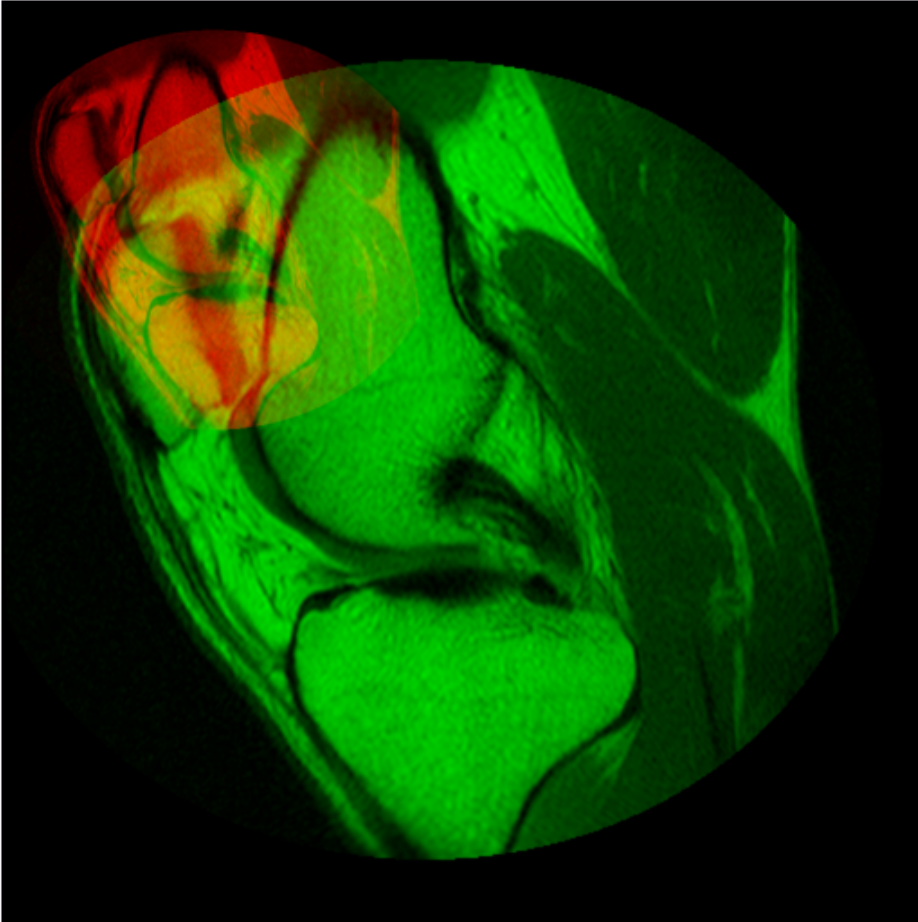
```
RB.XWorldLimits = RA.XWorldLimits;  
RB.YWorldLimits = RA.YWorldLimits;
```

Create a blended overlay image using color to indicate areas of similar intensity. This example uses red for image A, green for image B, and yellow for areas of similar intensity between the two images.

```
C = imfuse(A,B,'falsecolor','Scaling','joint','ColorChannels',[1 2 0]);
```

Display the fused image. Note how the images do not appear to share many areas of similar intensity. For this example, the fused image is shrunk for easier display.

```
C = imresize(C,0.5);  
imshow(C)
```

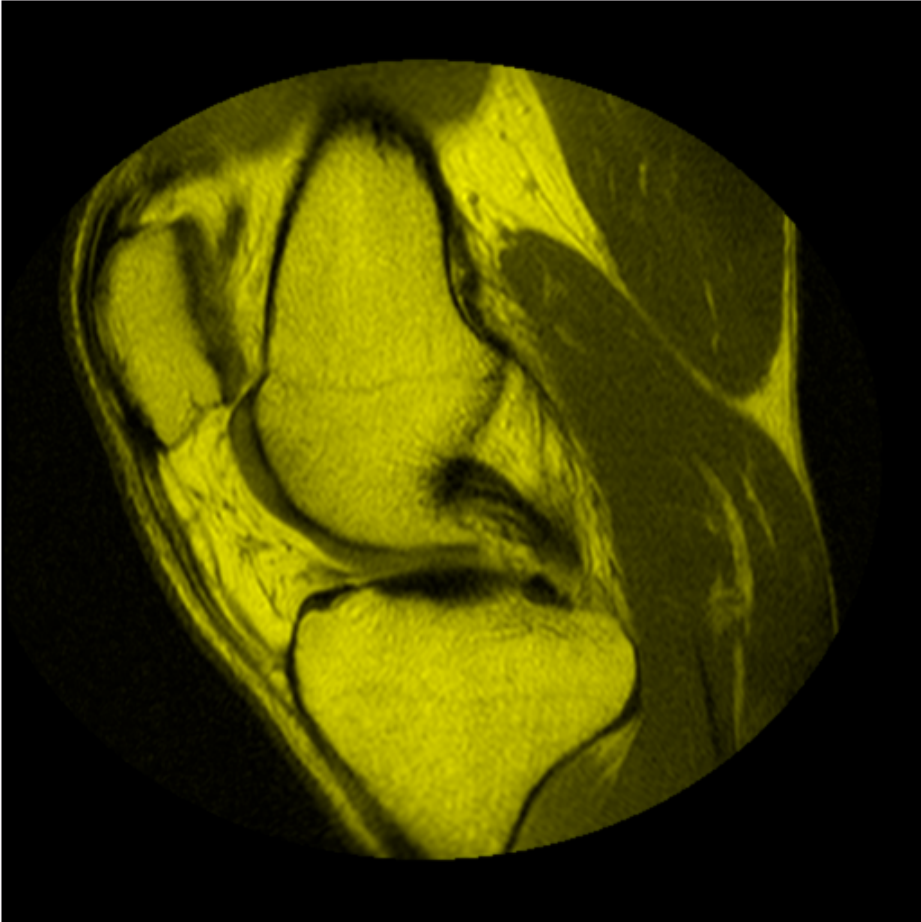


Create a new fused image, this time using the spatial referencing information in RA and RB.

```
[D,RD] = imfuse(A,RA,B,RB, 'ColorChannels', [1 2 0]);
```

Display the new fused image. In this version, the image appears yellow because the images A and B have the same extent in the world coordinate system. The images actually are aligned, even though B is twice the size of A. For this example, the fused image is shrunk for easier display.

```
D = imresize(D,0.5);  
imshow(D)
```



Input Arguments

A — Image to be combined into a composite image

grayscale image | truecolor image | binary image

Image to be combined into a composite image, specified as a grayscale, truecolor, or binary image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

B — Image to be combined into a composite image

grayscale image | truecolor image | binary image

Image to be combined into a composite image, specified as a grayscale, truecolor, or binary image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

RA — Spatial referencing information associated with the input image A

spatial referencing object

Spatial referencing information associated with the input image A, specified as a spatial referencing object of class `imref2d`.

RB — Spatial referencing information associated with the input image B

spatial referencing object

Spatial referencing information associated with the input image B, specified as a spatial referencing object of class `imref2d`.

method — Algorithm used to combine images

'falsecolor' (default) | 'blend' | 'diff' | 'montage'

Algorithm used to combine images, specified as one of the following values.

Method	Description
'falsecolor'	Creates a composite RGB image showing A and B overlaid in different color bands. Gray regions in the composite image show where the two images have the same intensities. Magenta and green regions show where the intensities are different. This is the default method.

Method	Description
'blend'	Overlays A and B using alpha blending.
'checkerboard'	Creates an image with alternating rectangular regions from A and B.
'diff'	Creates a difference image from A and B.
'montage'	Puts A and B next to each other in the same image.

Example: `C = imfuse(A,B,'montage')` places A and B next to each other in the output image.

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example: `'Scaling','joint'` scales the intensity values of A and B together as a single data set.

'Scaling' — Intensity scaling option

'independent' (default) | 'joint' | 'none'

Intensity scaling option, specified as one of the following values:

'independent'	Scales the intensity values of A and B independently when C is created.
'joint'	Scales the intensity values in the images jointly as if they were together in the same image. This option is useful when you want to visualize registrations of monomodal images, where one image contains fill values that are outside the dynamic range of the other image.
'none'	No additional scaling.

'ColorChannels' — Output color channel for each input image

'green-magenta' (default) | [R G B] | 'red-cyan'

Output color channel for each input image, specified as one of the following values:

<code>[R G B]</code>	A three element vector that specifies which image to assign to the red, green, and blue channels. The R, G, and B values must be 1 (for the first input image), 2 (for the second input image), and 0 (for neither image).
<code>'red-cyan'</code>	A shortcut for the vector <code>[1 2 2]</code> , which is suitable for red/cyan stereo anaglyphs
<code>'green-magenta'</code>	A shortcut for the vector <code>[2 1 2]</code> , which is a high contrast option, ideal for people with many kinds of color blindness

Output Arguments

C — Fused image that is a composite of the input images

grayscale image | truecolor image | binary image

Fused image that is a composite of the input images, returned as a grayscale, truecolor, or binary image.

Data Types: `uint8`

RC — Spatial referencing information associated with the output image

spatial referencing object

Spatial referencing information, returned as a spatial referencing object.

Tips

- Use `imfuse` to create composite visualizations that you can save to a file. Use `imshowpair` to display composite visualizations to the screen.

See Also

See Also

`imregister` | `imshowpair` | `imtransform`

Introduced in R2012a

imgaborfilt

Apply Gabor filter or set of filters to 2-D image

Syntax

```
[mag,phase] = imgaborfilt(A,wavelength,orientation)
[mag,phase] = imgaborfilt(A,gaborbank)
[mag,phase] = imgaborfilt(A,wavelength,orientation,name,value,...)
```

Description

`[mag,phase] = imgaborfilt(A,wavelength,orientation)` computes the magnitude and phase response of a Gabor filter for the input grayscale image **A**. **wavelength** describes the wavelength in pixels/cycle of the sinusoidal carrier. **orientation** is the orientation of the filter in degrees. The output **mag** and **phase** are the magnitude and phase responses of the Gabor filter.

`[mag,phase] = imgaborfilt(A,gaborbank)` applies the array of Gabor filters, **gaborbank**, to the input image **A**. **gaborbank** is a 1-by-*P* array of Gabor objects, called a *filter bank*. **mag** and **phase** are image stacks where each plane in the stack corresponds to one of the outputs of the filter bank. For inputs of size **A**, the outputs **mag** and **phase** contain the magnitude and phase response for each filter in **gaborbank** and are of size *M*-by-*N*-by-*P*. Each plane in the magnitude and phase responses, `mag(:, :, ind)`, `phase(:, :, ind)`, is the result of applying the Gabor filter of the same index, `gaborBank(ind)`.

`[mag,phase] = imgaborfilt(A,wavelength,orientation,name,value,...)` applies a single Gabor filter using name-value pairs to control various aspects of filtering.

Examples

Apply Single Gabor Filter to Input Image

Read image into the workspace.

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

Convert image to grayscale.

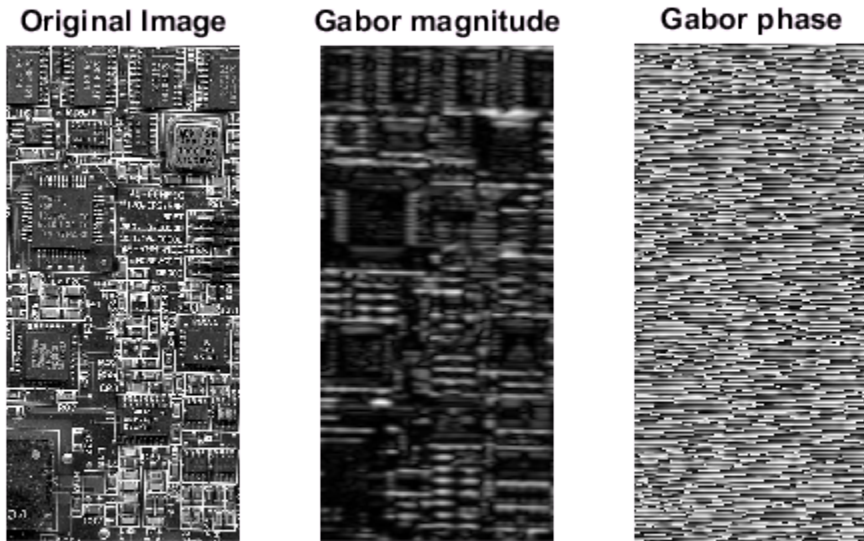
```
I = rgb2gray(I);
```

Apply Gabor filter to image.

```
wavelength = 4;  
orientation = 90;  
[mag,phase] = imgaborfilt(I,wavelength,orientation);
```

Display original image with plots of the magnitude and phase calculated by the Gabor filter.

```
figure  
subplot(1,3,1);  
imshow(I);  
title('Original Image');  
subplot(1,3,2);  
imshow(mag,[]);  
title('Gabor magnitude');  
subplot(1,3,3);  
imshow(phase,[]);  
title('Gabor phase');
```



Apply Array of Gabor Filters to Input Image

Read image into the workspace.

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

Create array of Gabor filters, called a *filter bank*. This filter bank contains two orientations and two wavelengths.

```
gaborArray = gabor([4 8],[0 90]);
```

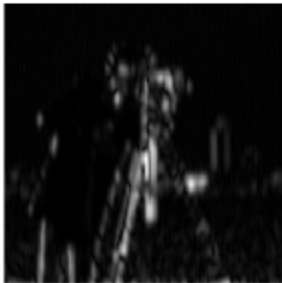
Apply filters to input image.

```
gaborMag = imgaborfilt(I,gaborArray);
```

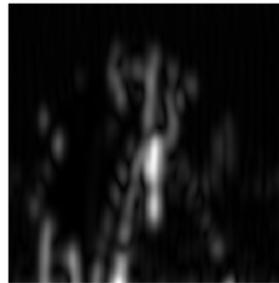
Display results. The figure shows the magnitude response for each filter.

```
figure
subplot(2,2,1);
for p = 1:4
    subplot(2,2,p)
    imshow(gaborMag(:,:,p),[]);
    theta = gaborArray(p).Orientation;
    lambda = gaborArray(p).Wavelength;
    title(sprintf('Orientation=%d, Wavelength=%d',theta,lambda));
end
```

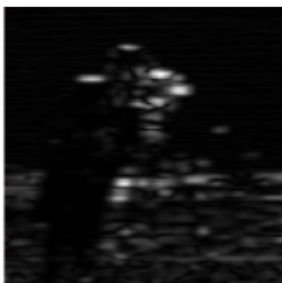
Orientation=0, Wavelength=4



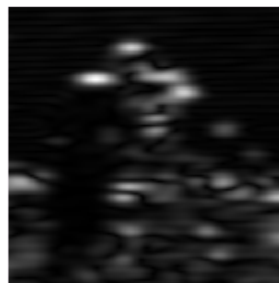
Orientation=0, Wavelength=8



Orientation=90, Wavelength=4



Orientation=90, Wavelength=8



- “Texture Segmentation Using Gabor Filters”

Input Arguments

A — Input grayscale image

real, nonsparse 2-D matrix

Input grayscale image, specified as a real, nonsparse 2-D matrix.

If the image contains `Inf`s or `NaN`s, the behavior of `imgaborfilt` is undefined because Gabor filtering is performed in the frequency domain.

For all input data types other than `single`, `imgaborfilt` performs the computation in `double`. Input images of type `single` are filtered in type `single`. Performance optimizations may result from casting the input image to `single` prior to calling `imgaborfilt`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

wavelength — Wavelength of the sinusoidal carrier

numeric scalar in the range `[2, Inf)`

Wavelength of the sinusoidal carrier, specified as a numeric scalar in the range `[2, Inf)`, in pixels/cycle.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

orientation — Orientation of filter in degrees

numeric scalar in the range `[0 360]`

Orientation of filter in degrees, specified as a numeric scalar in the range `[0 360]`, where the orientation is defined as the normal direction to the sinusoidal plane wave.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

gaborbank — Array of Gabor filters

`gabor` object

Array of Gabor filters, specified as a `gabor` object. You must use the `gabor` function to create an array of Gabor filters.

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, . . . , **NameN**, **ValueN**.

Example: `[mag,phase] = imgaborfilt(I,4,90,'SpatialFrequencyBandwidth',2);`

'SpatialFrequencyBandwidth' — Spatial frequency bandwidth

1.0 (default) | numeric scalar

Spatial frequency bandwidth, specified as a numeric scalar in units of octaves. The spatial frequency bandwidth determines the cutoff of the filter response as frequency content in the input image varies from the preferred frequency, $1/\lambda$. Typical values for spatial-frequency bandwidth are in the range [0.5 2.5].

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'SpatialAspectRatio' — Ratio of semi-major and semi-minor axes of Gaussian envelope

0.5 (default) | numeric scalar

Ratio of semi-major and semi-minor axes of Gaussian envelope (semi-minor/semi-major), specified as a numeric scalar. This parameter controls the ellipticity of the Gaussian envelope. Typical values for spatial aspect ratio are in the range [0.23 0.92].

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

mag — Magnitude response for the Gabor filter

numeric array of class `double`

Magnitude response for the Gabor filter, returned as a numeric array of class `double`.

phase — Phase response for the Gabor filter

numeric array of class `double`

Phase response for the Gabor filter, returned as a numeric array of class `double`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- The `wavelength`, `orientation`, `SpatialFrequencyBandwidth`, and `SpatialAspectRatio` must be compile-time constants.
- The filter bank syntax is not supported.

See Also

See Also

`edge` | `fspecial` | `gabor` | `imfilter` | `imgradient`

Topics

“Texture Segmentation Using Gabor Filters”

Introduced in R2015b

imgaussfilt

2-D Gaussian filtering of images

Syntax

```
B = imgaussfilt(A)
B = imgaussfilt(A,sigma)
B = imgaussfilt(__,Name,Value,...)
gpuarrayB = imgaussfilt(gpuarrayA, __)
```

Description

`B = imgaussfilt(A)` filters image `A` with a 2-D Gaussian smoothing kernel with standard deviation of 0.5. Returns `B`, the filtered image.

`B = imgaussfilt(A,sigma)` filters image `A` with a 2-D Gaussian smoothing kernel with standard deviation specified by `sigma`.

`B = imgaussfilt(__,Name,Value,...)` filters image `A` with Name-Value pairs used to control aspects of the filtering.

`gpuarrayB = imgaussfilt(gpuarrayA, __)` performs the filtering operation on a GPU. The input image must be a `gpuArray`. The function returns a `gpuArray`. This syntax requires the Parallel Computing Toolbox.

Examples

Smooth image with Gaussian filter

Read image to be filtered.

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

Filter the image with a Gaussian filter with standard deviation of 2.

```
Iblur = imgaussfilt(I, 2);
```

Display all results for comparison.

```
subplot(1,2,1)
imshow(I)
title('Original Image');
subplot(1,2,2)
imshow(Iblur)
title('Gaussian filtered image, \sigma = 2')
```

Original Image



Gaussian filtered image, $\sigma = 2$



Smooth Image with Gaussian Filter on a GPU

This example shows how to perform a Gaussian smoothing operation on a GPU.

Read image to be filtered into a `gpuArray`.

```
I = gpuArray(imread('cameraman.tif'));  
  
Perform Gaussian smoothing.  
  
Iblur = imgaussfilt(I, 2);  
  
Display all the results for comparison.  
  
subplot(1,2,1), imshow(I), title('Original Image');  
  
subplot(1,2,2), imshow(Iblur)  
title('Gaussian filtered image, \sigma = 2')
```

Input Arguments

A — Image to be filtered

real, nonsparse matrix of any dimension

Image to be filtered, specified as a real, nonsparse matrix of any dimension.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

sigma — Standard deviation of the Gaussian distribution

0.5 (default) | scalar or 2-element vector of positive values

Standard deviation of the Gaussian distribution, specified as a scalar or 2-element vector of positive values. If you specify a scalar, the Gaussian kernel is square.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

gpuarrayA — Input image for GPU

`gpuArray`

Input image for GPU, specified as a `gpuArray`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `B = imgaussfilt(A, 'FilterSize', 3);`

'FilterSize' — Size of the Gaussian filter

`2*ceil(2*SIGMA)+1` (default) | scalar or 2-element vector of positive, odd integers

Size of the Gaussian filter, specified as a scalar or 2-element vector of positive, odd integers. If you specify a scalar, the filter is a square.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'Padding' — Type of padding to be used on image before filtering

`'replicate'` (default) | numeric scalar | `'circular'` | `'symmetric'`

Padding to be used on image before filtering, specified as a one of the following values, or a numeric scalar. If you specify a scalar (X), input image values outside the bounds of the image are implicitly assumed to have the value X.

Value	Description
<code>'circular'</code>	Pad with circular repetition of elements within the dimension.
<code>'replicate'</code>	Pad by repeating border elements of array.
<code>'symmetric'</code>	Pad image with mirror reflections of itself.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

'FilterDomain' — Domain in which to perform filtering

`'auto'` (default) | `'spatial'` | `'frequency'`

Domain in which to perform filtering, specified as one of the following values:

Value	Description
<code>'auto'</code>	Perform convolution in the spatial or frequency domain, based on internal heuristics.
<code>'frequency'</code>	Perform convolution in the frequency domain.
<code>'spatial'</code>	Perform convolution in the spatial domain.

Data Types: `char`

Output Arguments

B — Filtered image

real, nonsparse matrix

Filtered image, returned as a real, nonsparse matrix, the same size and class as the input image.

gpuarrayB — Filtered image

gpuArray

Filtered image, returned as a gpuArray.

Tips

- If image **A** contains **Infs** or **NaNs**, the behavior of **imgaussfilt** for frequency domain filtering is undefined. This can happen if you set the **'FilterDomain'** parameter to **'frequency'** or if you set it to **'auto'** and **imgaussfilt** uses frequency domain filtering. To restrict the propagation of **Infs** and **NaNs** in the output in a manner similar to **imfilter**, consider setting the **'FilterDomain'** parameter to **'spatial'**.
- When you set the **'FilterDomain'** parameter to **'auto'**, **imgaussfilt** uses an internal heuristic to determine whether spatial or frequency domain filtering is faster. This heuristic is machine dependent and may vary for different configurations. For optimal performance, try both options, **'spatial'** and **'frequency'**, to determine the best filtering domain for your image and kernel size.
- If you do not specify the **'Padding'** parameter, **imgaussfilt** uses **'replicate'** padding by default, which is different from the default used by **imfilter**.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you set the `FilterDomain` parameter to `'frequency'`, this function supports the generation of C code using MATLAB Coder. If you set the `FilterDomain` parameter to `'spatial'` and you choose the generic `MATLAB Host Computer` target platform, generated code uses a precompiled, platform-specific shared library. For more information, see “Code Generation for Image Processing”.
- When generating code, all character vector input arguments must be compile-time constants.

See Also

See Also

`fspecial` | `imfilter` | `imgaussfilt3`

Introduced in R2015a

imgaussfilt3

3-D Gaussian filtering of 3-D images

Syntax

```
B = imgaussfilt3(A)
B = imgaussfilt3(A,sigma)
B = imgaussfilt3( ___,Name,Value,...)
gpuarrayB= imgaussfilt3(gpuarrayA, ___)
```

Description

`B = imgaussfilt3(A)` filters 3-D image `A` with a 3-D Gaussian smoothing kernel with standard deviation of 0.5.

`B = imgaussfilt3(A,sigma)` filters 3-D image `A` with a 3-D Gaussian smoothing kernel with standard deviation specified by `sigma`.

`B = imgaussfilt3(___,Name,Value,...)` filters 3-D image `A` with a 3-D Gaussian smoothing kernel with Name-Value pairs used to control aspects of the filtering.

`gpuarrayB= imgaussfilt3(gpuarrayA, ___)` performs the filtering operation on a GPU. The input image must be a `gpuArray`. The function returns a `gpuArray`. This syntax requires Parallel Computing Toolbox.

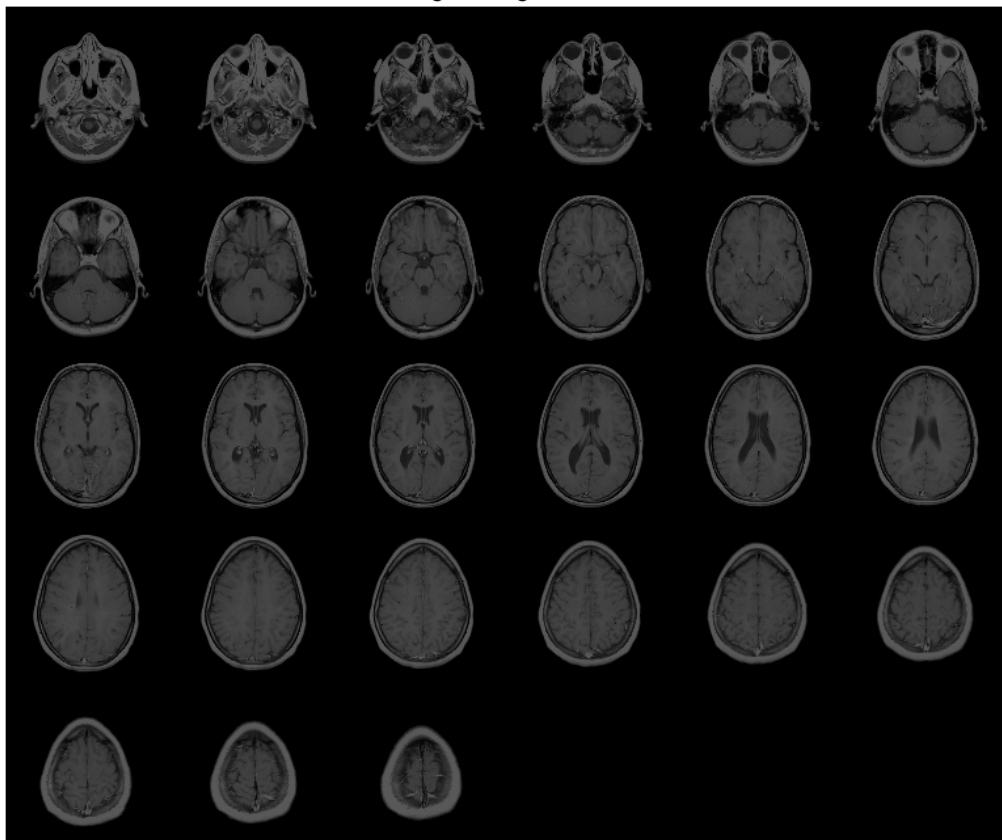
Examples

Smooth MRI volume with 3-D Gaussian filter

Load MRI data and display it.

```
vol = load('mri');
figure
montage(vol.D)
title('Original image volume')
```

Original image volume



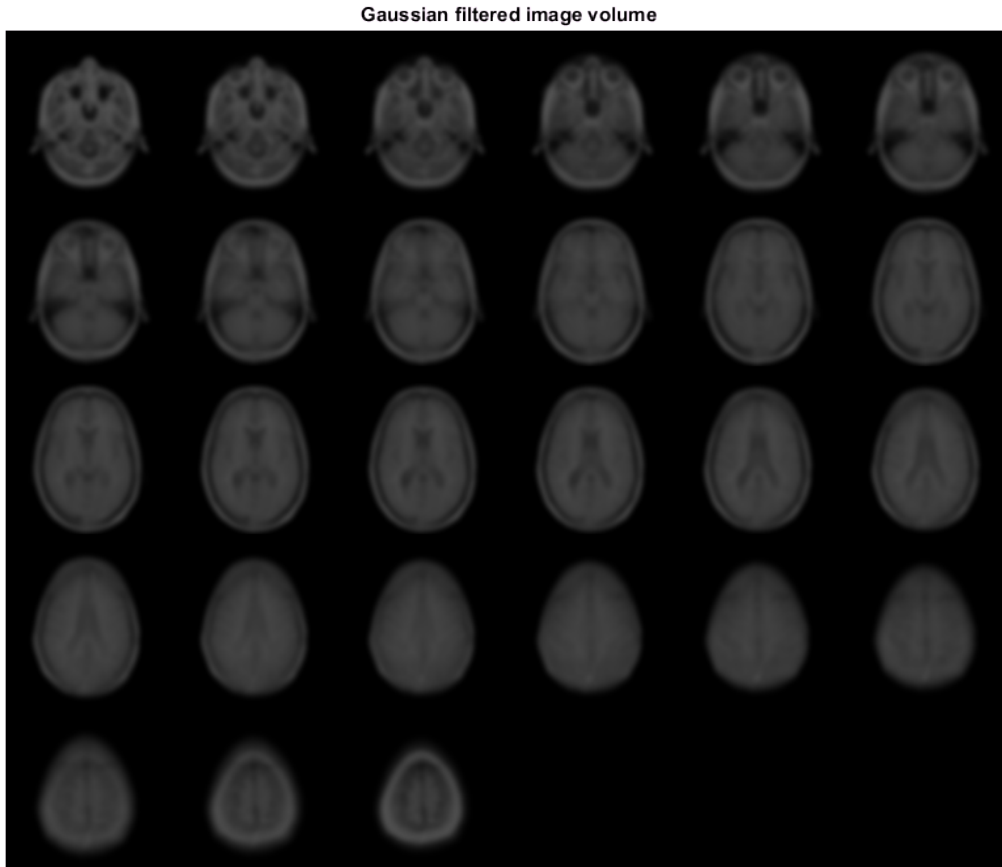
Smooth the image with a 3-D Gaussian filter.

```
siz = vol.siz;  
vol = squeeze(vol.D);  
sigma = 2;
```

```
volSmooth = imgaussfilt3(vol, sigma);
```

```
figure  
montage(reshape(volSmooth,siz(1),siz(2),1,siz(3)))
```

```
title('Gaussian filtered image volume')
```



Smooth MRI Volume with 3-D Gaussian Filter on a GPU

This example shows how to perform a 3-D Gaussian smoothing operation on a GPU.

Load MRI data to be filtered.

```
vol = load('mri');
```

```
figure, montage(vol.D), title('Original image volume')
```

Create a `gpuArray` containing the volume data and perform Gaussian smoothing.

```
siz = vol.siz;  
vol = gpuArray(squeeze(vol.D));  
sigma = 2;  
  
volSmooth = imgaussfilt3(vol, sigma);
```

Collect the smoothed data from the GPU (using the `gather` function) and display all the results for comparison.

```
figure, montage(reshape(gather(volSmooth),siz(1),siz(2),1,siz(3)))  
title('Gaussian filtered image volume')
```

Input Arguments

A — Image to be filtered

3-D, real, nonsparse array

Image to be filtered, specified as a 3-D, real, nonsparse array.

Example: `volSmooth = imgaussfilt3(vol,2);`

Data Types: `single` | `double` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`

sigma — Standard deviation of the Gaussian distribution

0.5 (default) | numeric, real, positive scalar or a 3-element vector

Standard deviation of the Gaussian distribution, specified as a numeric, real, positive scalar or a 3-element vector. If `sigma` is a scalar, `imgaussfilt3` uses a cubic Gaussian kernel.

Example: `volSmooth = imgaussfilt3(vol, 2);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

gpuarrayA — Input image for GPU

`gpuArray`

Input image for GPU, specified as a `gpuArray`.

```
Example: gpuarrayA = gpuArray(imread('cameraman.tif')); gpuarrayB =
imgaussfilt3(gpuarrayA);
```

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

```
Example: Smooth = imgaussfilt3(vol,sigma,'padding','circular');
```

'FilterSize' — Size of the Gaussian filter

$2*\text{ceil}(2*\text{sigma})+1$ (default) | real, positive, odd, integer scalar or 3-element vector

Size of the Gaussian filter, specified as a scalar or 3-element vector of real, positive, odd, integers. If you specify a scalar **Q**, **imgaussfilt3** uses a cubic Gaussian filter of size [**Q Q Q**].

```
Example: volSmooth = imgaussfilt3(vol,sigma,'FilterSize',5);
```

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

'Padding' — Type of padding to be used on image before filtering

'replicate' (default) | 'circular' | 'symmetric' | real, numeric, or logical scalar

Type of padding to be used on image before filtering, specified as one of the following values, or a numeric scalar. If you specify a scalar, **imgaussfilt3** uses that value for input image pixels that fall outside the bounds of the image.

Padding Type	Description
'circular'	Pad with circular repetition of elements within the dimension.
'replicate'	Pad by repeating border elements of array.
'symmetric'	Pad array with mirror reflections of itself.

```
Example: volSmooth = imgaussfilt3(vol,sigma,'padding','circular');
```

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char

'FilterDomain' — Domain in which to perform filtering

'auto' (default) | 'frequency' | 'spatial'

Domain in which to perform filtering, specified as one of the following values.

Filter Domain	Description
'auto'	Perform convolution in the spatial or frequency domain, based on internal heuristics.
'frequency'	Perform convolution in the frequency domain.
'spatial'	Perform convolution in the spatial domain.

Example: `Smooth = imgaussfilt3(vol,sigma,'FilterDomain','frequency');`

Data Types: char

Output Arguments

B — Filtered image

numeric array

Filtered image, returned as a numeric array, the same class and size as input image.

gpuarrayB — Filtered image

gpuArray

Filtered image, returned as a gpuArray.

Tips

- If image **A** contains `Infs` or `Nans`, the behavior of `imgaussfilt3` for frequency domain filtering is undefined. This can happen if you set the `'FilterDomain'` parameter to `'frequency'` or if you set it to `'auto'` and `imgaussfilt3` uses frequency domain filtering. To restrict the propagation of `Infs` and `Nans` in the output in a manner similar to `imfilter`, consider setting the `'FilterDomain'` parameter to `'spatial'`.
- When you set the `'FilterDomain'` parameter to `'auto'`, `imgaussfilt3` uses an internal heuristic to determine whether spatial or frequency domain filtering is faster.

This heuristic is machine-dependent and may vary for different configurations. For optimal performance, try both options, 'spatial' and 'frequency', to determine the best filtering domain for your image and kernel size.

- If you do not specify the 'Padding' parameter, `imgaussfilt3` uses 'replicate' padding by default, which is different from the default used by `imfilter`.

See Also

See Also

`imfilter` | `imgaussfilt`

Introduced in R2015a

imgca

Get current axes containing image

Syntax

```
h = imgca
h = imgca(fig)
```

Description

`h = imgca` returns the current axes that contains an image. The current axes can be in a regular figure window or in an Image Tool window.

If no figure contains an axes that contains an image, `imgca` creates a new axes.

`h = imgca(fig)` returns the current axes that contains an image in the specified figure. (It need not be the current figure.)

Note

`imgca` can be useful in returning the axes object in the Image Tool. You cannot retrieve this axes using `gca`.

Examples

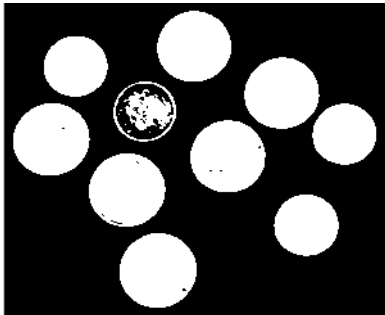
Compute the centroid of each coin, and superimpose its location on the image. View the results using `imtool` and `imgca`.

```
I = imread('coins.png');
figure, imshow(I)
```




Original Image

```
bw = im2bw(I, graythresh(getimage));
figure, imshow(bw)
```

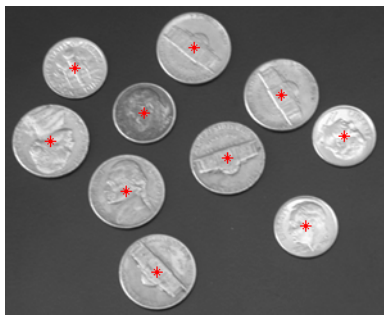


Binary Image

```
bw2 = imfill(bw, 'holes');
s = regionprops(bw2, 'centroid');
centroids = cat(1, s.Centroid);
```

Display original image I and superimpose centroids:

```
imshow(I)
hold(imgca, 'on')
plot(imgca, centroids(:,1), centroids(:,2), 'r*')
hold(imgca, 'off')
```



Centroids of Coins

See also

`gca`, `gcf`, `imgcf`, `imhandles`

Introduced before R2006a

imgcf

Get current figure containing image

Syntax

```
hfig = imgcf
```

Description

`hfig = imgcf` returns the current figure that contains an image. The figure may be a regular figure window that contains at least one image or an Image Tool window.

If none of the figures currently open contains an image, `imgcf` creates a new figure.

Note

`imgcf` can be useful in getting the figure used by the Image Tool. You cannot retrieve the tool figure using `gcf`.

Examples

```
% In this example, use the handle of a figure containing an Image Tool window to center
imshow(rice.png)
sz = get(gcf,'ScreenSize');
pos = get(imgcf,'Position');
pos = [(sz(3)-pos(3))/2 (sz(4)-pos(4))/2 pos(3) pos(4)];
set(imgcf,'Position',pos)
```

See also

`gca`, `gcf`, `imgca`, `imhandles`

Introduced before R2006a

imgetfile

Display Open Image dialog box

Syntax

```
[filename,user_canceled] = imgetfile  
[filename,user_canceled] = imgetfile(Name,Value)
```

Description

`[filename,user_canceled] = imgetfile` displays the Open Image dialog box. Use this dialog box in imaging applications to get the name of the image file a user wants to open. The Open Image dialog box includes only files that use supported image file formats (listed in `imformats`) and DICOM files. When the user selects a file and clicks **Open**, `imgetfile` returns the full path of the file in `filename` and sets the `user_canceled` return value to `false`. If the user clicks **Cancel**, `imgetfile` returns an empty character vector (`' '`) in `filename` and sets the `user_canceled` return value to `true`.

Note: The Open Image dialog box is modal; it blocks the MATLAB command line until the user responds.

`[filename,user_canceled] = imgetfile(Name,Value)` supports name-value parameter arguments that you can use to control aspects of its behavior.

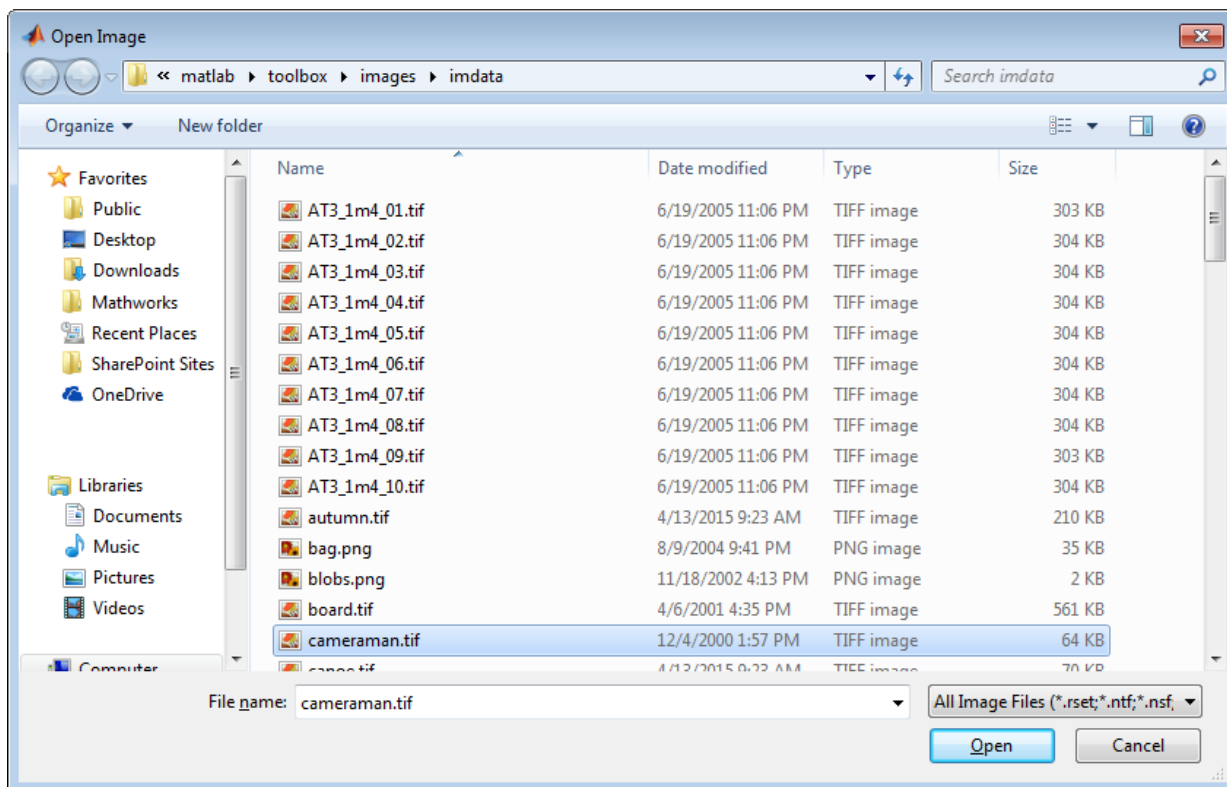
Examples

Get Name of File Selected from Specified Folder

Open the Open Image dialog box, and show the folder that contains the Image Processing Toolbox sample images.

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

```
[filename,user_canceled] = imgetfile('InitialPath',sample_image_folder)
```



Select an image in the list, and click **Open**. `imgetfile` returns the full path of the image file selected as a character vector. The `user_canceled` return value is set to `false`.

```
filename =
```

```
C:\Program Files\MATLAB\R2016b\toolbox\images\imdata\cameraman.tif
```

```
user_canceled =
```

```
logical
```

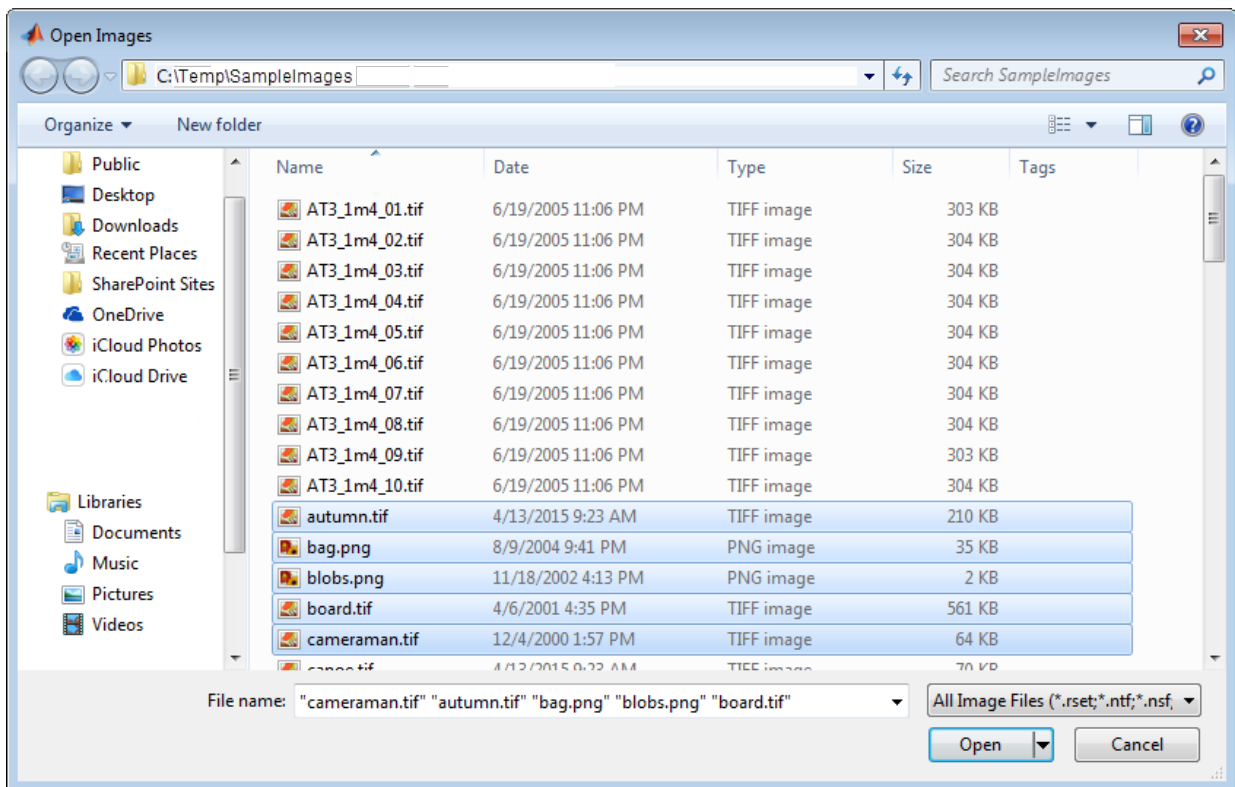
0

Get Names of Multiple Files from Specified Folder

Open the Open Image dialog box. This example assumes you have a folder that contains sample images on your system C: drive.

```
[filename,user_canceled] = imgetfile('InitialPath','C:\Temp\SampleImages','MultiSelect
```

Select several images in the list using **Shift+Click** or **Ctrl+Click**.



Click **Open**. `imgetfile` returns a cell array of character vectors that contain the full path of each image file. The `user_canceled` return value is set to `false`.

```
filename =
```

```

1×5 cell array

Columns 1 through 3
    'C:\Temp\SampleIma...'    'C:\Temp\SampleIma...'    'C:\Temp\SampleIma...'

Columns 4 through 5
    'C:\Temp\SampleIma...'    'C:\Temp\SampleIma...'

user_canceled =

    logical

    0

```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example: `[fname,user_canc] = imgetfile('InitialPath','C:\temp')`

'InitialPath' — Folder displayed when the Open Image dialog box opens

character vector

Folder displayed when the Open Image dialog box opens, specified as a character vector. If you do not specify an initial path, `imgetfile` opens the dialog box at the last location where an image was successfully selected.

Data Types: `char`

'MultiSelect' — Selection mode

`false` (default) | `true` | `'on'` | `'off'`

Selection mode, specified as a Boolean scalar or the character vector `'on'` or `'off'`. The value `true` or `'on'` turns on multiple selection, enabling a user to select more than one

image in the dialog box using **Shift+click** or **Ctrl+click**. The value `false` or `'off'` turns off multiple selection. If multiple selection is on, the output parameter `filename` is a cell array of character vectors containing the full paths to the selected files.

Data Types: `logical` | `char`

Output Arguments

filename — Full path of image or images selected by the user

character vector | cell array of character vectors

Full path of image or images selected by the user, returned as a character vector or cell array of character vectors. If the user clicked **Cancel**, `filename` is an empty character vector (`''`).

user_canceled — User clicked Cancel

`false` | `true`

User clicked **Cancel**, returned as a Boolean scalar. The value is `true` if the user clicked **Cancel** or `false` if the user selected an image or images.

See Also

See Also

`imformats` | `imputfile` | `imtool` | `uigetfile`

Introduced before R2006a

imgradient

Gradient magnitude and direction of an image

Syntax

```
[Gmag,Gdir] = imgradient(I)
[Gmag,Gdir] = imgradient(I,method)
[gpuarrayGmag,gpuarrayGdir] = imgradient(gpuarrayI, ___ )
[Gmag,Gdir] = imgradient(Gx,Gy)
[gpuarrayGmag,gpuarrayGdir] = imgradient(gpuarrayGx,gpuarrayGy)
```

Description

`[Gmag,Gdir] = imgradient(I)` returns the gradient magnitude, `Gmag`, and the gradient direction, `Gdir`, for the grayscale or binary image `I`.

`[Gmag,Gdir] = imgradient(I,method)` returns the gradient magnitude and direction using specified `method`.

`[gpuarrayGmag,gpuarrayGdir] = imgradient(gpuarrayI, ___)` performs the operation on a GPU. The input image and the return values are `gpuArrays`. This syntax requires the Parallel Computing Toolbox.

`[Gmag,Gdir] = imgradient(Gx,Gy)` returns the gradient magnitude and direction using directional gradients along the x -axis, `Gx`, and the y -axis, `Gy`, such as that returned by `imgradientxy`. The x -axis points in the direction of increasing column subscripts and the y -axis points in the direction of increasing row subscripts.

`[gpuarrayGmag,gpuarrayGdir] = imgradient(gpuarrayGx,gpuarrayGy)` performs the operation on a GPU. The input x and y gradients and the return values are `gpuArrays`. This syntax requires the Parallel Computing Toolbox.

Examples

Calculate Gradient Magnitude and Direction Using Prewitt Method

Read an image into workspace.

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

Calculate the gradient magnitude and direction, specifying the Prewitt gradient operator.

```
[Gmag, Gdir] = imgradient(I,'prewitt');
```

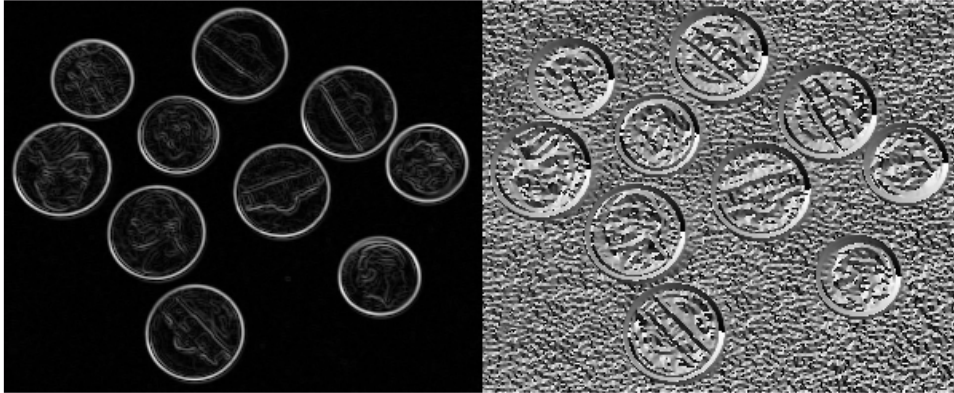
Display the gradient magnitude and direction.

```
figure
```

```
imshowpair(Gmag, Gdir, 'montage');
```

```
title('Gradient Magnitude, Gmag (left), and Gradient Direction, Gdir (right), using Prewitt method');
```

Gradient Magnitude, Gmag (left), and Gradient Direction, Gdir (right), using Prewitt method



Calculate Gradient Magnitude and Direction Using Prewitt Method on a GPU

Read image and compute gradient magnitude and gradient direction using Prewitt's gradient operator.

Read image.

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

Calculate gradients and display.

```
[Gmag, Gdir] = imgradient(I, 'prewitt');
figure, imshow(Gmag, []), title('Gradient magnitude')
figure, imshow(Gdir, []), title('Gradient direction')
```

Calculate Gradient Magnitude and Direction Using Directional Gradients

Read an image into workspace.

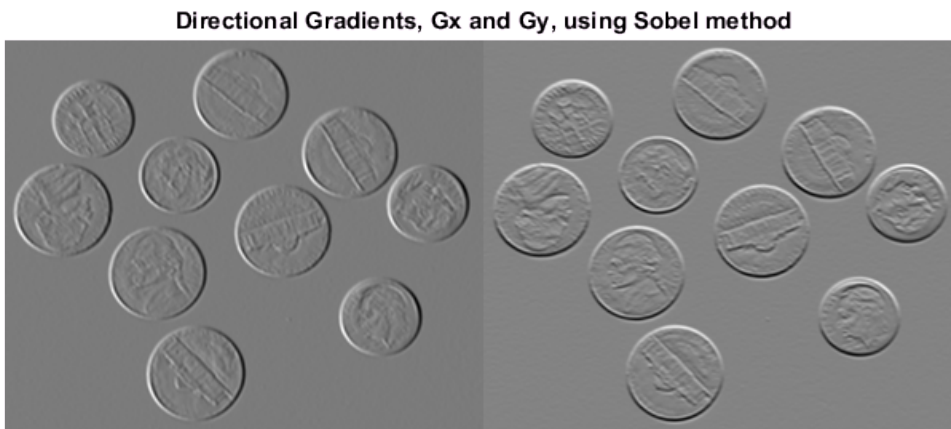
```
I = imread('coins.png');
```

Calculate x- and y-directional gradients, using the Sobel gradient operator by default.

```
[Gx, Gy] = imgradientxy(I);
```

Display the directional gradients.

```
figure
imshowpair(Gx, Gy, 'montage')
title('Directional Gradients, Gx and Gy, using Sobel method')
```



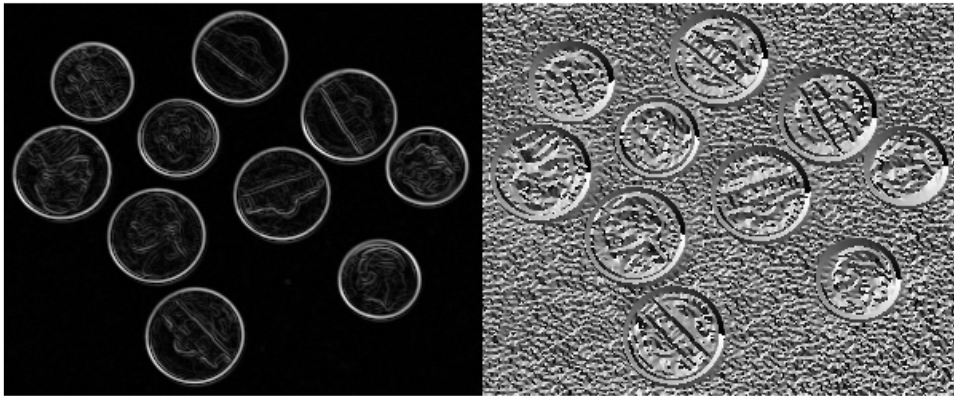
Calculate the gradient magnitude and direction using the directional gradients.

```
[Gmag, Gdir] = imgradient(Gx, Gy);
```

Display the gradient magnitude and direction.

```
figure
imshowpair(Gmag, Gdir, 'montage')
title('Gradient Magnitude, Gmag (left), and Gradient Direction, Gdir (right), using Sobel
```

Gradient Magnitude, Gmag (left), and Gradient Direction, Gdir (right), using Sobel method



Calculate Gradient Magnitude and Direction Using Directional Gradients on a GPU

Read image and return directional gradients, G_x and G_y , as well as gradient magnitude and direction, G_{mag} and G_{dir} , utilizing default Sobel gradient operator.

Read image.

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

Calculate gradients and display them. Note that when you specify a `gpuArray` to `imgradientxy`, it returns G_x and G_y as `gpuArrays`. The results are the same as the previous example.

```
[Gx, Gy] = imgradientxy(I);
[Gmag, Gdir] = imgradient(Gx, Gy);
```

```
figure, imshow(Gmag, []), title('Gradient magnitude')
```

```
figure, imshow(Gdir, []), title('Gradient direction')
figure, imshow(Gx, []), title('Directional gradient: X axis')
figure, imshow(Gy, []), title('Directional gradient: Y axis')
```

Input Arguments

I — Input image

grayscale image | binary image

Input image, specified as a grayscale or binary image, that is, a numeric or logical 2-D matrix that must be nonsparse.

Data Types: single | double | int8 | int32 | uint8 | uint16 | uint32 | logical

gpuarrayI — Input image

gpuArray

Input image, specified as a 2-D grayscale or binary gpuArray image.

Data Types: single | double | int8 | int32 | uint8 | uint16 | uint32 | logical

method — Gradient operator

'sobel' (default) | 'prewitt' | 'central' | 'intermediate' | 'roberts'

Gradient operator, specified as one of the following values.

Method	Description
'sobel'	Sobel gradient operator (default)
'prewitt'	Prewitt gradient operator
'central'	Central difference gradient: $dI/dx = (I(x+1) - I(x-1))/2$
'intermediate'	Intermediate difference gradient: $dI/dx = I(x+1) - I(x)$
'roberts'	Roberts gradient operator

Data Types: char

Gx — Directional gradients along x-axis (horizontal)

matrix

Directional gradient along x -axis (horizontal), specified as non-sparse matrix equal in size to image **I**, typically returned by `imgradientxy`.

Data Types: `single` | `double` | `int8` | `int32` | `uint8` | `uint16` | `uint32`

Gy — Directional gradients along the y -axis (vertical)

`matrix`

Directional gradient along y -axis (vertical), specified as non-sparse matrix equal in size to image **I**, typically returned by `imgradientxy`.

Data Types: `single` | `double` | `int8` | `int32` | `uint8` | `uint16` | `uint32`

gpuarrayGx — Directional gradients along x -axis

`gpuArray`

Directional gradient along x -axis, specified as a `gpuArray`, typically returned by `imgradientxy`.

Data Types: `single` | `double` | `int8` | `int32` | `uint8` | `uint16` | `uint32`

gpuarrayGy — Directional gradients along the y -axis

`gpuArray`

Directional gradient along y -axis, specified as a `gpuArray`, typically returned by `imgradientxy`.

Data Types: `single` | `double` | `int8` | `int32` | `uint8` | `uint16` | `uint32`

Output Arguments

Gmag — Gradient magnitude

`matrix`

Gradient magnitude, returned as a non-sparse matrix the same size as image **I**. **Gmag** is of class `double`, unless the input image **I** is of class `single`, in which case it is of class `single`.

Data Types: `double` | `single`

gpuarrayGmag — Gradient magnitude

`gpuArray`

Gradient magnitude, returned as a non-sparse `gpuArray` the same size as image `I`. `Gmag` is of class `double`, unless the input image `I` is of class `single`, in which case it is of class `single`.

Data Types: `double` | `single`

Gdir — Gradient direction

`matrix` | `gpuArray`

Gradient direction, returned as a nonsparse matrix the same size as image `I`. `Gdir` contains angles in degrees within the range `[-180 180]` measured counterclockwise from the positive x -axis. (The x -axis points in the direction of increasing column subscripts.) `Gdir` is of class `double`, unless the input image `I` is of class `single`, in which case it is of class `single`.

When `I` or `Gx` and `Gy` are `gpuArrays`, `Gdir` is a `gpuArray`.

Data Types: `double` | `single`

gpuarrayGdir — Gradient direction

`gpuArray`

Gradient direction, returned as a nonsparse `gpuArray` the same size as image `I`. `Gdir` contains angles in degrees within the range `[-180 180]` measured counterclockwise from the positive x -axis. (The x -axis points in the direction of increasing column subscripts.) `Gdir` is of class `double`, unless the input image `I` is of class `single`, in which case it is of class `single`.

Data Types: `double` | `single`

Tips

- When applying the gradient operator at the boundaries of the image, values outside the bounds of the image are assumed to equal the nearest image border value. This is similar to the `'replicate'` boundary option in `imfilter`.

Algorithms

The algorithmic approach taken in `imgradient` for each of the listed gradient methods is to first compute directional gradients, `Gx` and `Gy`, with respect to the x -axis and y -axis.

The x -axis is defined along the columns going right and the y -axis is defined along the rows going down. The gradient magnitude and direction are then computed from their orthogonal components G_x and G_y .

`imgradient` does not normalize the gradient output. If the range of the gradient output image has to match the range of the input image, consider normalizing the gradient image, depending on the `method` argument used. For example, with a Sobel kernel, the normalization factor is $1/8$, for Prewitt, it is $1/6$, and for Roberts it is $1/2$.

See Also

See Also

`edge` | `fspecial` | `gpuArray` | `gpuArray` | `imgradient3` | `imgradientxy` | `imgradientxyz`

Introduced in R2012b

imgradient3

Find 3-D gradient magnitude and direction of images

Syntax

```
[Gmag,Gazimuth,Gelevation] = imgradient3(I)
[Gmag,Gazimuth,Gelevation] = imgradient3(I,method)
[Gmag,Gazimuth,Gelevation] = imgradient3(Gx,Gy,Gz)
```

Description

`[Gmag,Gazimuth,Gelevation] = imgradient3(I)` returns the gradient magnitude, `Gmag`, gradient direction, `Gazimuth`, and gradient elevation `Gelevation` for the grayscale or binary 3-D image `I`.

`[Gmag,Gazimuth,Gelevation] = imgradient3(I,method)` calculates the gradient magnitude and direction using the specified `method`.

`[Gmag,Gazimuth,Gelevation] = imgradient3(Gx,Gy,Gz)` calculates the gradient magnitude and direction from the directional gradients along the x -axis, `Gx`, y -axis, `Gy` and z -axis, `Gz`.

Examples

Compute 3-D Gradient Magnitude and Direction Using Sobel Method

Read 3-D data into the workspace and prepare it for processing.

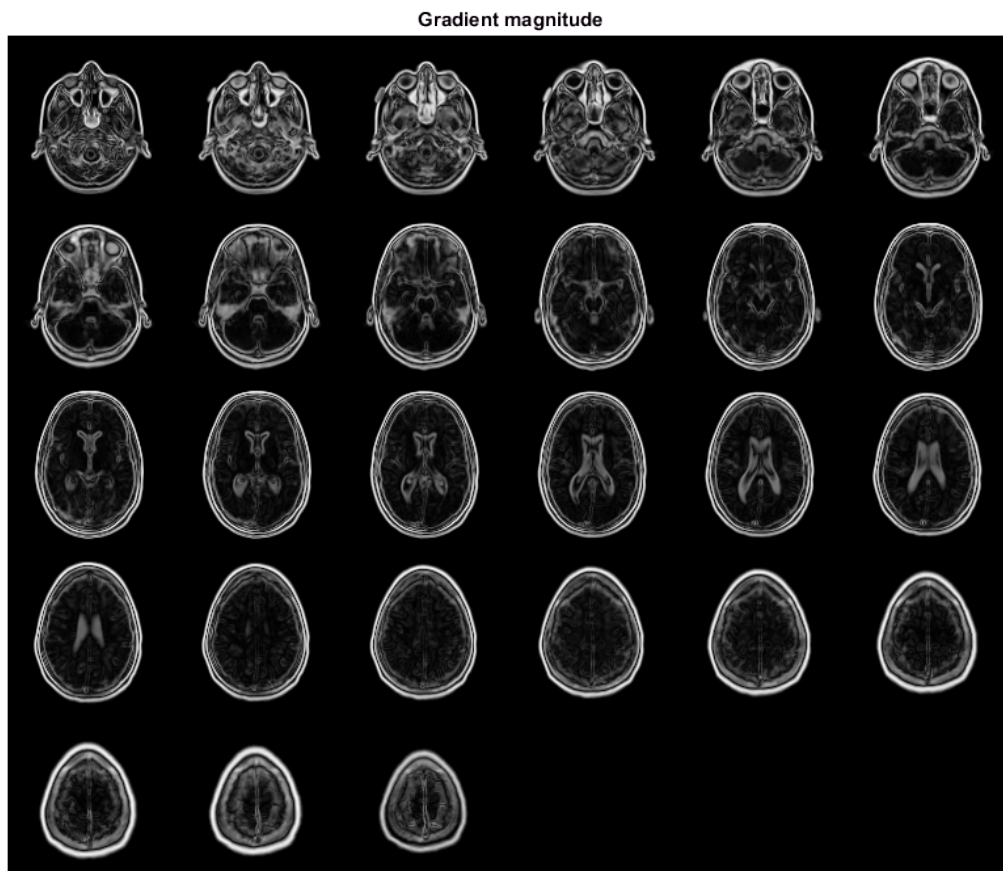
```
volData = load('mri');
sz = volData.siz;
vol = squeeze(volData.D);
```

Calculate the gradients.

```
[Gmag, Gaz, Gelev] = imgradient3(vol);
```

Visualize the gradient magnitude as a montage.

```
figure,  
montage(reshape(Gmag,sz(1),sz(2),1,sz(3)),'DisplayRange',[,])  
title('Gradient magnitude')
```



Input Arguments

I — Input image

nonsparse, numeric or logical, 3-D matrix

Input image, specified as a nonsparse, numeric or logical, 3-D matrix

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

method — Gradient operator

'sobel' (default) | 'prewitt' | 'central' | 'intermediate'

Gradient operator, specified as one of the following values.

Value	Meaning
'sobel'	Sobel gradient operator (default)
'prewitt'	Prewitt gradient operator
'central'	Central difference gradient. $dI/dx = (I(x+1) - I(x-1)) / 2$
'intermedia'	Intermediate difference gradient. $dI/dx = I(x+1) - I(x)$

When applying the gradient operator at the boundaries of the image, `imgradient3` assumes values outside the bounds of the image equal the nearest image border value. This behavior is similar to the 'replicate' boundary option in `imfilter`.

Data Types: char

Gx — Directional gradients along x-axis (horizontal)

nonsparse, numeric or logical, 3-D matrix

Directional gradient along x -axis (horizontal), specified as a nonsparse, numeric or logical, 3-D matrix. The x -axis points in the direction of increasing column subscripts. The matrix must be equal in size to `Gy` and `Gz`. `imgradientxyz` returns `Gx`.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Gy — Directional gradients along the y-axis (vertical)

nonsparse, numeric or logical, 3-D matrix

Directional gradient along y -axis (vertical), specified as a nonsparse, numeric or logical, 3-D matrix. The y -axis points in the direction of increasing row subscripts. The matrix must be equal in size to `Gx` and `Gz`. `imgradientxyz` returns `Gy`.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Gz — Directional gradients along the z-axis

nonsparse, numeric or logical, 3-D matrix

Directional gradient along z -axis, specified as a nonsparse, numeric or logical, 3-D matrix. The matrix must be equal in size to G_x and G_y . `imgradientxyz` returns G_z .

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

Gmag — Magnitude of the gradient vector

nonsparse matrix

Magnitude of the gradient vector, returned as a nonsparse matrix the same size as image I .

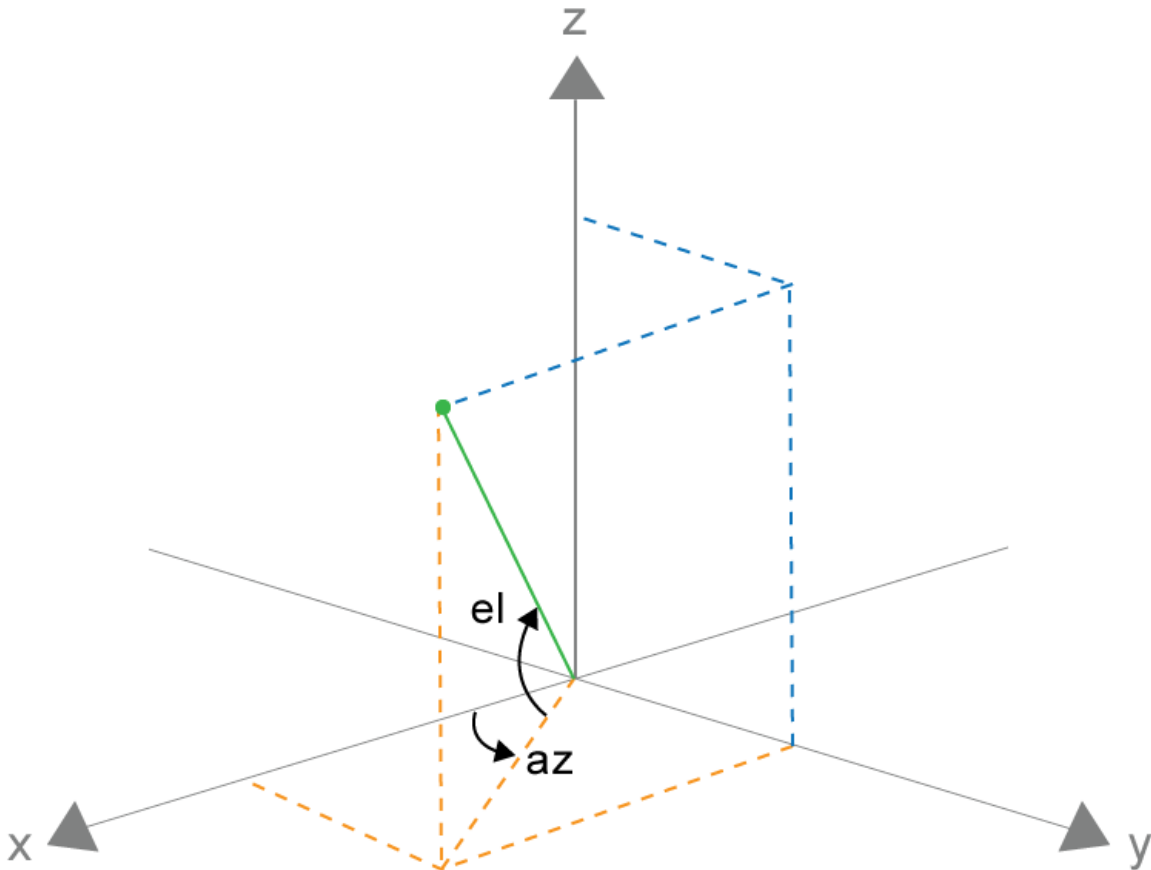
G_{mag} is of class `double`, unless the input image I or any of the directional gradients G_x , G_y , or G_z are of class `single`. In this case, G_{mag} is of class `single`.

Gazimuth — Azimuthal angle

nonsparse matrix

Azimuthal angle, returned as a nonsparse matrix the same size as image I . $G_{azimuth}$ contains angles in degrees within the range $[-180\ 180]$ measured between positive x -axis and the projection of the point on the x - y plane.

$G_{azimuth}$ is of class `double`, unless the input image I or any of the directional gradients G_x , G_y , or G_z are of class `single`. In this case, $G_{azimuth}$ is of class `single`.



Gazimuth and Gelevation

Gelevation — Gradient elevation

nonsparse matrix

Gradient elevation, returned as a nonsparse matrix the same size as image I. Gelevation contains angles in degrees within the range [-90 90] measured between the radial line and the x - y plane.

`Gelevation` is of class `double`, unless the input image `I` or any of the directional gradients `Gx`, `Gy`, or `Gz` are of class `single`. In this case, `Gelevation` is of class `single`.

Algorithms

`imgradient3` does not normalize the gradient output. If the range of the gradient output image has to match the range of the input image, consider normalizing the gradient image, depending on the `method` argument used. For example, with a Sobel kernel, the normalization factor is $1/44$ and for Prewitt, the normalization factor is $1/18$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, the input argument `method` must be a compile-time constant.

See Also

See Also

`imgradient` | `imgradientxy` | `imgradientxyz`

Introduced in R2016a

imgradientxy

Directional gradients of an image

Syntax

```
[Gx,Gy] = imgradientxy(I)  
[Gx,Gy] = imgradientxy(I,method)  
[gpuarrayGx,gpuarrayGy] = imgradientxy(gpuarrayI, ___)
```

Description

`[Gx,Gy] = imgradientxy(I)` returns the directional gradients, `Gx` and `Gy`, the same size as the input image `I`.

When applying the gradient operator at the boundaries of the image, values outside the bounds of the image are assumed to equal the nearest image border value.

`[Gx,Gy] = imgradientxy(I,method)` returns the directional gradients using the specified `method`.

`[gpuarrayGx,gpuarrayGy] = imgradientxy(gpuarrayI, ___)` performs the operation on a GPU. The input image and the return values are `gpuArrays`. This syntax requires the Parallel Computing Toolbox

Examples

Calculate Directional Gradients Using Prewitt Method

Read an image into workspace.

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

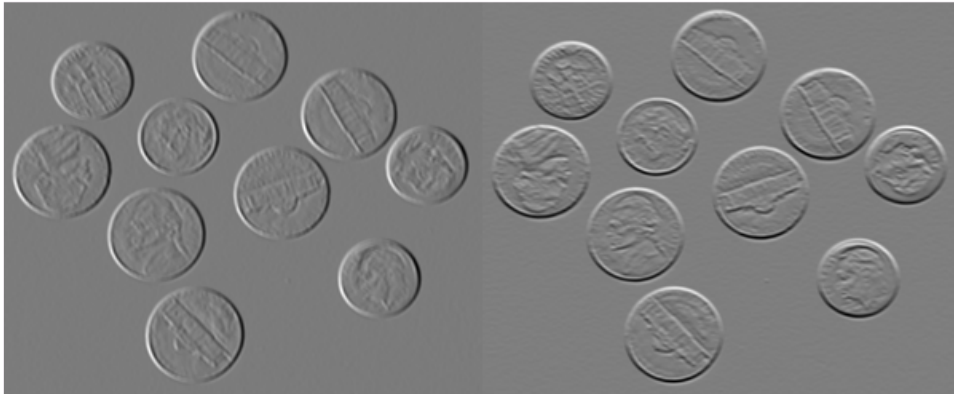
Calculate the *x*- and *y*-directional gradients using the Prewitt gradient operator.

```
[Gx, Gy] = imgradientxy(I,'prewitt');
```

Display the directional gradients.

```
figure
imshowpair(Gx, Gy, 'montage');
title('Directional Gradients: x-direction, Gx (left), y-direction, Gy (right), using Prewitt method')
```

Directional Gradients: x-direction, Gx (left), y-direction, Gy (right), using Prewitt method



Calculate directional gradients on a GPU

Read image into a gpuArray.

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

Calculate gradient magnitude and gradient direction using Prewitt's gradient operator and display images.

```
[Gx, Gy] = imgradientxy(I, 'prewitt');

figure, imshow(Gx, []), title('Directional gradient: X axis')
figure, imshow(Gy, []), title('Directional gradient: Y axis')
```

Calculate Gradient Magnitude and Direction Using Directional Gradients

Read an image into workspace.

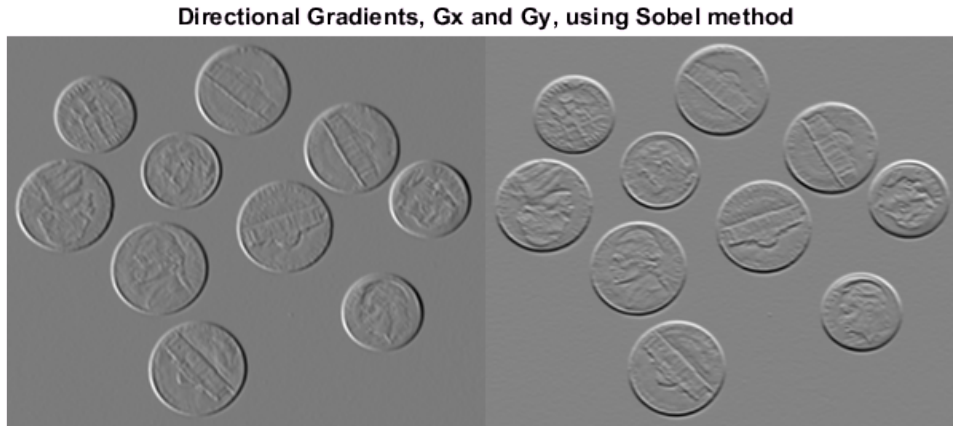

```
I = imread('coins.png');
```

Calculate x - and y -directional gradients, using the Sobel gradient operator by default.

```
[Gx, Gy] = imgradientxy(I);
```

Display the directional gradients.

```
figure
imshowpair(Gx, Gy, 'montage')
title('Directional Gradients, Gx and Gy, using Sobel method')
```



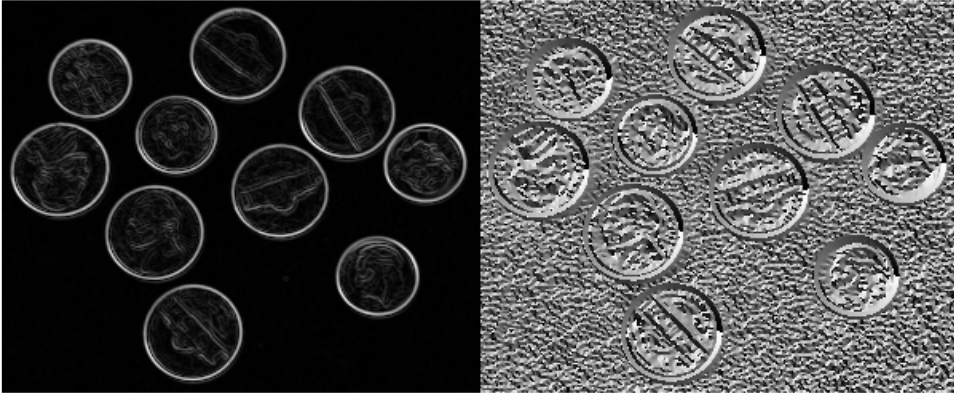
Calculate the gradient magnitude and direction using the directional gradients.

```
[Gmag, Gdir] = imgradient(Gx, Gy);
```

Display the gradient magnitude and direction.

```
figure
imshowpair(Gmag, Gdir, 'montage')
title('Gradient Magnitude, Gmag (left), and Gradient Direction, Gdir (right), using Sobel method')
```

Gradient Magnitude, Gmag (left), and Gradient Direction, Gdir (right), using Sobel method



Calculate gradient magnitude and direction in addition to directional gradients on a GPU

Read image and return directional gradients, G_x and G_y , as well as gradient magnitude and direction, G_{mag} and G_{dir} , utilizing default Sobel gradient operator.

Read image into a `gpuArray`.

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

Calculate gradient and display images.

```
[Gx, Gy] = imgradientxy(I);  
[Gmag, Gdir] = imgradient(Gx, Gy);  
  
figure, imshow(Gmag, []), title('Gradient magnitude')  
figure, imshow(Gdir, []), title('Gradient direction')  
figure, imshow(Gx, []), title('Directional gradient: X axis')
```

```
figure, imshow(Gy, []), title('Directional gradient: Y axis')
```

Input Arguments

I — Input image

grayscale image | binary image

Input image, specified as a grayscale or binary image, that is, a numeric or logical 2-D matrix that must be nonsparse, or a `gpuArray`.

Data Types: `single` | `double` | `int8` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

gpuarrayI — Input image

`gpuArray`

Input image, specified as a 2-D grayscale or binary `gpuArray` image.

Data Types: `single` | `double` | `int8` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

method — Gradient operator

'sobel' (default) | 'prewitt' | 'central' | 'intermediate'

Gradient operator, specified as one of the following values.

Method	Description
'sobel'	Sobel gradient operator (default)
'prewitt'	Prewitt gradient operator
'central'	Central difference gradient: $dI/dx = (I(x+1) - I(x-1))/2$
'intermediate'	Intermediate difference gradient: $dI/dx = I(x+1) - I(x)$

Data Types: `char`

Output Arguments

Gx — Directional gradients along x-axis

matrix

Directional gradient along the x -axis, returned as non-sparse matrix equal in size to image **I**. The x -axis points in the direction of increasing column subscripts. The output matrices are of class **double**, unless the input image is of class **single**, in which case they are of class **single**.

Data Types: `single` | `double`

gpuarrayGx — Directional gradients along x -axis

`gpuArray`

Directional gradient along the x -axis, returned as non-sparse `gpuArray` equal in size to image **I**. The x -axis points in the direction of increasing column subscripts. The output matrices are of class **double**, unless the input image is of class **single**, in which case they are of class **single**.

Data Types: `single` | `double`

Gy — Directional gradient along y -axis

`matrix`

Directional gradients along the y -axis, returned as non-sparse matrix equal in size to image **I**. The y -axis points in the direction of increasing row subscripts. The output matrices are of class **double**, unless the input image is of class **single**, in which case they are of class **single**.

Data Types: `single` | `double`

gpuarrayGy — Directional gradient along y -axis

`gpuArray`

Directional gradients along the y -axis, returned as non-sparse `gpuArray` equal in size to image **I**. The y -axis points in the direction of increasing row subscripts. The output matrices are of class **double**, unless the input image is of class **single**, in which case they are of class **single**.

Data Types: `single` | `double`

Tips

- When applying the gradient operator at the boundaries of the image, values outside the bounds of the image are assumed to equal the nearest image border value.

Algorithms

The algorithmic approach is to compute directional gradients with respect to the x -axis and y -axis. The x -axis is defined along the columns going right and the y -axis is defined along the rows going down.

`imgradientxy` does not normalize the gradient output. If the range of the gradient output image has to match the range of the input image, consider normalizing the gradient image, depending on the `method` argument used. For example, with a Sobel kernel, the normalization factor is $1/8$, and for Prewitt, it is $1/6$.

See Also

See Also

`edge` | `fspecial` | `gpuArray` | `imgradient` | `imgradient3` | `imgradientxyz`

Introduced in R2012b

imgradientxyz

Find the directional gradients of a 3-D image

Syntax

```
[Gx,Gy,Gz] = imgradientxyz(I)  
[Gx,Gy,Gz] = imgradientxyz(I,method)
```

Description

`[Gx,Gy,Gz] = imgradientxyz(I)` returns the gradient along the x -axis, Gx , the y -axis, Gy and the z -axis, Gz , for the grayscale or binary 3-D image I .

`[Gx,Gy,Gz] = imgradientxyz(I,method)` calculates the directional gradients using the specified method.

Examples

Compute 3-D Directional Image Gradients Using Sobel Method

Read 3-D data and prepare it for processing.

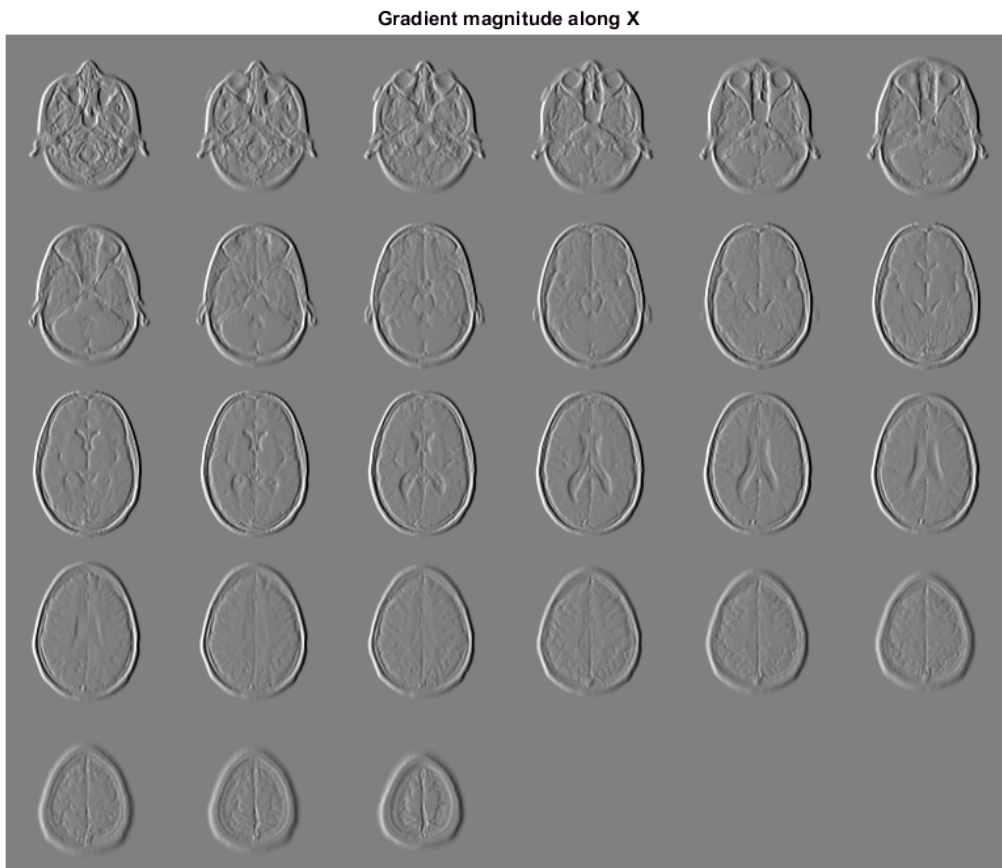
```
volData = load('mri');  
sz = volData.siz;  
vol = squeeze(volData.D);
```

Calculate the directional gradients.

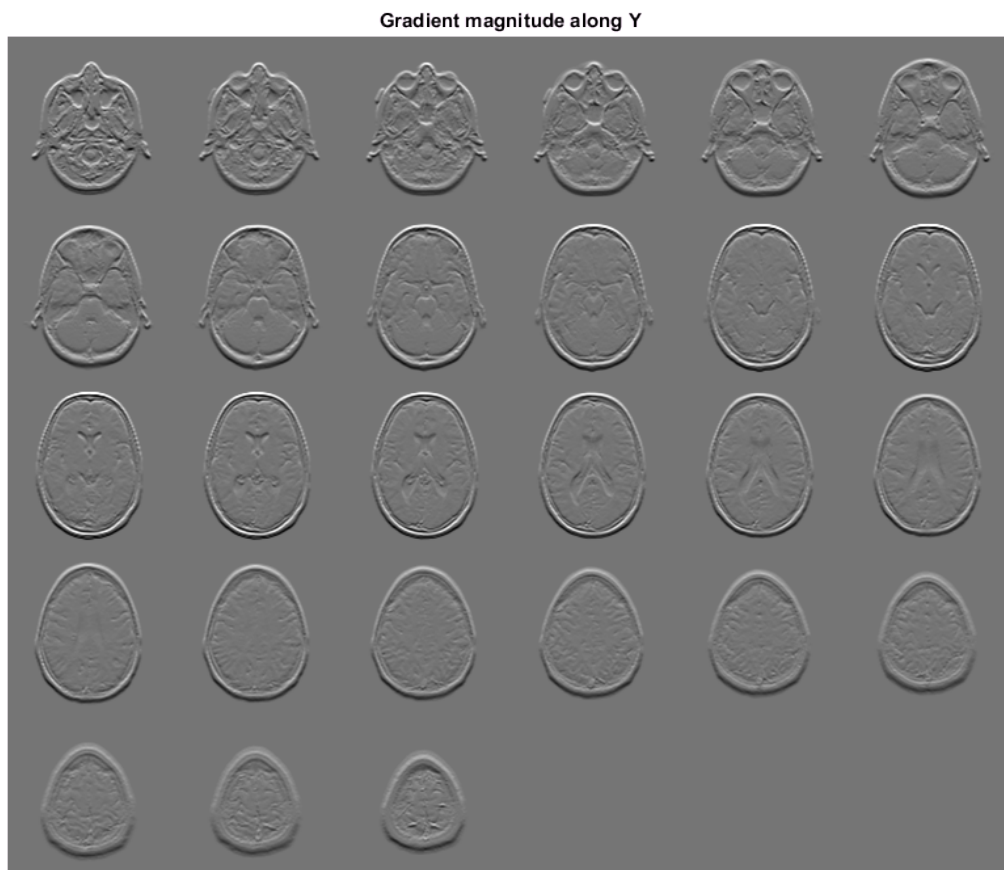
```
[Gx, Gy, Gz] = imgradientxyz(vol);
```

Visualize the directional gradients as a montage.

```
figure, montage(reshape(Gx,sz(1),sz(2),1,sz(3)), 'DisplayRange', [])  
title('Gradient magnitude along X')
```

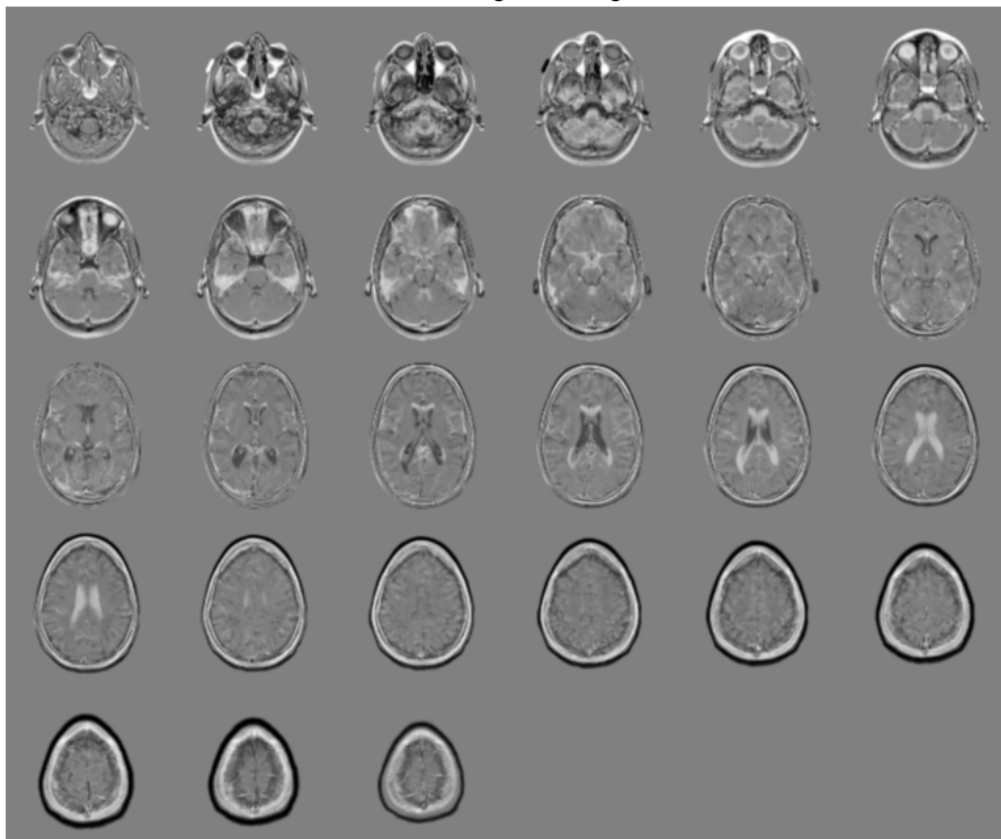


```
figure, montage(reshape(Gy,sz(1),sz(2),1,sz(3)), 'DisplayRange', [])  
title('Gradient magnitude along Y')
```



```
figure, montage(reshape(Gz,sz(1),sz(2),1,sz(3)), 'DisplayRange', [])  
title('Gradient magnitude along Z')
```


Gradient magnitude along Z



Input Arguments

I — Input image

nonsparse, numeric or logical, 3-D matrix

Input image, specified as a nonsparse, numeric or logical, 3-D matrix

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

method — Gradient operator

'sobel' (default) | 'prewitt' | 'central' | 'intermediate'

Gradient operator, specified as one of the following values.

Value	Meaning
'sobel'	Sobel gradient operator (default)
'prewitt'	Prewitt gradient operator
'central'	Central difference gradient. $dI/dx = (I(x+1) - I(x-1)) / 2$
'intermedia'	Intermediate difference gradient. $dI/dx = I(x+1) - I(x)$

When applying the gradient operator at the boundaries of the image, `imgradientxyz` assumes values outside the bounds of the image are equal to the nearest image border value. This behavior is similar to the 'replicate' boundary option in `imfilter`.

Data Types: `char`

Output Arguments

Gx — Directional gradients along x-axis (horizontal)

nonsparse matrix

Directional gradient along *x*-axis (horizontal), returned as nonsparse matrix equal in size to image *I*. The *X*-axis points in the direction of increasing column subscripts. **Gx** is of class `double`, unless the input image *I* is of class `single`, in which case **Gx** is of class `single`.

Gy — Directional gradients along the y-axis (vertical)

nonsparse matrix

Directional gradient along *y*-axis (vertical), returned as non-sparse matrix equal in size to image *I*. *Y*-axis points in the direction of increasing row subscripts. **Gy** is of class `double`, unless the input image *I* is of class `single`, in which case **Gy** is of class `single`.

Gz — Directional gradients along the z-axis

nonsparse matrix

Directional gradient along z -axis, returned as nonsparse matrix equal in size to image `I`. The Z -axis points in the direction of increasing third dimension subscripts. `Gz` is of class `double`, unless the input image `I` is of class `single`, in which case `Gz` is of class `single`.

Algorithms

`imgradientxyz` does not normalize the gradient output. If the range of the gradient output image has to match the range of the input image, consider normalizing the gradient image, depending on the `method` argument used. For example, with a Sobel kernel, the normalization factor is $1/44$, for Prewitt, the normalization factor is $1/18$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, the input argument `method` must be a compile-time constant.

See Also

See Also

`imgradient` | `imgradient3` | `imgradientxy`

Introduced in R2016a

imguidedfilter

Guided filtering of images

Syntax

```
B = imguidedfilter(A,G)
B = imguidedfilter(A)
B = imguidedfilter(__,Name,Value,...)
```

Description

`B = imguidedfilter(A,G)` filters binary, grayscale, or RGB image `A` using the guided filter, where the filtering process is guided by image `G`. `G` can be a binary, grayscale or RGB image and must have the same number of rows and columns as `A`.

`B = imguidedfilter(A)` filters input image `A` under self-guidance, using `A` itself as the guidance image. This can be used for edge-preserving smoothing of image `A`.

`B = imguidedfilter(__,Name,Value,...)` filters the image `A` using name-value pairs to control aspects of guided filtering. Parameter names can be abbreviated.

Examples

Perform Edge-Preserving Smoothing Using Guided Filtering

This example shows how to perform edge-preserving smoothing using a guide filter.

Read an image into the workspace.

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

Smooth the image using `imguifilter`. In this syntax, `imguidedfilter` uses the image itself as the guidance image.

```
Ismooth = imguidedfilter(A);
```

Display the original image and the smoothed image side-by-side.

```
imshowpair(A,Ismooth,'montage');
```



- “Perform Flash/No-flash Denoising with Guided Filter”

Input Arguments

A — Image to be filtered

binary image | grayscale image | RGB image

Image to be filtered, specified as a nonsparse, binary, grayscale, or RGB image.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | logical

G — Image to use as a guide during filtering

binary image | grayscale image | RGB image

Image to use as a guide during filtering, specified as a nonsparse, binary, grayscale, or RGB image.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | logical

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `Ismooth = imguidedfilter(A,'NeighborhoodSize',[4 4]);`

'NeighborhoodSize' — Size of the rectangular neighborhood around each pixel used in guided filtering`[5 5]` (default) | scalar or two-element vector of positive integers

Size of the rectangular neighborhood around each pixel used in guided filtering, specified as a scalar or a two-element vector, `[M N]`, of positive integers. If you specify a scalar value, such as `Q`, the neighborhood is a square of size `[Q Q]`.

Example: `Ismooth = imguidedfilter(A,'NeighborhoodSize',[4 4]);`

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

'DegreeOfSmoothing' — Amount of smoothing in the output image`0.01*diff(getrangedfromclass(G)).^2` (default) | positive scalar

Amount of smoothing in the output image, specified as a positive scalar. If you specify a small value, only neighborhoods with small variance (uniform areas) will get smoothed and neighborhoods with larger variance (such as around edges) will not be smoothed. If you specify a larger value, high variance neighborhoods, such as stronger edges, will get smoothed in addition to the relatively uniform neighborhoods. Start with the default value, check the results, and adjust the default up or down to achieve the effect you desire.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Output Arguments

B — Filtered image

array the same size and type as A

Filtered image, returned as an array of the same size and type as A

Tips

- The parameter `DegreeOfSmoothing` specifies a soft threshold on variance for the given neighborhood. If a pixel's neighborhood has variance much lower than the threshold, it will see some amount of smoothing. If a pixel's neighborhood has variance much higher than the threshold it will have little to no smoothing.
- Input images **A** and **G** can be of different classes. If either **A** or **G** is of class integer or logical, `imguiddfilter` converts them to floating-point precision for internal computation.
- Input images **A** and **G** can have different number of channels.
 - If **A** is an RGB image and **G** is a grayscale or binary image, `imguiddfilter` uses **G** for guidance for all the channels of **A** independently.
 - If both **A** and **G** are RGB images, `imguiddfilter` uses each channel of **G** for guidance for the corresponding channel of **A**, i.e. plane-by-plane behavior.
 - If **A** is a grayscale or binary image and **G** is an RGB image, `imguiddfilter` uses all the three channels of **G** for guidance (color statistics) for filtering **A**.

References

- [1] Kaiming He, Jian Sun, Xiaoou Tang, *Guided Image Filtering*. IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume 35, Issue 6, pp. 1397-1409, June 2013

See Also

See Also

`edge` | `imfilter` | `imsharpen`

Topics

“Perform Flash/No-flash Denoising with Guided Filter”

“What is Guided Image Filtering?”

Introduced in R2014a

imhandles

Get all image objects

Syntax

```
imageobj = imhandles(parentobj)
```

Description

`imageobj = imhandles(parentobj)` returns all of the image objects whose ancestor is `parentobj`. `parentobj` can be an array of valid figures, axes, images, or uipanel objects.

`imhandles` ignores colorbars.

Note

`imhandles` returns an error if the image objects do not have the same figure as their parent.

Examples

Return the image object in the current axes.

```
figure, imshow('moon.tif');  
imageobj = imhandles(gca)
```

Display two images in a figure and use `imhandles` to get both of the image objects in the figure.

```
subplot(1,2,1), imshow('autumn.tif');  
subplot(1,2,2), imshow('glass.png');  
imageobjs = imhandles(gcf)
```

See Also

imgca | imgcf

Introduced before R2006a

imhist

Histogram of image data

Syntax

```
imhist(I)
imhist(I,n)
imhist(X,map)
[counts,binLocations] = imhist(I)
[counts,binLocations] = imhist(gpuarrayI, ___ )
```

Description

`imhist(I)` calculates the histogram for the intensity image `I` and displays a plot of the histogram. The number of bins in the histogram is determined by the image type.

`imhist(I,n)` calculates the histogram, where `I` specifies the number of bins used in the histogram. `n` also specifies the length of the colorbar displayed at the bottom of the histogram plot.

`imhist(X,map)` displays a histogram for the indexed image `X`. This histogram shows the distribution of pixel values above a colorbar of the colormap `map`. The colormap must be at least as long as the largest index in `X`. The histogram has one bin for each entry in the colormap.

`[counts,binLocations] = imhist(I)` returns the histogram counts in `counts` and the bin locations in `binLocations` so that `stem(binLocations,counts)` shows the histogram. For indexed images, `imhist` returns the histogram counts for each colormap entry. The length of `counts` is the same as the length of the colormap.

`[counts,binLocations] = imhist(gpuarrayI, ___)` performs the histogram calculation on a GPU. The input image and the return values are `gpuArrays`. This syntax requires the Parallel Computing Toolbox. When the input image is a `gpuArray`, `imhist` does not automatically display the histogram. To display the histogram, use `stem(binLocations,counts)`.

Examples

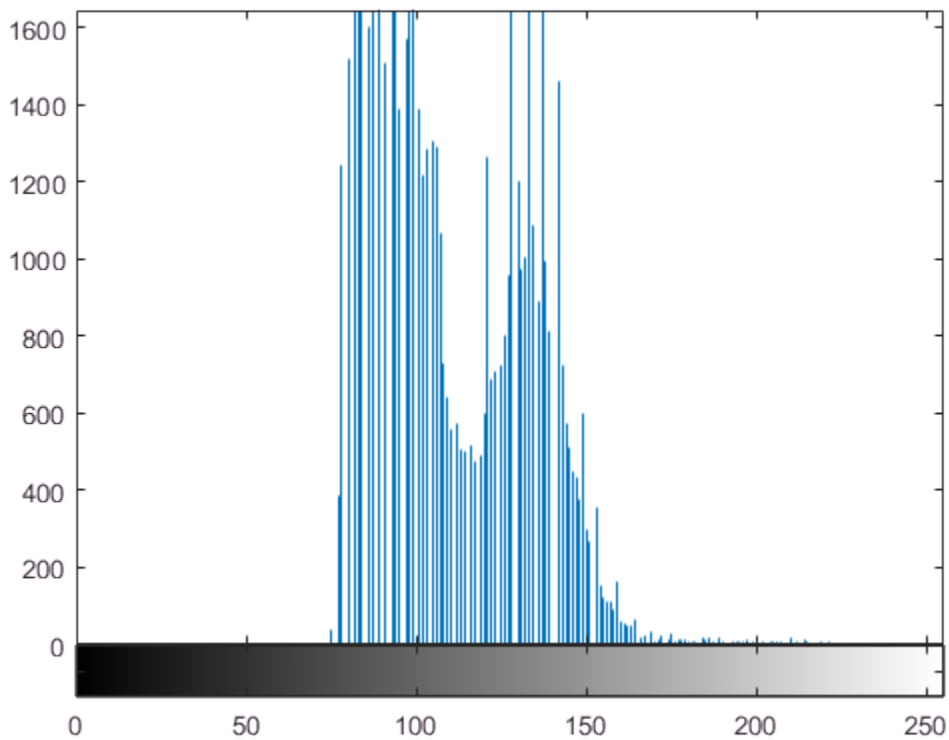
Calculate Histogram

Read a grayscale image into the workspace.

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

Display a histogram of the image. Since I is grayscale, by default the histogram will have 256 bins.

```
imhist(I)
```



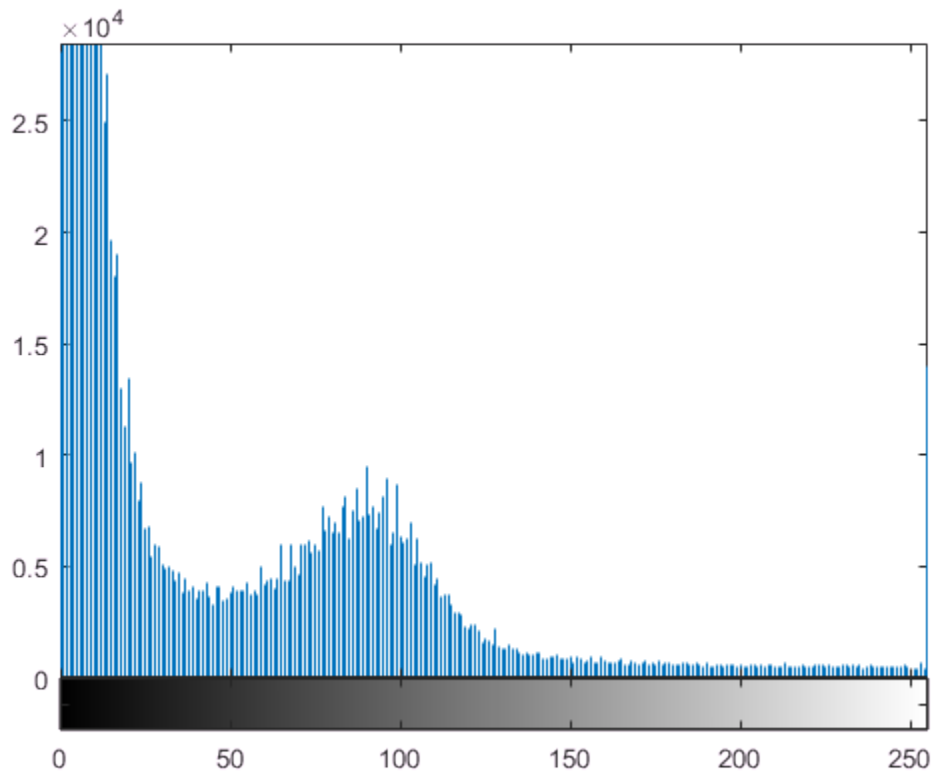
Display the Histogram of a 3-D Intensity Image

Load a 3-D dataset.

```
load mrystack
```

Display the histogram of the data. Since the image is grayscale, `imhist` uses 256 bins by default.

```
imhist(mrystack)
```



Calculate Histogram on a GPU

Create array of class `uint16`.

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

Calculate histogram. Because `imhist` does not automatically display the plot of the histogram when run on a GPU, this example uses `stem` to plot the histogram.

```
[counts,x] = imhist(I);  
stem(x,counts);
```

Input Arguments

I — Input intensity image

numeric array

Input intensity image, specified as a numeric array. **I** can be 2-D, 3-D, or N-D.

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

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

n — Number of bins

256 (for grayscale images) (default) | numeric scalar

Number of bins, specified as a numeric scalar. If **I** is a grayscale image, `imhist` uses a default value of 256 bins. If **I** is a binary image, `imhist` uses two bins.

Example: `[counts,x] = imhist(I,50);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

X — Input indexed image

numeric array

Input indexed image, specified as a numeric array. **X** can be 2-D, 3-D, or N-D.

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

Data Types: `single` | `double` | `uint8` | `uint16` | `logical`

map — Colormap associated with indexed image

p-by-3 array

Colormap associated with indexed image, specified as a *p*-by-3 array.

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

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

gpuarrayI — Input image

`gpuArray`

Input image, specified as a `gpuArray`.

Example: `gpuarrayI = gpuArray(imread('cameraman.tif'));`

Output Arguments

counts — histogram counts

numeric array

Histogram counts, returned as a numeric array.

binLocations — Bin locations

numeric array

Bin locations, returned as a numeric array.

Tips

- For intensity images, the n bins of the histogram are each half-open intervals of width $A/(n-1)$. In particular, the p th bin is the half-open interval

$$\frac{A(p-1.5)}{(n-1)} - B \leq x < \frac{A(p-0.5)}{(n-1)} - B,$$

where x is the intensity value. The scale factor A and offset B depend on the type of the image class as follows:

	double	single	int8	int16	int32	uint8	uint16	uint32	logical
A	1	1	255	65535	4294967295	255	65535	4294967295	1
B	0	0	128	32768	2147483648	0	0	0	0

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- If the first input is a binary image, then `n` must be a scalar constant of value 2 at compile time.
- Nonprogrammatic syntaxes are not supported. For example, the syntax `imhist(I)`, where `imhist` displays the histogram, is not supported.

See Also

See Also

`gpuArray` | `histeq` | `histogram`

Introduced before R2006a

imhistmatch

Adjust histogram of 2-D image to match histogram of reference image

Syntax

```
B = imhistmatch(A,ref)
B = imhistmatch(A,ref,nbins)
[B,hgram] = imhistmatch( ___ )
```

Description

`B = imhistmatch(A,ref)` transforms the 2-D grayscale or truecolor image `A` returning output image `B` whose histogram approximately matches the histogram of the reference image `ref`.

- If both `A` and `ref` are truecolor RGB images, `imhistmatch` matches each color channel of `A` independently to the corresponding color channel of `ref`.
- If `A` is a truecolor RGB image and `ref` is a grayscale image, `imhistmatch` matches each channel of `A` against the single histogram derived from `ref`.
- If `A` is a grayscale image, `ref` must also be a grayscale image.
-

Images `A` and `ref` can be any of the permissible data types and need not be equal in size.

`B = imhistmatch(A,ref,nbins)` uses `nbins` equally spaced bins within the appropriate range for the given image data type. The returned image `B` has no more than `nbins` discrete levels.

- If the data type of the image is either `single` or `double`, the histogram range is `[0, 1]`.
- If the data type of the image is `uint8`, the histogram range is `[0, 255]`.
- If the data type of the image is `uint16`, the histogram range is `[0, 65535]`.
- If the data type of the image is `int16`, the histogram range is `[-32768, 32767]`.

`[B,hgram] = imhistmatch(___)` returns the histogram of the reference image `ref` used for matching in `hgram`. `hgram` is a 1-by-`nbins` (when `ref` is grayscale) or a 3-

by-`nbins` (when `ref` is `truecolor`) matrix, where `nbins` is the number of histogram bins. Each row in `hgram` stores the histogram of a single color channel of `ref`.

Examples

Match Histogram of Aerial Images

These aerial images, taken at different times, represent overlapping views of the same terrain in Concord, Massachusetts. This example demonstrates that input images `A` and `Ref` can be of different sizes and image types.

Load an RGB image and a reference grayscale image.

```
A = imread('concordaerial.png');  
Ref = imread('concordorthophoto.png');
```

Get the size of `A`.

```
size(A)  
  
ans =  
  
        2036        3060         3
```

Get the size of `Ref`.

```
size(Ref)  
  
ans =  
  
        2215        2956
```

Note that image `A` and `Ref` are different in size and type. Image `A` is a truecolor RGB image, while image `Ref` is a grayscale image. Both images are of data type `uint8`.

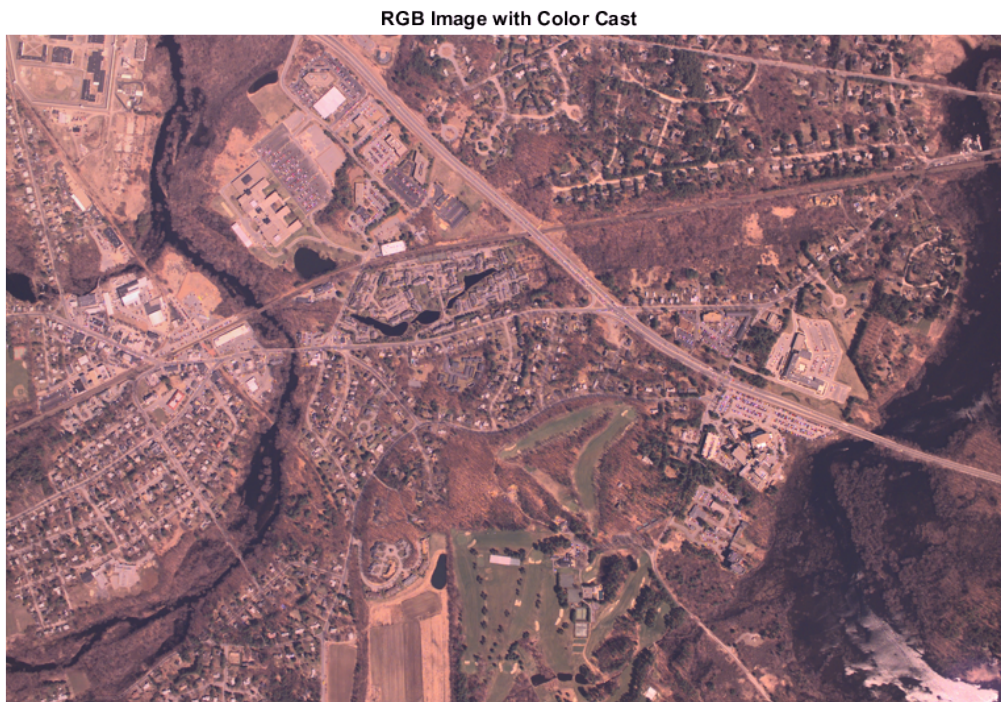
Generate the histogram matched output image. The example matches each channel of `A` against the single histogram of `Ref`. Output image `B` takes on the characteristics of image `A` - it is an RGB image whose size and data type is the same as image `A`. The number of distinct levels present in each RGB channel of image `B` is the same as the

number of bins in the histogram built from grayscale image `Ref`. In this example, the histogram of `Ref` and `B` have the default number of bins, 64.

```
B = imhistmatch(A,Ref);
```

Display the RGB image `A`, the reference image `Ref`, and the histogram matched RGB image `B`. The images are resized before display.

```
figure  
imshow(imresize(A, 0.25))  
title('RGB Image with Color Cast')
```



```
figure  
imshow(imresize(Ref, 0.25))  
title('Reference Grayscale Image')
```

Reference Grayscale Image



```
figure  
imshow(imresize(B, 0.25))  
title('Histogram Matched RGB Image')
```

Histogram Matched RGB Image



Multiple N Values Applied to RGB Images

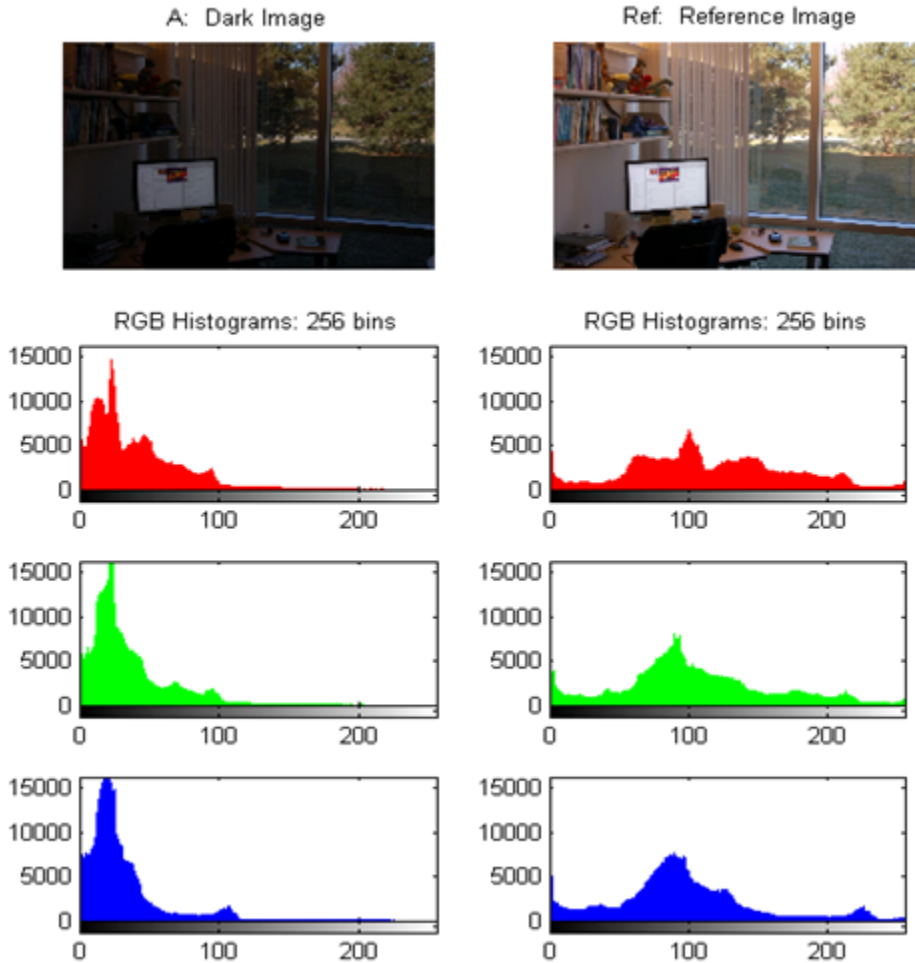
In this example, you will see the effect on output image **B** of varying the number of equally spaced bins in the target histogram of image **Ref**, from its default value 64 to the maximum value of 256 for `uint8` pixel data.

The following images were taken with a digital camera and represent two different exposures of the same scene.

```
A = imread('office_2.jpg'); % Dark Image
Ref = imread('office_4.jpg'); % Reference image
```

Image **A**, being the darker image, has a preponderance of its pixels in the lower bins. The reference image, **Ref**, is a properly exposed image and fully populates all of the available

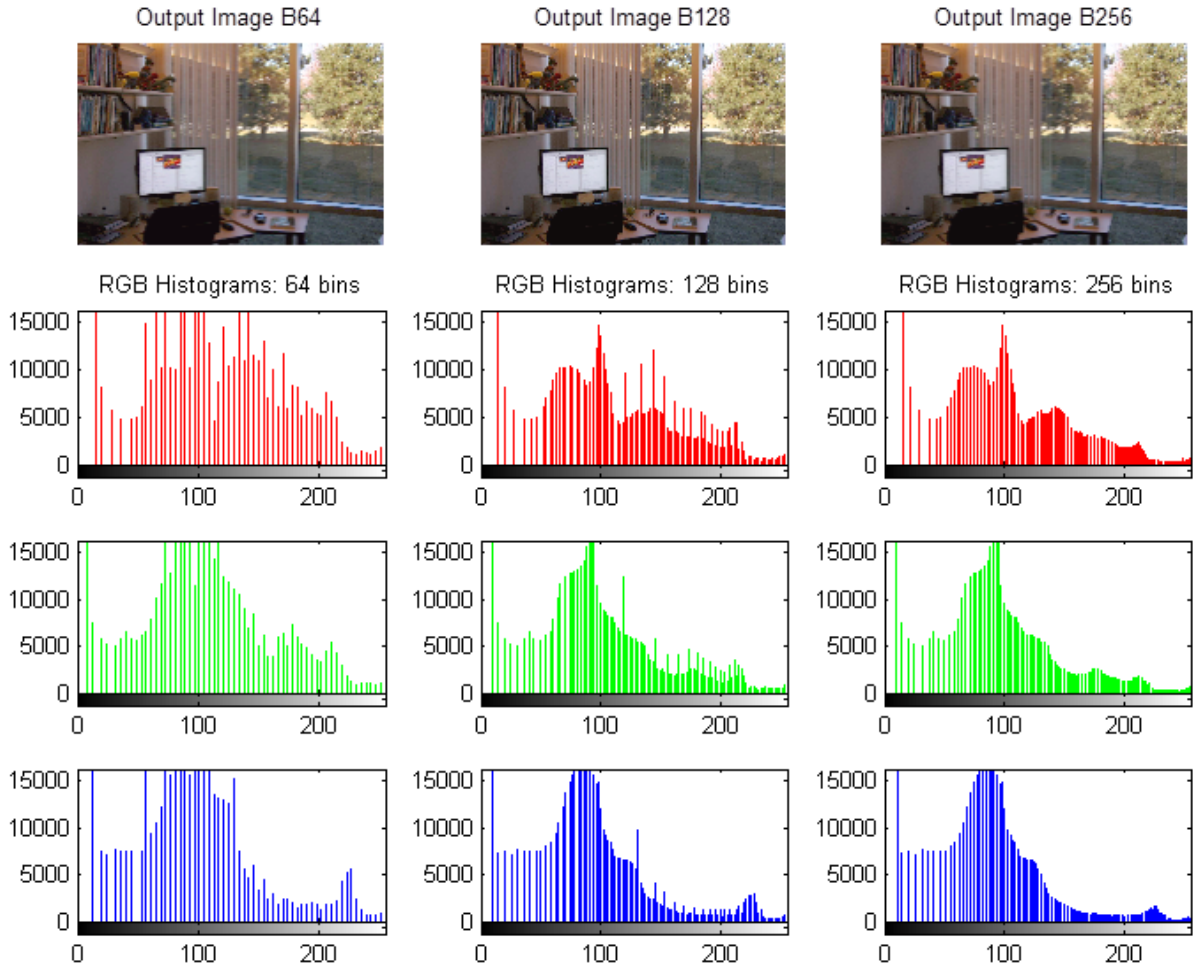
bins values in all three RGB channels: as shown in the table below, all three channels have 256 unique levels for 8-bit pixel values.



The unique 8-bit level values for the red channel is 205 for A and 256 for Ref. The unique 8-bit level values for the green channel is 193 for A and 256 for Ref. The unique 8-bit level values for the blue channel is 224 for A and 256 for Ref.

The example generates the output image **B** using three different values of `nbins`: 64, 128 and 256. The objective of function `imhistmatch` is to transform image **A** such that the histogram of output image **B** is a match to the histogram of `Ref` built with `nbins` equally spaced bins. As a result, `nbins` represents the upper limit of the number of discrete data levels present in image **B**.

```
[B64, hgram] = imhistmatch(A, Ref, 64);  
[B128, hgram] = imhistmatch(A, Ref, 128);  
[B256, hgram] = imhistmatch(A, Ref, 256);
```



The unique 8-bit level values for the red channel for `nbins=[64 128 256]` are 57 for output image B64, 101 for output image B128, and 134 for output image B256. The unique 8-bit level values for the green channel for `nbins=[64 128 256]` are 57 for output image B64, 101 for output image B128, and 134 for output image B256. The unique 8-bit level values for the blue channel for `nbins=[64 128 256]` are 57 for output image B64, 101 for output image B128, and 134 for output image B256. Note that

as nbins increases, the number of levels in each RGB channel of output image B also increases.

Match Histogram of 16-Bit Grayscale MRI Image

This example shows how to perform histogram matching with different numbers of bins.

Load a 16-bit DICOM image of a knee imaged via MRI.

```
K = dicomread('knee1.dcm'); % read in original 16-bit image
LevelsK = unique(K(:)); % determine number of unique code values
disp(['image K: ', num2str(length(LevelsK)), ' distinct levels']);
```

```
image K: 448 distinct levels
```

```
disp(['max level = ' num2str( max(LevelsK) )]);
```

```
max level = 473
```

```
disp(['min level = ' num2str( min(LevelsK) )]);
```

```
min level = 0
```

All 448 discrete values are at low code values, which causes the image to look dark. To rectify this, scale the image data to span the entire 16-bit range of [0, 65535].

```
Kdouble = double(K); % cast uint16 to double
kmult = 65535/(max(max(Kdouble(:)))); % full range multiplier
Ref = uint16(kmult*Kdouble); % full range 16-bit reference image
```

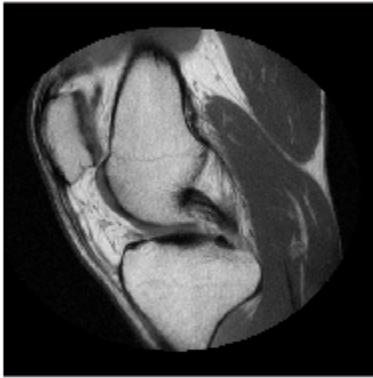
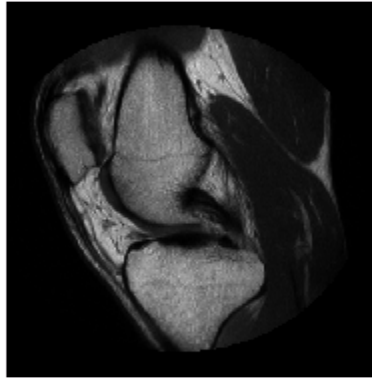
Darken the reference image Ref to create an image A that can be used in the histogram matching operation.

```
%Build concave bow-shaped curve for darkening |Ref|.
ramp = [0:65535]/65535;
ppconcave = spline([0 .1 .50 .72 .87 1],[0 .025 .25 .5 .75 1]);
Ybuf = ppval( ppconcave, ramp);
Lut16bit = uint16( round( 65535*Ybuf ) );
% Pass image |Ref| through a lookup table (LUT) to darken the image.
A = intlut(Ref,Lut16bit);
```

View the reference image Ref and the darkened image A. Note that they have the same number of discrete code values, but differ in overall brightness.

```
subplot(1,2,1)
imshow(Ref)
title('Ref: Reference Image')
```

```
subplot(1,2,2)
imshow(A)
title('A: Darkened Image');
```

Ref: Reference Image**A: Darkened Image**

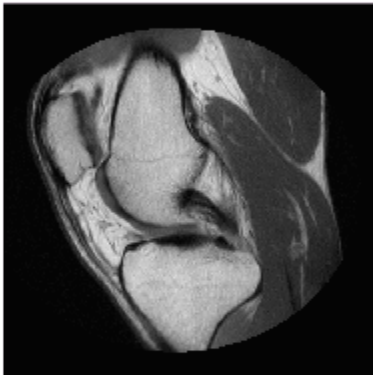
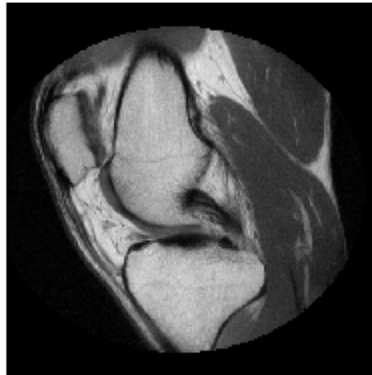
Generate histogram-matched output images using histograms with different number of bins. First use the default number of bins, 64. Then use the number of values present in image A, 448 bins.

```
B16bit64 = imhistmatch(A(:,:,1),Ref(:,:,1)); % default: 64 bins
```

```
N = length(LevelsK); % number of unique 16-bit code values in image A.
B16bitUniq = imhistmatch(A(:,:,1),Ref(:,:,1),N);
```

View the results of the two histogram matching operations.

```
figure
subplot(1,2,1)
imshow(B16bit64)
title('B16bit64: 64 bins')
subplot(1,2,2)
imshow(Ref)
title(['B16bitUniq: ',num2str(N),' bins'])
```

B16bit64: 64 bins**B16bitUniq: 448 bins**

Input Arguments

A — Input image

2-D truecolor image | 2-D grayscale image

Input image to be transformed, specified as a 2-D truecolor or grayscale image. The returned image will take the data type class of the input image.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

ref — Reference image whose histogram is the reference histogram

2-D truecolor image | 2-D grayscale image

Reference image whose histogram is the reference histogram, specified as a 2-D truecolor or grayscale image. The reference image provides the equally spaced `nbins` bin reference histogram which output image `B` is trying to match.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

nbins — Number of equally spaced bins in reference histogram

64 (default) | positive integer

Number of equally spaced bins in reference histogram, specified as a positive integer. In addition to specifying the number of equally spaced bins in the histogram for image `ref`, `nbins` also represents the upper limit of the number of discrete data levels present in output image `B`.

Data Types: `double`

Output Arguments

B — Output image

2-D truecolor RGB image | 2-D grayscale image

Output image, returned as a 2-D truecolor or grayscale image. The output image is derived from image `A` whose histogram is an approximate match to the histogram of input image `ref` built with `nbins` equally spaced bins. Image `B` is of the same size and data type as input image `A`. Input argument `nbins` represents the upper limit of the number of discrete levels contained in image `B`.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

hgram — Histogram counts derived from reference image `ref`

vector | matrix

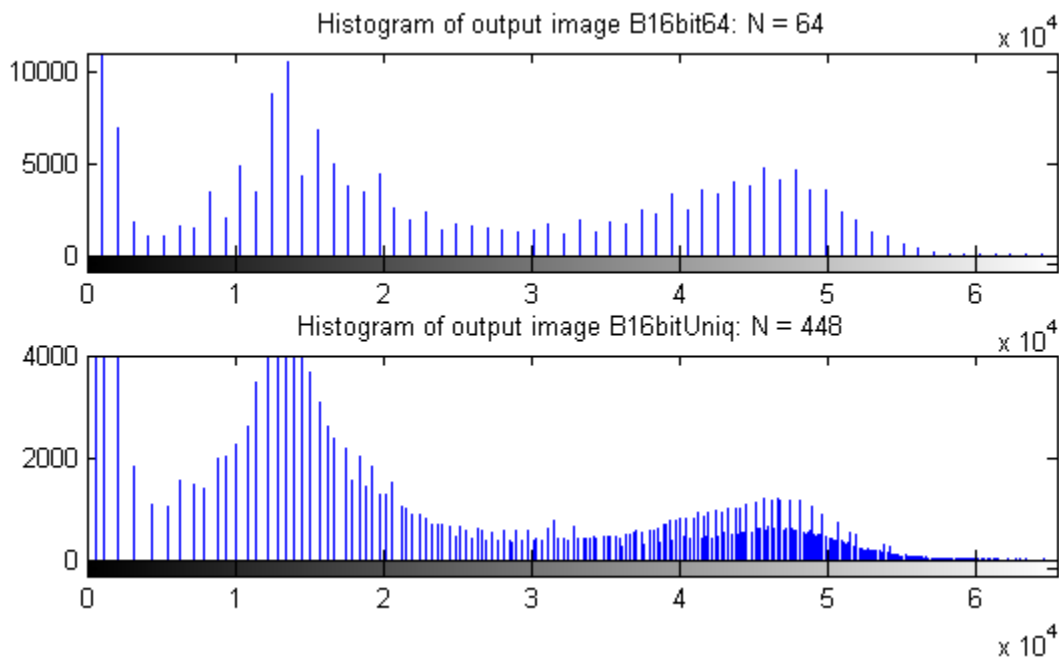
Histogram counts derived from reference image `ref`, specified as a vector or matrix. When `ref` is a truecolor image, `hgram` is a 3-by-`nbins` matrix. When `ref` is a grayscale image, `hgram` is a 1-by-`nbins` vector.

Data Types: double

Algorithms

The objective of `imhistmatch` is to transform image A such that the histogram of image B matches the histogram derived from image `ref`. It consists of `nbins` equally spaced bins which span the full range of the image data type. A consequence of matching histograms in this way is that `nbins` also represents the upper limit of the number of discrete data levels present in image B.

An important behavioral aspect of this algorithm to note is that as `nbins` increases in value, the degree of rapid fluctuations between adjacent populated peaks in the histogram of image B tends to increase. This can be seen in the following histogram plots taken from the 16-bit grayscale MRI example.



An optimal value for `nbins` represents a trade-off between more output levels (larger values of `nbins`) while minimizing peak fluctuations in the histogram (smaller values of `nbins`).

See Also

See Also

`histeq` | `imadjust` | `imhist` | `imhistmatchn`

Introduced in R2012b

imhistmatchn

Adjust histogram of N-D image to match histogram of reference image

Syntax

```
B = imhistmatchn(A,ref)
B = imhistmatchn(A,ref,nbins)
[B,hgram] = imhistmatchn( ___ )
```

Description

`B = imhistmatchn(A,ref)` transforms the N-D grayscale image `A` and returns output image `B` whose histogram approximately matches the histogram of the reference image `ref`. Both `A` and `ref` must be grayscale images, but they do not need to have the same data type, size, or number of dimensions.

`B = imhistmatchn(A,ref,nbins)` uses `nbins` equally spaced bins within the appropriate range for the given image data type. The returned image `B` has no more than `nbins` discrete levels.

If the data type of the image is:

- `single` or `double`, the histogram range is `[0, 1]`.
- `uint8`, the histogram range is `[0, 255]`.
- `uint16`, the histogram range is `[0, 65535]`.
- `int16`, the histogram range is `[-32768, 32767]`.

`[B,hgram] = imhistmatchn(___)` returns the histogram of the reference image `ref` used for matching in `hgram`. `hgram` is a 1-by-`nbins` vector, where `nbins` is the number of histogram bins.

Examples

Match Histograms of Multidimensional Images

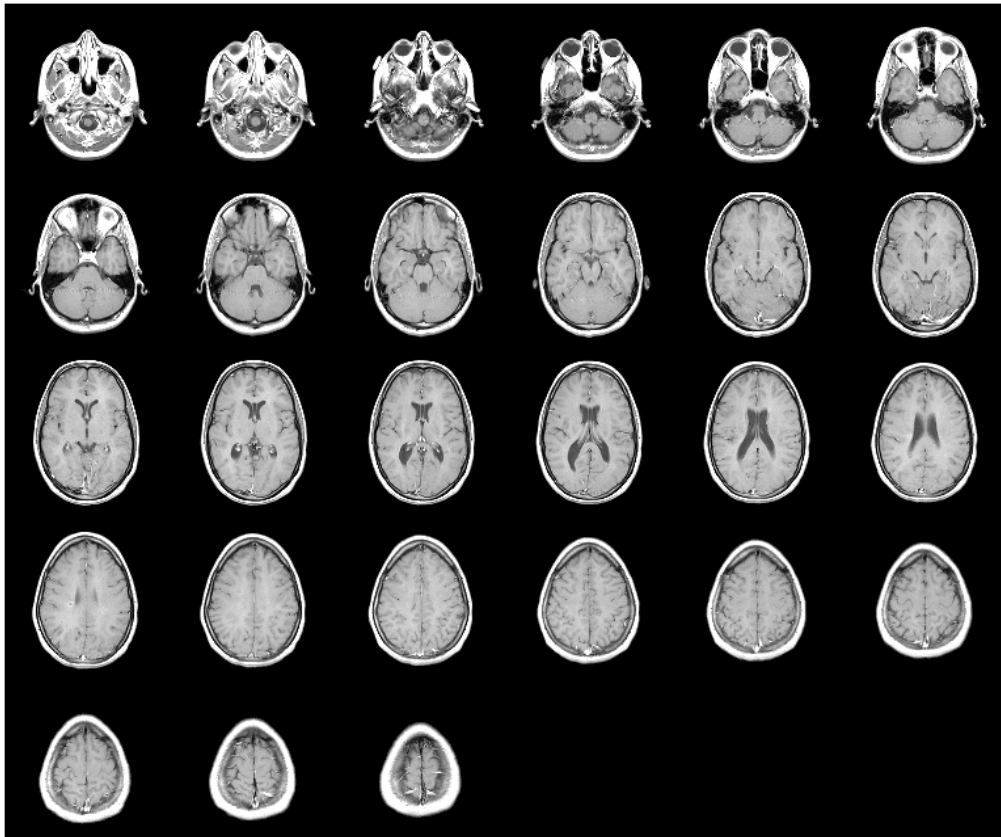
Load an N-D grayscale image into the workspace. Also load a grayscale image to provide a reference histogram.

```
load mri D  
load mristack
```

Display the original volume as slices.

```
figure  
montage(D, 'DisplayRange', [])  
title('Original 3-D Image')
```


Original 3-D Image



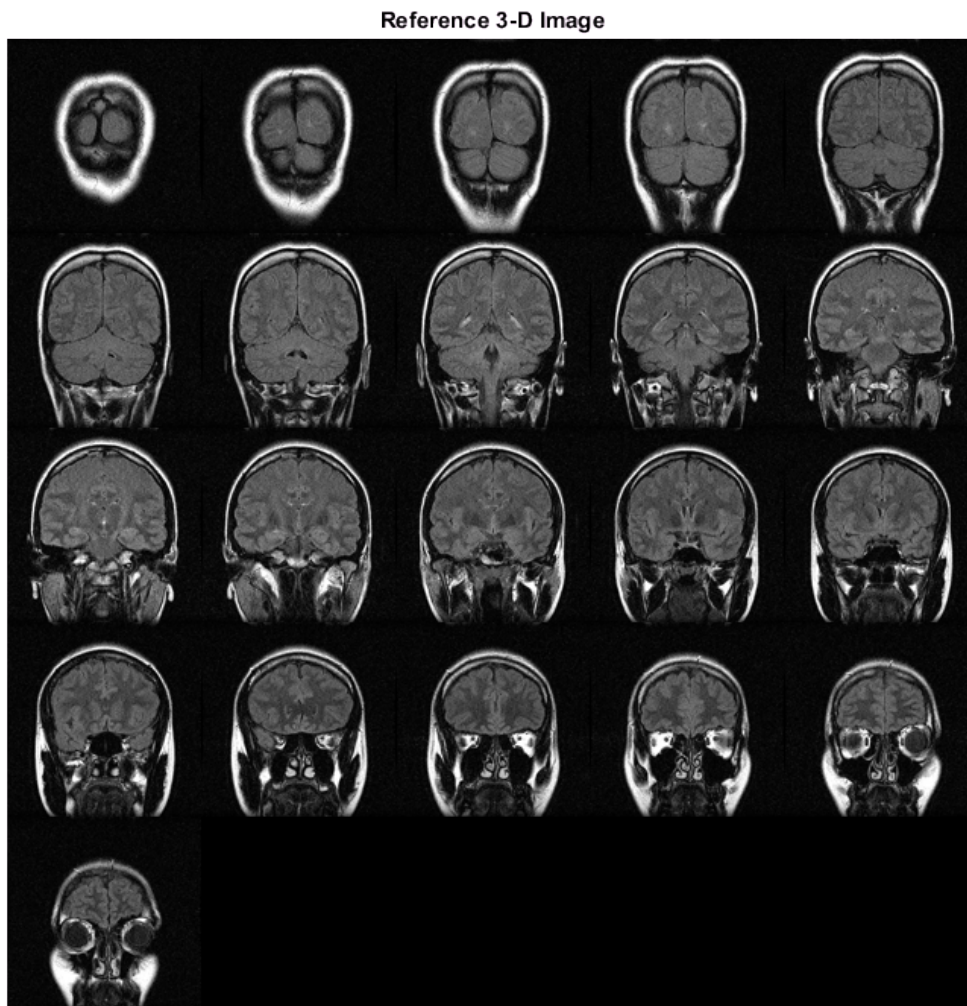
Reshape the reference as a stack of grayscale slices for display.

```
ref = reshape(mristack,[256,256,1,21]);
```

Display the reference volume as slices. To display correctly on the screen, the reference volume is downsized by a factor of 0.5 using `imresize`.

```
ref_downsized = imresize(ref,0.5);  
figure  
montage(ref_downsized,'DisplayRange',[1])
```

```
title('Reference 3-D Image')
```

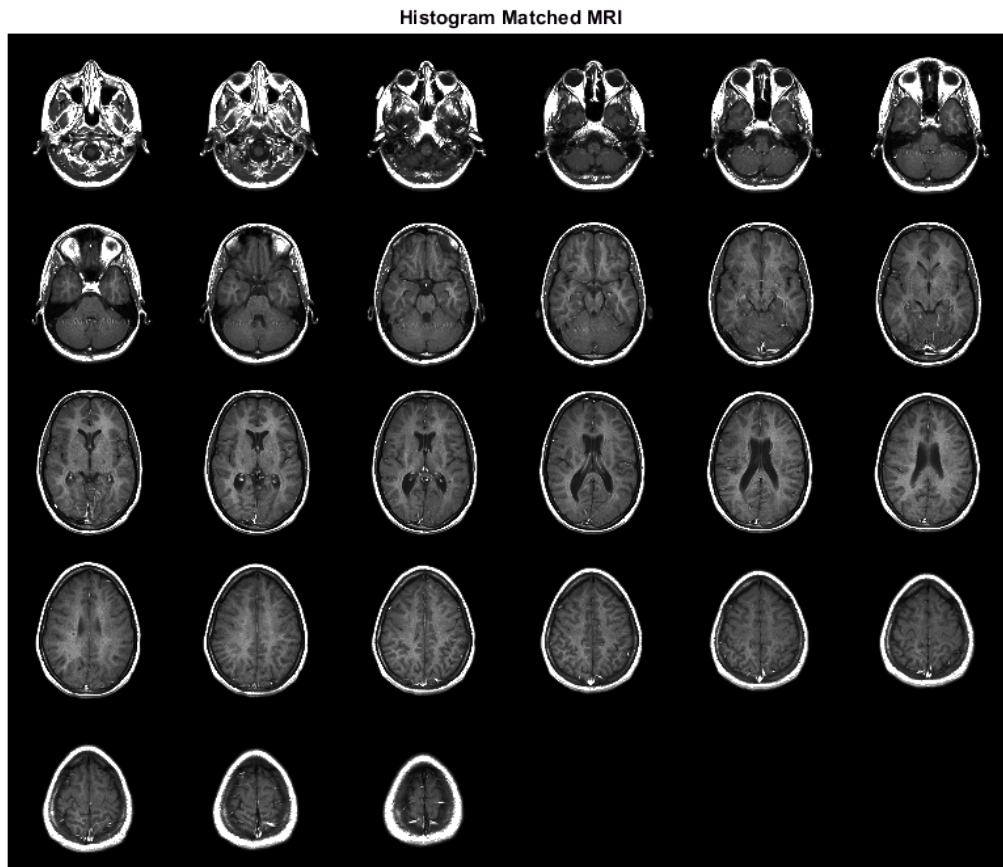


Match the histogram of D to the histogram of the fullsize ref.

```
Dmatched = imhistmatchn(D,ref);
```

Display the output. Observe that the brightness levels of the output more closely match the reference image than the original image.

```
figure  
montage(Dmatched, 'DisplayRange', [])  
title('Histogram Matched MRI')
```



Input Arguments

A — Input image

N-D grayscale image

Input image to be transformed, specified as an N-D grayscale image.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

ref — Reference image whose histogram is the reference histogram

grayscale image

Reference image whose histogram is the reference histogram, specified as a grayscale image. The reference image provides the equally spaced `nbins` bin reference histogram which output image **B** is trying to match.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

nbins — Number of equally spaced bins in reference histogram

64 (default) | positive integer

Number of equally spaced bins in reference histogram, specified as a positive integer. `nbins` also represents the upper limit of the number of discrete data levels present in output image **B**.

Data Types: `double`

Output Arguments

B — Output image

N-D grayscale image

Output image, returned as an N-D grayscale image. The output image is derived from image **A** whose histogram is an approximate match to the histogram of input image `ref` built with `nbins` equally spaced bins. Image **B** is of the same size and data type as input image **A**. Input argument `nbins` represents the upper limit of the number of discrete levels contained in image **B**.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

hgram — Histogram counts derived from reference image `ref`

1-by-`nbins` vector

Histogram counts derived from reference image `ref`, returned as a 1-by-nbins vector.

Data Types: `double`

See Also

See Also

`histeq` | `imadjust` | `imhist` | `imhistmatch`

Introduced in R2017a

imhmax

H-maxima transform

Syntax

```
I2 = imhmax(I,h)
I2 = imhmax(I,h,conn)
```

Description

`I2 = imhmax(I,h)` suppresses all maxima in the intensity image `I` whose height is less than `h`, where `h` is a scalar. Regional maxima are connected components of pixels with a constant intensity value, and whose external boundary pixels all have a lower value. By default, `imhmax` uses 8-connected neighborhoods for 2-D images, and 26-connected neighborhoods for 3-D images. For higher dimensions, `imhmax` uses `conndef(ndims(I), 'maximal')`.

`I2 = imhmax(I,h,conn)` computes the H-maxima transform, where `conn` specifies the connectivity.

Examples

Create H-Maxima Transform

Create simple sample array of zeros with several maxima.

```
a = zeros(10,10);
a(2:4,2:4) = 3;
a(6:8,6:8) = 8
```

a =

```
0 0 0 0 0 0 0 0 0 0
0 3 3 3 0 0 0 0 0 0
0 3 3 3 0 0 0 0 0 0
0 3 3 3 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

```

0  0  0  0  0  8  8  8  0  0
0  0  0  0  0  8  8  8  0  0
0  0  0  0  0  8  8  8  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0

```

Calculate the maxima equal to 4 or more. Note how the area of the image set to 3 is not included.

```
b = imhmax(a,4)
```

```
b =
```

```

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  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  4  4  4  0  0
0  0  0  0  0  4  4  4  0  0
0  0  0  0  0  4  4  4  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0

```

Input Arguments

I — Input image

nonsparse numeric array of any dimension

Input array, specified as a nonsparse numeric array of any dimension.

Example: `I = imread('glass.png');` `BW = imhmax(I,80);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

h — h-maxima transform

nonnegative scalar

h-maxima transform, specified as a nonnegative scalar.

Example: `b = imhmax(a,4);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

conn — Connectivity

8 (default) | 4 | 6 | 18 | 26 | 3-by-3-by- ...-by-3 matrix of zeroes and ones

Connectivity, specified as a one of the scalar values in the following table. By default, `imhmax` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imhmax` uses `conndef(numel(size(I)), 'maximal')`. Connectivity can be defined in a more general way for any dimension by using for `conn` a 3-by-3-by- ...-by-3 matrix of 0s and 1s. The 1-valued elements define neighborhood locations relative to the center element of `conn`. Note that `conn` must be symmetric around its center element.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Example: `b = imhmax(a,4,4);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

I2 — Transformed image

nonsparse numeric array of any class

Transformed image, returned as a nonsparse numeric array of any class, the same size as `I`.

References

- [1] Soille, P., *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 170-171.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the optional third input argument, **conn**, must be a compile-time constant.

See Also

See Also

`conndef` | `imextendedmax` | `imhmin` | `imreconstruct` | `imregionalmax`

Introduced before R2006a

imhmin

H-minima transform

Syntax

```
I2 = imhmin(I,h)
I2 = imhmin(I,h,conn)
```

Description

`I2 = imhmin(I,h)` suppresses all minima in the intensity image `I` whose depth is less than `h`, where `h` is a scalar. Regional minima are connected components of pixels with a constant intensity value, t , whose external boundary pixels all have a value greater than t . By default, `imhmin` uses 8-connected neighborhoods for 2-D images, and 26-connected neighborhoods for 3-D images. For higher dimensions, `imhmin` uses `conndef(ndims(I), 'maximal')`.

`I2 = imhmin(I,h,conn)` computes the H-minima transform, where `conn` specifies the connectivity.

Examples

Calculate H-Minima Transform

Create a sample image with two regional minima.

```
a = 10*ones(10,10);
a(2:4,2:4) = 7;
a(6:8,6:8) = 2
```

`a =`

```
10  10  10  10  10  10  10  10  10  10
10   7   7   7  10  10  10  10  10  10
10   7   7   7  10  10  10  10  10  10
10   7   7   7  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
```

```

10  10  10  10  10  2  2  2  10  10
10  10  10  10  10  2  2  2  10  10
10  10  10  10  10  2  2  2  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10

```

Suppress all minima below a specified value. Note how the region with pixels valued 7 disappears in the transformed image because its depth is less than the specified h value.

```
b = imhmin(a,4)
```

```
b =
```

```

10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  6  6  6  10  10
10  10  10  10  10  6  6  6  10  10
10  10  10  10  10  6  6  6  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10

```

Input Arguments

I — Input image

nonsparse numeric array of any dimension

Input array, specified as a nonsparse numeric array of any dimension.

Example: `I = imread('glass.png');` `BW = imhmin(I,80);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

h — h-minima transform

nonnegative scalar

h-minima transform, specified as a nonnegative scalar.

Example: `b = imhmin(a,4)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

conn — Connectivity

8 (default) | 4 | 6 | 18 | 26 | 3-by-3-by- ...-by-3 matrix of zeroes and ones

Connectivity, specified as a one of the scalar values in the following table. By default, `imhmin` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imhmin` uses `conndef(numel(size(I)), 'maximal')`. Connectivity can be defined in a more general way for any dimension by using for `conn` a 3-by-3-by- ...-by-3 matrix of 0s and 1s. The 1-valued elements define neighborhood locations relative to the center element of `conn`. Note that `conn` must be symmetric around its center element.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Example: `b = imhmin(a,4,4)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

I2 — Transformed image

nonsparse numeric array of any class

Transformed image, returned as a nonsparse numeric array of any class, the same size as `I`.

References

- [1] Soille, P., *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, 1999, pp. 170-171.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the optional third input argument, **conn**, must be a compile-time constant.

See Also

See Also

`conndef` | `imextendedmin` | `imhmax` | `imreconstruct` | `imregionalmin`

Introduced before R2006a

imimposemin

Impose minima

Syntax

```
I2 = imimposemin(I,BW)
I2 = imimposemin(I,BW,conn)
```

Description

`I2 = imimposemin(I,BW)` modifies the intensity image `I` using morphological reconstruction so it only has regional minima wherever `BW` is nonzero. `BW` is a binary image the same size as `I`.

By default, `imimposemin` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imimposemin` uses `conndef(ndims(I), 'minimum')`.

`I2 = imimposemin(I,BW,conn)` specifies the connectivity, where `conn` can have any of the following scalar values.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can also be defined in a more general way for any dimension by using for `conn` a 3-by-3-by-...-by-3 matrix of 0's and 1's. The 1-valued elements define

neighborhood locations relative to the center element of `conn`. Note that `conn` must be symmetric about its center element.

Class Support

`I` can be of any nonsparse numeric class and any dimension. `BW` must be a nonsparse numeric array with the same size as `I`. `I2` has the same size and class as `I`.

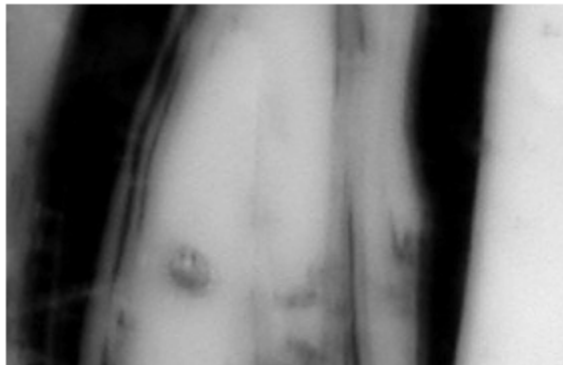
Examples

Impose Regional Minimum at One Location

This example shows how to modify an image so that one area is always a regional minimum.

Read an image and display it. This image is called the *mask* image.

```
mask = imread('glass.png');  
imshow(mask)
```



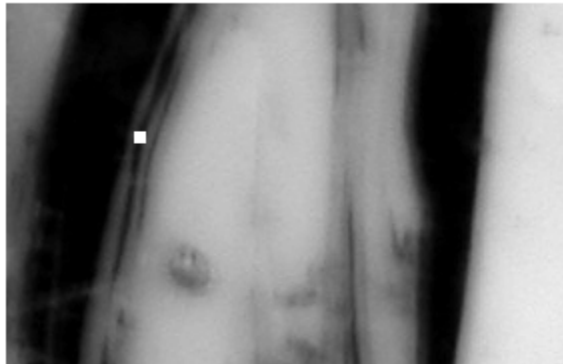
Create a binary image that is the same size as the mask image and sets a small area of the binary image to 1. These pixels define the location in the mask image where a regional minimum will be imposed. The resulting image is called the *marker* image.

```
marker = false(size(mask));  
marker(65:70,65:70) = true;
```

Superimpose the marker over the mask to show where these pixels of interest fall on the original image. The small white square marks the spot. This code is not essential to the `impose minima` operation.

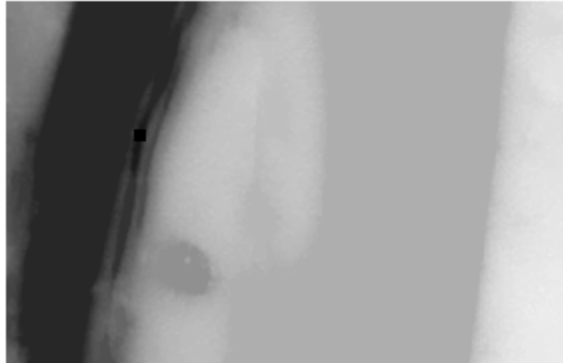
```
J = mask;  
J(marker) = 255;  
figure  
imshow(J)  
title('Marker Image Superimposed on Mask')
```

Marker Image Superimposed on Mask



Impose the regional minimum on the input image using the `imimposemin` function. Note how all the dark areas of the original image, except the marked area, are lighter.

```
K = imimposemin(mask,marker);  
figure  
imshow(K)
```

To illustrate how this operation removes all minima in the original image except the imposed minimum, compare the regional minima in the original image with the regional minimum in the processed image. These calls to `imregionalmin` return binary images that specify the locations of all the regional minima in both images.

```
BW = imregionalmin(mask);  
figure  
subplot(1,2,1)  
imshow(BW)  
title('Regional Minima in Original Image')  
  
BW2 = imregionalmin(K);  
subplot(1,2,2)  
imshow(BW2)  
title('Regional Minima After Processing')
```

Regional Minima in Original Image Regional Minima After Processing



Algorithms

`imimposemin` uses a technique based on morphological reconstruction.

See Also

`conndef` | `imreconstruct` | `imregionalmin`

Introduced before R2006a

imlincomb

Linear combination of images

Syntax

```
Z = imlincomb(K1,A1,K2,A2,...,Kn,An)
Z = imlincomb(K1,A1,K2,A2,...,Kn,An,K)
Z = imlincomb( ____,output_class)
gpuarrayZ = imlincomb(gpuarrayK,gpuarrayA, ____,output_class)
```

Description

`Z = imlincomb(K1,A1,K2,A2,...,Kn,An)` computes

$$K1*A1 + K2*A2 + \dots + Kn*An$$

where `K1`, `K2`, through `Kn` are real, double scalars and `A1`, `A2`, through `An` are real, nonsparse, numeric arrays with the same class and size. `Z` has the same class and size as `A1` unless `A1` is logical, in which case `Z` is double.

`Z = imlincomb(K1,A1,K2,A2,...,Kn,An,K)` computes

$$K1*A1 + K2*A2 + \dots + Kn*An + K$$

where `imlincomb` adds `K`, a real, double scalar, to the sum of the products of `K1` through `Kn` and `A1` through `An`.

`Z = imlincomb(____,output_class)` lets you specify the class of `Z`. `output_class` is a character vector containing the name of a numeric class.

`gpuarrayZ = imlincomb(gpuarrayK,gpuarrayA, ____,output_class)` performs the operation on a GPU, where the input values, `gpuarrayK` and `gpuarrayA`, are `gpuArrays` and the output value, `gpuarrayZ` is a `gpuArray`. This syntax requires the Parallel Computing Toolbox

When performing a series of arithmetic operations on a pair of images, you can achieve more accurate results if you use `imlincomb` to combine the operations, rather than

nesting calls to the individual arithmetic functions, such as `imadd`. When you nest calls to the arithmetic functions, and the input arrays are of an integer class, each function truncates and rounds the result before passing it to the next function, thus losing accuracy in the final result. `imlincomb` computes each element of the output `Z` individually, in double-precision floating point. If `Z` is an integer array, `imlincomb` truncates elements of `Z` that exceed the range of the integer type and rounds off fractional values.

Examples

Scale an Image Using Linear Combinations

Read an image into the workspace.

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

Scale the image using a coefficient of 1.5 in the linear combination.

```
J = imlincomb(1.5,I);
```

Display the original image and the processed image.

```
imshow(I)
```



figure
imshow(J)



Form a Difference Image with Zero Value Shifted to 128

Read an image into the workspace.

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

Create a low-pass filtered copy of the image.

```
J = uint8(filter2(fspecial('gaussian'), I));
```

Find the difference image and shift the zero value to 128 using a linear combination of I and J.

```
K = imlincomb(1,I,-1,J,128); %K(r,c) = I(r,c) - J(r,c) + 128
```

Display the resulting difference image.

```
imshow(K)
```



Add Two Images and Specify Output Class Using Linear Combinations

Read two grayscale `uint8` images into the workspace.

```
I = imread('rice.png');  
J = imread('cameraman.tif');
```

Add the images using a linear combination. Specify the output as type `uint16` to avoid truncating the result.

```
K = imlincomb(1,I,1,J,'uint16');
```

Display the result.

```
imshow(K,[])
```



Add Two Images and Specify Output Class Using Linear Combinations on a GPU

Read two grayscale `uint8` images into the workspace and convert them to GPUarrays.

```
I = gpuArray(imread('rice.png'));  
J = gpuArray(imread('cameraman.tif'));
```

Add the images using a linear combination on a GPU. Specify the output as type `uint16` to avoid truncating the result.

```
K = imlincomb(1,I,1,J,'uint16');
```

Display the result.

```
figure
```



```
imshow(K, [])
```

Compare Methods for Averaging Images

This example shows the difference between nesting calls and using linear combinations when performing a series of arithmetic operations on images. To illustrate how `imlincomb` performs all the arithmetic operations before truncating the result, compare the results of calculating the average of two arrays, `X` and `Y`, using nested arithmetic functions and using `imlincomb`.

Create two arrays.

```
X = uint8([ 255 0 75; 44 225 100]);
Y = uint8([ 50 50 50; 50 50 50 ]);
```

Average the arrays using nested arithmetic functions. To calculate the average returned in `Z(1,1)`, the function `imadd` adds 255 and 50 and truncates the result to 255 before passing it to `imdivide`. The average returned in `Z(1,1)` is 128.

```
Z = imdivide(imadd(X,Y),2)
```

```
Z = 2×3 uint8 matrix
```

```
   128    25    63
    47   128    75
```

In contrast, `imlincomb` performs the addition and division in double precision and only truncates the final result. The average returned in `Z2(1,1)` is 153.

```
Z2 = imlincomb(.5,X,.5,Y)
```

```
Z2 = 2×3 uint8 matrix
```

```
   153    25    63
    47   138    75
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- You can specify up to 4 input image arguments.
- The `output_class` argument must be a compile-time constant.

See Also

`imadd` | `imcomplement` | `imdivide` | `immultiply` | `imsubtract` | `gpuArray`

Introduced before R2006a

imline

Create draggable, resizable line

Syntax

```
h = imline
h = imline(hparent)
h = imline(hparent, position)
h = imline(hparent, x, y)
h = imline(..., param1, val1,...)
```

Description

`h = imline` begins interactive placement of a line on the current axes. The function returns `h`, a handle to an `imline` object. The line has a context menu associated with it that controls aspects of its appearance and behavior—see “Interactive Behavior” on page 1-926. Right-click on the line to access this context menu.

`h = imline(hparent)` begins interactive placement of a line on the object specified by `hparent`. `hparent` specifies the HG parent of the line graphics, which is typically an axes but can also be any other object that can be the parent of an `hggroup`

`h = imline(hparent, position)` creates a draggable, resizable line on the object specified by `hparent`. `position` is a 2-by-2 array that specifies the initial endpoint positions of the line in the form `[X1 Y1; X2 Y2]`.

`h = imline(hparent, x, y)` creates a line on the object specified by `hparent`. `x` and `y` are two-element vectors that specify the initial endpoint positions of the line in the form `x = [X1 X2]`, `y = [Y1 Y2]`.

`h = imline(..., param1, val1, ...)` creates a draggable, resizable line, specifying parameters and corresponding values that control the behavior of the line. The following table lists the parameter available. Parameter names can be abbreviated, and case does not matter.

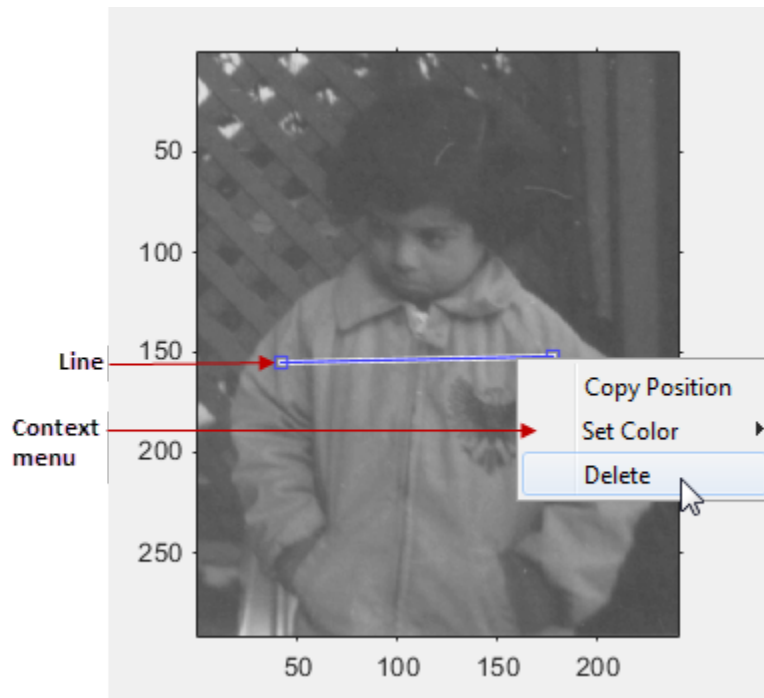
Parameter	Description
'PositionConstraintFcn'	Function handle <code>fcn</code> that is called whenever the object is dragged using the mouse. You can use this



Parameter	Description
	function to control where the line can be dragged. See the help for the setPositionConstraintFcn method for information about valid function handles.

Interactive Behavior

When you call `imline` with an interactive syntax, the pointer changes to a cross hairs

⊕ when over the image. Click and drag the mouse to specify the position and length of the line. The line supports a context menu that you can use to control aspects of its appearance and behavior. For more information about these interactive features, see the following table.



Interactive Behavior	Description
Moving the line.	Move the pointer over the line. The pointer changes to a fleur shape  . Click and drag the mouse to move the line.
Moving the endpoints of the line.	Move the pointer over either end of the line. The pointer changes to the pointing finger,  . Click and drag the mouse to resize the line.
Changing the color used to display the line.	Move the pointer over the line. Right-click and select Set Color from the context menu.
Retrieving the coordinates of the endpoints of the line.	Move the pointer over the line. Right-click and select Copy Position from the context menu. <code>imline</code> copies a 2-by-2 array to the clipboard specifying the coordinates of the endpoints of the line in the form [X1 Y1; X2 Y2].
Deleting the line	Move the pointer on top of the line. Right-click and select Delete from the context menu. To remove this option from the context menu, set the <code>Deletable</code> property to false: <code>h = imline(); h.Deletable = false;</code>

Methods

Each `imline` object supports a number of methods. Type `methods imline` to see a list of the methods.

addNewPositionCallback — Add new-position callback to ROI object

See `imroi` for information.

delete — Delete ROI object

See `imroi` for information.

getColor — Get color used to draw ROI object.

See `imroi` for information.

getPosition — Return current position of line

Returns the endpoint positions of the line.

```
pos = api.getPosition()
```

pos is a 2-by-2 array [X1 Y1; X2 Y2].

getPositionConstraintFcn — Return function handle to current position constraint function

See imroi for information.

removeNewPositionCallback — Remove new-position callback from ROI object.

See imroi for information.

resume — Resume execution of MATLAB command line

See imroi for information.

setColor — Set color used to draw ROI object

See imroi for information.

setConstrainedPosition — Set ROI object to new position

See imroi for information.

setPosition — Set line to new position

setPosition(h, pos) sets the line h to a new position. The new position, pos, has the form, [X1 Y1; X2 Y2].

setPosition(h, x, y) sets the line h to a new position. x and y specify the endpoint positions of the line in the form x = [x1 x2], y = [y1 y2].

setPositionConstraintFcn — Set position constraint function of ROI object.

See imroi for information.

wait — Block MATLAB command line until ROI creation is finished

See imroi for information.

Examples

Example 1

Use a custom color for displaying the line. Use `addNewPositionCallback` method. Move the line, note that the 2-by-2 position vector of the line is displayed in the title above the image. Explore the context menu of the line by right clicking on the line.

```
figure, imshow('pout.tif');
h = imline(gca,[10 100], [100 100]);
setColor(h,[0 1 0]);
id = addNewPositionCallback(h,@(pos) title(mat2str(pos,3)));

% After observing the callback behavior, remove the callback.
% using the removeNewPositionCallback API function.
removeNewPositionCallback(h,id);
```

Example 2

Interactively place a line by clicking and dragging. Use `wait` to block the MATLAB command line. Double-click on the line to `resume` execution of the MATLAB command line

```
figure, imshow('pout.tif');
h = imline;
position = wait(h);
```

Tips

If you use `imline` with an axes that contains an image object, and do not specify a position constraint function, users can drag the line outside the extent of the image and lose the line. When used with an axes created by the `plot` function, the axis limits automatically expand to accommodate the movement of the line.

See Also

`imellipse` | `imfreehand` | `impoint` | `impoly` | `imrect` | `imroi` | `makeConstrainToRectFcn`

Introduced before R2006a

immagbox

Magnification box for scroll panel

Syntax

```
hbox = immagbox(hparent,himage)
```

Description

`hbox = immagbox(hparent,himage)` creates a Magnification box for the image displayed in a scroll panel created by `imscrollpanel`. `hparent` is a handle to the figure or uipanel object that will contain the Magnification box. `himage` is a handle to the target image (the image in the scroll panel). `immagbox` returns `hbox`, which is a handle to the Magnification box uicontrol object

A Magnification box is an editable text box uicontrol that contains the current magnification of the target image. When you enter a new value in the magnification box, the magnification of the target image changes. When the magnification of the target image changes for any reason, the magnification box updates the magnification value.

API Functions

A Magnification box contains a structure of function handles, called an API. You can use the functions in this API to manipulate magnification box. To retrieve this structure, use the `iptgetapi` function.

```
api = iptgetapi(hbox)
```

The API for the Magnification box includes the following function.

Function	Description
<code>setMagnification</code>	Sets the magnification in units of screen pixels per image pixel. <code>setMagnification(new_mag)</code>

Function	Description
	where <code>new_mag</code> is a scalar magnification factor. Multiply <code>new_mag</code> by 100 to get percent magnification. For example if you call <code>setMagnification(2)</code> , the magnification box will show '200%'.

Examples

Add a magnification box to a scrollable image. Because the toolbox scrollable navigation is incompatible with standard MATLAB figure window navigation tools, the example suppresses the toolbar and menu bar in the figure window. The example positions the scroll panel in the figure window to allow room for the magnification box.

```
hFig = figure('Toolbar','none',...
             'Menubar','none');
hIm = imshow('pears.png');
hSP = imscrollpanel(hFig,hIm);
set(hSP,'Units','normalized',...
     'Position',[0 .1 1 .9])
```

```
hMagBox = immagbox(hFig,hIm);
pos = get(hMagBox,'Position');
set(hMagBox,'Position',[0 0 pos(3) pos(4)])
```

Change the magnification of the image in the scroll panel, using the scroll panel API function `setMagnification`. Notice how the magnification box updates.

```
apiSP = iptgetapi(hSP);
apiSP.setMagnification(2)
```

See also

`imscrollpanel`, `iptgetapi`

Introduced before R2006a

immovie

Make movie from multiframe image

Note: `immovie(D,size)` is an obsolete syntax and is no longer supported. Use `immovie(X,map)` instead.

Syntax

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

Description

`mov = immovie(X,map)` returns the movie structure array `MOV` from the images in the multiframe indexed image `X` with the colormap `map`. For details about the movie structure array, see the reference page for `getframe`. To play the movie, call `implay`.

`X` comprises multiple indexed images, all having the same size and all using the colormap `map`. `X` is an `m-by-n-by-1-by-k` array, where `k` is the number of images.

`mov = immovie(RGB)` returns the movie structure array `MOV` from the images in the multiframe, truecolor image `RGB`.

`RGB` comprises multiple truecolor images, all having the same size. `RGB` is an `m-by-n-by-3-by-k` array, where `k` is the number of images.

Class Support

An indexed image can be `uint8`, `uint16`, `single`, `double`, or `logical`. A truecolor image can be `uint8`, `uint16`, `single`, or `double`. `mov` is a MATLAB movie structure.

Examples

```
load mri
mov = immovie(D,map);
imshow(mov)
```

Tips

To create a movie that can be played outside the MATLAB environment, use the `VideoWriter` class.

See Also

See Also

`getframe` | `montage` | `movie` | `VideoWriter`

Introduced before R2006a

immse

Mean-squared error

Syntax

```
err = immse(X,Y)
```

Description

`err = immse(X,Y)` calculates the mean-squared error (MSE) between the arrays `X` and `Y`. `X` and `Y` can be arrays of any dimension, but must be of the same size and class.

Examples

Calculate Mean-Squared Error in Noisy Image

Read image and display it.

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



Create another image by adding noise to a copy of the reference image.

```
A = imnoise(ref, 'salt & pepper', 0.02);  
imshow(A)
```



Calculate mean-squared error between the two images.

```
err = immse(A, ref);  
fprintf('\n The mean-squared error is %0.4f\n', err);
```

The mean-squared error is 353.7631

Input Arguments

X — Input array

numeric array

Input array, specified as a nonsparse, numeric array.

Example: `err = immse(I,I2);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Y — Input array

nonsparse, numeric array

Input arrays, specified as a nonsparse, numeric array.

Example: `err = immse(I,I2);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output Arguments

err — Mean-squared error

`double` | `single`

Mean-squared error, returned as a scalar of class `double`. If the input arguments are of class `single`, `err` is of class `single`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.

See Also

See Also

`mean` | `median` | `psnr` | `ssim` | `sum` | `var`

Introduced in R2014b

immultiply

Multiply two images or multiply image by constant

Syntax

```
Z = immultiply(X,Y)
```

Description

`Z = immultiply(X,Y)` multiplies each element in array `X` by the corresponding element in array `Y` and returns the product in the corresponding element of the output array `Z`.

If `X` and `Y` are real numeric arrays with the same size and class, then `Z` has the same size and class as `X`. If `X` is a numeric array and `Y` is a scalar double, then `Z` has the same size and class as `X`.

If `X` is logical and `Y` is numeric, then `Z` has the same size and class as `Y`. If `X` is numeric and `Y` is logical, then `Z` has the same size and class as `X`.

`immultiply` computes each element of `Z` individually in double-precision floating point. If `X` is an integer array, then elements of `Z` exceeding the range of the integer type are truncated, and fractional values are rounded.

If `X` and `Y` are numeric arrays of the same size and class, you can use the expression `X.*Y` instead of `immultiply`.

Examples

Multiply an Image by Itself

Read a grayscale image into the workspace, then convert the image to `uint8`.

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

```
I16 = uint16(I);
```

Multiply the image by itself. Note that `immultiply` converts the class of the image from `uint8` to `uint16` before performing the multiplication to avoid truncating the results.

```
J = immultiply(I16,I16);
```

Show the original image and the processed image.

```
imshow(I)
```



`figure`
`imshow(J)`



Scale an Image by a Constant Factor

Read an image into the workspace.

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

Scale each value of the image by a constant factor of 0.5.

```
J = immultiply(I,0.5);
```

Display the original image and the processed image.

```
imshow(I)
```



`figure`
`imshow(J)`



See Also

`imabsdiff` | `imadd` | `imcomplement` | `imdivide` | `imlincomb` | `imsubtract`

Introduced before R2006a

imnoise

Add noise to image

Syntax

```
J = imnoise(I,type)
J = imnoise(I,type,parameters)
J = imnoise(I,'gaussian',M,V)
J = imnoise(I,'localvar',V)
J = imnoise(I,'localvar',image_intensity,var)
J = imnoise(I,'poisson')
J = imnoise(I,'salt & pepper',d)
J = imnoise(I,'speckle',v)
gpuarrayJ = imnoise(gpuarrayI, ___ )
```

Description

`J = imnoise(I,type)` adds noise of a given type to the intensity image `I`. `type` specifies any of the following types of noise. Note that certain types of noise support additional parameters. See the related syntax.

Value	Description
'gaussian'	Gaussian white noise with constant mean and variance
'localvar'	Zero-mean Gaussian white noise with an intensity-dependent variance
'poisson'	Poisson noise
'salt & pepper'	On and off pixels
'speckle'	Multiplicative noise

`J = imnoise(I,type,parameters)` Depending on `type`, you can specify additional parameters to `imnoise`. All numerical parameters are normalized— they correspond to operations with images with intensities ranging from 0 to 1.

`J = imnoise(I,'gaussian',M,V)` adds Gaussian white noise of mean `m` and variance `v` to the image `I`. The default is zero mean noise with 0.01 variance.

`J = imnoise(I, 'localvar', V)` adds zero-mean, Gaussian white noise of local variance `V` to the image `I`. `V` is an array of the same size as `I`.

`J = imnoise(I, 'localvar', image_intensity, var)` adds zero-mean, Gaussian noise to an image `I`, where the local variance of the noise, `var`, is a function of the image intensity values in `I`. The `image_intensity` and `var` arguments are vectors of the same size, and `plot(image_intensity, var)` plots the functional relationship between noise variance and image intensity. The `image_intensity` vector must contain normalized intensity values ranging from 0 to 1.

`J = imnoise(I, 'poisson')` generates Poisson noise from the data instead of adding artificial noise to the data. If `I` is double precision, then input pixel values are interpreted as means of Poisson distributions scaled up by `1e12`. For example, if an input pixel has the value `5.5e-12`, then the corresponding output pixel will be generated from a Poisson distribution with mean of 5.5 and then scaled back down by `1e12`. If `I` is single precision, the scale factor used is `1e6`. If `I` is `uint8` or `uint16`, then input pixel values are used directly without scaling. For example, if a pixel in a `uint8` input has the value 10, then the corresponding output pixel will be generated from a Poisson distribution with mean 10.

`J = imnoise(I, 'salt & pepper', d)` adds salt and pepper noise to the image `I`, where `d` is the noise density. This affects approximately `d*numel(I)` pixels. The default for `d` is 0.05.

`J = imnoise(I, 'speckle', v)` adds multiplicative noise to the image `I`, using the equation $J = I+n*I$, where `n` is uniformly distributed random noise with mean 0 and variance `v`. The default for `v` is 0.04.

Note The mean and variance parameters for `'gaussian'`, `'localvar'`, and `'speckle'` noise types are always specified as if the image were of class `double` in the range [0, 1]. If the input image is of class `uint8` or `uint16`, the `imnoise` function converts the image to `double`, adds noise according to the specified type and parameters, and then converts the noisy image back to the same class as the input.

`gpuarrayJ = imnoise(gpuarrayI, ___)` adds noise to the `gpuArray` intensity image `gpuarrayI`, performing the operation on a GPU. Returns a `gpuArray` image `J` of the same class. This syntax requires the Parallel Computing Toolbox.

Class Support

For most noise types, the input image **I** can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. For Poisson noise, `int16` is not allowed. The output image **J** is of the same class as **I**. If **I** has more than two dimensions it is treated as a multidimensional intensity image and not as an RGB image.

An input `gpuArray` image **I** can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. For Poisson noise, `int16` is not allowed. The output `gpuArray` image **J** is of the same class as **I**. If **I** has more than two dimensions it is treated as a multidimensional intensity image and not as an RGB `gpuArray` image.

Examples

Add noise to an image.

Add noise to an image.

```
I = imread('eight.tif');  
J = imnoise(I,'salt & pepper',0.02);  
figure, imshow(I)
```



`figure, imshow(J)`



Add Noise to an Image Performing Operation on a GPU

```
I = gpuArray(imread('eight.tif'));  
J = imnoise(I, 'salt & pepper', 0.02);  
  
figure, imshow(I);  
figure, imshow(J);
```

See Also

rand | randn | gpuArray

Introduced before R2006a

imopen

Morphologically open image

Syntax

```
IM2 = imopen(IM,SE)
IM2 = imopen(IM,NHOOD)
gpuarrayIM2 = imopen(gpuarrayIM, ___ )
```

Description

`IM2 = imopen(IM,SE)` performs morphological opening on the grayscale or binary image `IM` with the structuring element `SE`. The argument `SE` must be a single structuring element object, as opposed to an array of objects. The morphological open operation is an erosion followed by a dilation, using the same structuring element for both operations.

`IM2 = imopen(IM,NHOOD)` performs opening with the structuring element `strel(NHOOD)`, where `NHOOD` is an array of 0's and 1's that specifies the structuring element neighborhood.

`gpuarrayIM2 = imopen(gpuarrayIM, ___)` performs the operation on a graphics processing unit (GPU) with the structuring element `strel(NHOOD)`, if `NHOOD` is an array of 0s and 1s that specifies the structuring element neighborhood, or `strel(gather(NHOOD))` if `NHOOD` is a `gpuArray` object that specifies the structuring element neighborhood. This syntax requires the Parallel Computing Toolbox.

Class Support

`IM` can be any numeric or logical class and any dimension, and must be nonsparse. If `IM` is logical, then `SE` must be flat.

`gpuarrayIM` must be a `gpuArray` of type `uint8` or `logical`. When used with a `gpuarray`, the structuring element must be flat and two-dimensional.

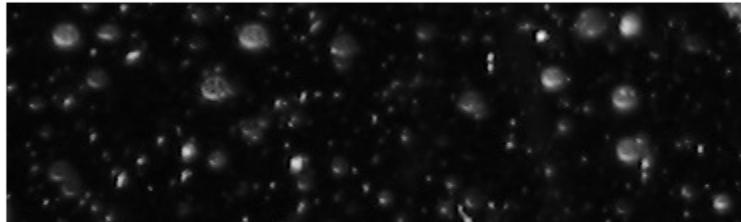
The output has the same class as the input.

Examples

Morphologically Open Image with a Disk-Shaped Structuring Element

Read the image into the workspace and display it.

```
original = imread('snowflakes.png');  
imshow(original);
```

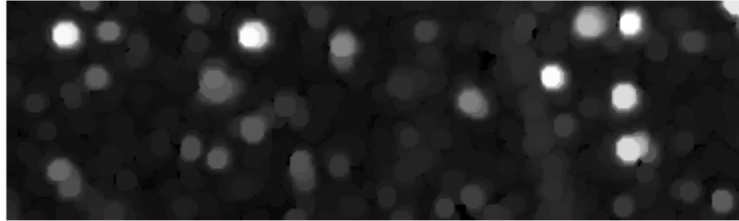


Create a disk-shaped structuring element with a radius of 5 pixels.

```
se = strel('disk',5);
```

Remove snowflakes having a radius less than 5 pixels by opening it with the disk-shaped structuring element.

```
afterOpening = imopen(original,se);  
figure  
imshow(afterOpening,[]);
```



Morphologically Open Image with Disk-shaped Structuring Element on a GPU

Read an image.

```
original = imread('snowflakes.png');
```

Create a disk-shaped structuring element.

```
se = strel('disk',5);
```

Morphologically open the image on a GPU, using a `gpuArray` object, and display the images.

```
afterOpening = imopen(gpuArray(original),se);  
figure, imshow(original), figure, imshow(afterOpening,[])
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the image input argument, **IM**, must be 2-D or 3-D and the structuring element input argument, **SE**, must be a compile-time constant.

See Also

[imclose](#) | [imdilate](#) | [imerode](#) | [strel](#) | [offsetstrel](#) | [gpuArray](#)

Introduced before R2006a

imoverlay

Burn binary mask into 2-D image

Syntax

```
B = imoverlay(A,BW)
B = imoverlay( ____,color)
```

Description

`B = imoverlay(A,BW)` fills the grayscale or RGB input image, `A`, with a solid color where the input binary mask, `BW`, is `true`.

`B = imoverlay(____,color)` lets you specify the color that `imoverlay` uses to fill the image. `color` is a valid MATLAB color specification.

Examples

Burn Binary Image into Grayscale Image

Read grayscale image into the workspace.

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

Read binary image into the workspace.

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

Burn the binary image into the grayscale image, choosing the color to be used.

```
B = imoverlay(A,BW,'yellow');
```

Display the result.

```
figure
```

```
imshow(B)
```



Input Arguments

A — Input image

real, nonsparse 2-D matrix

Input image, specified as a real, nonsparse 2-D matrix.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

BW — Mask image

2-D logical matrix

Mask image, specified 2-D logical matrix the same size as **A**.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

color — Color used for the overlay

MATLAB color specification

Color used for the overlay, specified as a MATLAB color specification. For example, if you want to specify the color red, you could use any of the following specifications: 'red', 'r', or [1 0 0].

Output Arguments

B — Output image

2-D matrix

Output image, returned as a 2-D matrix of class `uint8`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, if you specify color as a character vector, it must be a compile-time constant.

See Also

See Also

`boundarymask` | `superpixels`

Introduced in R2016a

imoverview

Overview tool for image displayed in scroll panel

Syntax

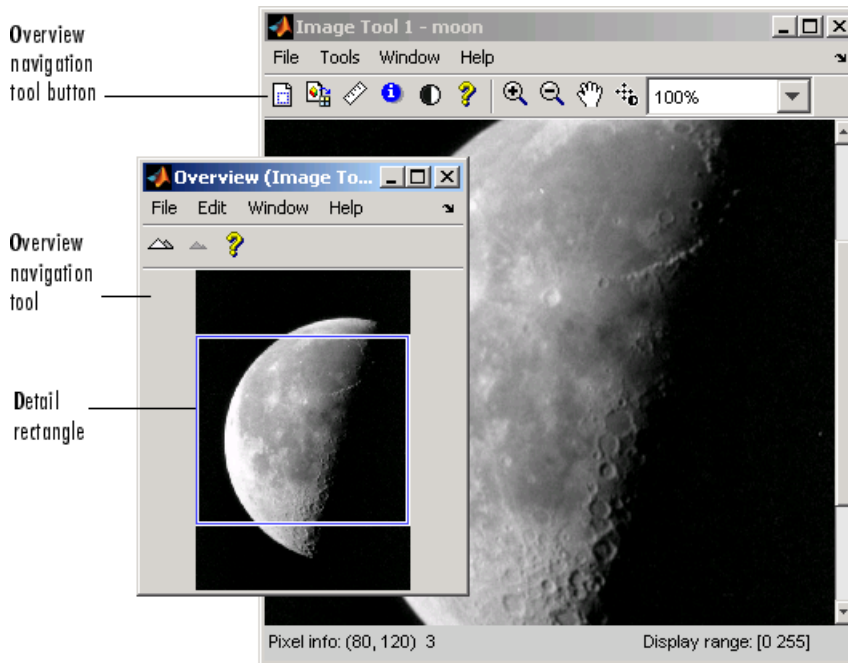
```
imoverview(himage)  
hfig = imoverview(...)
```

Description

`imoverview(himage)` creates an Overview tool associated with the image specified by the handle `himage`, called the target image. The target image must be contained in a scroll panel created by `imscrollpanel`.

The Overview tool is a navigation aid for images displayed in a scroll panel. `imoverview` creates the tool in a separate figure window that displays the target image in its entirety, scaled to fit. Over this scaled version of the image, the tool draws a rectangle, called the detail rectangle, that shows the portion of the target image that is currently visible in the scroll panel. To view portions of the image that are not currently visible in the scroll panel, move the detail rectangle in the Overview tool.

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



`hfig = imoverview(...)` returns a handle to the Overview tool figure.

Note

To create an Overview tool that can be embedded in an existing figure or uipanel object, use `imoverviewpanel`.

Examples

Create a figure, disabling the toolbar and menubar, because the toolbox navigation tools are not compatible with the standard MATLAB zoom and pan tools. Then create a scroll panel in the figure and use scroll panel API functions to set the magnification.

```
hFig = figure('Toolbar','none',...
'Menubar','none');
hIm = imshow('tape.png');
```



```
hSP = imscrollpanel(hFig,hIm);  
api = iptgetapi(hSP);  
api.setMagnification(2) % 2X = 200%  
imoverview(hIm)
```

See Also

[imoverviewpanel](#) | [imscrollpanel](#)

Introduced before R2006a

imoverviewpanel

Overview tool panel for image displayed in scroll panel

Syntax

```
hpanel = imoverviewpanel(hparent,himage)
```

Description

`hpanel = imoverviewpanel(hparent,himage)` creates an Overview tool panel associated with the image specified by the handle `himage`, called the target image. `himage` must be contained in a scroll panel created by `imsrollpanel`. `hparent` is a handle to the figure or `uipanel` object that will contain the Overview tool panel. `imoverviewpanel` returns `hpanel`, a handle to the Overview tool `uipanel` object.

The Overview tool is a navigation aid for images displayed in a scroll panel. `imoverviewpanel` creates the tool in a `uipanel` object that can be embedded in a figure or `uipanel` object. The tool displays the target image in its entirety, scaled to fit. Over this scaled version of image, the tool draws a rectangle, called the detail rectangle, that shows the portion of the target image that is currently visible in the scroll panel. To view portions of the image that are not currently visible in the scroll panel, move the detail rectangle in the Overview tool.

Note

To create an Overview tool in a separate figure, use `imoverview`. When created using `imoverview`, the Overview tool includes zoom-in and zoom-out buttons.

Examples

Create an Overview tool that is embedded in the same figure that contains the target image.

```
hFig = figure('Toolbar','none','Menubar','none');
```

```
hIm = imshow('tissue.png');  
hSP = imscrollpanel(hFig,hIm);  
set(hSP,'Units','normalized','Position',[0 .5 1 .5])  
hOvPanel = imoverviewpanel(hFig,hIm);  
set(hOvPanel,'Units','Normalized',...  
'Position',[0 0 1 .5])
```

See Also

[imoverview](#) | [imscrollpanel](#)

Introduced before R2006a

impixel

Pixel color values

Syntax

```
impixel(I)
P = impixel(I,c,r)
P = impixel(X,map)
P = impixel(X,map,c,r)
[c,r,P] = impixel(____)
P = impixel(x,y,I,xi,yi)
[xi,yi,P] = impixel(x,y,I,xi,yi)
```

Description

`impixel(I)` returns the value of pixels in the specified image `I`, where `I` can be a grayscale, binary, or RGB image. `impixel` displays the image specified and waits for you to select the pixels in the image using the mouse. If you omit the input arguments, `impixel` operates on the image in the current axes.

Use normal button clicks to select pixels. Press **Backspace** or **Delete** to remove the previously selected pixel. To finish selecting pixels, adding a final pixel, press shift-click, right-click, or double-click. To finish selecting pixels without adding a final pixel, press **Return**.

When you finish selecting pixels, `impixel` returns an `m`-by-3 matrix of RGB values in the supplied output argument. If you do not supply an output argument, `impixel` returns the matrix in `ans`. `impixel` always returns pixel values as RGB triplets, regardless of the image type:

- For an RGB image, `impixel` returns the actual data for the pixel. The values are either `uint8` integers or `double` floating-point numbers, depending on the class of the image array.
- For an indexed image, `impixel` returns the RGB triplet stored in the row of the colormap that the pixel value points to. The values are `double` floating-point numbers.

- For grayscale images, `impixel` returns the intensity value as an RGB triplet, where $R=G=B$. The values are either `uint8` integers or `double` floating-point numbers, depending on the class of the image array.
- For binary images, `impixel` returns the intensity value as an RGB triplet, where $R=G=B$. The values are `double` floating-point numbers.

`P = impixel(I,c,r)` returns the values of pixels specified by the row and column vectors `r` and `c`. `r` and `c` must be equal-length vectors. The k th row of `P` contains the RGB values for the pixel $(r(k),c(k))$.

`P = impixel(X,map)` returns the value of pixels in the specified indexed image `I` with corresponding colormap, `map`.

`P = impixel(X,map,c,r)` returns the value of pixels specified by the row and column vectors `r` and `c`.

`[c,r,P] = impixel(___)` returns the coordinates of the selected pixels.

`P = impixel(x,y,I,xi,yi)` returns the values of pixels in the specified image, `I`, where `x` and `y` are two-element vectors specifying the image `XData` and `YData`. `xi` and `yi` are equal-length vectors specifying the spatial coordinates of the pixels whose values are returned in `P`

`[xi,yi,P] = impixel(x,y,I,xi,yi)` returns the coordinates of the selected pixels.

Class Support

The input image can be of class `uint8`, `uint16`, `int16`, `single`, `double`, or `logical`. All other inputs are of class `double`.

If the input is `double`, the output `P` is `double`. For all other input classes the output is `single`. The rest of the outputs are `double`.

Examples

Return Individual Pixel Values from Image

Read a truecolor image into the workspace.

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

Determine the column `c` and row `r` indices of the pixels to extract.

```
c = [1 12 146 410];  
r = [1 104 156 129];
```

Return the data at the selected pixel locations.

```
pixels = impixel(RGB,c,r)
```

```
pixels =
```

```
    62    29    64  
    62    34    63  
   166    54    60  
    59    28    47
```

See Also

`improfile`

Introduced before R2006a

impixelinfo

Pixel Information tool

Syntax

```
impixelinfo
impixelinfo(h)
impixelinfo(hparent,himage)
hpanel = impixelinfo(...)
```

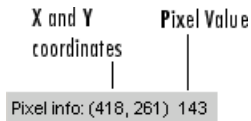
Description

`impixelinfo` creates a Pixel Information tool in the current figure. The Pixel Information tool displays information about the pixel in an image that the pointer is positioned over. The tool can display pixel information for all the images in a figure.

The Pixel Information tool is a `uipanel` object, positioned in the lower-left corner of the figure. The tool contains the text label `Pixel info:` followed by the pixel information. Before you move the pointer over the image, the tool contains the default pixel information text `(X,Y) Pixel Value`. Once you move the pointer over the image, the information displayed varies by image type, as shown in the following table. If you move the pointer off the image, the pixel information tool displays the default pixel information label for that image type.

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

For example, for grayscale (intensity) images, the pixel information tool displays the x and y coordinates of the pixel and its value, as shown in the following figure.



If you want to display the pixel information without the “Pixel Info” label, use the `impixelinfoval` function.

`impixelinfo(h)` creates a Pixel Information tool in the figure specified by `h`, where `h` is a handle to an image, axes, uipanel, or figure object. Axes, uipanel, or figure objects must contain at least one image object.

`impixelinfo(hparent,himage)` creates a Pixel Information tool in `hparent` that provides information about the pixels in `himage`. `himage` is a handle to an image or an array of image handles. `hparent` is a handle to the figure or uipanel object that contains the pixel information tool.

`hpanel = impixelinfo(...)` returns a handle to the Pixel Information tool uipanel.

Note

To copy the pixel information label to the clipboard, right-click while the pointer is positioned over a pixel. In the context menu displayed, choose **Copy pixel info**.

Examples

Display an image and add a Pixel Information tool to the figure. The example shows how you can change the position of the tool in the figure using properties of the tool uipanel object.

```
h = imshow('hestain.png');
hp = impixelinfo;
set(hp,'Position',[5 1 300 20]);
```

Use the Pixel Information tool in a figure containing multiple images of different types.

```
figure
subplot(1,2,1), imshow('liftingbody.png');
subplot(1,2,2), imshow('autumn.tif');
```


`impixelinfo;`

See Also

`impixelinfoval` | `imtool`

Introduced before R2006a

impixelinfoval

Pixel Information tool without text label

Syntax

```
hcontrol = impixelinfoval(parentobj,imageobj)
```

Description

`hcontrol = impixelinfoval(parentobj,imageobj)` creates a Pixel Information tool in `parentobj` that provides information about the pixels in the image specified by `imageobj`. `parentobj` is a figure or uipanel object. `imageobj` can be an image object or array of image objects.

The Pixel Information tool displays information about the pixel in an image that the pointer is positioned over. If the figure contains multiple images, the tool displays pixel information for all the images.

When created with `impixelinfo`, the tool is a uipanel object, positioned in the lower-left corner of the figure, that contains the text label **Pixel Info:** followed by the *x*- and *y*-coordinates of the pixel and its value. When created with `impixelinfoval`, the tool is a uicontrol object positioned in the lower-left corner of the figure, that displays the pixel information without the text label, as shown in the following figure.

X and Y coordinates	Pixel Value
(167, 251)	114

The information displayed depends on the image type. See `impixelinfo` for details.

To copy the pixel value label to the Clipboard, right-click while the pointer is positioned over a pixel. In the context menu displayed, choose **Copy pixel info**.

Examples

Add a Pixel Information tool to a figure. Note how you can change the style and size of the font used to display the value in the tool using standard graphics object properties.

```
ankle = dicomread('CT-MON02-16-ankle.dcm');  
h = imshow(ankle,[]);  
hText = impixelinfoval(gcf,h);  
set(hText,'FontWeight','bold')  
set(hText,'FontSize',10)
```

See also

`impixelinfo`

Introduced before R2006a

impixelregion

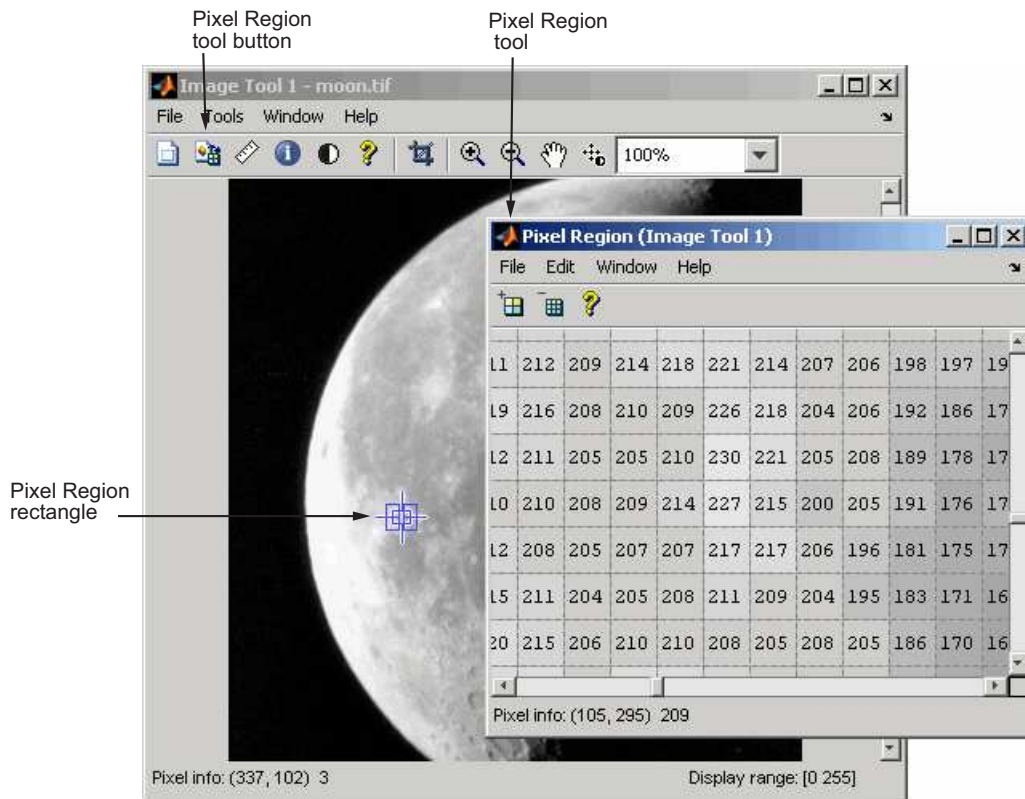
Pixel Region tool

Syntax

```
impixelregion  
impixelregion(h)  
hfig = impixelregion(...)
```

Description

`impixelregion` creates a Pixel Region tool associated with the image displayed in the current figure, called the target image. The Pixel Region tool opens a separate figure window containing an extreme close-up view of a small region of pixels in the target image, as shown in the following figure.



The Pixel Region rectangle defines the area of the target image that is displayed in the Pixel Region tool. You can move this rectangle over the target image using the mouse to view different regions. To get a closer view of the pixels displayed in the tool, use the zoom buttons on the Pixel Region tool toolbar or change the size of the Pixel Region rectangle using the mouse. You can also resize the Pixel Region tool itself to view more or fewer pixels. If the size of the pixels allows, the tool superimposes the numeric value of the pixel over each pixel.

To get the current position of the Pixel Region rectangle, right-click on the rectangle and select **Copy Position** from the context menu. The Pixel Region tool copies a four-element position vector to the clipboard. To change the color of the Pixel Region rectangle, right-click and select **Set Color**.

`impixelregion(h)` creates a Pixel Region tool associated with the object specified by the handle `h`. `h` can be a handle to a figure, axes, uipanel, or image object. If `h` is a handle to an axes or figure, `impixelregion` associates the tool with the first image found in the axes or figure.

`hfig = impixelregion(...)` returns `hfig`, a handle of the Pixel Region tool figure.

Note

To create a Pixel Region tool that can be embedded in an existing figure window or uipanel, use `impixelregionpanel`.

Examples

Display an image and then create a Pixel Region tool associated with the image.

```
imshow peppers.png
impixelregion
```

See Also

`impixelinfo` | `imtool` | `impixelregionpanel`

Introduced before R2006a

impixelregionpanel

Pixel Region tool panel

Syntax

```
hpanel = impixelregionpanel(hparent,himage)
```

Description

`hpanel = impixelregionpanel(hparent,himage)` creates a Pixel Region tool panel associated with the image specified by the handle `himage`, called the target image. This is the image whose pixels are to be displayed. `hparent` is the handle to the figure or `uipanel` object that will contain the Pixel Region tool panel. `hpanel` is the handle to the Pixel Region tool scroll panel.

The Pixel Region tool is a `uipanel` object that contains an extreme close-up view of a small region of pixels in the target image. If the size of the pixels allows, the tool superimposes the numeric value of the pixel over each pixel. To define the region being examined, the tool overlays a rectangle on the target image, called the pixel region rectangle. To view pixels in a different region, click and drag the rectangle over the target image. See `impixelregion` for more information.

Note

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

Examples

```
himage = imshow('peppers.png');  
hfigure = figure;  
hpanel = impixelregionpanel(hfigure, himage);
```

Set the panel's position to the lower-left quadrant of the figure.

```
set(hpanel, 'Position', [0 0 .5 .5])
```

See Also

`impixelregion` | `imrect` | `imscrollpanel`

Introduced before R2006a

implay

Play movies, videos, or image sequences

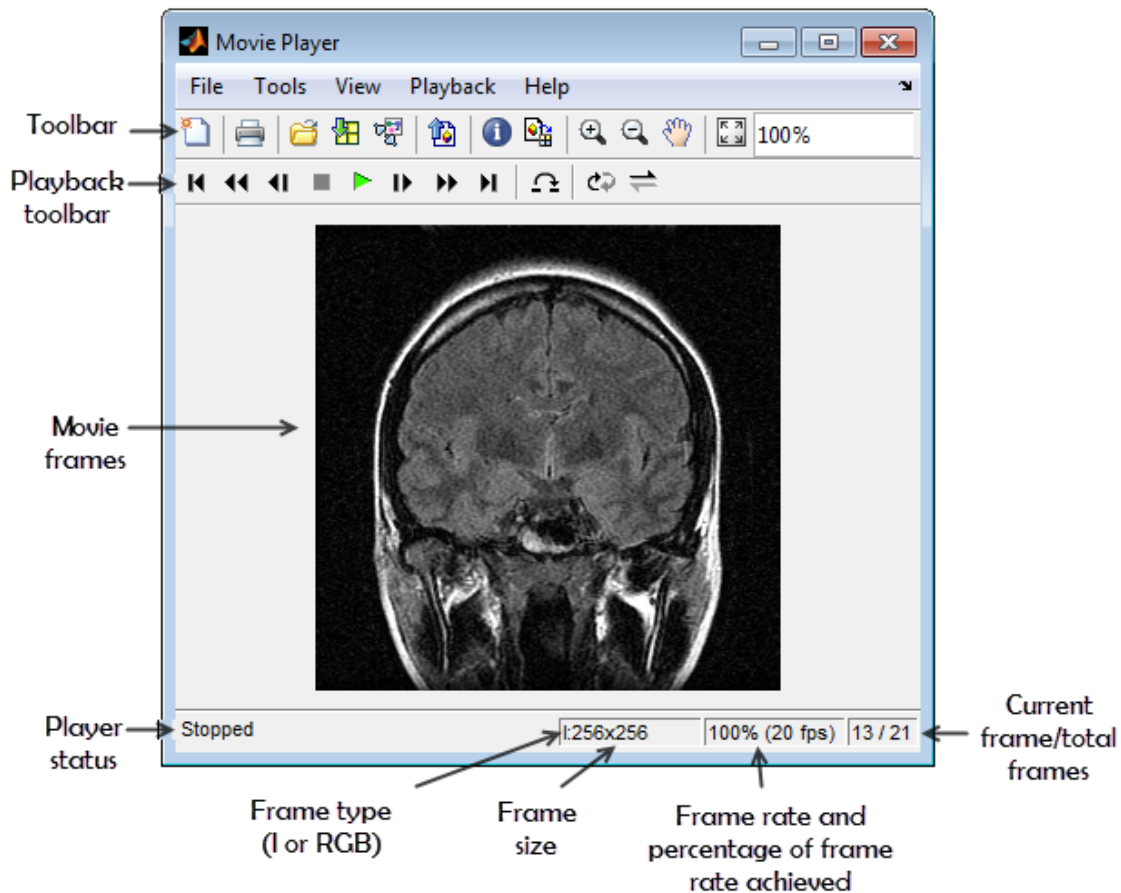
Syntax

```
implay  
implay(filename)  
implay(I)  
implay(..., FPS)
```

Description

`implay` opens the Video Viewer app. You can use Video Viewer to show MATLAB movies, videos, or image sequences (also called image stacks). To select the movie or image sequence that you want to play, use the Video Viewer **File** menu. To play the movie, jump to a specific frame in the image sequence, or change the frame rate of the display use the Video Viewer toolbar buttons or menu options. You can open multiple Video Viewers to view different movies simultaneously.

The following figure shows the Video Viewer app containing an image sequence.



`imshow(filename)` opens the Video Viewer app, displaying the content of the file specified by `filename`. The file can be an Audio Video Interleaved (AVI) file. The Video Viewer reads one frame at a time, conserving memory during playback. The Video Viewer does not play audio tracks.

`imshow(I)` opens the Video Viewer app, displaying the first frame in the multiframe image array specified by `I`. `I` can be a MATLAB movie structure, or a sequence of binary, grayscale, or truecolor images. A binary or grayscale image sequence can be an M -by- N -by-1-by- K array or an M -by- N -by- K array. A truecolor image sequence must be an M -by- N -by-3-by- K array.

`implay(..., FPS)` specifies the rate at which you want to view the movie or image sequence. The frame rate is specified as frames-per-second. If omitted, the Video Viewer uses the frame rate specified in the file or the default value 20.

Class Support

It can be numeric but `uint8` is preferred. The actual data type used to display pixels may differ from the source data type.

Examples

Animate a sequence of images.

```
load cellsequence
implay(cellsequence,10);
```

Visually explore a stack of MRI images.

```
load mrystack
implay(mrystack);
```

Play an AVI file.

```
implay('rhinos.avi');
```

Tips

- You can also open the Video Viewer app through the Apps tab. Navigate to the Image Processing and Computer Vision group and click Video Viewer.

Introduced in R2007b

impoint

Create draggable point

Syntax

```
h = impoint
h = impoint(hparent)
h = impoint(hparent,position)
h = impoint(hparent,x, y)
h = impoint(..., param,val)
```

Description

`h = impoint` begins interactive placement of a draggable point on the current axes. The function returns `h`, a handle to an `impoint` object. The point has a context menu associated with it that controls aspects of its appearance and behavior—see “Interactive Behavior” on page 1-983. Right-click on the point to access this context menu.

`h = impoint(hparent)` begins interactive placement of a point on the object specified by `hparent`. `hparent` specifies the HG parent of the point graphics, which is typically an axes but can also be any other object that can be the parent of an `hggroup`.

`h = impoint(hparent,position)` creates a draggable point on the object specified by `hparent`. `position` is a 1-by-2 array of the form `[x y]` that specifies the initial position of the point.

`h = impoint(hparent,x, y)` creates a draggable point on the object specified by `hparent`. `x` and `y` are both scalars that together specify the initial position of the point.


`h = impoint(..., param,val)` creates a draggable point, specifying parameters and corresponding values that control the behavior of the point. The following table lists the parameter available. Parameter names can be abbreviated, and case does not matter.

Parameter	Description
'PositionConstraintFcn'	Function handle <code>fcn</code> that is called whenever the point is dragged using the mouse. You can use this


Parameter	Description
	function to control where the point can be dragged. See the help for the <code>setPositionConstraintFcn</code> method for information about valid function handles.

Interactive Behavior

When you call `impoint` with an interactive syntax, the pointer changes to a cross

hairs  when over the image. Click and drag the mouse to specify the position of the point. The point supports a context menu that you can use to control aspects of its appearance and behavior. The following table describes the interactive behavior of the point, including the right-click context menu options.



Interactive Behavior	Description
Moving the point.	Move the mouse pointer over the point. The mouse pointer changes to a fleur shape  . Click and drag the mouse to move the point.
Changing the color used to display the point.	Move the mouse pointer over the point. Right-click and select Set Color from the context menu and specify the color you want to use.
Retrieving the coordinates of the point.	Move the mouse pointer over the point. Right-click and select Copy Position from the context menu to copy a 1-by-2 array to the clipboard specifying the coordinates of the point [X Y].
Deleting the point	Move the pointer on top of the point. Right-click and select Delete from the context menu. To remove this option from the context menu, set the <code>Deletable</code> property to false: <code>h = impoint(); h.Deletable = false;</code>

Methods

The following lists the methods supported by the `impoint` object. Type `methods impoint` to see a complete list of all the methods.

addNewPositionCallback — Add new-position callback to ROI object

See `imroi` for information.

delete — Delete ROI object

See `imroi` for information.

getColor — Get color used to draw ROI object

See `imroi` for information.

getPosition — Return current position of point

`pos = getPosition(h)` returns the current position of the point `h`. The returned position, `pos`, is a two-element vector [x y].

getPositionConstraintFcn — Return function handle to current position constraint function

See `imroi` for information.

removeNewPositionCallback — Remove new-position callback from ROI object.

See `imroi` for information.

resume — Resume execution of MATLAB command line

See `imroi` for information.

setColor — Set color used to draw ROI object

See `imroi` for information.

setConstrainedPosition — Set ROI object to new position

See `imroi` for information.

setPosition — Set point to new position

`setPosition(h, pos)` sets the point `h` to a new position. The new position, `pos`, has the form, `[x y]`.

`setPosition(h, new_x, new_y)` sets the point `h` to a new position. `new_x` and `new_y` are both scalars that together specify the position of the point.

setPositionConstraintFcn — Set position constraint function of ROI object.

See `imroi` for information.

setString — Set text label for point

`setString(h, s)` sets a text label for the point `h`. The character vector, `s`, is placed to the lower right of the point.

wait — Block MATLAB command line until ROI creation is finished

See `imroi` for information.

Examples

Example 1

Use `impoint` methods to set custom color, set a label, enforce a boundary constraint, and update position in title as point moves.

```
figure, imshow rice.png
h = impoint(gca,100,200);
% Update position in title using newPositionCallback
addNewPositionCallback(h,@(h) title(sprintf('%1.0f,%1.0f',h(1),h(2))));
% Construct boundary constraint function
fcn = makeConstrainToRectFcn('impoint',get(gca,'XLim'),get(gca,'YLim'));
% Enforce boundary constraint function using setPositionConstraintFcn
setPositionConstraintFcn(h,fcn);
setColor(h,'r');
setString(h,'Point label');
```

Example 2

Interactively place a point. Use `wait` to block the MATLAB command line. Double-click on the point to resume execution of the MATLAB command line.

```
figure, imshow('pout.tif');
h = impoint(gca,[]);
position = wait(h);
```

Tips

If you use `impoint` with an axes that contains an image object, and do not specify a drag constraint function, users can drag the point outside the extent of the image and lose the point. When used with an axes created by the `plot` function, the axes limits automatically expand to accommodate the movement of the point.

See Also

`imellipse` | `imline` | `impoly` | `imrect` | `imroi` | `imfreehand` | `makeConstrainToRectFcn`

Introduced before R2006a

impoly

Create draggable, resizable polygon

Syntax

```
h = impoly
h = impoly(hparent)
h = impoly(hparent, position)
h = impoly(..., param1, val1, ...)
```

Description

`h = impoly` begins interactive placement of a polygon on the current axes. The function returns `h`, a handle to an `impoly` object. The polygon has a context menu associated with it that controls aspects of its appearance and behavior—see “Interactive Behavior” on page 1-988. Right-click on the polygon to access this context menu.

`h = impoly(hparent)` begins interactive placement of a polygon on the object specified by `hparent`. `hparent` specifies the HG parent of the polygon graphics, which is typically an axes but can also be any other object that can be the parent of an `hggroup`.


`h = impoly(hparent, position)` creates a draggable, resizable polygon on the object specified by `hparent`. `position` is an n -by-2 array that specifies the initial position of the vertices of the polygon. `position` has the form `[X1,Y1;...;XN,YN]`.

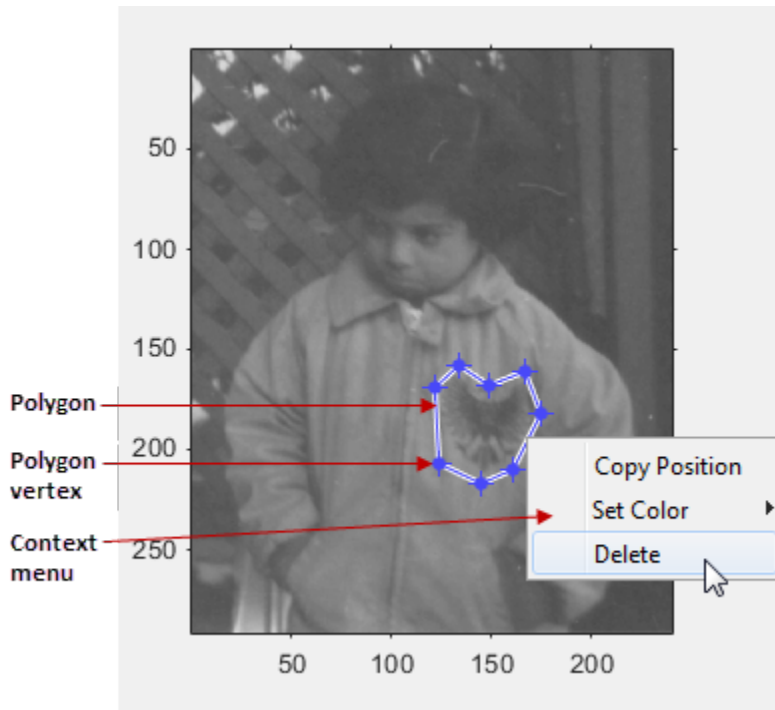
`h = impoly(..., param1, val1, ...)` creates a draggable, resizable polygon, specifying parameters and corresponding values that control the behavior of the polygon. The following table lists available parameters. Parameter names can be abbreviated, and case does not matter.

Parameter	Description
'Closed'	Scalar logical that controls whether the polygon is closed. When set to <code>true</code> (the default), <code>impoly</code> creates a closed polygon, that is, it draws a straight


Parameter	Description
	line between the last vertex specified and the first vertex specified to create a closed region. When <code>Closed</code> is <code>false</code> , <code>impoly</code> does not connect the last vertex with the first vertex, creating an open polygon (or polyline).
'PositionConstraintFcn'	Function handle <code>fcn</code> that is called whenever the object is dragged using the mouse. You can use this function to control where the line can be dragged. See the help for the <code>setPositionConstraintFcn</code> method for information about valid function handles.





Interactive Behavior

When you call `impoly` with an interactive syntax, the pointer changes to a cross hairs  when over the image. Click and drag the mouse to define the vertices of the polygon and adjust the size, shape, and position of the polygon. The polygon also supports a context menu that you can use to control aspects of its appearance and behavior. The choices in the context menu vary whether you position the pointer on an edge of the polygon (or anywhere inside the region) or on one of the vertices. The following figure shows the context menu when the pointer is on the polygon, not on a vertex.



The following table lists the interactive behaviors supported by `impoly`.

Interactive Behavior	Description
Closing the polygon.	<p>Use any of the following mechanisms:</p> <ul style="list-style-type: none"> • Move the pointer over the initial vertex of the polygon that you selected. The pointer changes to a circle . Click either mouse button. • Double-click the left mouse button. This action creates a vertex at the point under the mouse and draws a straight line connecting this vertex with the initial vertex. • Click the right mouse button. This action draws a line connecting the last vertex selected with the initial vertex; it does not create a new vertex.

Interactive Behavior	Description
Adding a new vertex.	Move the pointer over an edge of the polygon and press the A key. The shape of the pointer changes  . Click the left mouse button to create a new vertex at that position on the line.
Moving a vertex. (Reshaping the polygon.)	Move the pointer over a vertex. The pointer changes to a circle  . Click and drag the vertex to its new position.
Deleting a vertex.	Move the pointer over a vertex. The shape changes to a circle  . Right-click and select Delete Vertex from the vertex context menu. This action deletes the vertex and adjusts the shape of the polygon, drawing a new straight line between the two vertices that were neighbors of the deleted vertex.
Deleting the polygon	Move the pointer inside the polygon or on one of the lines that define the polygon, not on a vertex. Right-click and select Delete from the context menu. To remove this option from the context menu, set the Deletable property to false: <code>h = impoly(); h.Deletable = false;</code>
Moving the polygon.	Move the pointer inside the polygon. The pointer changes to a fleur shape  . Click and drag the mouse to move the polygon.
Changing the color of the polygon	Move the pointer inside the polygon. Right-click and select Set Color from the context menu.
Retrieving the coordinates of the vertices	Move the pointer inside the polygon. Right-click and select Copy Position from the context menu. <code>impoly</code> copies an <i>n</i> -by-2 array containing the <i>x</i> - and <i>y</i> -coordinates of each vertex to the clipboard. <i>n</i> is the number of vertices you specified.

Methods

Each `impoly` object supports a number of methods, listed below. Methods inherited from the base class are links to that class.

addNewPositionCallback — Add new-position callback to ROI object

See `imroi` for information.

createMask — Create mask within image

See `imroi` for information.

delete — Delete ROI object

See `imroi` for information.

getColor — Get color used to draw ROI object

See `imroi` for information.

getPosition — Return current position of polygon

`pos = getPosition(h)` returns the current position of the polygon `h`. The returned position, `pos`, is an N-by-2 array [`X1 Y1; . . . ; XN YN`].

getPositionConstraintFcn — Return function handle to current position constraint function

See `imroi` for information.

removeNewPositionCallback — Remove new-position callback from ROI object.

See `imroi` for information.

resume — Resume execution of MATLAB command line

See `imroi` for information.

setClosed — Set geometry of polygon

`setClosed(TF)` sets the geometry of the polygon. `TF` is a logical scalar. `true` means that the polygon is closed. `false` means that the polygon is an open polyline.

setColor — Set color used to draw ROI object

See `imroi` for information.

setPosition — Set ROI object to new position

See `imroi` for information.

setPosition — Set polygon to new position

`setPosition(h, pos)` sets the polygon `h` to a new position. The new position, `pos`, is an n -by-2 array, `[x1 y1; ..; xn yn]` where each row specifies the position of a vertex of the polygon.

setPositionConstraintFcn — Set position constraint function of ROI object.

See `imroi` for information.

setVerticesDraggable — Control whether vertices may be dragged

`setVerticesDraggable(h, TF)` sets the interactive behavior of the vertices of the polygon `h`. `TF` is a logical scalar. `True` means that the vertices of the polygon are draggable. `False` means that the vertices of the polygon are not draggable.

wait — Block MATLAB command line until ROI creation is finished

See `imroi` for information.

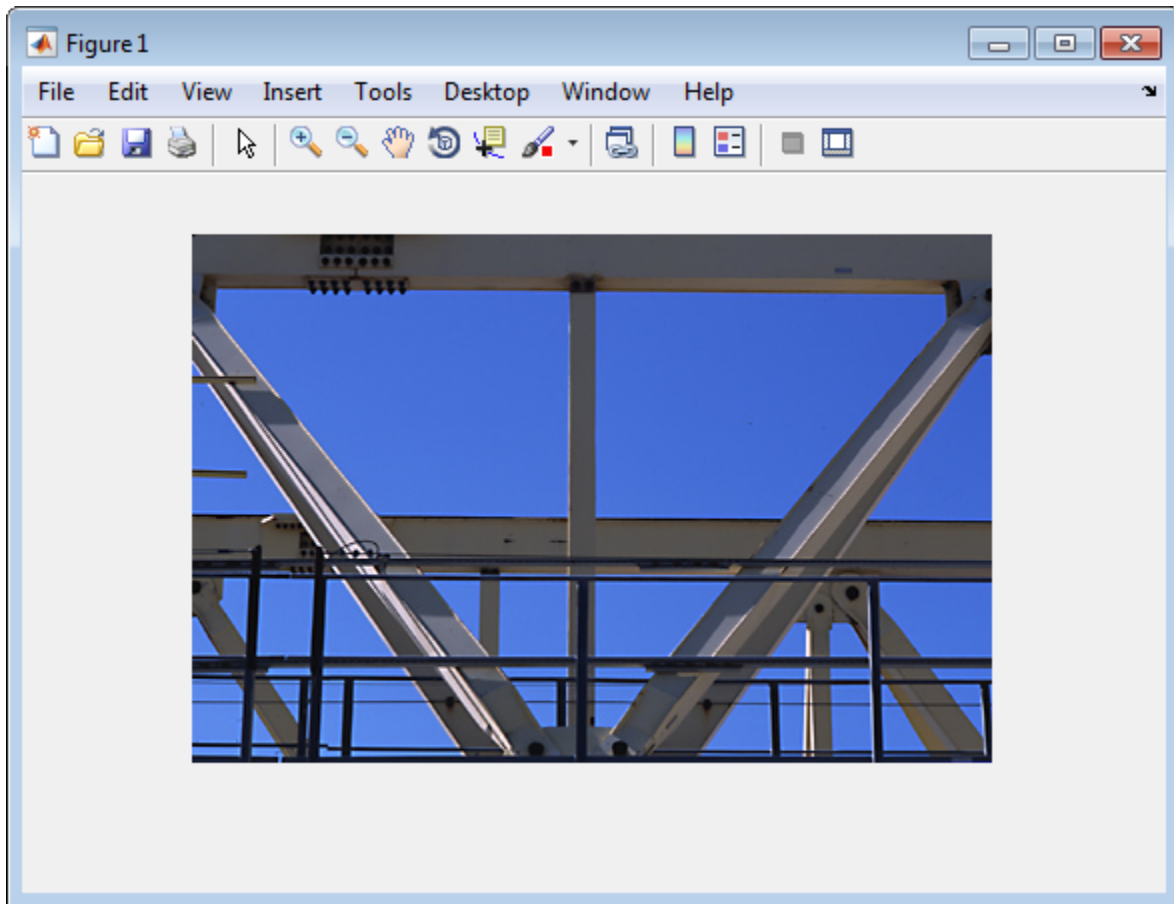
Examples

Draw Polygon on Image and Specify Position Constraint Function

Draw a polygon on an image and specify a position constraint function using `makeConstrainToRectFcn` to keep the polygon inside the original `xlim` and `ylim` ranges. Display updated position of the polygon in the title.

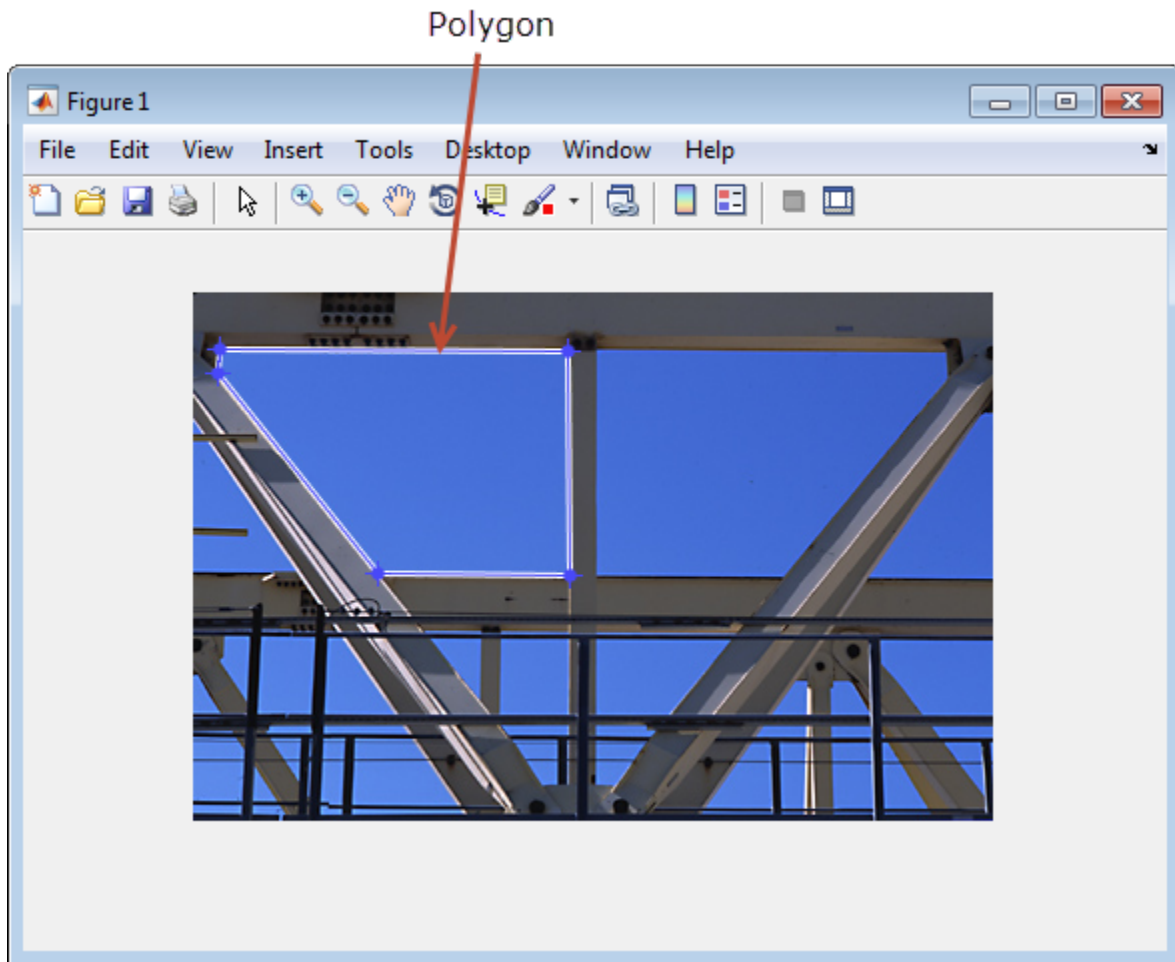
Display image.

```
figure
imshow('gantrycrane.png');
```



Draw polygon on the image, specifying location of vertices.

```
h = impoly(gca, [188,30; 189,142; 93,141; 13,41; 14,29]);
```



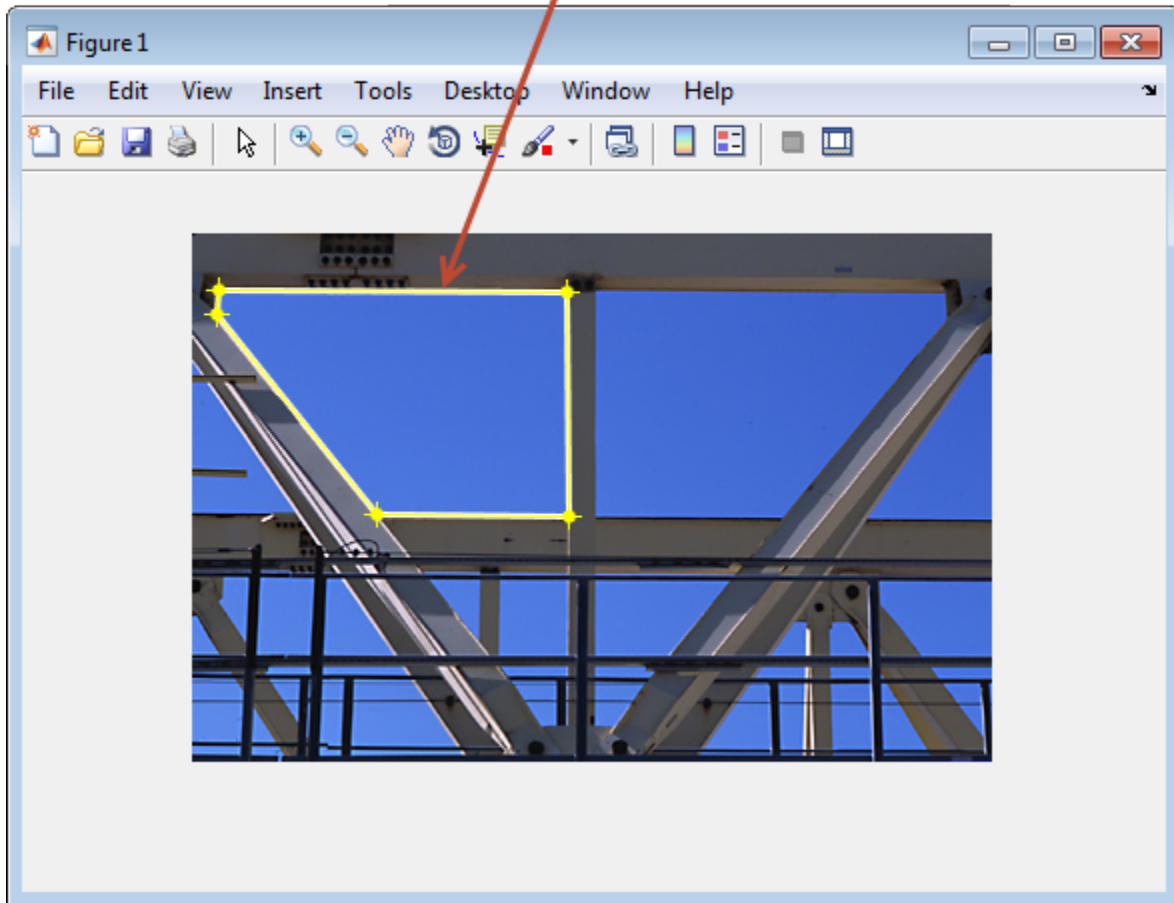
Get the API associated with the polygon so that you can modify properties of the polygon.

```
api = iptgetapi(h);
```

Set the color of the polygon to yellow.

```
api.setColor('yellow');
```

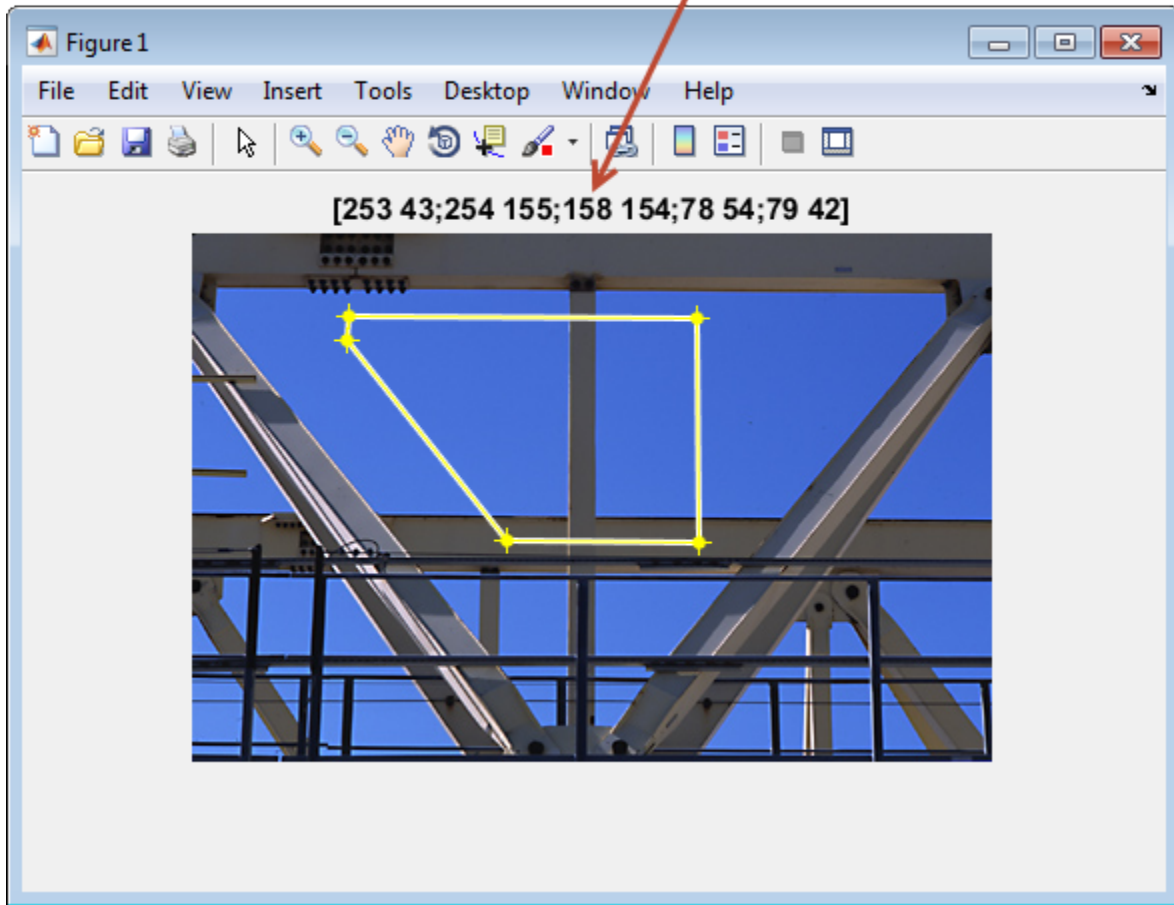

Color of polygon changed to yellow



Define a function for the new position callback. This function displays the current position of the polygon whenever it is moved.

```
api.addNewPositionCallback(@(p) title(mat2str(p,3)));
```

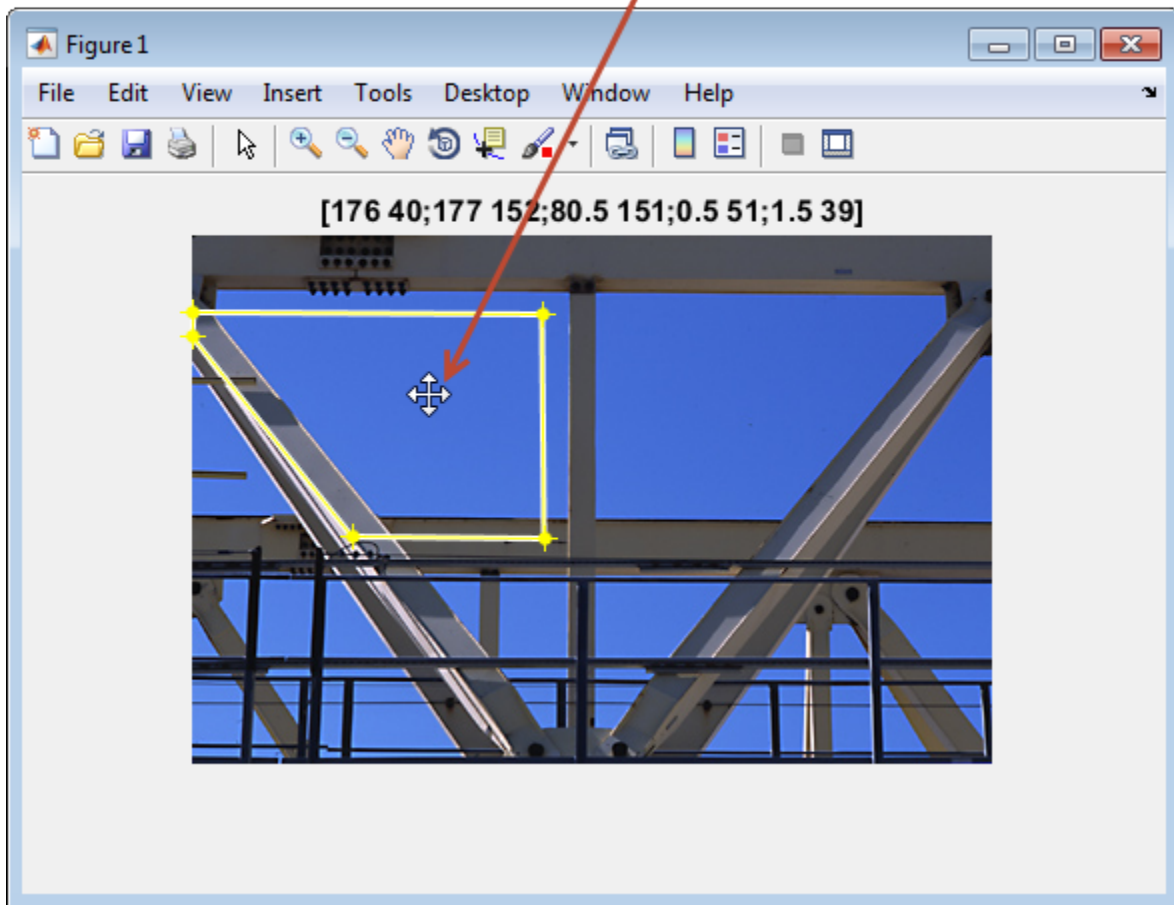
Title with current position of polygon



Create the function that constrains the movement of the polygon, specifying the boundary of the image as the limits, and then set the value of the `setPositionConstraintFcn` property.

```
fcn = makeConstrainToRectFcn('impoly',get(gca,'XLim'),get(gca,'YLim'));
api.setPositionConstraintFcn(fcn);
```

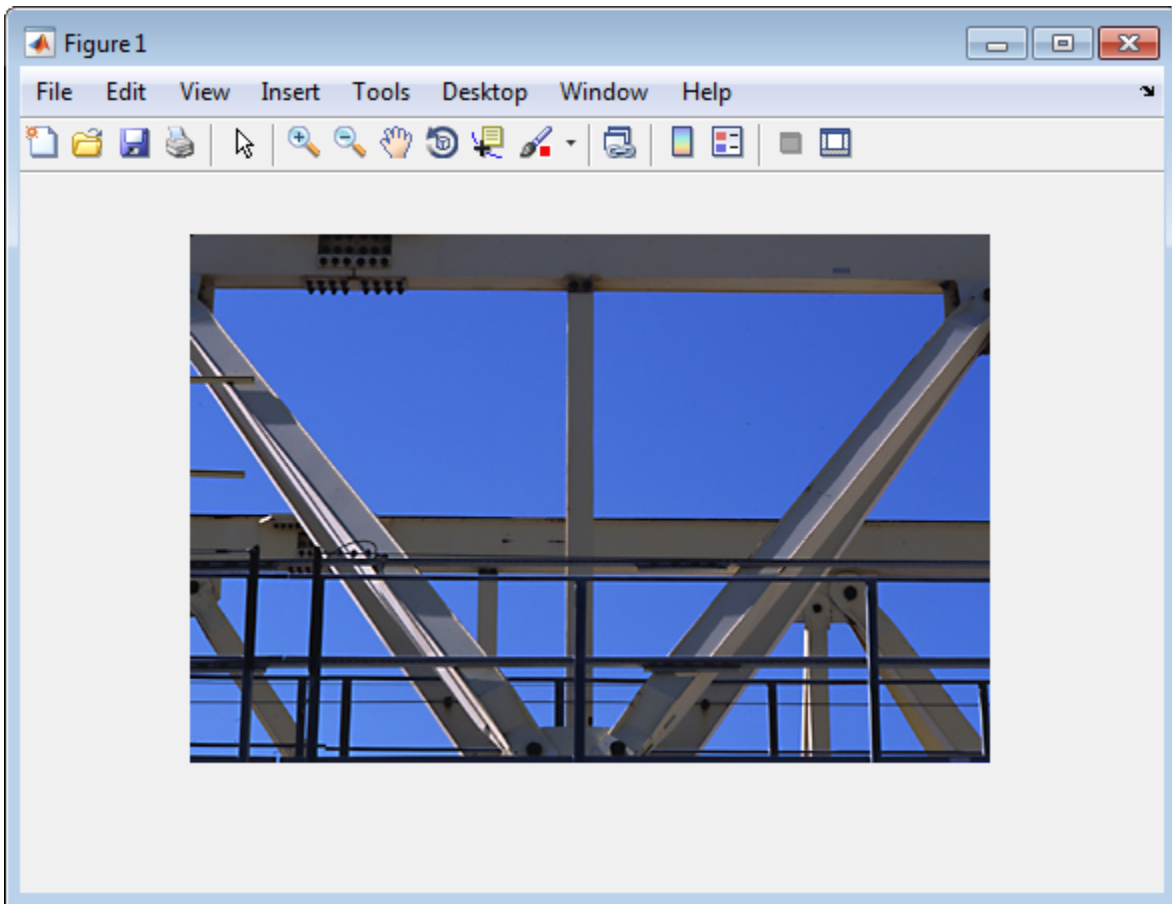
Can't move polygon past image border



Interactively Create a Polygon by Clicking to Specify Vertex Locations

Display image.

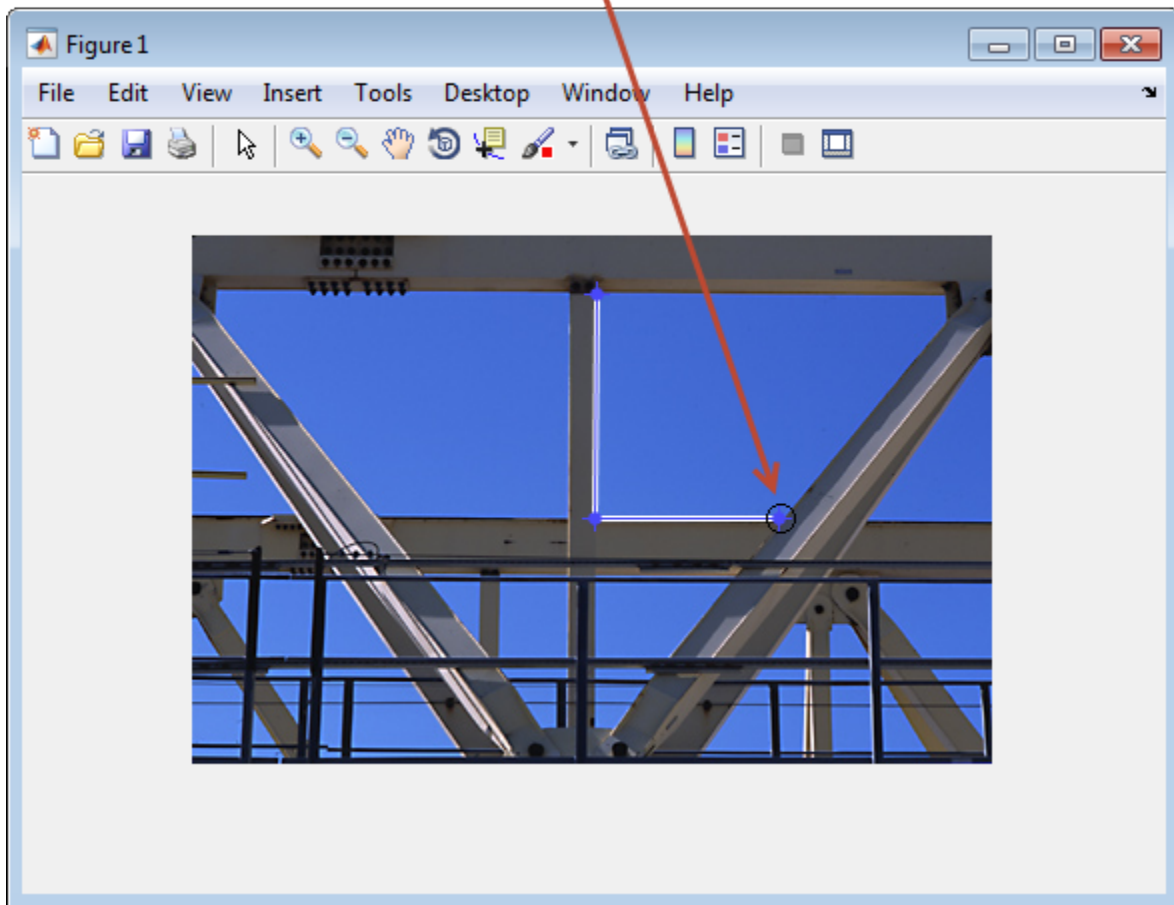
```
figure  
imshow('gantrycrane.png');
```



Create a polygon, specifying several vertices, but leave it unfinished so that you can finish it interactively. The example sets `Closed` to `false` so that the polygon is left open. When you move the cursor over one of the endpoints of the polygon, the cursor shape changes to a circle.

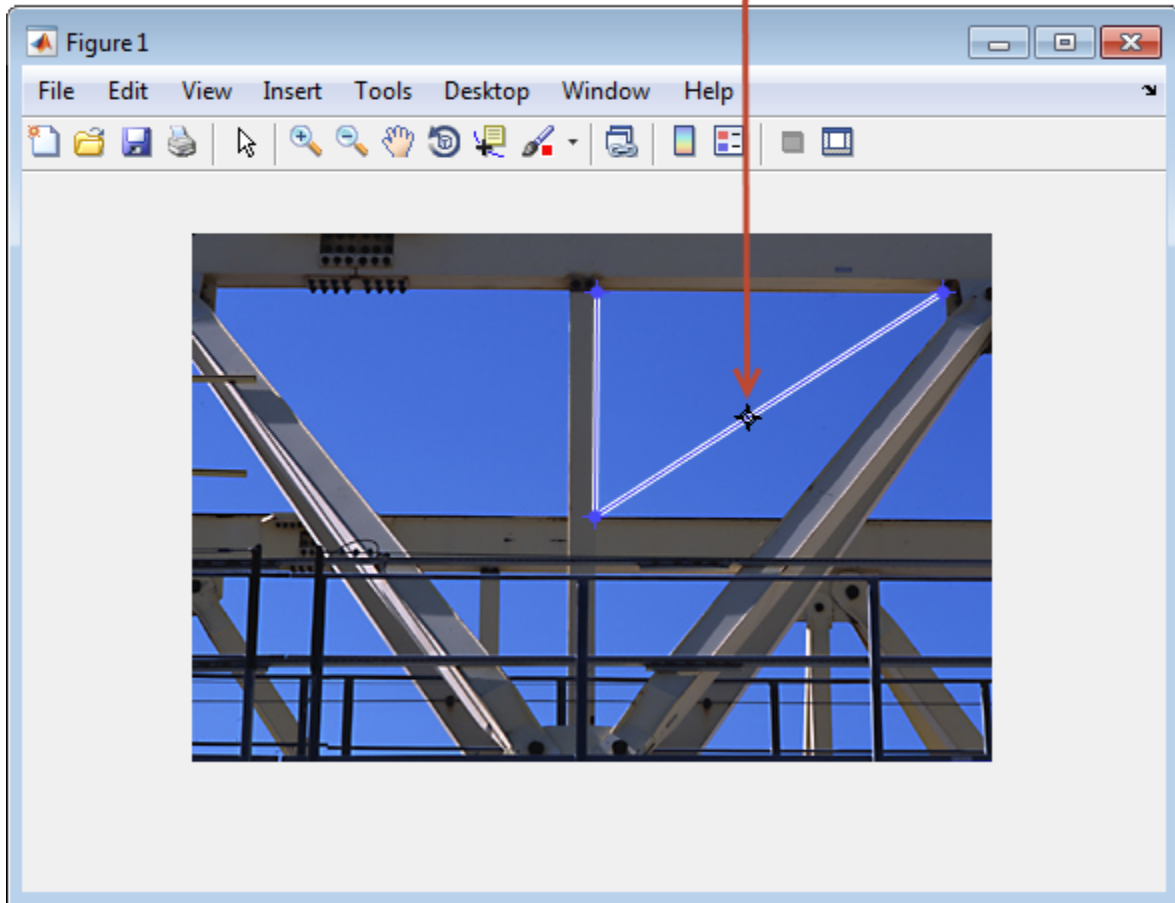
```
h = impoly(gca,[203,30; 202,142; 294,142], 'Closed',false);
```

Cursor changes to circle.



Complete the polygon. Grab one of the ends of the existing lines. Extend the line by dragging it to another corner of the shape you want to create. Then, while positioning the cursor over the line, press the **A** key to add a vertex to the line. Once you create the vertex you can drag it anywhere you want to create the shape you want. Continue dragging the line and adding vertices as you want. For more information, see “Interactive Behavior” on page 1-988.

Press and hold the A key to get the add vertex cursor. Then drag the new vertex where desired.



See also

`imellipse`, `imfreehand`, `imline`, `impoint`, `imrect`, `imroi`,
`makeConstrainToRectFcn`

Tips

If you use `impoly` with an axes that contains an image object, and do not specify a position constraint function, users can drag the polygon outside the extent of the image and lose the polygon. When used with an axes created by the `plot` function, the axes limits automatically expand when the polygon is dragged outside the extent of the axes.

Introduced in R2007b

impositionrect

Create draggable position rectangle

Syntax

```
H = impositionrect(hparent,position)
```

Note This function is obsolete and may be removed in future versions. Use `imrect` instead.

Description

`H = impositionrect(hparent,position)` creates a position rectangle on the object specified by `hparent`. The function returns `H`, a handle to the position rectangle, which is an hggroup object. `hparent` specifies the hggroup's parent, which is typically an axes object, but can also be any other object that can be the parent of an hggroup. `position` is a four-element position vector that specifies the initial location of the rectangle. `position` has the form `[XMIN YMIN WIDTH HEIGHT]`.

All measurements are in units specified by the `Units` property axes object. When you do not specify the `position` argument, `impositionrect` uses `[0 0 1 1]` as the default value.

API Function Syntaxes

A position rectangle contains a structure of function handles, called an API, that can be used to manipulate it. To retrieve this structure from the position rectangle, use the `iptgetapi` function.

```
API = iptgetapi(H)
```

The following lists the functions in the position rectangle API in the order they appear in the API structure.

Function	Description
setPosition	<p>Sets the position rectangle to a new position.</p> <pre>api.setPosition(new_position)</pre> <p>where <code>new_position</code> is a four-element position vector.</p>
getPosition	<p>Returns the current position of the position rectangle.</p> <pre>position = api.getPosition()</pre> <p><code>position</code> is a four-element position vector.</p>
delete	<p>Deletes the position rectangle associated with the API.</p> <pre>api.delete()</pre>
setColor	<p>Sets the color used to draw the position rectangle.</p> <pre>api.setColor(new_color)</pre> <p>where <code>new_color</code> can be a three-element vector specifying an RGB triplet, or a character vector specifying the long or short names of a predefined color, such as 'white' or 'w'. For a complete list of these predefined colors and their short names, see <code>ColorSpec</code>.</p>
addNewPositionCallback	<p>Adds the function handle <code>fun</code> to the list of new-position callback functions.</p> <pre>id = api.addNewPositionCallback(fun)</pre> <p>Whenever the position rectangle changes its position, each function in the list is called with the syntax:</p> <pre>fun(position)</pre> <p>The return value, <code>id</code>, is used only with <code>removeNewPositionCallback</code>.</p>

Function	Description
<code>removeNewPositionCallback</code>	<p>Removes the corresponding function from the new-position callback list.</p> <pre>api.removeNewPositionCallback(id)</pre> <p>where <code>id</code> is the identifier returned by <code>api.addNewPositionCallback</code></p>
<code>setDragConstraintCallback</code>	<p>Sets the drag constraint function to be the specified function handle, <code>fcn</code>.</p> <pre>api.setDragConstraintCallback(fcn)</pre> <p>Whenever the position rectangle is moved because of a mouse drag, the constraint function is called using the syntax:</p> <pre>constrained_position = fcn(new_position)</pre> <p>where <code>new_position</code> is a four-element position vector. This allows a client, for example, to control where the position rectangle may be dragged.</p>

Examples

Display in the command window the updated position of the position rectangle as it moves in the axes.

```
close all, plot(1:10)
h = impositionrect(gca, [4 4 2 2]);
api = iptgetapi(h);
api.addNewPositionCallback(@(p) disp(p));
```

Constrain the position rectangle to move only up and down.

```
close all, plot(1:10)
h = impositionrect(gca, [4 4 2 2]);
api = getappdata(h, 'API');
api.setDragConstraintCallback(@(p) [4 p(2:4)]);
```

Specify the color of the position rectangle.

```
close all, plot(1:10)
```

```
h = impositionrect(gca, [4 4 2 2]);  
api = iptgetapi(h, 'API');  
api.setColor([1 0 0]);
```

When the position rectangle occupies only a few pixels on the screen, the rectangle is drawn in a different style to increase its visibility.

```
close all, imshow cameraman.tif  
h = impositionrect(gca, [100 100 10 10]);
```

Tips

A position rectangle can be dragged interactively using the mouse. When the position rectangle occupies a small number of screen pixels, its appearance changes to aid visibility.

The position rectangle has a context menu associated with it that you can use to copy the current position to the clipboard and change the color used to display the rectangle.

See Also

`iptgetapi`

Introduced before R2006a

improfile

Pixel-value cross-sections along line segments

Syntax

```
improfile
improfile(n)
improfile(I,xi,yi)
improfile(I,xi,yi,n)
c = improfile( ___ )
[ cx,cy,c ] = improfile(I,xi,yi,n)
[ cx,cy,c,xi,yi ] = improfile(I,xi,yi,n)
[ ___ ] = improfile(x,y,I,xi,yi)
[ ___ ] = improfile(x,y,I,xi,yi,n)
[ ___ ] = improfile( ___,method)
```

Description

`improfile` retrieves the intensity values of pixels along a line or a multiline path in the grayscale, binary, or RGB image in the current axes and displays a plot of the intensity values. If the specified path consists of a single line segment, `improfile` creates a two-dimensional plot of intensity values versus the distance along the line segment. If the path consists of two or more line segments, `improfile` creates a three-dimensional plot of the intensity values versus their *x*- and *y*-coordinates.

With this syntax, you specify the line or path using the mouse, by clicking points in the image. Press **Backspace** or **Delete** to remove the previously selected point. To finish selecting points, adding a final point, press shift-click, right-click, or double-click. To finish selecting points without adding a final point, press **Return**.

`improfile(n)` retrieves the intensity values, where *n* specifies the number of points to include. If you do not provide this argument, `improfile` chooses a value for *n*, roughly equal to the number of pixels the path traverses.

`improfile(I,xi,yi)` retrieves pixel intensity values, where *I* specifies an image, and *xi* and *yi* are equal-length vectors specifying the spatial coordinates of the endpoints of the line segments.

`improfile(I,xi,yi,n)` returns pixel intensity values, where `n` specifies the number of points to include.

`c = improfile(____)` returns the intensity values in `c`, an `n`-by-1 vector, if the input is a grayscale intensity image, or an `n`-by-1-by-3 array if the input is an RGB image.

`[cx,cy,c] = improfile(I,xi,yi,n)` additionally returns the spatial coordinates of the pixels, `cx` and `cy`, of length `n`.

`[cx,cy,c,xi,yi] = improfile(I,xi,yi,n)` additionally returns two equal-length vectors specifying the spatial coordinates of the endpoints of the line segments. `xi` and `yi`.

`[____] = improfile(x,y,I,xi,yi)` enables the definition of a nondefault spatial coordinate system by specifying two, 2-element vector, `x` and `y`, containing the image XData and YData.

`[____] = improfile(x,y,I,xi,yi,n)` defines a nondefault spatial coordinate system and specifies the number of points to include, `n`.

`[____] = improfile(____,method)` specifies the interpolation method:

- 'nearest' — Nearest-neighbor interpolation (the default)
- 'bilinear' — Bilinear interpolation
- 'bicubic' — Bicubic interpolation

Class Support

The input image can be `uint8`, `uint16`, `int16`, `single`, `double`, or `logical`. All other inputs and outputs must be `double`.

Examples

Plot Multisegment Line from Image

Read image.

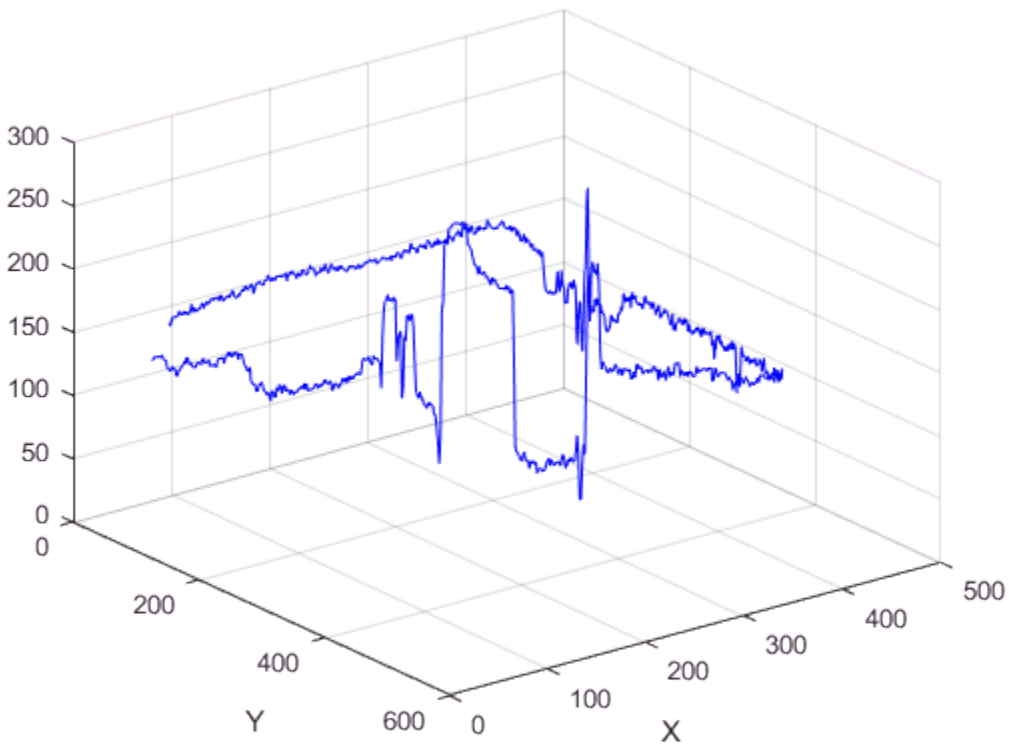
```
I = imread('liftingbody.png');
```

Specify x - and y -coordinates that define line segments.

```
x = [19 427 416 77];  
y = [96 462 37 33];
```

Display 3-D plot of the pixel values of these line segments.

```
improfile(I,x,y),grid on;
```



See Also

`impixel` | `interp2`

Introduced before R2006a

imputfile

Display Save Image dialog box

Syntax

```
[filename,ext,user_canceled] = imputfile
```

Description

[filename,ext,user_canceled] = imputfile displays the **Save Image** dialog box which you can use to specify the full path and format of a file. Using the dialog box, you can navigate to folders in a file system and select a particular file or specify the name of a new file. imputfile limits the types of files displayed in the dialog box to the image file format selected in the Files of Type menu.

When you click **Save**, imputfile returns the full path to the file in filename and the file extension associated with the file format selected from the **Files of Type** menu in ext. imputfile automatically adds the file name extension (such as .jpg) to the file name.

If the user clicks **Cancel** or closes the **Save Image** dialog box, imputfile closes and returns control to MATLAB, setting user_canceled to True (1), and setting filename and ext to empty character vectors (' '); otherwise, user_canceled is False (0).

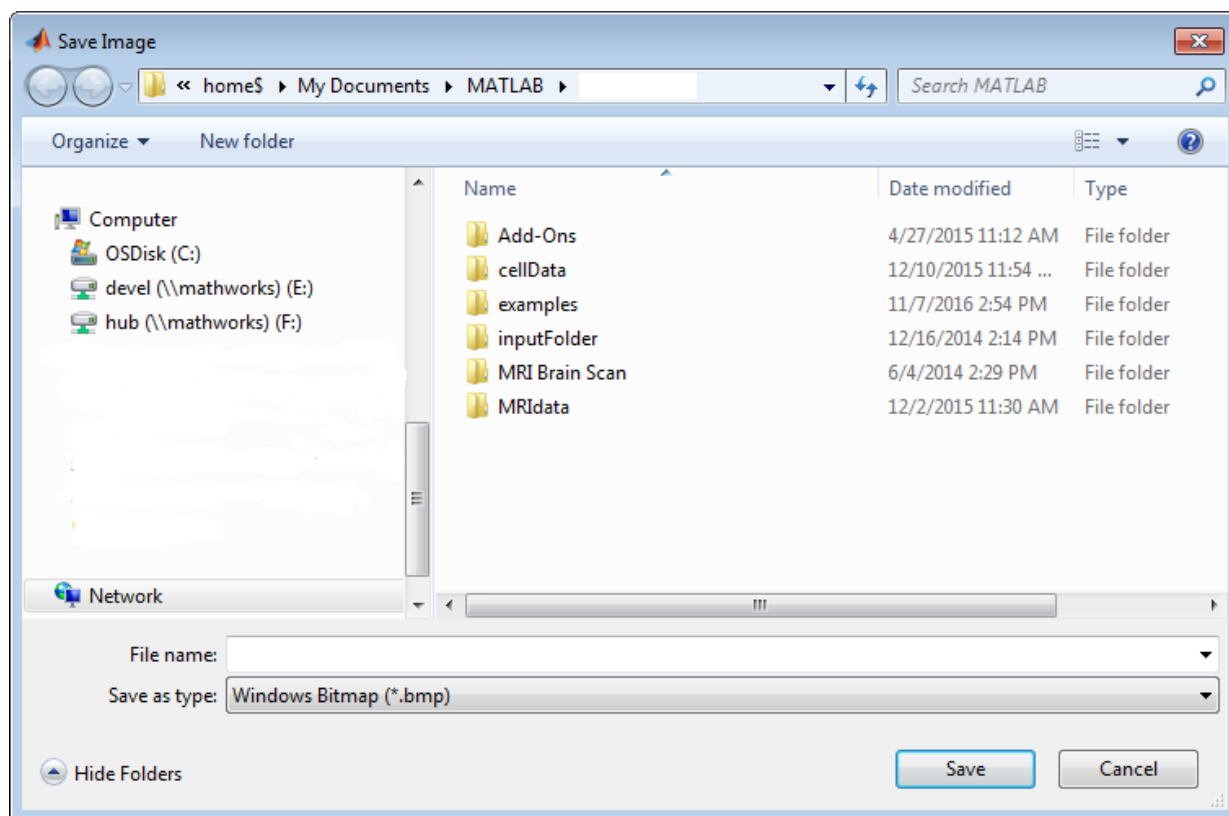
Note: The **Save Image** dialog box is modal; it blocks the MATLAB command line until you click **Save** or cancel the operation.

Examples

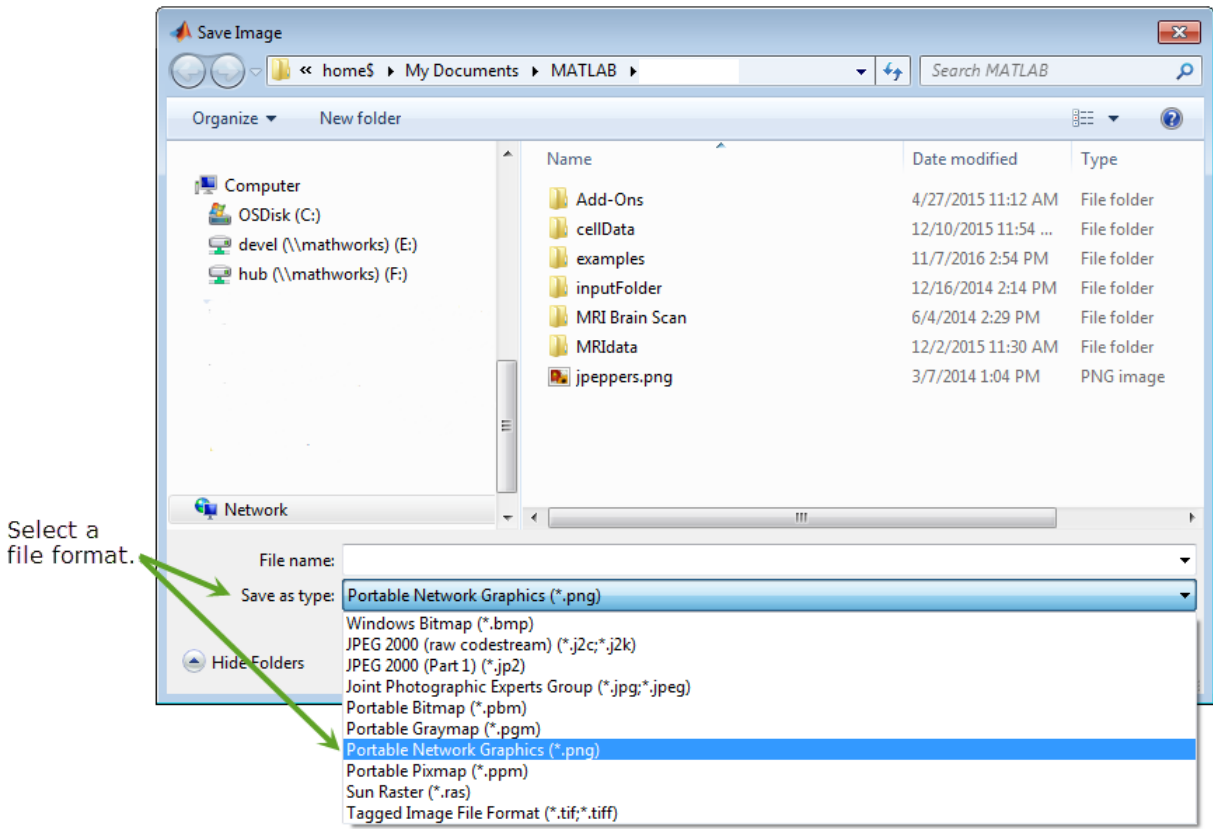
Get User-Specified File Name

Open the **Save Image** dialog box. This dialog box is modal—control in the command window is suspended until you respond to the **Save Image** dialog box.

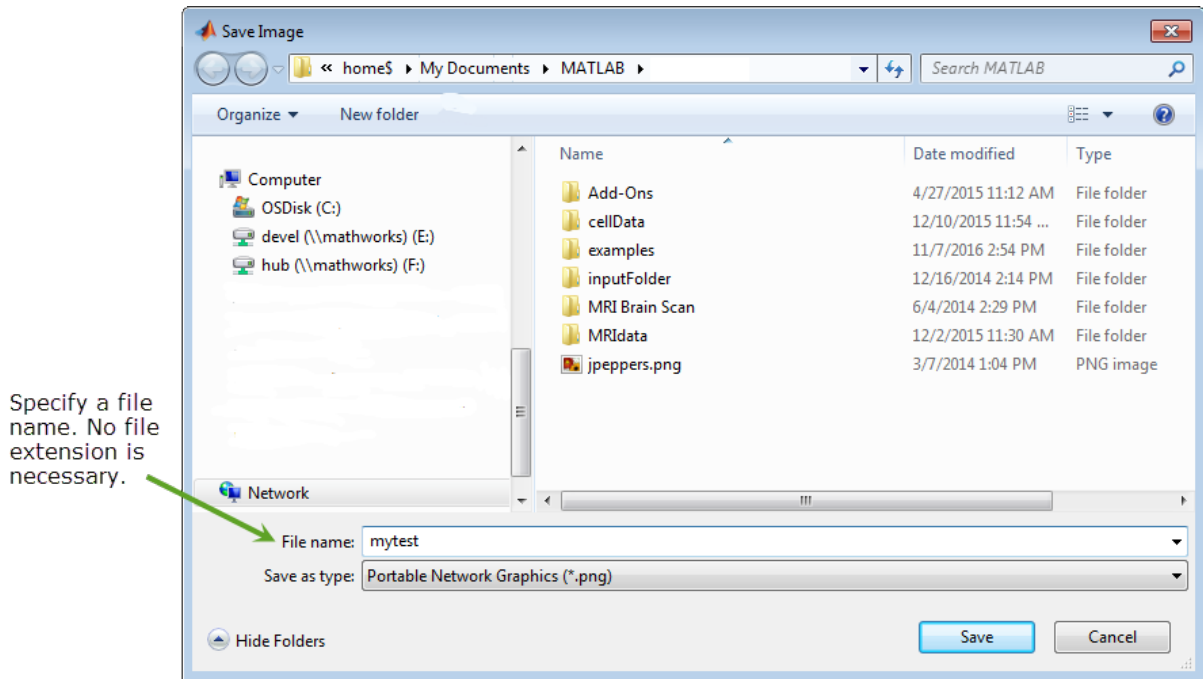

```
[fn, ext, ucancel] = imputfile
```



To view only images in Portable Network Graphics format, select the format from the **Save as type** menu.



Specify a new filename and click **Save**. `imwrite` returns the full path of the filename you specified, the file extension, and the Boolean value `false`, meaning that you didn't click **Cancel**. Note that `imwrite` automatically adds the file extension of the format you selected to the file name.



```
fn =
    1×37 char array
    '\\home$\Documents\MATLAB\mytest.png'

ext =
    1×3 char array
    'png'

ucancel =
    logical
```

0

Output Arguments

filename — Name of file selected

character array

Name of file selected, returned as a character array.

ext — File extension of a supported file format

character array

File extension of a supported file format, returned as a character array.

user_canceled — Flag indicating if user chose to cancel dialog

logical

Flag indicating if user chose to cancel dialog, returned as a Boolean logical value `true` or `false`.

See Also

See Also

`imformats` | `imgetfile` | `imsave` | `imtool`

Introduced in R2007b

impyramid

Image pyramid reduction and expansion

Syntax

```
B = impyramid(A,direction)
```

Description

`B = impyramid(A,direction)` computes a Gaussian pyramid reduction or expansion of `A` by one level. `direction` determines whether `impyramid` performs a reduction or an expansion.

Examples

Compute Four-level Multiresolution Pyramid of Image

Read image into the workspace.

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

Perform a series of reductions. The first call reduces the original image. The other calls to `impyramid` use the previously reduced image.

```
I1 = impyramid(I, 'reduce');  
I2 = impyramid(I1, 'reduce');  
I3 = impyramid(I2, 'reduce');
```

View the original image and the reduced versions.

```
figure, imshow(I)
```



`figure, imshow(I1)`



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



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



Input Arguments

A — Image to be reduced or expanded

numeric or logical array

Image to reduced or expanded, specified as a numeric or logical array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

direction — Reduction or expansion

'reduce' | 'expand'

Reduction or expansion, specified as one of the following values:

Value	Description
'reduce'	Return an image, smaller than the original image.
'expand'	Return an image that is larger than the original image.

Data Types: char

Output Arguments

B — Reduced or expanded image

numeric or logical array

Reduced or expanded image, returned as a numeric or logical array, the same class as A.

Algorithms

If A is m -by- n and direction is 'reduce', the size of B is $\text{ceil}(M/2)$ -by- $\text{ceil}(N/2)$. If direction is 'expand', the size of B is $(2*M-1)$ -by- $(2*N-1)$.

Reduction and expansion take place only in the first two dimensions. For example, if A is 100-by-100-by-3 and direction is 'reduce', then B is 50-by-50-by-3.

impyramid uses the kernel specified on page 533 of the Burt and Adelson paper:

$w = \left[\frac{1}{4} - \frac{a}{2}, \frac{1}{4}, a, \frac{1}{4}, \frac{1}{4} - \frac{a}{2} \right]$, where $a = 0.375$. The parameter a is set to 0.375 so that the equivalent weighting function is close to a Gaussian shape. In addition, the weights can be readily applied using fixed-point arithmetic.

References

- [1] Burt and Adelson, "The Laplacian Pyramid as a Compact Image Code," *IEEE Transactions on Communications*, Vol. COM-31, no. 4, April 1983, pp. 532-540.
- [2] Burt, "Fast Filter Transforms for Image Processing," *Computer Graphics and Image Processing*, Vol. 16, 1981, pp. 20-51

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- `direction` must be a compile-time constant.

See Also

See Also

`imresize`

Introduced in R2007b

imquantize

Quantize image using specified quantization levels and output values

Syntax

```
quant_A = imquantize(A,levels)
quant_A = imquantize( ____,values)
[quant_A,index] = imquantize( ____ )
```

Description

`quant_A = imquantize(A,levels)` quantizes image `A` using specified quantization values contained in the `N` element vector `levels`. Output image `quant_A` is the same size as `A` and contains `N + 1` discrete integer values in the range 1 to `N + 1` which are determined by the following criteria:

- If $A(k) \leq levels(1)$, then $quant_A(k) = 1$.
- If $levels(m-1) < A(k) \leq levels(m)$, then $quant_A(k) = m$.
- If $A(k) > levels(N)$, then $quant_A(k) = N + 1$.

Note that `imquantize` assigns values to the two implicitly defined end intervals:

- $A(k) \leq levels(1)$
- $A(k) > levels(N)$

`quant_A = imquantize(____,values)` adds the `N + 1` element vector `values` where `N = length(levels)`. Each of the `N + 1` elements of `values` specify the quantization value for one of the `N + 1` discrete pixel values in `quant_A`.

- If $A(k) \leq levels(1)$, then $quant_A(k) = values(1)$.
- If $levels(m-1) < A(k) \leq levels(m)$, then $quant_A(k) = values(m)$.
- If $A(k) > levels(N)$, then $quant_A(k) = values(N + 1)$.

`[quant_A,index] = imquantize(____)` returns an array `index` such that:
`quant_A = values(index)`

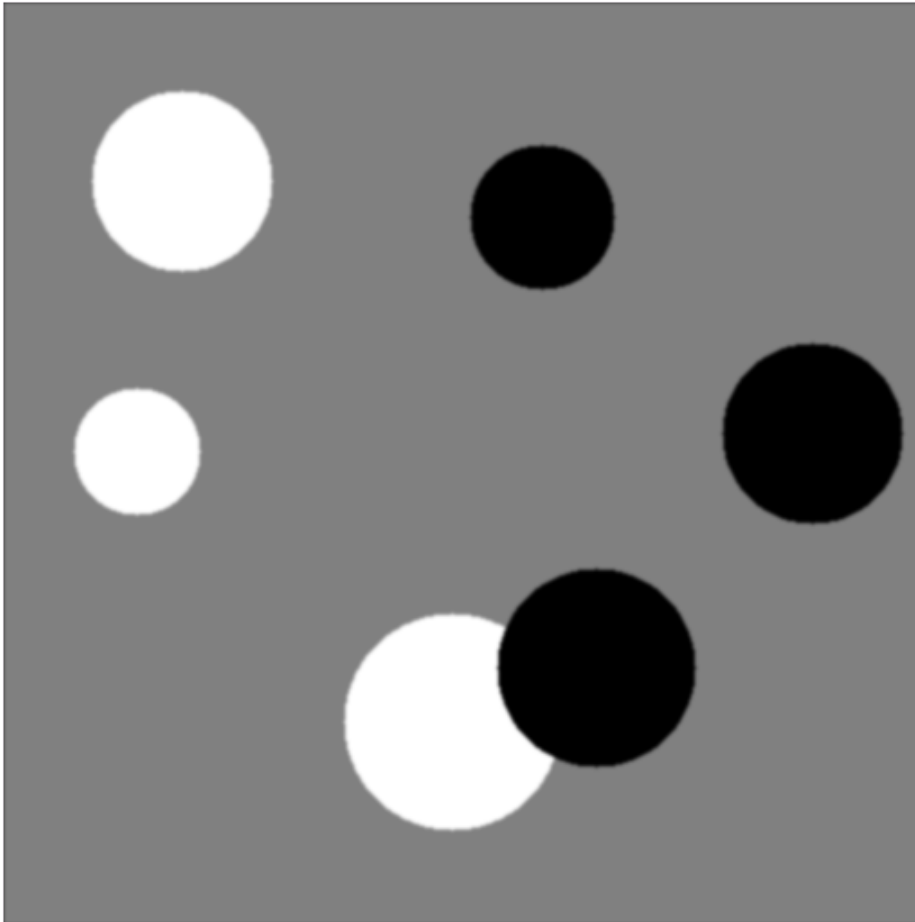
Examples

Segment Image into Three Levels Using Two Thresholds

Read image and display it.

```
I = imread('circlesBrightDark.png');  
imshow(I)  
axis off  
title('Original Image')
```

Original Image



Calculate two threshold levels.

```
thresh = multithresh(I,2);
```

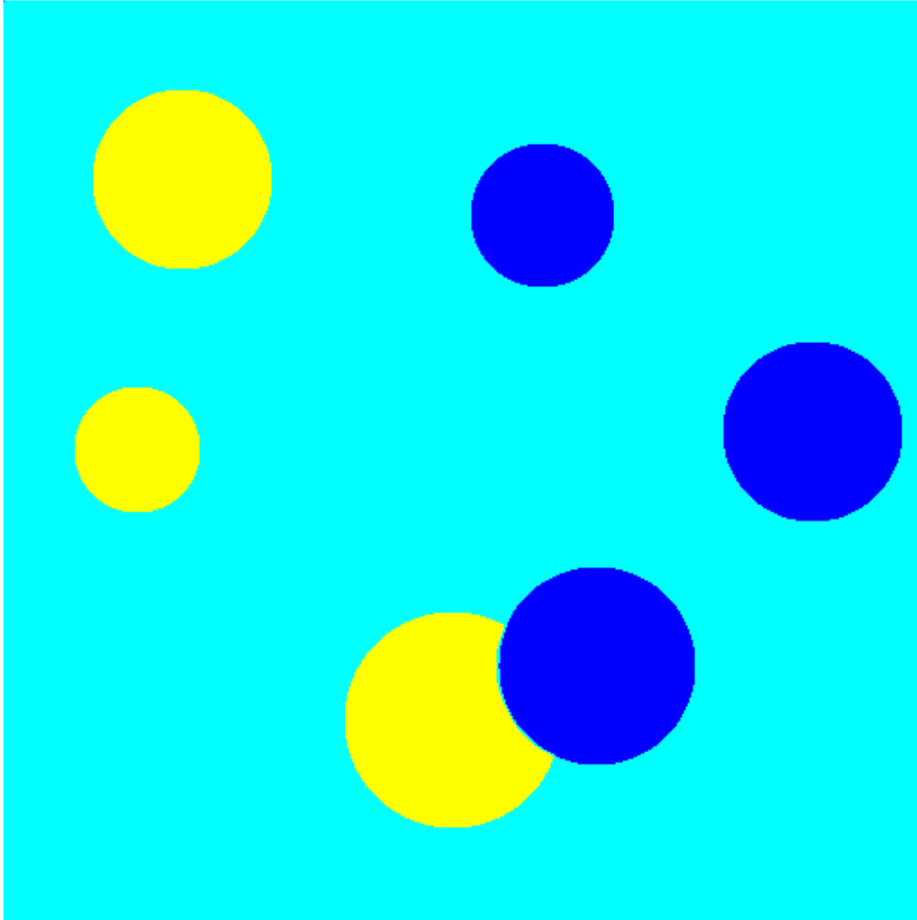
Segment the image into three levels using `imquantize` .

```
seg_I = imquantize(I,thresh);
```

Convert segmented image into color image using `label2rgb` and display it.

```
RGB = label2rgb(seg_I);  
figure;  
imshow(RGB)  
axis off  
title('RGB Segmented Image')
```

RGB Segmented Image



Compare Thresholding Entire Image Versus Plane-by-Plane Thresholding

Read truecolor (RGB) image and display it.

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

```
imshow(I)
axis off
title('RGB Image');
```

RGB Image



Generate thresholds for seven levels from the entire RGB image.

```
threshRGB = multithresh(I,7);
```

Generate thresholds for each plane of the RGB image.

```
threshForPlanes = zeros(3,7);
```

```
for i = 1:3
```

```

    threshForPlanes(i,:) = multithresh(I(:,:,i),7);
end

```

Process the entire image with the set of threshold values computed from entire image.

```

value = [0 threshRGB(2:end) 255];
quantRGB = imquantize(I, threshRGB, value);

```

Process each RGB plane separately using the threshold vector computed from the given plane. Quantize each RGB plane using threshold vector generated for that plane.

```

quantPlane = zeros( size(I) );

for i = 1:3
    value = [0 threshForPlanes(i,2:end) 255];
    quantPlane(:,:,i) = imquantize(I(:,:,i),threshForPlanes(i,:),value);
end

```

```

quantPlane = uint8(quantPlane);

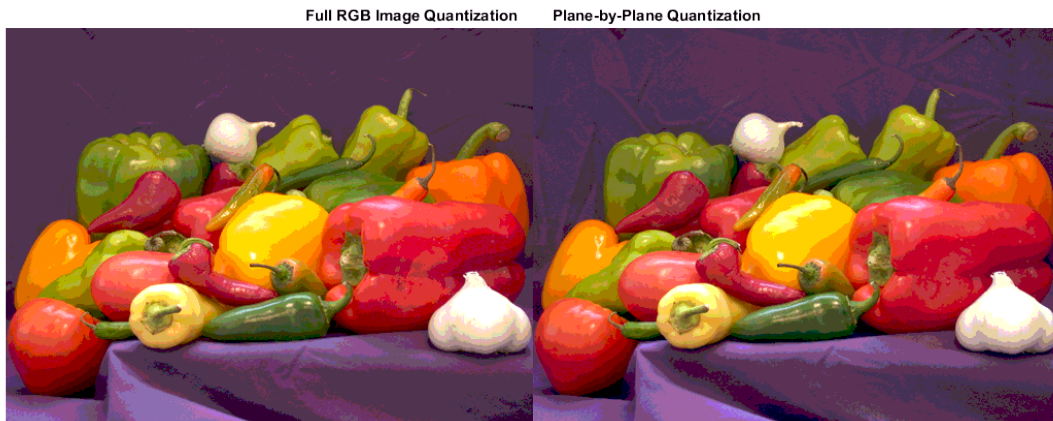
```

Display both posterized images and note the visual differences in the two thresholding schemes.

```

imshowpair(quantRGB,quantPlane,'montage')
axis off
title('Full RGB Image Quantization          Plane-by-Plane Quantization')

```



To compare the results, calculate the number of unique RGB pixel vectors in each output image. Note that the plane-by-plane thresholding scheme yields about 23% more colors than the full RGB image scheme.

```
dim = size( quantRGB );
quantRGBmx3 = reshape(quantRGB, prod(dim(1:2)), 3);
quantPlanemx3 = reshape(quantPlane, prod(dim(1:2)), 3);

colorsRGB = unique(quantRGBmx3, 'rows' );
colorsPlane = unique(quantPlanemx3, 'rows' );

disp(['Unique colors in RGB image          : ' int2str(length(colorsRGB))]);
Unique colors in RGB image          : 188

disp(['Unique colors in Plane-by-Plane image : ' int2str(length(colorsPlane))]);
Unique colors in Plane-by-Plane image : 231
```

Threshold grayscale image from 256 to 8 levels

Reduce the number of discrete levels in an image from 256 to 8. This example uses two different methods for assigning values to each of the eight output levels.

Read image and display it.

```
I = imread('coins.png');
imshow(I)
axis off
title('Grayscale Image')
```

Grayscale Image

Split the image into eight levels by obtaining seven thresholds from `multithresh`.

```
thresh = multithresh(I,7);
```

Construct the `valuesMax` vector such that the maximum value in each quantization interval is assigned to the eight levels of the output image.

```
valuesMax = [thresh max(I(:))]
```

```
valuesMax = 1×8 uint8 row vector
```

```
    65    88   119   149   169   189   215   255
```

```
[quant8_I_max, index] = imquantize(I,thresh,valuesMax);
```

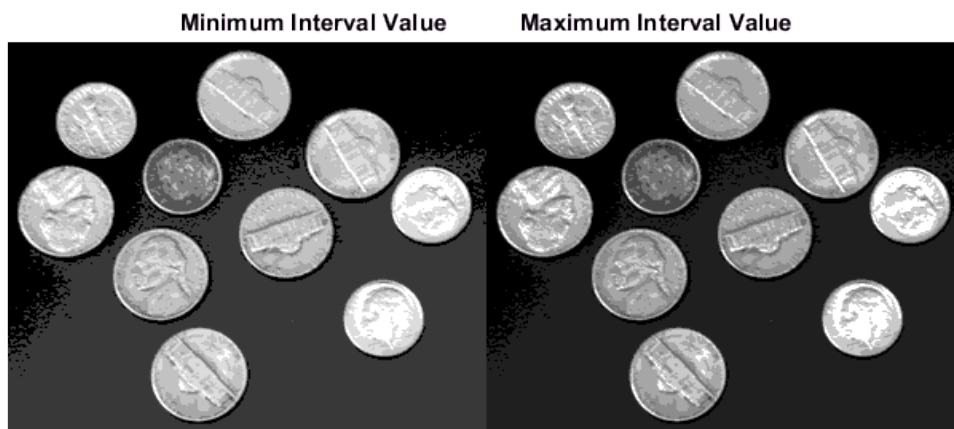
Similarly, construct the `valuesMin` vector such that the minimum value in each quantization interval is assigned to the eight levels of the output image. Instead of calling `imquantize` again with the vector `valuesMin`, use the output argument `index` to assign those values to the output image.

```
valuesMin = [min(I(:)) thresh]
valuesMin = 1×8 uint8 row vector
    23    65    88   119   149   169   189   215
```

```
quant8_I_min = valuesMin(index);
```

Display both eight-level output images side by side.

```
imshowpair(quant8_I_min,quant8_I_max,'montage')
title('Minimum Interval Value          Maximum Interval Value')
```



Input Arguments

A — Input image

image

Input image, specified as a numeric array of any dimension.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

levels — Quantization levels

vector

Quantization levels, specified as an N element vector. Values of the discrete quantization levels must be in monotonically increasing order.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

values — Quantization values

vector

Quantization values, specified as an $N + 1$ element vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

quant_A — Quantized output image

image

Quantized output image, returned as a numeric array the same size as A . If input argument *values* is specified, then *quant_A* is the same data type as *values*. If *values* is not specified, then *quant_A* is of class `double`.

index — Mapping array

array

Mapping array, returned as an array the same size as input image A . It contains integer indices which access *values* to construct the output image: $\text{quant_A} = \text{values}(\text{index})$. If input argument *values* is not defined, then $\text{index} = \text{quant_A}$.

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.

See Also

See Also

[label2rgb](#) | [multithresh](#) | [rgb2ind](#)

Introduced in R2012b

imreconstruct

Morphological reconstruction

Syntax

```
IM = imreconstruct(marker,mask)
IM = imreconstruct(marker,mask,conn)
IM = imreconstruct(gpuarrayMarker,gpuarrayMask)
```

Description

`IM = imreconstruct(marker,mask)` performs morphological reconstruction of the image `marker` under the image `mask`. `marker` and `mask` can be two intensity images or two binary images with the same size. The returned image `IM` is an intensity image or a binary image, depending on the input images, and is the same size as the input images.

`marker` must be the same size as `mask`, and its elements must be less than or equal to the corresponding elements of `mask`. If the values in `marker` are greater than corresponding elements in `mask`, `imreconstruct` clips the values to the `mask` level before starting the procedure.

By default, `imreconstruct` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imreconstruct` uses `conndef(ndims(I), 'maximal')`.

`IM = imreconstruct(marker,mask,conn)` performs morphological reconstruction with the specified connectivity.

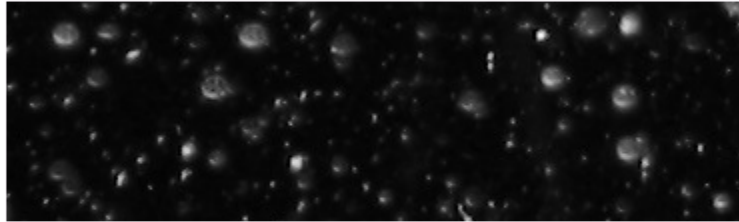
`IM = imreconstruct(gpuarrayMarker,gpuarrayMask)` performs morphological reconstruction on a GPU. The input `marker` image and `mask` image must be `gpuArrays`. This syntax requires the Parallel Computing Toolbox.

Examples

Perform Opening-by-Reconstruction to Identify High Intensity Objects

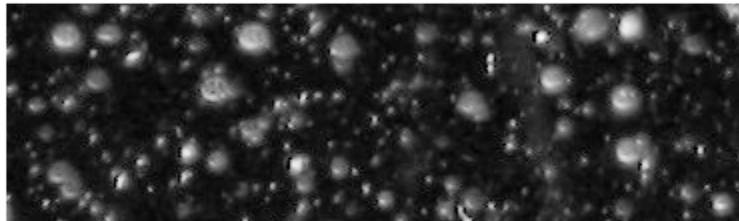
Read a grayscale image and display it.

```
I = imread('snowflakes.png');  
figure  
imshow(I)
```



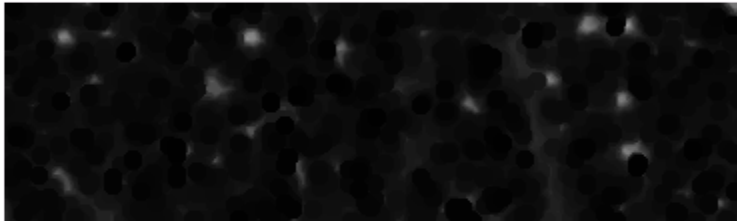
Adjust the contrast of the image to create the mask image and display results.

```
mask = adapthisteq(I);  
figure  
imshow(mask)
```



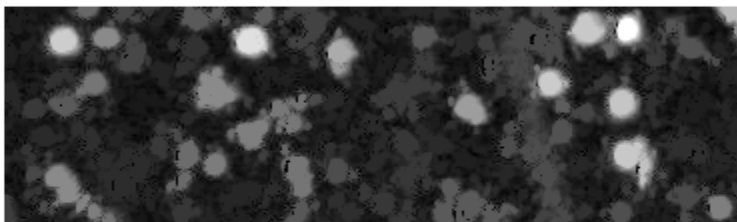
Create a marker image that identifies high-intensity objects in the image using morphological erosion and display results.

```
se = strel('disk',5);  
marker = imerode(mask,se);  
imshow(marker)
```



Perform morphological opening on the mask image, using the marker image to identify high-intensity objects in the mask. Display results.

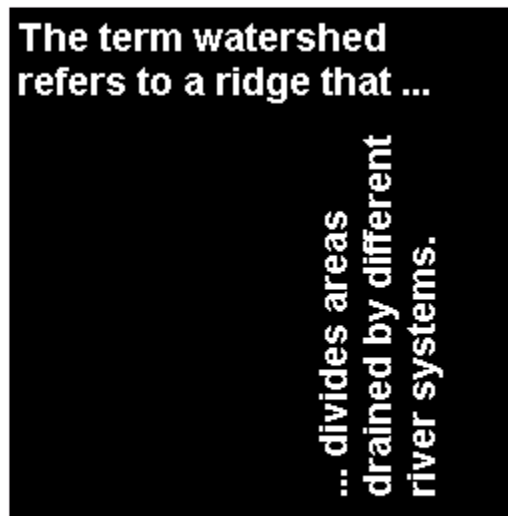
```
obr = imreconstruct(marker,mask);  
figure  
imshow(obr,[])
```



Use Reconstruction to Segment an Image

Read a logical image into workspace and display it. This is the mask image.

```
mask = imread('text.png');  
figure  
imshow(mask)
```



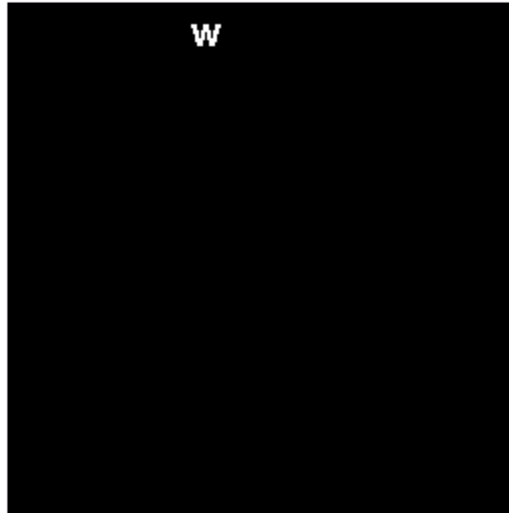
Create a marker image that identifies the object in the image you want to extract through segmentation. For this example, identify the "w" in the word "watershed".

```
marker = false(size(mask));  
marker(13,94) = true;
```

Perform segmentation of the mask image using the marker image.

```
im = imreconstruct(marker,mask);  
figure
```

```
imshow(im)
```



Use Reconstruction to Segment an Image on a GPU

Read mask image and create gpuArray.

```
mask = gpuArray(imread('text.png'));  
figure, imshow(mask),
```

Create marker image gpuArray.

```
marker = gpuArray.false(size(mask));  
marker(13,94) = true;
```

Perform the segmentation and display the result.

```
im = imreconstruct(marker,mask);
```

```
figure, imshow(im)
```

Input Arguments

marker — Input image

nonsparse numeric or logical array

Input image, specified as a nonsparse numeric or logical array.

Example: `se = strel('disk',5); marker = imerode(mask,se);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

mask — Mask image

nonsparse numeric or logical array

Mask image, specified as a nonsparse numeric or logical array.

Example: `mask = imread('text.png');`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

conn — Connectivity

8 (for 2-D images) (default) | 4 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix

Connectivity, specified as one of the values in this table.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can also be defined in a more general way for any dimension by using for `conn` a 3-by-3-by- ... -by-3 matrix of 0s and 1s. The 1-valued elements define neighborhood locations relative to the center element of `conn`. Note that `conn` must be symmetric about its center element.

Example: `obr = imreconstruct(marker,mask,4);`

Data Types: `double` | `logical`

gpuarrayMarker — Input image on a GPU

`gpuArray`

Input image on a GPU, specified as a `gpuArray`.

Example: `marker = gpuArray(imread('text.png'));`

gpuarrayMask — Mask image on a GPU

`gpuArray`

Mask image on a GPU, specified as a `gpuArray`.

Example: `mask = gpuArray(imread('text.png'));`

Output Arguments

IM — Reconstructed image

numeric or logical array

Reconstructed image, returned as a numeric or logical array, depending on the input image, that is the same size as the input image.

Tips

- Morphological reconstruction is the algorithmic basis for several other Image Processing Toolbox functions, including `imclearborder`, `imextendedmax`, `imextendedmin`, `imfill`, `imhmax`, `imhmin`, and `imimposemin`.
- **Performance note:** This function may take advantage of hardware optimization for data types `logical`, `uint8`, `uint16`, `single`, and `double` to run faster. Hardware optimization requires `marker` and `mask` to be 2-D images and `conn` to be either 4 or 8.

Algorithms

imreconstruct uses the fast hybrid grayscale reconstruction algorithm described in [1].

References

- [1] Vincent, L., "Morphological Grayscale Reconstruction in Image Analysis: Applications and Efficient Algorithms," *IEEE Transactions on Image Processing*, Vol. 2, No. 2, April, 1993, pp. 176-201.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the optional third input argument, **conn**, must be a compile-time constant, and can only take the value **4** or **8**.

See Also

See Also

imclearborder | imextendedmax | imextendedmin | imfill | imhmax | imhmin | imimposemin

Topics

“Understanding Morphological Reconstruction”

Introduced before R2006a

imrect

Create draggable rectangle

Syntax

```
h = imrect
h = imrect(hparent)
h = imrect(hparent, position)
h = imrect(..., param1, val1,...)
```

Description

`h = imrect` begins interactive placement of a rectangle on the current axes. The function returns `h`, a handle to an `imrect` object. The rectangle has a context menu associated with it that controls aspects of its appearance and behavior—see “Interactive Behavior” on page 1-1042. Right-click on the rectangle to access this context menu.

`h = imrect(hparent)` begins interactive placement of a rectangle on the object specified by `hparent`. `hparent` specifies the HG parent of the rectangle graphics, which is typically an axes but can also be any other object that can be the parent of an `hggroup`.

`h = imrect(hparent, position)` creates a draggable rectangle on the object specified by `hparent`. `position` is a four-element vector that specifies the initial size and location of the rectangle. `position` has the form `[xmin ymin width height]`.

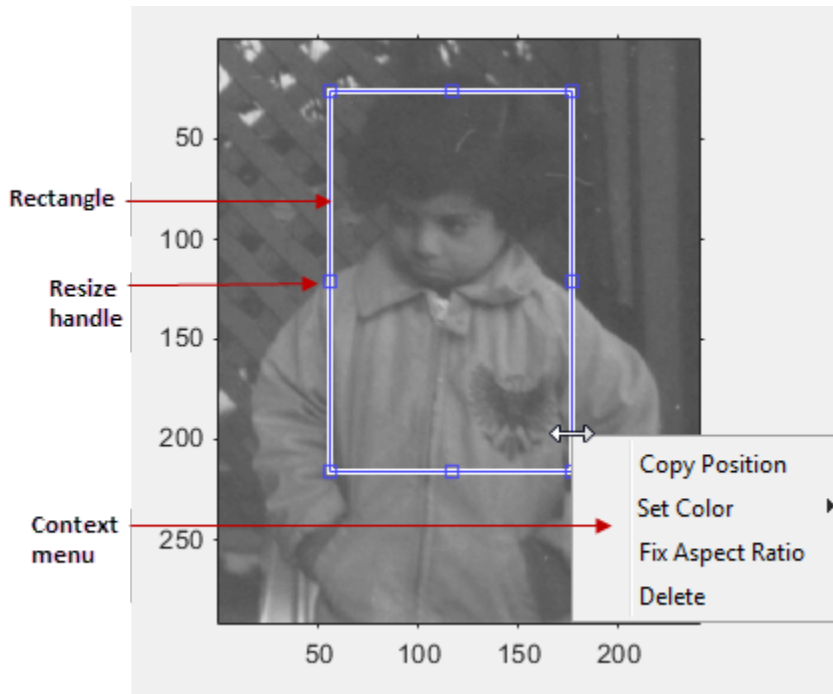
`h = imrect(..., param1, val1,...)` creates a draggable rectangle, specifying parameters and corresponding values that control the behavior of the rectangle. The following table lists the parameter available. Parameter names can be abbreviated, and case does not matter.

Parameter	Description
'PositionConstraintFcn'	Function handle <code>fcn</code> that is called whenever the mouse is dragged. You can use this function to control where the rectangle can be dragged. See the help for the <code>setPositionConstraintFcn</code> method for information about valid function handles.


Interactive Behavior


When you call `imrect` with an interactive syntax, the pointer changes to a cross hairs

⊕ when over the image. You can create the rectangle and adjust its size and position using the mouse. The rectangle also supports a context menu that you can use to control aspects of its appearance and behavior. The following figure shows the rectangle.



The following table lists the interactive behaviors supported by `imrect`.

Interactive Behavior	Description
Moving the rectangle.	Move the pointer inside the rectangle. The pointer changes to a fleur shape  . Click and drag the mouse to move the rectangle.

Interactive Behavior	Description
Resizing the rectangle.	Move the pointer over any of the edges or corners of the rectangle, the shape changes to a double-ended arrow,  . Click and drag the edge or corner using the mouse.
Changing the color of the rectangle.	Move the pointer inside the rectangle. Right-click and select Set Color from the context menu.
Retrieving the coordinates of the current position	Move the pointer inside the polygon. Right-click and select Copy Position from the context menu. <code>imrect</code> copies a four-element position vector to the clipboard.
Preserve the current aspect ratio of the rectangle during interactive resizing.	Move the pointer inside the rectangle. Right-click and select Fix Aspect Ratio from the context menu.
Deleting the rectangle	Move the pointer inside the rectangle or on an edge of the rectangle. Right-click and select Delete from the context menu. To remove this option from the context menu, set the <code>Deletable</code> property to false: <code>h = imrect(); h.Deletable = false;</code>

Methods

Each `imrect` object supports a number of methods, listed below. Type methods `imrect` to see a list of the methods.

addNewPositionCallback — Add new-position callback to ROI object

See `imroi` for information.

createMask — Create mask within image

See `imroi` for information.

delete — Delete ROI object

See `imroi` for information.

getColor — Get color used to draw ROI object

See `imroi` for information.

getPosition — Return current position of rectangle

`pos = getPosition(h)` returns the current position of the rectangle `h`. The returned position, `pos`, is a 1-by-4 array [`xmin` `ymin` `width` `height`].

getPositionConstraintFcn — Return function handle to current position constraint function

See `imroi` for information.

removeNewPositionCallback — Remove new-position callback from ROI object.

See `imroi` for information.

resume — Resume execution of MATLAB command line

See `imroi` for information.

setColor — Set color used to draw ROI object

See `imroi` for information.

setConstrainedPosition — Set ROI object to new position

See `imroi` for information.

setFixedAspectRatioMode — Control whether aspect ratio preserved during resize

`setFixedAspectRatioMode(h,TF)` sets the interactive resize behavior of the rectangle `h`. `TF` is a logical scalar. `True` means that the current aspect ratio is preserved during interactive resizing. `False` means that interactive resizing is not constrained.

setPosition — Set rectangle to new position

`setPosition(h,pos)` sets the rectangle `h` to a new position. The new position, `pos`, has the form [`xmin` `ymin` `width` `height`].

setPositionConstraintFcn — Set position constraint function of ROI object

See `imroi` for information.

setResizable — Set resize behavior of rectangle

`setResizable(h,TF)` sets whether the rectangle `h` may be resized interactively. `TF` is a logical scalar.

wait — Block MATLAB command line until ROI creation is finished

See `imroi` for information.

Examples

Example 1

Display updated position in the title. Specify a position constraint function using `makeConstrainToRectFcn` to keep the rectangle inside the original `Xlim` and `Ylim` ranges.

```
figure, imshow('cameraman.tif');  
h = imrect(gca, [10 10 100 100]);  
addNewPositionCallback(h,@(p) title(mat2str(p,3)));  
fcn = makeConstrainToRectFcn('imrect',get(gca,'XLim'),get(gca,'YLim'));  
setPositionConstraintFcn(h,fcn);
```

Now drag the rectangle using the mouse.

Example 2

Interactively place a rectangle by clicking and dragging. Use `wait` to block the MATLAB command line. Double-click on the rectangle to resume execution of the MATLAB command line.

```
figure, imshow('pout.tif');  
h = imrect;  
position = wait(h);
```

Tips

If you use `imrect` with an axes that contains an image object, and do not specify a position constraint function, users can drag the rectangle outside the extent of the image. When used with an axes created by the `plot` function, the axes limits automatically expand to accommodate the movement of the rectangle.

When the API function `setResizable` is used to make the rectangle non-resizable, the **Fix Aspect Ratio** context menu item is not provided.

See Also

`imellipse` | `imline` | `imfreehand` | `impoint` | `impoly` | `imroi` |
`makeConstrainToRectFcn`

Introduced before R2006a

imref2d class

Reference 2-D image to world coordinates

Description

An `imref2d` object encapsulates the relationship between the intrinsic coordinates anchored to the rows and columns of a 2-D image and the spatial location of the same row and column locations in a world coordinate system. The image is sampled regularly in the planar world- x and world- y coordinate system such that intrinsic- x values align with world- x values, and intrinsic- y values align with world- y values. The pixel spacing from row to row need not equal the pixel spacing from column to column.

The intrinsic coordinate values (x,y) of the center point of any pixel are identical to the values of the column and row subscripts for that pixel. For example, the center point of the pixel in row 5, column 3 has intrinsic coordinates $x = 3.0$, $y = 5.0$. Be aware, however, that the order of coordinate specification (3.0,5.0) is reversed in intrinsic coordinates relative to pixel subscripts (5,3). Intrinsic coordinates are defined on a continuous plane while the subscript locations are discrete locations with integer values.

Construction

`R = imref2d()` creates an `imref2d` object with default property settings.

`R = imref2d(imageSize)` creates an `imref2d` object given an image size. This syntax constructs a spatial referencing object for the default case in which the world coordinate system is co-aligned with the intrinsic coordinate system.

`R = imref2d(imageSize,pixelExtentInWorldX,pixelExtentInWorldY)` creates an `imref2d` object given an image size and the resolution in each dimension, specified by `pixelExtentInWorldX` and `pixelExtentInWorldY`.

`R = imref2d(imageSize,xWorldLimits,yWorldLimits)` creates an `imref2d` object given an image size and the world coordinate limits in each dimension, specified by `xWorldLimits` and `yWorldLimits`.

Input Arguments

imageSize — Size of the image associated with the object

two-element positive row vector

Size of the image associated with the object, specified as a two-element positive row vector. You can use the vector returned by the `size` function.

Example: `imageSize = [128 128];`

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `int8` | `int16` | `int32`

pixelExtentInWorldX — Size of a single pixel in the x-dimension measured in the world coordinate system

positive scalar

Size of a single pixel in the x-dimension measured in the world coordinate system, specified as a positive scalar.

Data Types: `double` | `single`

pixelExtentInWorldY — Size of a single pixel in the y-dimension measured in the world coordinate system

positive scalar

Size of a single pixel in the y-dimension measured in the world coordinate system, specified as a positive scalar.

Data Types: `double` | `single`

xWorldLimits — Minimum and maximum coordinate values in the x-dimension in the world coordinate system

two-element numeric row vector

Minimum and maximum coordinate values in the x-dimension in the world coordinate system, specified as a two-element numeric row vector.

Example: `xWorldLimits = [0.5 256.5];`

Data Types: `double` | `single`

yWorldLimits — Minimum and maximum coordinate values in the y-dimension in the world coordinate system

two-element numeric row vector

Minimum and maximum coordinate values in the y -dimension in the world coordinate system, specified as a two-element numeric row vector.

Example: `yWorldLimits = [0.5 256.5];`

Data Types: `double` | `single`

Properties

ImageExtentInWorldX — Span of image in the x -dimension in the world coordinate system

numeric scalar

Span of image in the x -dimension in the world coordinate system, specified as a numeric scalar. The `imref2d` object calculates this value as `PixelExtentInX * ImageSize(2)`.

Data Types: `double`

ImageExtentInWorldY — Span of image in the y -dimension in the world coordinate system

numeric scalar

Span of image in the y -dimension in the world coordinate system, specified as a numeric scalar. The `imref2d` object calculates this value as `PixelExtentInY * ImageSize(1)`.

Data Types: `double`

ImageSize — Number of elements in each spatial dimension

two-element positive row vector

Number of elements in each spatial dimension, specified as a two-element positive row vector. `ImageSize` is the same form as that returned by the `size` function.

Data Types: `double`

PixelExtentInWorldX — Size of a single pixel in the x -dimension measured in the world coordinate system

positive scalar

Size of a single pixel in the x -dimension measured in the world coordinate system, specified as a positive scalar.

Data Types: double

PixelExtentInWorldY — Size of a single pixel in the *y*-dimension measured in the world coordinate system

positive scalar

Size of a single pixel in the *y*-dimension measured in the world coordinate system, specified as a positive scalar.

Data Types: double

XWorldLimits — Limits of image in world *x*-dimension

two-element numeric row vector

Limits of image in world *x*-dimension, specified as a two-element row numeric vector [xMin xMax].

Data Types: double

YWorldLimits — Limits of image in world *y*-dimension

two-element numeric row vector

Limits of image in world *y*-dimension, specified as a two-element numeric row vector [yMin yMax].

Data Types: double

XIntrinsicLimits — Limits of image in intrinsic units in the *x*-dimension

two-element row vector

Limits of image in intrinsic units in the *x*-dimension, specified as a two-element row vector [xMin xMax]. For an *m*-by-*n* image (or an *m*-by-*n*-by-*p* image), XIntrinsicLimits equals [0.5, n + 0.5].

Data Types: double

YIntrinsicLimits — Limits of image in intrinsic units in the *y*-dimension

two-element row vector

Limits of image in intrinsic units in the *y*-dimension, specified as a two-element row vector [yMin yMax]. For an *m*-by-*n* image (or an *m*-by-*n*-by-*p* image), YIntrinsicLimits equals [0.5, m + 0.5].

Data Types: double

Methods

<code>contains</code>	True if image contains points in world coordinate system
<code>intrinsicToWorld</code>	Convert from intrinsic to world coordinates
<code>sizesMatch</code>	True if object and image are size-compatible
<code>worldToIntrinsic</code>	Convert from world to intrinsic coordinates
<code>worldToSubscript</code>	Convert world coordinates to row and column subscripts

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#) in the MATLAB documentation.

Examples

Create `imref2d` Object Given Knowledge of Image Size and World Limits

Read image.

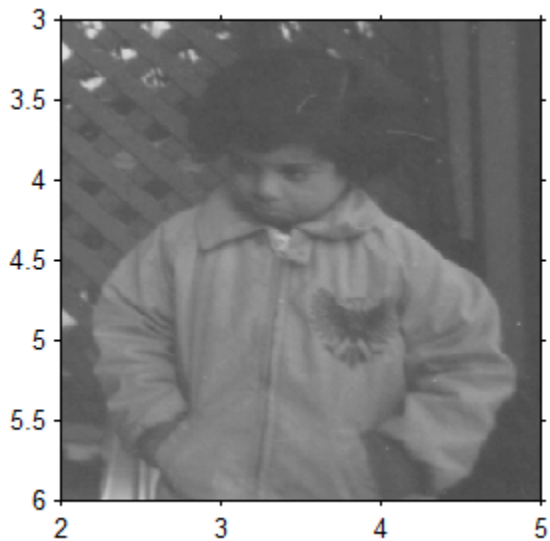
```
A = imread('pout.tif');
```

Create an `imref2d` object, specifying the size and world limits of the image with which you want to associate the object.

```
xWorldLimits = [2 5];  
yWorldLimits = [3 6];  
RA = imref2d(size(A),xWorldLimits,yWorldLimits);
```

Display the image, specifying the spatial referencing object. Note how the axes coordinates reflect the world coordinates.

```
figure, imshow(A,RA);
```



Create `imref2d` Object Given Knowledge of Image Size and Resolution

Read image.

```
m = dicominfo('knee1.dcm');
A = dicomread(m);
```

Create an `imref2d` object associated with the image, specifying the size of the pixels. The DICOM file contains a metadata field `PixelSpacing` that specifies the image resolution in each dimension in millimeters/pixel.

```
RA = imref2d(size(A),m.PixelSpacing(2),m.PixelSpacing(1))
```

RA =

`imref2d` with properties:

```
    XWorldLimits: [0.1563 160.1563]
    YWorldLimits: [0.1563 160.1563]
    ImageSize: [512 512]
    PixelExtentInWorldX: 0.3125
    PixelExtentInWorldY: 0.3125
```

```
ImageExtentInWorldX: 160
ImageExtentInWorldY: 160
  XIntrinsicLimits: [0.5000 512.5000]
  YIntrinsicLimits: [0.5000 512.5000]
```

Examine the extent of the image in each dimension in millimeters.

```
RA.ImageExtentInWorldX
RA.ImageExtentInWorldY
```

```
ans =
```

```
    160
```

```
ans =
```

```
    160
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, you can only specify singular objects—arrays of objects are not supported.

See Also

See Also

imref3d | imshow | imwarp

contains

Class: `imref2d`

True if image contains points in world coordinate system

Syntax

`TF = contains(R,xWorld,yWorld)`

Description

`TF = contains(R,xWorld,yWorld)` returns a logical array `TF` having the same size as `xWorld`, `yWorld`, such that `TF(k)` is true if and only if the point `(xWorld(k),yWorld(k))` falls within the bounds of the image associated with spatial referencing object `R`.

Input Arguments

R

Spatial referencing object, `imref2d`, associated with an image.

xWorld

Coordinates along the *X* dimension in world system, specified as a numeric scalar or vector

yWorld

Coordinates along the *Y* dimension in world system, specified as a numeric scalar or vector

Output Arguments

TF

Logical scalar or vector, the same size as the input values.

Examples

Check if Image Contains Specified World Coordinates

Read an image.

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

Create an `imref2d` object and associate it with the image.

```
R = imref2d(size(I))
```

```
R =
```

```
imref2d
```

```
Properties:
```

```
    XWorldLimits: [0.5000 256.5000]
    YWorldLimits: [0.5000 256.5000]
    ImageSize: [256 256]
PixelExtentInWorldX: 1
PixelExtentInWorldY: 1
ImageExtentInWorldX: 256
ImageExtentInWorldY: 256
    XIntrinsicLimits: [0.5000 256.5000]
    YIntrinsicLimits: [0.5000 256.5000]
```

Test to see if certain world coordinates are in the image.

```
res = contains(R,[5 8],[5 10])
```

```
res =
```

```
    1    1
```

Try some coordinates that are deliberately outside the image.

```
res = contains(R,[5 8],[5 257])
```

```
res =
```

```
    1    0
```

See Also

intrinsicToWorld

Class: `imref2d`

Convert from intrinsic to world coordinates

Syntax

```
[xWorld,yWorld] = intrinsicToWorld(R,xIntrinsic,yIntrinsic)
```

Description

`[xWorld,yWorld] = intrinsicToWorld(R,xIntrinsic,yIntrinsic)` maps point locations from the intrinsic system (`xIntrinsic`, `yIntrinsic`) to the world system (`xWorld`, `yWorld`) based on the relationship defined by the spatial referencing object `R`. If the input includes values that fall outside the limits of the image in the intrinsic system, the `intrinsicToWorld` method extrapolates `worldX` and `worldY` outside the bounds of the image in the world system.

Input Arguments

R

Spatial referencing object, `imref2d`, associated with an image.

xIntrinsic

Coordinate in *X* dimension in intrinsic system, specified as a numeric scalar or vector

yIntrinsic

Coordinate in *Y* dimension in intrinsic system, specified as a numeric scalar or vector

Output Arguments

xWorld

Coordinates along the X dimension in world system, specified as a numeric scalar or vector

yWorld

Coordinates along the Y dimension in world system, specified as a numeric scalar or vector

Examples

Convert Intrinsic Coordinates to World Coordinates

Read an image.

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

Create an `imref2d` object and associate it with the image.

```
R = imref2d(size(I))
```

```
R =
```

```
imref2d
```

```
Properties:
```

```

    XWorldLimits: [0.5000 256.5000]
    YWorldLimits: [0.5000 256.5000]
    ImageSize: [256 256]
    PixelExtentInWorldX: 1
    PixelExtentInWorldY: 1
    ImageExtentInWorldX: 256
    ImageExtentInWorldY: 256
    XIntrinsicLimits: [0.5000 256.5000]
    YIntrinsicLimits: [0.5000 256.5000]
```

Convert intrinsic coordinates to world coordinates. In this example, intrinsic coordinates align with world coordinates.

```
[val1, val2] = intrinsicToWorld(R,5,5)
```

```
val1 =
```

```
5
```

```
val2 =
```

```
5
```

See Also

sizesMatch

Class: `imref2d`

True if object and image are size-compatible

Syntax

`TF = sizesMatch(R,A)`

Description

`TF = sizesMatch(R,A)` returns `true` if the size of the image `A` is consistent with the `ImageSize` property of the spatial referencing object `R`. That is, `R.ImageSize == [size(A,1) size(A,2)]`.

Input Arguments

R

Spatial referencing object, `imref2d`, associated with an image.

A

Input image

Output Arguments

TF

Logical array. True if the size of the image `A` is consistent with the referencing object `R`.

Examples

Check Image Size with Spatial Referencing Object

Read an image.

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

Create an `imref2d` object.

```
R = imref2d(size(I))
```

```
R =
```

```
imref2d
```

```
Properties:
```

```
    XWorldLimits: [0.5000 256.5000]
    YWorldLimits: [0.5000 256.5000]
      ImageSize: [256 256]
PixelExtentInWorldX: 1
PixelExtentInWorldY: 1
ImageExtentInWorldX: 256
ImageExtentInWorldY: 256
    XIntrinsicLimits: [0.5000 256.5000]
    YIntrinsicLimits: [0.5000 256.5000]
```

See if the size of the image matches the size in the object.

```
res = sizesMatch(R,I)
```

```
res =
```

```
    1
```

Read another image that is a different size.

```
I2 = imread('coins.png');
```

Check if the size of this image matches the size in the object. It should return `false`.

```
res2 = sizesMatch(R,I2)
```

```
res2 =
```

```
    0
```

See Also

worldToIntrinsic

Class: `imref2d`

Convert from world to intrinsic coordinates

Syntax

```
[xIntrinsic,yIntrinsic] = worldToIntrinsic(R,xWorld,yWorld)
```

Description

`[xIntrinsic,yIntrinsic] = worldToIntrinsic(R,xWorld,yWorld)` maps point locations from the world system (`xWorld`, `yWorld`) to the intrinsic system (`xIntrinsic`, `yIntrinsic`) based on the relationship defined by the spatial referencing object `R`. If the input includes values that fall outside the limits of the image in the world system, the `worldToIntrinsic` method extrapolates `xWorld` and `yWorld` outside the bounds of the image in the intrinsic system.

Input Arguments

R

Spatial referencing object, `imref2d`, associated with an image.

xWorld

Coordinates in *X* dimension in world system, specified as a numeric scalar or vector.

yWorld

Coordinate in *Y* dimension in world system, specified as a numeric scalar or vector.

Output Arguments

xIntrinsic

Coordinates in *X* dimension in intrinsic system, returned as a numeric scalar or vector.

yIntrinsic

Coordinates in *Y* dimension in intrinsic system, returned as a numeric scalar or vector.

Examples

Convert World Coordinates to Intrinsic

Read an image.

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

Create an `imref2d` object.

```
R = imref2d(size(I))
```

```
R =
```

```
imref2d
```

```
Properties:
```

```
    XWorldLimits: [0.5000 256.5000]
    YWorldLimits: [0.5000 256.5000]
      ImageSize: [256 256]
PixelExtentInWorldX: 1
PixelExtentInWorldY: 1
ImageExtentInWorldX: 256
ImageExtentInWorldY: 256
  XIntrinsicLimits: [0.5000 256.5000]
  YIntrinsicLimits: [0.5000 256.5000]
```

Convert coordinates from world to intrinsic. In this example, world coordinates align with intrinsic coordinates.

```
[xi, yi] = worldToIntrinsic(R,5,5)
```

```
xi =
```

```
5
```

```
yi =
```

5

See Also

worldToSubscript

Class: `imref2d`

Convert world coordinates to row and column subscripts

Syntax

```
[I,J] = worldToSubscript(R,xWorld,yWorld)
```

Description

`[I,J] = worldToSubscript(R,xWorld,yWorld)` maps point locations from the world system (`xWorld,yWorld`) to subscript arrays `I` and `J` based on the relationship defined by the spatial referencing object `R`. `I` and `J` are the row and column subscripts of the image pixels identified by their world coordinates (`xWorld, yWorld`). `xWorld` and `yWorld` must have the same size. `I` and `J` have the same size as `xWorld` and `yWorld`.

For an m -by- n image, $1 \leq I \leq m$ and $1 \leq J \leq n$. If a point `xWorld(k)`, `yWorld(k)` falls outside the image, as defined by `contains(R,xWorld,yWorld)`, `worldToSubscript` sets the returned subscripts, `I(k)` and `J(k)`, to `NaN`.

Input Arguments

R

Spatial referencing object, `imref2d`, associated with an image.

xWorld

Coordinates along the X dimension in world system, specified as a numeric scalar or vector

yWorld

Coordinates along the Y dimension in world system, specified as a numeric scalar or vector

Output Arguments

I

Row coordinates, returned as a numeric scalar or vector the same size as `yWorld`.

J

Column coordinates, returned as a numeric scalar or vector, the same size as `xWorld`.

Examples

Convert World Coordinates to Subscripts

Read an image.

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

Create an `imref2d` object and associate it with the image.

```
R = imref2d(size(I))
```

```
R =
```

```
imref2d
```

```
Properties:
```

```

    XWorldLimits: [0.5000 256.5000]
    YWorldLimits: [0.5000 256.5000]
    ImageSize: [256 256]
PixelExtentInWorldX: 1
PixelExtentInWorldY: 1
ImageExtentInWorldX: 256
ImageExtentInWorldY: 256
    XIntrinsicLimits: [0.5000 256.5000]
    YIntrinsicLimits: [0.5000 256.5000]
```

Convert the world coordinates to subscripts. Note that, when expressed as subscripts, the order of the coordinates is reversed.

```
[m, n] = worldToSubscript(R,100,200)
```

```
m =
```

200

n =

100

See Also

imref3d class

Reference 3-D image to world coordinates

Description

An `imref3d` object encapsulates the relationship between the intrinsic coordinates anchored to the columns, rows, and planes of a 3-D image and the spatial location of the same column, row, and plane locations in a world coordinate system. The image is sampled regularly in the planar world- x , world- y , and world- z coordinates of the coordinate system such that intrinsic- x values align with world- x values, intrinsic- y values align with world- y values, and intrinsic- z values align with world- z values. The pixel spacing in each dimension may be different.

The intrinsic coordinate values (x,y,z) of the center point of any pixel are identical to the values of the column, row, and plane subscripts for that pixel. For example, the center point of the pixel in row 5, column 3, plane 4 has intrinsic coordinates $x = 3.0$, $y = 5.0$, $z = 4.0$. Be aware, however, that the order of the coordinate specification (3.0,5.0,4.0) is reversed in intrinsic coordinates relative to pixel subscripts (5,3,4). Intrinsic coordinates are defined on a continuous plane while the subscript locations are discrete locations with integer values.

Construction

`R = imref3d()` creates an `imref3d` object with default property settings.

`R = imref3d(imageSize)` creates an `imref3d` object given an image size. This syntax constructs a spatial referencing object for the default case in which the world coordinate system is co-aligned with the intrinsic coordinate system.

`R = imref3d(imageSize,pixelExtentInWorldX,pixelExtentInWorldY,pixelExtentInWorldZ)` creates an `imref3d` object given an image size and the resolution in each dimension, specified by `pixelExtentInWorldX`, `pixelExtentInWorldY`, and `pixelExtentInWorldZ`.

`R = imref3d(imageSize,xWorldLimits,yWorldLimits,zWorldLimits)` creates an `imref3d` object given an image size and the world limits in each dimension, specified by `xWorldLimits`, `yWorldLimits` and `zWorldLimits`.

Input Arguments

imageSize — Size of the image associated with the object

three-element positive row vector

Size of the image associated with the object, specified as a three-element positive row vector. You can use the vector returned by the `size` function.

Example: `imageSize = [128 128 27];`

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `int8` | `int16` | `int32`

pixelExtentInWorldX — Size of a single pixel in the x-dimension measured in the world coordinate system

positive scalar

Size of a single pixel in the x-dimension measured in the world coordinate system, specified as a positive scalar.

Data Types: `double` | `single`

pixelExtentInWorldY — Size of a single pixel in the y-dimension measured in the world coordinate system

positive scalar

Size of a single pixel in the y-dimension measured in the world coordinate system, specified as a positive scalar.

Data Types: `double` | `single`

pixelExtentInWorldZ — Size of a single pixel in the z-dimension measured in the world coordinate system

positive scalar

Size of a single pixel in the z-dimension measured in the world coordinate system, specified as a positive scalar.

Data Types: `double` | `single`

xWorldLimits — Minimum and maximum coordinate values in the x-dimension in the world coordinate system

two-element numeric row vector

Minimum and maximum coordinate values in the x-dimension in the world coordinate system, specified as a two-element numeric row vector.

Example: `xWorldLimits = [0.5 256.5];`

Data Types: `double` | `single`

yWorldLimits — Minimum and maximum coordinate values in the *y*-dimension in the world coordinate system

two-element numeric row vector

Minimum and maximum coordinate values in the *y*-dimension in the world coordinate system, specified as a two-element numeric row vector.

Example: `yWorldLimits = [0.5 256.5];`

Data Types: `double` | `single`

zWorldLimits — Minimum and maximum coordinate values in the *z*-dimension in the world coordinate system

two-element numeric row vector

Minimum and maximum coordinate values in the *z*-dimension in the world coordinate system, specified as a two-element numeric row vector.

Example: `zWorldLimits = [0.5 27.5];`

Data Types: `double` | `single`

Properties

ImageExtentInWorldX — Span of image in the *x*-dimension in the world coordinate system

numeric scalar

Span of image in the *x*-dimension in the world coordinate system, specified as a numeric scalar. The `imref3d` object calculates this value as `PixelExtentInX * ImageSize(2)`.

Data Types: `double`

ImageExtentInWorldY — Span of image in the *y*-dimension in the world coordinate system

numeric scalar

Span of image in the *y*-dimension in the world coordinate system, specified as a numeric scalar. The `imref3d` object calculates this value as `PixelExtentInY * ImageSize(1)`.

Data Types: double

ImageExtentInWorldZ — Span of image in the z-dimension in the world coordinate system
numeric scalar

Span of image in the z-dimension in the world coordinate system, specified as a numeric scalar. The `imref3d` object calculates this value as `PixelExtentInZ * ImageSize(3)`.

Data Types: double

ImageSize — Number of elements in each spatial dimension
three-element positive row vector

Number of elements in each spatial dimension, specified as a three-element positive row vector. `ImageSize` is the same form as that returned by the `size` function.

Data Types: double

PixelExtentInWorldX — Size of a single pixel in the x-dimension measured in the world coordinate system
positive scalar

Size of a single pixel in the x-dimension measured in the world coordinate system, specified as a positive scalar.

Data Types: double

PixelExtentInWorldY — Size of a single pixel in the y-dimension measured in the world coordinate system
positive scalar

Size of a single pixel in the y-dimension measured in the world coordinate system, specified as a positive scalar.

Data Types: double

PixelExtentInWorldZ — Size of a single pixel in the z-dimension measured in the world coordinate system
positive scalar

Size of a single pixel in the z-dimension measured in the world coordinate system, specified as a positive scalar.

Data Types: double

XWorldLimits — Limits of image in world *x*-dimension

two-element numeric row vector

Limits of image in world *x*, specified as a two-element row vector, [*xMin* *xMax*].

Data Types: double

YWorldLimits — Limits of image in world *y*-dimension

two-element numeric row vector

Limits of image in world *y*, specified as a two-element row vector, [*yMin* *yMax*].

Data Types: double

ZWorldLimits — Limits of image in world *z*-dimension

two-element numeric row vector

Limits of image in world *z*, specified as a two-element row vector, [*zMin* *zMax*].

Data Types: double

XIntrinsicLimits — Limits of image in intrinsic units in the *x*-dimension

two-element row vector

Limits of image in intrinsic units in the *x*-dimension, specified as a two-element row vector [*xMin* *xMax*]. For an *m*-by-*n*-by-*p* image, it equals [0.5, *n* + 0.5].

Data Types: double

YIntrinsicLimits — Limits of image in intrinsic units in the *y*-dimension

two-element row vector

Limits of image in intrinsic units in the *y*-dimension, specified as a two-element row vector [*yMin* *yMax*]. For an *m*-by-*n*-by-*p* image, it equals [0.5, *m* + 0.5].

Data Types: double

ZIntrinsicLimits — Limits of image in intrinsic units in the *z*-dimension

two-element row vector

Limits of image in intrinsic units in the *z*-dimension, specified as a two-element row vector [*zMin* *zMax*]. For an *m*-by-*n*-by-*p* image, it equals [0.5, *p* + 0.5].

Data Types: `double`

Methods

<code>contains</code>	True if image contains points in world coordinate system
<code>intrinsicToWorld</code>	Convert from intrinsic to world coordinates
<code>sizesMatch</code>	True if object and image are size-compatible
<code>worldToIntrinsic</code>	Convert from world to intrinsic coordinates
<code>worldToSubscript</code>	World coordinates to row and column subscripts

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#) in the MATLAB documentation.

Examples

Create `imref3d` Object Given Knowledge of Image Size and Resolution in each Dimension

Read image.

```
m = analyze75info('brainMRI.hdr');  
A = analyze75read(m);
```

Create an `imref3d` object associated with the image, specifying the size of the pixels. The `PixelDimensions` field of the metadata of the file specifies the resolution in each dimension in millimeters/pixel.

```
RA = imref3d(size(A),m.PixelDimensions(2),m.PixelDimensions(1),m.PixelDimensions(3));  
RA =
```

imref3d with properties:

```
XWorldLimits: [0.5000 128.5000]
YWorldLimits: [0.5000 128.5000]
ZWorldLimits: [0.5000 27.5000]
  ImageSize: [128 128 27]
PixelExtentInWorldX: 1
PixelExtentInWorldY: 1
PixelExtentInWorldZ: 1
ImageExtentInWorldX: 128
ImageExtentInWorldY: 128
ImageExtentInWorldZ: 27
  XIntrinsicLimits: [0.5000 128.5000]
  YIntrinsicLimits: [0.5000 128.5000]
  ZIntrinsicLimits: [0.5000 27.5000]
```

Examine the extent of the image in each dimension in millimeters.

```
RA.ImageExtentInWorldX
RA.ImageExtentInWorldY
RA.ImageExtentInWorldZ
```

```
ans =
```

```
128
```

```
ans =
```

```
128
```

```
ans =
```

```
27
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, you can only specify singular objects—arrays of objects are not supported.

See Also

See Also

`imref2d`

contains

Class: `imref3d`

True if image contains points in world coordinate system

Syntax

`TF = contains(R,xWorld,yWorld,zWorld)`

Description

`TF = contains(R,xWorld,yWorld,zWorld)` returns a logical array `TF` having the same size as `xWorld`, `yWorld`, and `zWorld` such that `TF(k)` is true if and only if the point `(xWorld(k), yWorld(k), zWorld(k))` falls within the bounds of the image associated with spatial referencing object `R`.

Input Arguments

R

Spatial referencing object, `imref3d`, associated with an image.

xWorld

Coordinate in the *X* dimension, specified as a numeric scalar or vector.

yWorld

Coordinate in the *Y* dimension, specified as a numeric scalar or vector.

zWorld

Coordinate in the *Z* dimension, specified as a numeric scalar or vector.

Default:

Output Arguments

TF

Logical scalar or vector, the same size as the input values.

Examples

Check if Image Contains Specified World Coordinates

Read an image.

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

Create an `imref3d` object and associate it with the image.

```
R3d = imref3d(size(I))
```

R =

```
imref3d
```

Properties:

```
    XWorldLimits: [0.5000 512.5000]
    YWorldLimits: [0.5000 384.5000]
    ZWorldLimits: [0.5000 3.5000]
    ImageSize: [384 512 3]
PixelExtentInWorldX: 1
PixelExtentInWorldY: 1
PixelExtentInWorldZ: 1
ImageExtentInWorldX: 512
ImageExtentInWorldY: 384
ImageExtentInWorldZ: 3
    XIntrinsicLimits: [0.5000 512.5000]
    YIntrinsicLimits: [0.5000 384.5000]
    ZIntrinsicLimits: [0.5000 3.5000]
```

Test to see if certain world coordinates are in the image.

```
res = contains(R,5,5,1)
```

```
res =
```

1

Try a point that is deliberately outside the image.

```
res = contains(R,513,5,1)
```

```
res =
```

0

See Also

intrinsicToWorld

Class: `imref3d`

Convert from intrinsic to world coordinates

Syntax

```
[xWorld, yWorld, zWorld] =  
intrinsicToWorld(R,xIntrinsic,yIntrinsic,zIntrinsic)
```

Description

`[xWorld, yWorld, zWorld] = intrinsicToWorld(R,xIntrinsic,yIntrinsic,zIntrinsic)` maps point locations from the intrinsic system (`xIntrinsic`, `yIntrinsic`, `zIntrinsic`) to the world system (`xWorld`, `yWorld`, `zWorld`) based on the relationship defined by the spatial referencing object `R`. If the input includes values that fall outside the limits of the image in the intrinsic system, the `intrinsicToWorld` method extrapolates `worldX`, `worldY`, and `worldZ` outside the bounds of the image in the world system.

Input Arguments

R

Spatial referencing object, `imref3d`, associated with an image.

xIntrinsic

Coordinate in *X* dimension in intrinsic system, specified as a numeric scalar or vector

yIntrinsic

Coordinate in *Y* dimension in intrinsic system, specified as a numeric scalar or vector

zIntrinsic

Coordinate in *Z* dimension in intrinsic system, specified as a numeric scalar or vector

Output Arguments

xWorld

Coordinates along the *X* dimension in world system, returned as a numeric scalar or vector the same size as `xIntrinsic`.

yWorld

Coordinates along the *Y* dimension in world system, returned as a numeric scalar or vector the same size as `xIntrinsic`.

zWorld

Coordinates along the *Z* dimension in world system, returned as a numeric scalar or vector the same size as `xIntrinsic`.

Examples

Convert Intrinsic Coordinates to World Coordinates

Read an image.

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

Create an `imref3d` object and associate it with the image.

```
R = imref3d(size(I))
```

```
R =
```

```
imref3d
```

```
Properties:
```

```
    XWorldLimits: [0.5000 512.5000]  
    YWorldLimits: [0.5000 384.5000]  
    ZWorldLimits: [0.5000 3.5000]  
      ImageSize: [384 512 3]  
PixelExtentInWorldX: 1  
PixelExtentInWorldY: 1  
PixelExtentInWorldZ: 1  
ImageExtentInWorldX: 512
```

```
ImageExtentInWorldY: 384
ImageExtentInWorldZ: 3
  XIntrinsicLimits: [0.5000 512.5000]
  YIntrinsicLimits: [0.5000 384.5000]
  ZIntrinsicLimits: [0.5000 3.5000]
```

Convert intrinsic coordinates to world coordinates. In this example, intrinsic coordinates align with world coordinates.

```
[x, y, z] = intrinsicToWorld(R,5,5,3)
```

```
x =
```

```
5
```

```
y =
```

```
5
```

```
z =
```

```
3
```

See Also

sizesMatch

Class: `imref3d`

True if object and image are size-compatible

Syntax

`TF = sizesMatch(R,A)`

Description

`TF = sizesMatch(R,A)` returns true if the size of the image `A` is consistent with the `ImageSize` property of the spatial referencing object `R`. That is, `R.ImageSize == [size(A,1) size(A,2) size(A,3)]`.

Input Arguments

R

Spatial referencing object, `imref3d`, associated with an image.

A

Input image

Output Arguments

TF

Logical scalar

Examples

Check if Image and Spatial Referencing Object Are Size-Compatible

Read an image.

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

Create an `imref3d` object and associate it with the image.

```
R = imref3d(size(I))
```

```
R =
```

```
imref3d
```

```
Properties:
```

```
    XWorldLimits: [0.5000 512.5000]  
    YWorldLimits: [0.5000 384.5000]  
    ZWorldLimits: [0.5000 3.5000]  
    ImageSize: [384 512 3]  
PixelExtentInWorldX: 1  
PixelExtentInWorldY: 1  
PixelExtentInWorldZ: 1  
ImageExtentInWorldX: 512  
ImageExtentInWorldY: 384  
ImageExtentInWorldZ: 3  
    XIntrinsicLimits: [0.5000 512.5000]  
    YIntrinsicLimits: [0.5000 384.5000]  
    ZIntrinsicLimits: [0.5000 3.5000]
```

Check if the size of the image matches the size in the object.

```
tf = sizesMatch(R,I)
```

```
tf =
```

```
    1
```

Read another image that is a different size.

```
I2 = imread('coins.png');
```

Check if the size of this image matches the size in the object. It should return **false**.

```
tf = sizesMatch(R,I2)
```

```
tf =
```

```
    0
```

See Also

worldToIntrinsic

Class: `imref3d`

Convert from world to intrinsic coordinates

Syntax

```
[xIntrinsic,yIntrinsic,zIntrinsic] =  
worldToIntrinsic(R,xWorld,yWorld,zWorld)
```

Description

```
[xIntrinsic,yIntrinsic,zIntrinsic] =  
worldToIntrinsic(R,xWorld,yWorld,zWorld)
```

 maps point locations from the world system (`xWorld`, `yWorld`, `zWorld`) to the intrinsic system (`xIntrinsic`, `yIntrinsic`, `zIntrinsic`) based on the relationship defined by the spatial referencing object `R`. If the input includes values that fall outside the limits of the image in the world system, the `worldToIntrinsic` method extrapolates `worldX`, `worldY`, and `worldZ` outside the bounds of the image in the intrinsic system.

Input Arguments

R

Spatial referencing object, `imref3d`, associated with an image.

xWorld

Coordinates along the *X* dimension in world system, specified as a numeric scalar or vector

yWorld

Coordinates along the *Y* dimension in world system, specified as a numeric scalar or vector

zWorld

Coordinates along the Z dimension in world system, specified as a numeric scalar or vector

Output Arguments

xIntrinsic

Coordinates in X dimension in intrinsic system, returned as a numeric scalar or vector.

yIntrinsic

Coordinates in Y dimension in intrinsic system, returned as a numeric scalar or vector.

zIntrinsic

Coordinates in Z dimension in intrinsic system, returned as a numeric scalar or vector.

Examples

Illustrate the world to intrinsic method

Read an image.

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

Create an `imref3d` object associated with the image.

```
R = imref3d(size(I))
```

```
R =
```

```
imref3d
```

```
Properties:
```

```
    XWorldLimits: [0.5000 512.5000]
```

```
    YWorldLimits: [0.5000 384.5000]
```

```
    ZWorldLimits: [0.5000 3.5000]
```

```
    ImageSize: [384 512 3]
```

```
    PixelExtentInWorldX: 1
```

```
PixelExtentInWorldY: 1
PixelExtentInWorldZ: 1
ImageExtentInWorldX: 512
ImageExtentInWorldY: 384
ImageExtentInWorldZ: 3
  XIntrinsicLimits: [0.5000 512.5000]
  YIntrinsicLimits: [0.5000 384.5000]
  ZIntrinsicLimits: [0.5000 3.5000]
```

Convert coordinates from world to intrinsic. In this example, world coordinates align with intrinsic coordinates.

```
[x, y, z] = worldToIntrinsic(R,5,5,3)
```

```
x =
```

```
    5
```

```
y =
```

```
    5
```

```
z =
```

```
    3
```

See Also

worldToSubscript

Class: `imref3d`

World coordinates to row and column subscripts

Syntax

`[I,J,K] = worldToSubscript(R,xWorld,yWorld,zWorld)`

Description

`[I,J,K] = worldToSubscript(R,xWorld,yWorld,zWorld)` maps point locations from the world system (`xWorld,yWorld,zWorld`) to subscript arrays `I`, `J`, and `K`, based on the relationship defined by the spatial referencing object `R`. `I`, `J`, and `K` are the row, column, and plane subscripts of the image voxels identified by their world coordinates (`xWorld`, `yWorld`, `zWorld`). `xWorld`, `yWorld`, and `zWorld` must have the same size. `I`, `J`, and `K` have the same size as `xWorld`, `yWorld`, and `zWorld`. For an m -by- n -by- p image, $1 \leq I \leq M$, $1 \leq J \leq N$, and $1 \leq K \leq P$. If a point `xWorld(k)`, `yWorld(k)`, `zWorld(k)` falls outside the image, as defined by `contains(R,xWorld, yWorld, zWorld)`, `worldToSubscript` sets the returned subscripts, `I(k)`, `J(k)`, and `K(k)` to `NaN`.

Input Arguments

R

Spatial referencing object, `imref3d`, associated with an image.

xWorld

Coordinates along the X dimension in world system, specified as a numeric scalar or vector

yWorld

Coordinates along the Y dimension in world system, specified as a numeric scalar or vector

zWorld

Coordinates along the Z dimension in world system, specified as a numeric scalar or vector

Output Arguments

I

Row coordinates, returned as a numeric scalar or vector the same size as `yWorld`.

J

Column coordinates, returned as a numeric scalar or vector the same size as `xWorld`.

K

Plane coordinates, returned as a numeric scalar or vector the same size as `zWorld`.

Examples

Convert World Coordinates to Subscripts

Read an image.

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

Create an `imref3d` object and associate it with the image.

```
R = imref3d(size(I))
```

```
R =
```

```
imref3d
```

```
Properties:
```

```
  XWorldLimits: [0.5000 512.5000]
```

```
  YWorldLimits: [0.5000 384.5000]
```

```
  ZWorldLimits: [0.5000 3.5000]
```

```
  ImageSize: [384 512 3]
```

```
  PixelExtentInWorldX: 1
```

```
PixelExtentInWorldY: 1
PixelExtentInWorldZ: 1
ImageExtentInWorldX: 512
ImageExtentInWorldY: 384
ImageExtentInWorldZ: 3
  XIntrinsicLimits: [0.5000 512.5000]
  YIntrinsicLimits: [0.5000 384.5000]
  ZIntrinsicLimits: [0.5000 3.5000]
```

Convert the world coordinates to subscripts.

```
[m, n, p] = worldToSubscript(R,30,50,3)
```

```
m =
```

```
50
```

```
n =
```

```
30
```

```
p =
```

```
3
```

See Also

imregionalmax

Regional maxima

Syntax

```
BW = imregionalmax(I)
BW = imregionalmax(I,conn)
gpuarrayBW = imregionalmax(gpuarrayI, ___)
```

Description

`BW = imregionalmax(I)` returns the binary image `BW` that identifies the regional maxima in `I`. Regional maxima are connected components of pixels with a constant intensity value, t , whose external boundary pixels all have a value less than t . In `BW`, pixels that are set to 1 identify regional maxima; all other pixels are set to 0.

`BW = imregionalmax(I,conn)` computes the regional maxima, where `conn` specifies the connectivity. By default, `imregionalmax` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images.

`gpuarrayBW = imregionalmax(gpuarrayI, ___)` performs the operation on a GPU. The input image must be a `gpuArray`. The function returns a `gpuArray`. This syntax requires Parallel Computing Toolbox.

Examples

Find Regional Maxima in Simple Sample Image

Create a simple sample image with several regional maxima.

```
A = 10*ones(10,10);
A(2:4,2:4) = 22;
A(6:8,6:8) = 33;
A(2,7) = 44;
A(3,8) = 45;
A(4,9) = 44
```

A =

```

10  10  10  10  10  10  10  10  10  10
10  22  22  22  10  10  44  10  10  10
10  22  22  22  10  10  10  45  10  10
10  22  22  22  10  10  10  10  44  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  33  33  33  10  10
10  10  10  10  10  33  33  33  10  10
10  10  10  10  10  33  33  33  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10

```

Find the regional maxima. Note that the result includes the regional maxima at (3,8).

```
regmax = imregionalmax(A)
```

regmax = 10×10 logical array

```

0  0  0  0  0  0  0  0  0  0
0  1  1  1  0  0  0  0  0  0
0  1  1  1  0  0  0  1  0  0
0  1  1  1  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  1  1  1  0  0
0  0  0  0  0  1  1  1  0  0
0  0  0  0  0  1  1  1  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0

```

Find Regional Maxima in Simple Sample Image on a GPU

Create a 10-by-10 pixel sample image that contains two regional maxima.

```

A = 10*gpuArray.ones(10,10);
A(2:4,2:4) = 22;    % maxima 12 higher than surrounding pixels
A(6:8,6:8) = 33;    % maxima 23 higher than surrounding pixels
A(2,7) = 44;
A(3,8) = 45;        % maxima 1 higher than surrounding pixels
A(4,9) = 44

```

A =

```

10  10  10  10  10  10  10  10  10  10

```



```

10  22  22  22  10  10  44  10  10  10
10  22  22  22  10  10  10  45  10  10
10  22  22  22  10  10  10  10  44  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  33  33  33  10  10
10  10  10  10  10  33  33  33  10  10
10  10  10  10  10  33  33  33  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10

```

Pass the sample image `A` to `imregionalmax`. The function returns a binary image, the same size as `A`, in which pixels with the value `1` represent the regional maxima in `A`. `imregionalmax` sets all other pixels in to `0`.

```
regmax = imregionalmax(A)
```

```
regmax =
```

```

0  0  0  0  0  0  0  0  0  0
0  1  1  1  0  0  0  0  0  0
0  1  1  1  0  0  0  1  0  0
0  1  1  1  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  1  1  1  0  0
0  0  0  0  0  1  1  1  0  0
0  0  0  0  0  1  1  1  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0

```

Input Arguments

I — Input image

nonsparse numeric array of any dimension

Input array, specified as a nonsparse numeric array of any dimension.

```
Example: I = imread('glass.png'); BW = imregionalmax(I);
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

conn — Connectivity

8 (default) | 4 | 6 | 18 | 26 | 3-by-3-by- ...-by-3 matrix of zeroes and ones

Connectivity, specified as a one of the scalar values in the following table. By default, `imregionalmax` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imregionalmax` uses `conndef(ndims(I), 'maximal')`. Connectivity can be defined in a more general way for any dimension by using for `conn` a 3-by-3-by- ...-by-3 matrix of 0s and 1s. The 1-valued elements define neighborhood locations relative to the center element of `conn`. Note that `conn` must be symmetric around its center element.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Example: `regmax = imregionalmax(A,4);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

gpuarrayI — Input image for GPU

`gpuArray`

Input image for GPU, specified as a `gpuArray`.

Example: `gpuarrayI = gpuArray(imread('cameraman.tif')); gpuarrayBW = imregionalmax(gpuarrayI);`

Output Arguments

BW — Transformed image

logical array

Transformed image, returned as a logical array the same size as `I`.

gpuarrayBW — Transformed image`gpuArray`

Transformed image, returned as a `gpuArray`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the optional second input argument, `conn`, must be a compile-time constant.

See Also

See Also

`conndef` | `imextendedmax` | `imhmax` | `imreconstruct` | `imregionalmin`

Introduced before R2006a

imregionalmin

Regional minima

Syntax

```
BW = imregionalmin(I)  
BW = imregionalmin(I,conn)  
gpuarrayBW = imregionalmin(gpuarrayI, ____ )
```

Description

`BW = imregionalmin(I)` returns the binary image `BW` that identifies the regional minima in `I`. Regional minima are connected components of pixels with a constant intensity value, and whose external boundary pixels all have a higher value. In `BW`, pixels that are set to 1 identify regional minima; all other pixels are set to 0.

`BW = imregionalmin(I,conn)` computes the regional minima, where `conn` specifies the desired connectivity. By default, `imregionalmin` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images

`gpuarrayBW = imregionalmin(gpuarrayI, ____)` performs the operation on a GPU. The input image must be a `gpuArray`. The function returns a `gpuArray`. This syntax requires Parallel Computing Toolbox.

Examples

Find Regional Minima in Simple Sample Image

Create a simple sample array with several regional minima.

```
A = 10*ones(10,10);  
A(2:4,2:4) = 3;  
A(6:8,6:8) = 8
```

A =

```

10  10  10  10  10  10  10  10  10  10
10  3   3   3   3  10  10  10  10  10
10  3   3   3   3  10  10  10  10  10
10  3   3   3   3  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  8   8   8  10  10
10  10  10  10  10  8   8   8  10  10
10  10  10  10  10  8   8   8  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10

```

Calculate the regional minima. The function returns a binary image, the same size as the input image, in which pixels with the value 1 represent the regional minima. `imregionalmin` sets all other pixels in to 0.

```
regmin = imregionalmin(A)
```

```

regmin = 10x10 logical array
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

```

Find Regional Minima in Simple Sample Image on a GPU

Create a 10-by-10 pixel sample image that contains two regional minima.

```

A = 10*gpuArray.ones(10,10);
A(2:4,2:4) = 3;           % minima 3 lower than surround
A(6:8,6:8) = 8           % minima 8 lower than surroundA(6:8,6:8) = 7;

```

```
A =
```

```

10  10  10  10  10  10  10  10  10  10
10  3   3   3   3  10  10  10  10  10

```

```

10   3   3   3   10  10  10  10  10  10
10   3   3   3   10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10   8   8   8  10  10
10  10  10  10  10   8   8   8  10  10
10  10  10  10  10   8   8   8  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10
10  10  10  10  10  10  10  10  10  10

```

Pass the sample image `A` to `imregionalmin`. The function returns a binary image, the same size as `A`, in which pixels with the value 1 represent the regional minima in `A`. `imregionalmin` sets all other pixels in to 0.

```
regmin = imregionalmin(A)
```

```
regmin =
```

```

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

```

Input Arguments

I — Input image

nonsparse numeric array of any dimension

Input array, specified as a nonsparse numeric array of any dimension.

```
Example: I = imread('glass.png'); BW = imregionalmin(I);
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

conn — Connectivity

8 (default) | 4 | 6 | 18 | 26 | 3-by-3-by- ...-by-3 matrix of zeroes and ones

Connectivity, specified as a one of the scalar values in the following table. By default, `imregionalmin` uses 8-connected neighborhoods for 2-D images and 26-connected neighborhoods for 3-D images. For higher dimensions, `imregionalmin` uses `conndef(ndims(I), 'maximal')`. Connectivity can be defined in a more general way for any dimension by using for `conn` a 3-by-3-by-...-by-3 matrix of 0s and 1s. The 1-valued elements define neighborhood locations relative to the center element of `conn`. Note that `conn` must be symmetric about its center element.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Example: `B = imregionalmin(A,4);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

gpuarrayI — Input image for GPU

`gpuArray`

Input image for GPU, specified as a `gpuArray`.

Example: `gpuarrayI = gpuArray(imread('cameraman.tif')); gpuarrayBW = imregionalmin(gpuarrayI);`

Output Arguments

BW — Transformed image

logical array

Transformed image, returned as a logical array the same size as `I`.

gpuarrayBW — Transformed image

`gpuArray`

Transformed image, returned as a `gpuArray`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the optional second input argument, `conn`, must be a compile-time constant.

See Also

See Also

`conndef` | `imextendedmin` | `imhmin` | `imimposemin` | `imreconstruct` | `imregionalmax`

Introduced before R2006a

imregconfig

Configurations for intensity-based registration

Syntax

```
[optimizer,metric] = imregconfig(modality)
```

Description

`[optimizer,metric] = imregconfig(modality)` creates `optimizer` and `metric` configurations that you pass to `imregister` to perform intensity-based image registration. `imregconfig` returns `optimizer` and `metric` with default settings to provide a basic registration configuration.

Examples

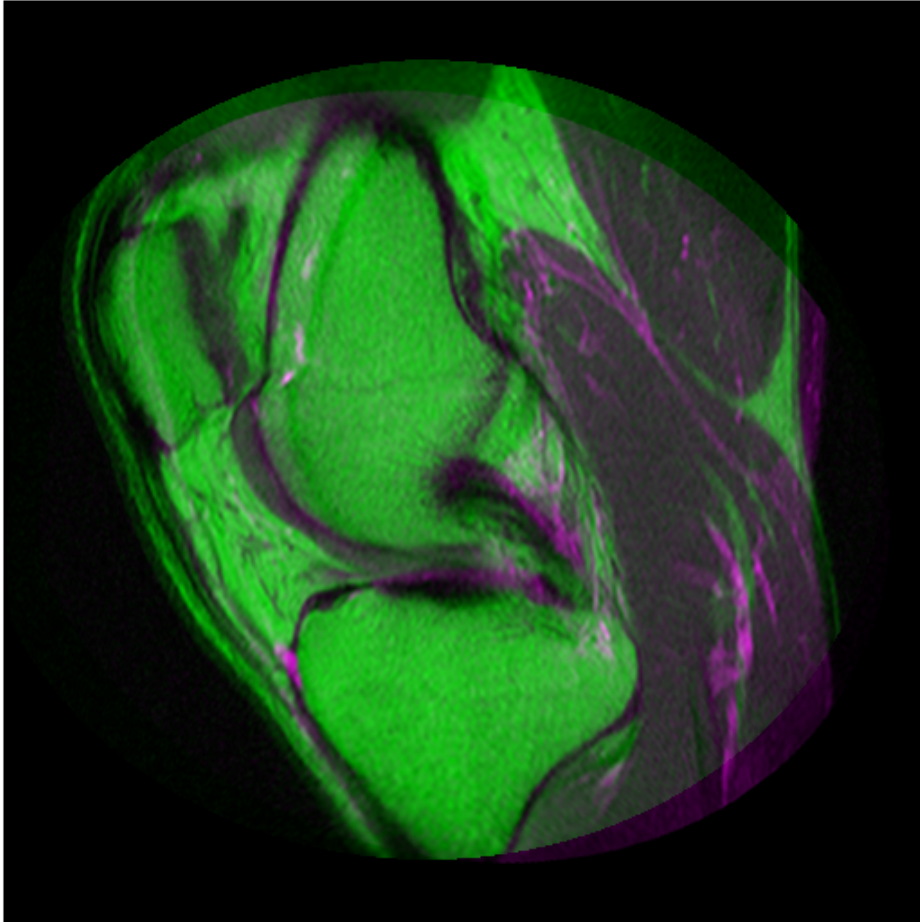
Register Multimodal MRI Images with Optimizer

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

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

View the misaligned images.

```
imshowpair(fixed, moving, 'Scaling', 'joint')
```



Create the optimizer and metric, setting the modality to 'multimodal' since the images come from different sensors.

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

```
optimizer =
    registration.optimizer.OnePlusOneEvolutionary

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

metric =
    registration.metric.MattesMutualInformation

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

Tune the properties of the optimizer to get the problem to converge on a global maxima and to allow for more iterations.

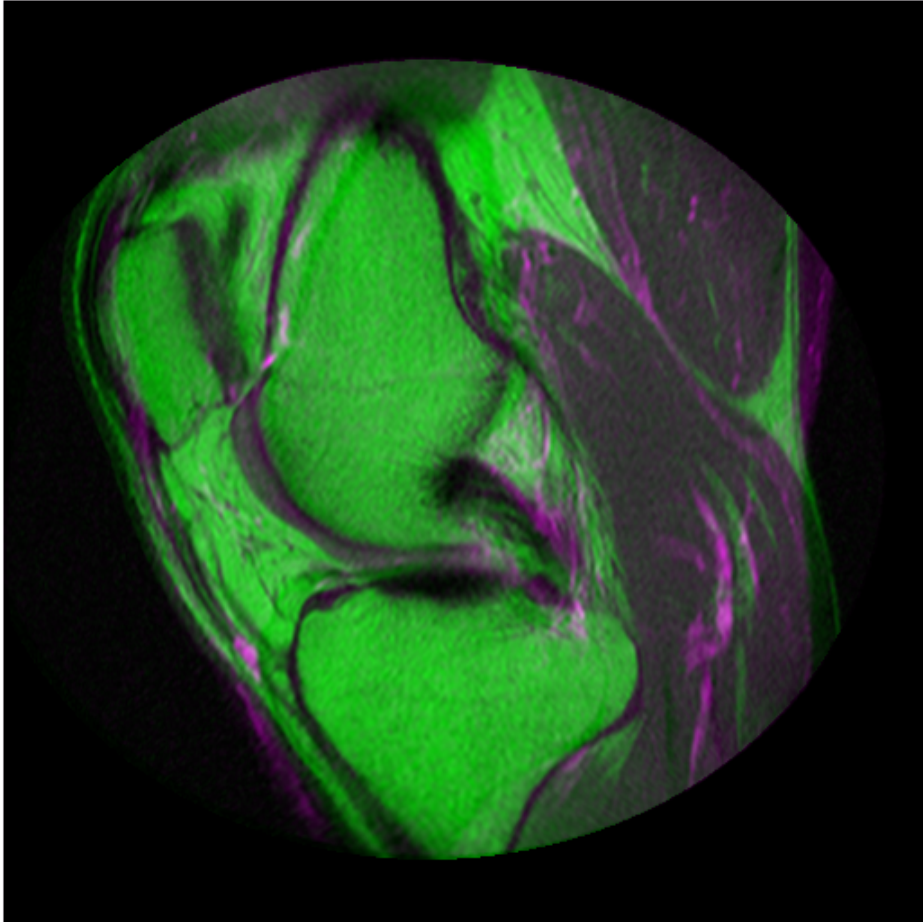
```
optimizer.InitialRadius = 0.009;
optimizer.Epsilon = 1.5e-4;
optimizer.GrowthFactor = 1.01;
optimizer.MaximumIterations = 300;
```

Perform the registration.

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

View the registered images.

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



Input Arguments

modality — Image capture modality

'monomodal' | 'multimodal'

Image capture modality describes how your images have been captured, specified as either 'monomodal' (with similar brightness and contrast) or 'multimodal' (with different brightness or contrast).

Output Arguments

optimizer — Optimization configuration

RegularStepGradientDescent or OnePlusOneEvolutionary optimizer object

Optimization configuration, returned as a RegularStepGradientDescent or OnePlusOneEvolutionary optimizer object. To learn more about the creation and properties of optimizer objects, see registration.optimizer.RegularStepGradientDescent and registration.optimizer.OnePlusOneEvolutionary.

metric — Metric configuration

MeanSquares or MattesMutualInformation metric object

Metric configuration describes the image similarity metric to be optimized during registration, returned as a MeanSquares or MattesMutualInformation metric object. To learn more about the creation and properties of metric objects, see registration.metric.MeanSquares and registration.metric.MattesMutualInformation.

Definitions

Monomodal

Monomodal images have similar brightness and contrast. The images are captured on the same type of scanner or sensor.

Multimodal

Multimodal images have different brightness and contrast. The images can come from two different types of devices, such as two camera models or two types of medical

imaging modalities (like CT and MRI). The images can also come from a single device, such as a camera using different exposure settings, or an MRI scanner using different imaging sequences.

Tips

- If you adjust the optimizer or metric parameters, the registration results can improve. For example, if you increase the number of iterations in the optimizer, reduce the optimizer step size, or change the number of samples in a stochastic metric, the registration improves to a point, at the expense of performance.

See Also

See Also

Apps

Registration Estimator

Functions

`imregister` | `imshowpair`

Classes

`registration.metric.MattesMutualInformation` | `registration.metric.MeanSquares`
| `registration.optimizer.RegularStepGradientDescent` |
`registration.optimizer.OnePlusOneEvolutionary`

Topics

“Create an Optimizer and Metric for Intensity-Based Image Registration”
“Intensity-Based Automatic Image Registration”

Introduced in R2012a

imregcorr

Estimate geometric transformation that aligns two 2-D images using phase correlation

Syntax

```
tform = imregcorr(moving, fixed)
tform = imregcorr(moving, fixed, transformtype)
tform = imregcorr(moving, Rmoving, fixed, Rfixed, ___)
tform = imregcorr( ___, Name, Value, ___)
```

Description

`tform = imregcorr(moving, fixed)` estimates the geometric transformation that aligns an image, `moving`, with a reference image, `fixed`. The function returns a geometric transformation object, `tform`, that maps pixels in `moving` to pixels in `fixed`.

`tform = imregcorr(moving, fixed, transformtype)` estimates the geometric transformation, where `transformtype` is a character vector that specifies the type of transformation.

`tform = imregcorr(moving, Rmoving, fixed, Rfixed, ___)` estimates the geometric transformation that aligns an image, `moving`, with a reference image, `fixed`. `Rmoving` and `Rfixed` are spatial referencing objects that contain spatial information about the `moving` and `fixed` images, respectively. The transformation object returned, `tform`, defines the point mapping in the world coordinate system.

`tform = imregcorr(___, Name, Value, ___)` registers the moving image to the fixed image using name-value pairs to control various aspects of the registration algorithm.

Examples

Register Images Using Phase Correlation

Read a reference image into the workspace.

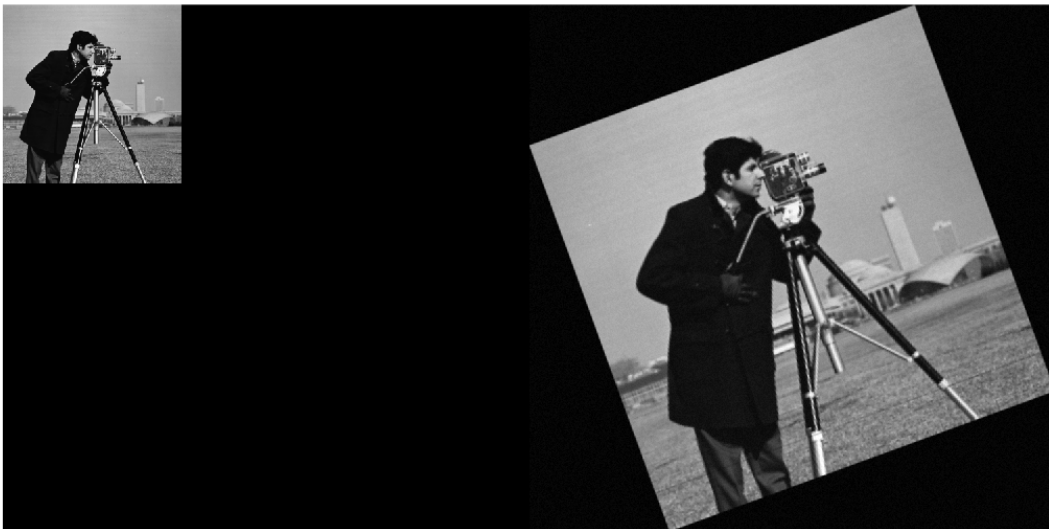
```
fixed = imread('cameraman.tif');
```

Create a synthetic moving image by scaling and rotating the fixed image.

```
theta = 20;  
S = 2.3;  
tform = affine2d([S.*cosd(theta) -S.*sind(theta) 0; ...  
                 S.*sind(theta)  S.*cosd(theta) 0; ...  
                 0 0 1]);  
moving = imwarp(fixed,tform);  
moving = moving + uint8(10*rand(size(moving)));
```

Display the fixed and the moving image alongside each other.

```
imshowpair(fixed,moving,'montage')
```



Estimate the transformation needed to align the images using `imregcorr`.

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

Apply estimated geometric transform to the moving image. This example uses the 'OutputView' parameter to obtain a registered image the same size and with the same world limits as the reference image.


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

View the original image and the registered image side-by-side to check the registration. Then view the registered image overlaid on the original using the 'falsecolor' option to highlight any areas where the images differ.

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





Input Arguments

moving — Image to be registered

grayscale image | binary image | RGB image

Image to be registered, specified as a grayscale, binary, or RGB image. If you specify an RGB image, `imregcorr` converts it to a grayscale image using `rgb2gray` before processing.

Note: The aspect ratio of `moving` affects the output transform `tform`. For best results, use a square image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

fixed — Reference image in the target orientation

grayscale image | binary image | RGB image

Reference image in the target orientation, specified as a grayscale, binary, or RGB image. If you specify an RGB image, `imregcorr` converts it to a grayscale image using `rgb2gray` before processing.

Note: The aspect ratio of `fixed` affects the output transform `tform`. For best results, use a square image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

transformtype — Type of transformation to estimate

'similarity' (default) | 'rigid' | 'translation'

Type of transformation to estimate, specified as one of the following values.

Value	Description
'translation'	Translation
'rigid'	Translation and rotation
'similarity'	Translation, rotation, and scaling When using the 'similarity' option, the phase correlation algorithm is only scale invariant within some range of scale difference between the fixed and moving images. <code>imregcorr</code> limits the search space to scale differences within the range [1/4, 4]. <code>imregcorr</code> does not detect scale differences less than 1/4 or greater than 4.

Data Types: `char`

Rmoving — Spatial referencing information associated with the image to be registered

spatial referencing object

Spatial referencing information associated with the image to be registered, specified as a spatial referencing object of type `imref2d`.

Rfixed — Spatial referencing information associated with the reference (fixed) image

spatial referencing object

Spatial referencing information associated with the reference (fixed) image, specified as a spatial referencing object of type `imref2d`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `tformEstimate = imregcorr(moving, fixed, 'Window', true);`

'Window' — Logical flag to control use of windowing to suppress spectral leakage effects in frequency domain

true (default) | scalar logical

Logical flag to control use of windowing to suppress spectral leakage effects in frequency domain, specified as the comma-separated pair consisting of 'Window' and a logical scalar. When set to `true`, `imregcorr` uses a Blackman window to increase the stability of registration results. If the common features you are trying to align in your images are oriented along the edges, setting 'Window' to `false` can sometimes provide superior registration results.

Example: `tformEstimate = imregcorr(moving, fixed, 'Window', true);`

Data Types: logical

Output Arguments

tform — Geometric transformation

geometric transformation object

Geometric transformation, returned as a geometric transformation object of type `affine2d`.

Tips

- If your image is of type `double`, you can achieve performance improvements by casting the image to `single` with `im2single` before registration. Input images of type `double` cause the algorithm to compute FFTs in `double`.

References

- [1] Reddy, B. S. and Chatterji, B. N., *An FFT-Based Technique for Translation, Rotation, and Scale-Invariant Image Registration*, IEEE Transactions on Image Processing, Vol. 5, No. 8, August 1996

See Also

See Also

Apps

Registration Estimator

Functions

imregister | imregtform | imshowpair | imwarp

Introduced in R2014a

imregdemons

Estimate displacement field that aligns two 2-D or 3-D images

Syntax

```
[D,moving_reg] = imregdemons(moving,fixed)
[___] = imregdemons(moving,fixed,N)
[gpuarrayD,gpuarrayMoving_reg] = imregdemons(gpuarrayMoving,
gpuarrayFixed,N)
[___] = imregdemons(___,Name,Value,...)
```

Description

`[D,moving_reg] = imregdemons(moving,fixed)` estimates the displacement field `D` that aligns the image to be registered, `moving`, with the reference image, `fixed`. `moving` and `fixed` are 2-D or 3-D intensity images.

The displacement vectors at each pixel location map locations from the `fixed` image grid to a corresponding location in the `moving` image. `moving_reg` is a warped version of the `moving` image that is warped according to the displacement field `D` and resampled using linear interpolation.

`[___] = imregdemons(moving,fixed,N)` specifies the number of iterations to be computed. This function does not use a convergence criterion and therefore is always guaranteed to run for the specified or default number of iterations.

`[gpuarrayD,gpuarrayMoving_reg] = imregdemons(gpuarrayMoving,gpuarrayFixed,N)` performs the estimation on a GPU.

`[___] = imregdemons(___,Name,Value,...)` registers the moving image using name-value pairs to control aspects of weight computation.

Examples

Register Two Images with Local Distortions

This example shows how to solve a registration problem in which the same hand has been photographed in two different poses. The misalignment of the images varies locally throughout each image. This is therefore a non-rigid registration problem.

Read the two images into the workspace.

```
fixed = imread('hands1.jpg');  
moving = imread('hands2.jpg');
```

Convert the images to grayscale for processing.

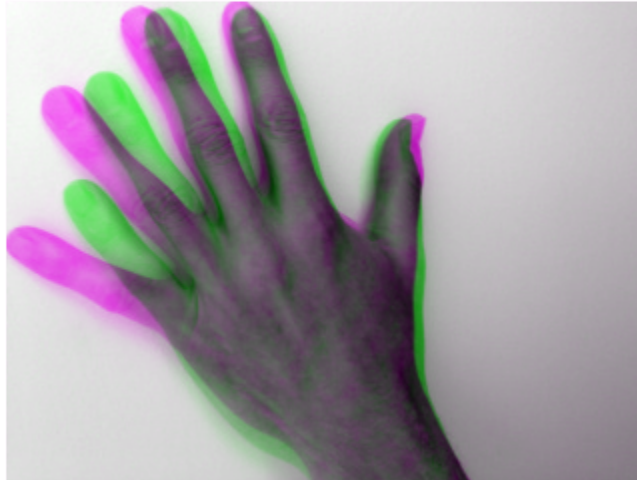
```
fixed = rgb2gray(fixed);  
moving = rgb2gray(moving);
```

Observe the initial misalignment. Fingers are in different poses. In the second figure, the two images are overlaid over each other to make it easy to see where the images differ. The differences are highlighted in green.

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



```
figure  
imshowpair(fixed,moving)
```



Correct illumination differences between the `moving` and `fixed` images using histogram matching. This is a common pre-processing step.

```
moving = imhistmatch(moving,fixed);
```

Estimate the transformation needed to bring the two images into alignment.

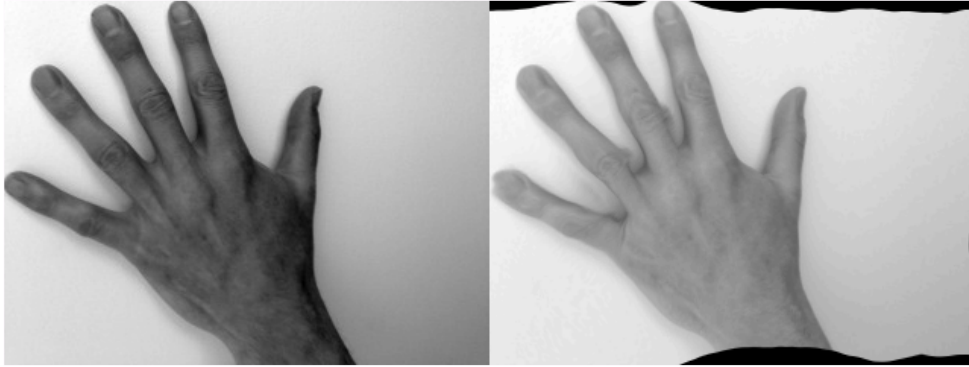
```
[~,movingReg] = imregdemons(moving,fixed,[500 400 200],...  
    'AccumulatedFieldSmoothing',1.3);
```

Display the results of the registration. In the first figure, the images are overlaid to show the alignment.

```
figure  
imshowpair(fixed,movingReg)
```




```
figure  
imshowpair(fixed,movingReg,'montage')
```



Register Two Images with Local Distortions on a GPU

Perform a nonrigid registration on a GPU.

Read images into the workspace.

```
fixed = imread('hands1.jpg');  
moving = imread('hands2.jpg');
```

Observe the initial misalignment. (Fingers are in different positions.)

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

Create `gpuArrays` and convert the images to grayscale.

```
fixedGPU = gpuArray(fixed);  
movingGPU = gpuArray(moving);  
  
fixedGPU = rgb2gray(fixedGPU);  
movingGPU = rgb2gray(movingGPU);
```

Use histogram matching to correct illumination differences between the moving and fixed images. This is a common preprocessing step.

```
fixedHist = imhist(fixedGPU);  
movingGPU = histeq(movingGPU, fixedHist);
```

Perform the registration.

```
[~,movingReg] = imregdemons(movingGPU, fixedGPU, [500 400 200], 'AccumulatedFieldSmoothing');
```

Bring the registered image back to the CPU.

```
movingReg = gather(movingReg);
```

View the results.

```
figure  
imshowpair(fixed, movingReg)  
figure  
imshowpair(fixed, movingReg, 'montage')
```

Input Arguments

moving — Image to be registered

2-D or 3-D grayscale image

Image to be registered, specified as a 2-D or 3-D grayscale image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

fixed — Reference image in the target orientation

2-D or 3-D grayscale image

Reference image in the target orientation, specified as a 2-D or 3-D grayscale image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

N — Number of iterations

100 (default) | positive integer scalar or vector

Number of iterations, specified as a positive integer scalar or vector.

When you specify a vector, **N** is the number of iterations per pyramid level (resolution level). For example, if there are 3 pyramid levels, you can specify the vector

[100,50,25], where `imregdemons` performs 100 iterations at the lowest resolution level, 50 iterations at the next pyramid level, and 25 iterations at the last iteration level—the level with full resolution. Because it takes less time to process the lower resolution levels, running more iterations at low resolution and fewer iterations at the higher resolutions of the pyramid can help performance.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

gpuarrayMoving — Input image for processing on a GPU

`gpuArray` containing a 2-D or 3-D grayscale image

Input image for processing on a GPU, specified as a 2-D or 3-D grayscale image.

gpuarrayFixed — Reference image for processing on a GPU

`gpuArray` containing a 2-D or 3-D grayscale image

Reference image for processing on a GPU, specified as a `gpuArray` containing 2-D or 3-D grayscale image.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[D,movingReg] = imregdemons(moving,fixed,[500 400 200],'AccumulatedFieldSmoothing',1.5);`

'AccumulatedFieldSmoothing' — Smoothing applied at each iteration

1.0 (default) | positive scalar

Smoothing applied at each iteration, specified as the comma-separated pair consisting of `'AccumulatedFieldSmoothing'` and a numeric value. This parameter controls the amount of diffusion-like regularization. `imregdemons` applies the standard deviation of the Gaussian smoothing to regularize the accumulated field at each iteration. Larger values result in smoother output displacement fields. Smaller values result in more localized deformation in the output displacement field. Values typically are in the range [0.5, 3.0]. When you specify multiple `PyramidLevels`, the standard deviation used in the Gaussian smoothing remains the same at each pyramid level.

Data Types: `double`

'PyramidLevels' — Number of multi-resolution image pyramid levels to use

3 (default) | positive integer scalar

Number of multi-resolution image pyramid levels to use, specified as the comma-separated pair consisting of 'PyramidLevels' and a positive integer scalar.

Data Types: double

'DisplayWaitbar' — Display waitbar to indicate progress

true (default) | false

Display waitbar to indicate progress, specified as the comma-separated pair consisting of 'DisplayWaitbar' and the value true or false. When set to true, `imregdemons` displays a waitbar to indicate progress for long-running operations. To prevent `imregdemons` from displaying a waitbar, set `DisplayWaitbar` to false.

Note: The 'DisplayWaitbar' parameter is not supported on a GPU.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

Output Arguments

D — Displacement field

numeric array

Displacement field, specified as a numeric array. Displacement values are in units of pixels.

- If `fixed` is a 2-D grayscale image of size m -by- n , the displacement field array is m -by- n -by-2. `D(:, :, 1)` contains displacements along the x -axis and `D(:, :, 2)` contains displacements along the y -axis.
- If `fixed` is a 3-D grayscale image of size m -by- n -by- p , the displacement field array is m -by- n -by- p -by-3. `D(:, :, :, 1)` contains displacements along the x -axis, `D(:, :, :, 2)` contains displacements along the y -axis, and `D(:, :, :, 3)` contains displacements along the z -axis.

Data Types: double

moving_reg — Aligned image

2-D or 3-D grayscale image

Registered image, returned as a 2-D or 3-D grayscale image, warped according to the displacement field `D` and resampled using linear interpolation.

gpuarrayD — Displacement field

`gpuArray` containing a matrix of class `double`

Displacement field, specified as a `gpuArray` containing a matrix of class `double`.

gpuarrayMoving_reg — Aligned image

`gpuArray` containing a 2-D or 3-D grayscale image

Registered image, returned as a `gpuArray` containing a 2-D or 3-D grayscale image, warped according to the displacement field `gpuarrayD` and resampled using linear interpolation.

Tips

- To transform an image using the displacement field `D`, use `imwarp`.

See Also

See Also

Apps

Registration Estimator

Functions

`imregcorr` | `imregister` | `imregtform` | `imshowpair` | `imwarp`

Introduced in R2014b

imregister

Intensity-based image registration

Syntax

```
moving_reg = imregister(moving, fixed, transformType, optimizer, metric)
[moving_reg, R_reg] = imregister(moving, Rmoving, fixed, Rfixed,
transformType, optimizer, metric)
___ = imregister( ___, Name, Value)
```

Description

`moving_reg = imregister(moving, fixed, transformType, optimizer, metric)` transforms the 2-D or 3-D image, `moving`, so that it is registered with the reference image, `fixed`. Both `moving` and `fixed` images must be of the same dimensionality, either 2-D or 3-D. `transformType` is a character vector that defines the type of transformation to perform. `optimizer` is an object that describes the method for optimizing the metric. `metric` is an object that defines the quantitative measure of similarity between the images to optimize. Returns the aligned image, `moving_reg`.

`[moving_reg, R_reg] = imregister(moving, Rmoving, fixed, Rfixed, transformType, optimizer, metric)` transforms the spatially referenced image `moving` so that it is registered with the spatially referenced image `fixed`. `Rmoving` and `Rfixed` are spatial referencing objects that describe the world coordinate limits and the resolution of `moving` and `fixed`.

`___ = imregister(___, Name, Value)` specifies additional options with one or more `Name, Value` pair arguments.

Examples

Register Multimodal MRI Images with Optimizer

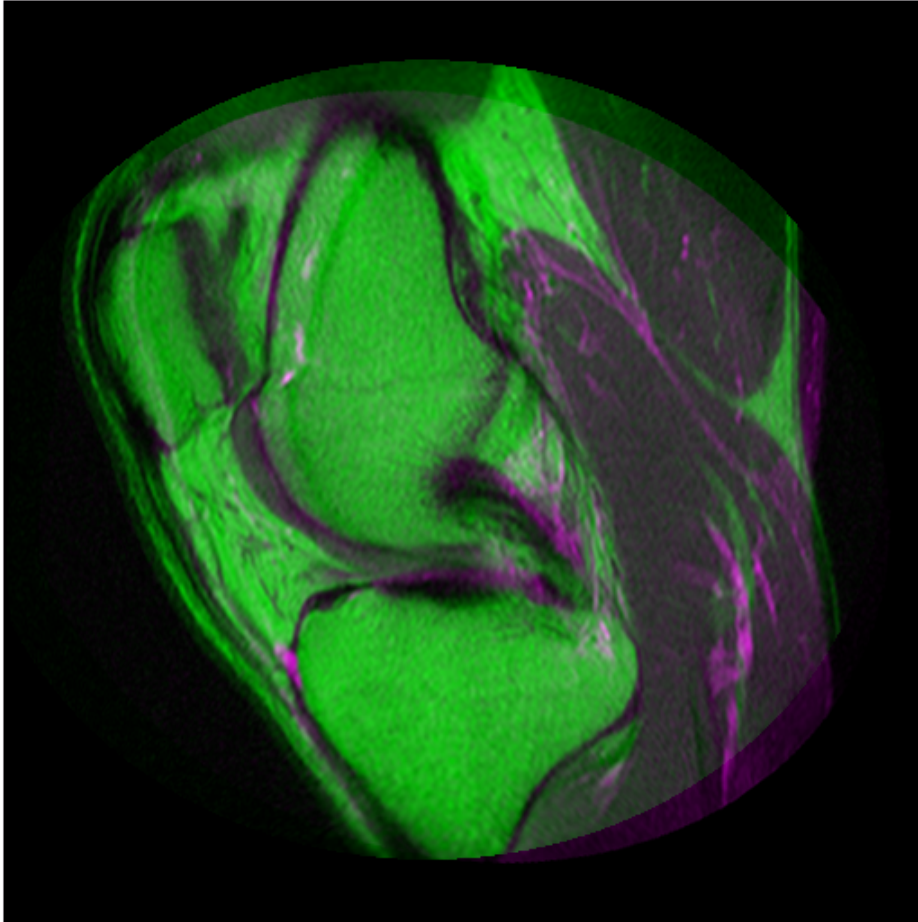
Read two images. This example uses two magnetic resonance (MRI) images of a knee. The fixed image is a spin echo image, while the moving image is a spin echo image

with inversion recovery. The two sagittal slices were acquired at the same time but are slightly out of alignment.

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

View the misaligned images.

```
imshowpair(fixed, moving, 'Scaling', 'joint')
```

Create the optimizer and metric, setting the modality to 'multimodal' since the images come from different sensors.

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

```
optimizer =
    registration.optimizer.OnePlusOneEvolutionary

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

metric =
    registration.metric.MattesMutualInformation

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

Tune the properties of the optimizer to get the problem to converge on a global maxima and to allow for more iterations.

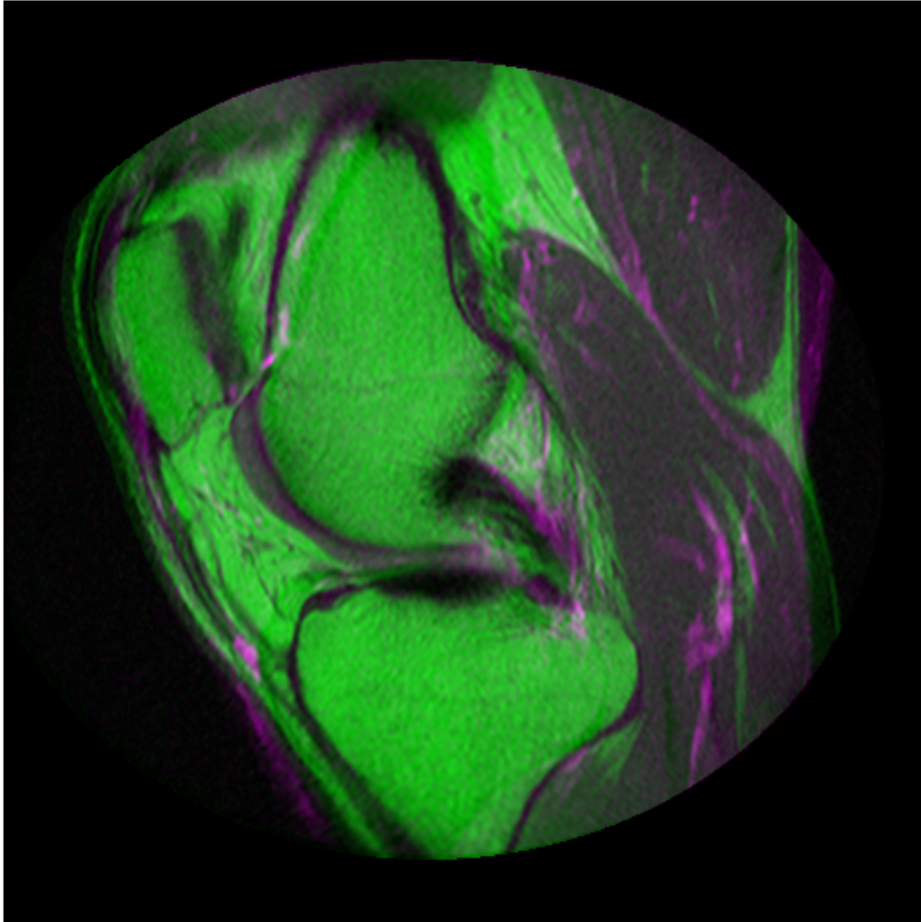
```
optimizer.InitialRadius = 0.009;
optimizer.Epsilon = 1.5e-4;
optimizer.GrowthFactor = 1.01;
optimizer.MaximumIterations = 300;
```

Perform the registration.

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

View the registered images.

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



Input Arguments

moving — Image to be registered

grayscale image

Image to be registered, specified as a 2-D or 3-D grayscale image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Rmoving — Spatial referencing information associated with the image to be registered

spatial referencing object

Spatial referencing information associated with image to be registered, specified as a spatial referencing object of type `imref2d` or `imref3d`.

fixed — Reference image in the target orientation

grayscale image

Reference image in the target orientation, specified as a grayscale image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Rfixed — Spatial referencing information associated with the reference image

spatial referencing object

Spatial referencing information associated with the reference image, specified as a spatial referencing object of type `imref2d` or `imref3d`.

transformType — Geometric transformation to be applied to the image to be registered

'translation' | 'rigid' | 'similarity' | 'affine'

Geometric transformation to be applied to the moving image, specified as one of the following values:

Value	Description
'translation'	(x,y) translation in 2-D, or (x,y,z) translation in 3-D.
'rigid'	Rigid transformation consisting of translation and rotation.
'similarity'	Nonreflective similarity transformation consisting of translation, rotation, and scale.
'affine'	Affine transformation consisting of translation, rotation, scale, and shear.

The 'similarity' and 'affine' transformation types always involve nonreflective transformations.

optimizer — Method for optimizing the similarity metric

RegularStepGradientDescent or OnePlusOneEvolutionary optimizer object

Method for optimizing the similarity metric, specified as a RegularStepGradientDescent or OnePlusOneEvolutionary optimizer object. To learn more about the creation and properties of optimizer objects, see registration.optimizer.RegularStepGradientDescent and registration.optimizer.OnePlusOneEvolutionary.

metric — Image similarity metric to be optimized during registration

MeanSquares or MattesMutualInformation metric object

Image similarity metric to be optimized during registration, specified as a MeanSquares or MattesMutualInformation metric object. To learn more about the creation and properties of metric objects, see registration.metric.MeanSquares and registration.metric.MattesMutualInformation.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: 'DisplayOptimization', 1 enables the verbose optimization mode.

'DisplayOptimization' — Verbose optimization flag

false (default) | true

Verbose optimization flag, specified as the comma-separated pair consisting of 'DisplayOptimization', and the logical value true or false. Controls whether imregister displays optimization information in the command window during the registration process.

Data Types: logical

'InitialTransformation' — Starting geometric transformation

affine2d or affine3d geometric transformation object

Starting geometric transformation, specified as the comma-separated pair consisting of 'InitialTransformation' and an affine2d or affine3d geometric transformation object.

'PyramidLevels' — Number of pyramid levels used during registration process

3 (default) | positive integer

Number of pyramid levels used during the registration process, specified as the comma-separated pair consisting of 'PyramidLevels' and a positive integer.

Example: 'PyramidLevels', 4 sets the number of pyramid levels to 4.

Data Types: double

Output Arguments

moving_reg — Transformed image

numeric matrix

Transformed image, returned as a matrix. Any fill pixels introduced that do not correspond to locations in the original image are 0.

R_reg — Spatial referencing information associated with output image

spatial referencing object

Spatial referencing information associated with output image, returned as a spatial referencing object of type imref2d or imref3d.

Tips

- Both `imregtform` and `imregister` use the same underlying registration algorithm. `imregister` performs the additional step of resampling `moving` to produce the registered output image from the geometric transformation estimate calculated by `imregtform`. Use `imregtform` when you want access to the geometric transformation that relates `moving` to `fixed`. Use `imregister` when you want a registered output image.
- Create an `optimizer` and `metric` with the `imregconfig` function before calling `imregister`. Getting good results from optimization-based image registration usually requires modifying optimizer or metric settings for the pair of images being

registered. The `imregconfig` function provides a default configuration that should only be considered a starting point. For example, if you increase the number of iterations in the optimizer, reduce the optimizer step size, or change the number of samples in a stochastic metric, the registration improves to a point, at the expense of performance. See the output of `imregconfig` for more information on the different parameters that you can modify.

- If the spatial scaling of your images differs by more than 10%, resize them with `imresize` before registering them.
- Use `imshowpair` or `imfuse` to visualize the results of registration.
- You can use `imregister` in an automated workflow to register several images.
- When you have spatial referencing information about the image to be registered, specify the information to `imregister` using spatial referencing objects. This helps `imregister` converge to better results more quickly because scale differences can be taken into account.

See Also

See Also

Apps

Registration Estimator

Functions

`imfuse` | `imregconfig` | `imregcorr` | `imregtform` | `imshowpair` | `imwarp`

Topics

“Create an Optimizer and Metric for Intensity-Based Image Registration”

“Intensity-Based Automatic Image Registration”

Introduced in R2012a

imregtform

Estimate geometric transformation that aligns two 2-D or 3-D images

Syntax

```
tform = imregtform(moving, fixed, transformType, optimizer, metric)
tform = imregtform(moving, Rmoving, fixed, Rfixed, transformType,
optimizer, metric)
tform = imregtform( ____, Name, Value)
```

Description

`tform = imregtform(moving, fixed, transformType, optimizer, metric)` estimates the geometric transformation that aligns the moving image `moving` with the fixed image `fixed`. `transformType` is a character vector that defines the type of transformation to estimate. `optimizer` is an object that describes the method for optimizing the metric. `metric` is an object that defines the quantitative measure of similarity between the images to optimize. The output `tform` is a geometric transformation object that maps `moving` to `fixed`.

`tform = imregtform(moving, Rmoving, fixed, Rfixed, transformType, optimizer, metric)` estimates the geometric transformation where `Rmoving` and `Rfixed` specify the spatial referencing objects associated with the `moving` and `fixed` images. The output `tform` is a geometric transformation object in units defined by the spatial referencing objects `Rmoving` and `Rfixed`.

`tform = imregtform(____, Name, Value)` estimates the geometric transformation using name-value pairs to control aspects of the operation.

Examples

Estimate Transformation Needed for Image Registration

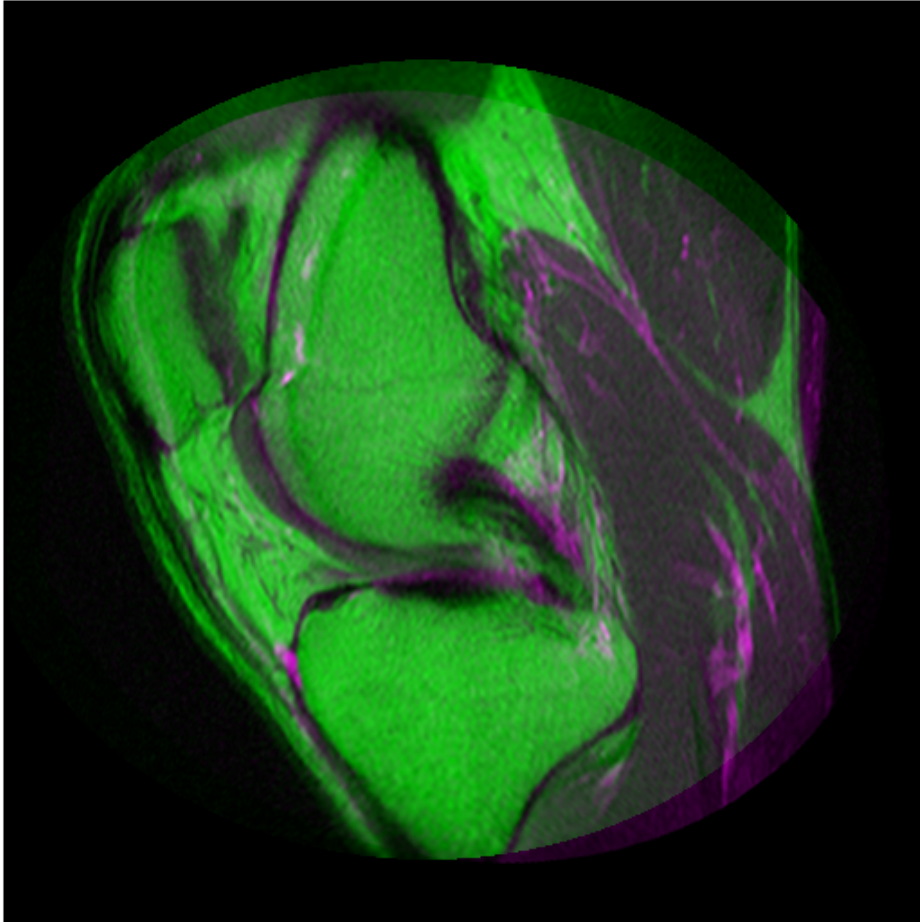
Read two images. This example uses two magnetic resonance (MRI) images of a knee. The fixed image is a spin echo image, while the moving image is a spin echo image

with inversion recovery. The two sagittal slices were acquired at the same time but are slightly out of alignment.

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

View the misaligned images.

```
imshowpair(fixed, moving, 'Scaling', 'joint')
```



Create the optimizer and metric, setting the modality to 'multimodal' since the images come from different sensors.

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

```

optimizer =
    registration.optimizer.OnePlusOneEvolutionary

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

metric =
    registration.metric.MattesMutualInformation

Properties:
    NumberOfSpatialSamples: 500
    NumberOfHistogramBins: 50
    UseAllPixels: 1

```

Tune the properties of the optimizer to get the problem to converge on a global maxima and to allow for more iterations.

```

optimizer.InitialRadius = 0.009;
optimizer.Epsilon = 1.5e-4;
optimizer.GrowthFactor = 1.01;
optimizer.MaximumIterations = 300;

```

Find the geometric transformation that maps the image to be registered (moving) to the reference image (fixed).

```
tform = imregtform(moving, fixed, 'affine', optimizer, metric)
```

```

tform =
    affine2d with properties:

        T: [3×3 double]
        Dimensionality: 2

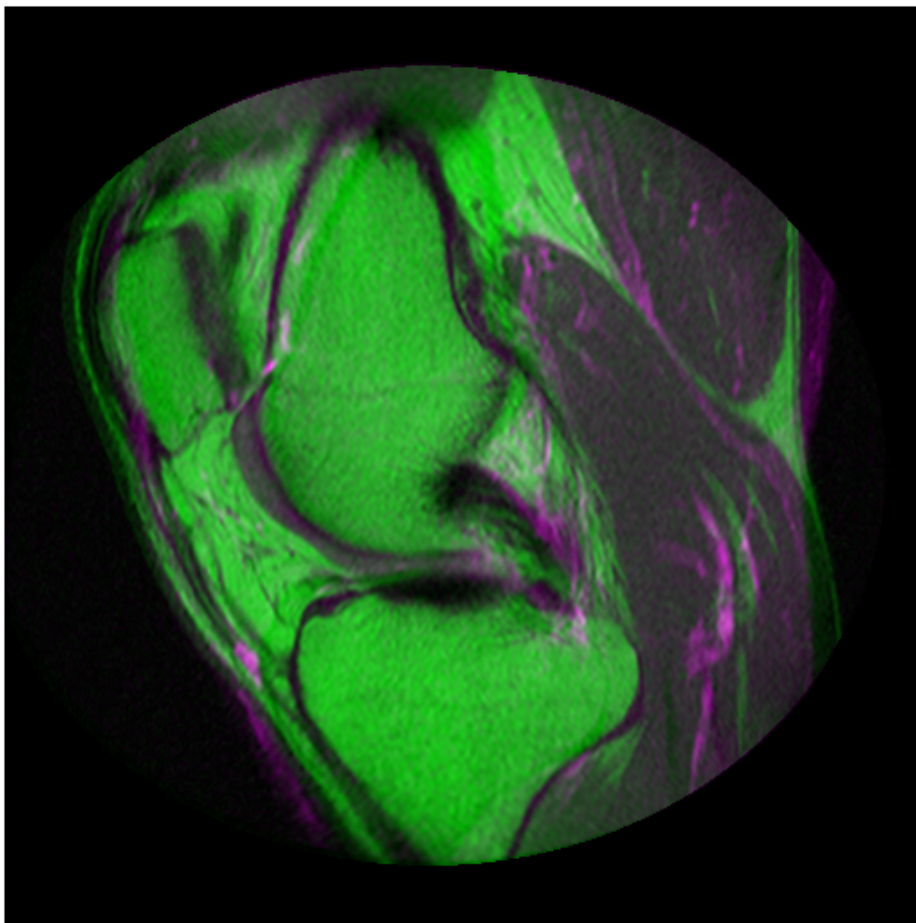
```

Apply the transformation to the image being registered (moving) using the `imwarp` function. The example uses the `'OutputView'` parameter to preserve world limits and resolution of the reference image when forming the transformed image.

```
movingRegistered = imwarp(moving,tform,'OutputView',imref2d(size(fixed)));
```

View the registered images.

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



Input Arguments

moving — Image to be registered

2-D or 3-D grayscale image

Image to be registered, specified as a 2-D or 3-D grayscale image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Rmoving — Spatial referencing information associated with the image to be registered

spatial referencing object

Spatial referencing information associated with the image to be registered, specified as a spatial referencing object of type `imref2d` or `imref3d`.

fixed — Reference image in the target orientation

2-D or 3-D grayscale image

Reference image in the target orientation, specified as a 2-D or 3-D grayscale image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Rfixed — Spatial referencing information associated with the reference (fixed) image

spatial referencing object of type `imref2d` or `imref3d`

Spatial referencing information associated with the reference (fixed) image, specified as a spatial referencing object of type `imref2d` or `imref3d`.

transformType — Geometric transformation to be applied to the image to be registered

`'translation'` | `'rigid'` | `'similarity'` | `'affine'`

Geometric transformation to be applied to the image to be registered, specified as one of the following values:

Value	Description
<code>'translation'</code>	(x,y) translation.
<code>'rigid'</code>	Rigid transformation consisting of translation and rotation.
<code>'similarity'</code>	Nonreflective similarity transformation consisting of translation, rotation, and scale.
<code>'affine'</code>	Affine transformation consisting of translation, rotation, scale, and shear.

The 'similarity' and 'affine' transformation types always involve nonreflective transformations.

optimizer — Method for optimizing the similarity metric

RegularStepGradientDescent or OnePlusOneEvolutionary optimizer object

Method for optimizing the similarity metric, specified as a RegularStepGradientDescent or OnePlusOneEvolutionary optimizer object. To learn more about the creation and properties of optimizer objects, see registration.optimizer.RegularStepGradientDescent and registration.optimizer.OnePlusOneEvolutionary.

metric — Image similarity metric to be optimized during registration

MeanSquares or MattesMutualInformation metric object

Image similarity metric to be optimized during registration, specified as a MeanSquares or MattesMutualInformation metric object. To learn more about the creation and properties of metric objects, see registration.metric.MeanSquares and registration.metric.MattesMutualInformation.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

Example: 'DisplayOptimization', 1 enables verbose optimization mode.

'DisplayOptimization' — Verbose optimization flag

false (default) | true

Verbose optimization flag, specified as the comma-separated pair consisting of 'DisplayOptimization', and the logical value true or false. Controls whether imregister displays optimization information in the command window during the registration process.

Data Types: logical

'InitialTransformation' — Starting geometric transformation

geometric transformation object

Starting geometric transformation, specified as the comma-separated pair consisting of 'InitialTransformation' and an affine2d or affine3d geometric transformation object.

'PyramidLevels' – Number of multi-level image pyramid levels used during the registration process

3 (default) | positive integer

Number of pyramid levels used during the registration process, specified as the comma-separated pair consisting of 'PyramidLevels' and a positive integer.

Example: 'PyramidLevels',4 sets the number of pyramid levels to 4.

Output Arguments

tform – Geometric transformation

geometric transformation object

Geometric transformation, returned as a geometric transformation object affine2d or affine3d. If the input matrices are 3-D, imregtform returns an affine3d object.

Tips

- When you have spatial referencing information available, it is important to provide this information to imregtform, using spatial referencing objects. This information helps imregtform converge to better results more quickly because scale differences can be considered.
- Both imregtform and imregister use the same underlying registration algorithm. imregister performs the additional step of resampling moving to produce the registered output image from the geometric transformation estimate calculated by imregtform. Use imregtform when you want access to the geometric transformation that relates moving to fixed. Use imregister when you want a registered output image.
- Getting good results from optimization-based image registration usually requires modifying optimizer and/or metric settings for the pair of images being registered. The imregconfig function provides a default configuration that should only be considered a starting point. See the output of the imregconfig for more information on the different parameters that can be modified.

See Also

See Also

Apps

Registration Estimator

Functions

`imregconfig` | `imregister` | `imshowpair` | `imwarp`

Classes

`affine2d` | `affine3d`

Topics

“Create an Optimizer and Metric for Intensity-Based Image Registration”

Introduced in R2013a

imresize

Resize image

Syntax

```
B = imresize(A, scale)
B = imresize(A, [numrows numcols])
[Y, newmap] = imresize(X, map, ___)
___ = imresize( ___, method)
___ = imresize( ___, Name, Value)

gpuarrayB = imresize(gpuarrayA, scale)
gpuarrayB = imresize(gpuarrayA, [numrows numcols])
```

Description

`B = imresize(A, scale)` returns image `B` that is `scale` times the size of `A`. The input image `A` can be a grayscale, RGB, or binary image. If `A` has more than two dimensions, `imresize` only resizes the first two dimensions. If `scale` is from 0 through 1.0, `B` is smaller than `A`. If `scale` is greater than 1.0, `B` is larger than `A`. By default, `imresize` uses bicubic interpolation.

`B = imresize(A, [numrows numcols])` returns image `B` that has the number of rows and columns specified by the two-element vector `[numrows numcols]`.

`[Y, newmap] = imresize(X, map, ___)` resizes the indexed image `X` where `map` is the colormap associated with the image. By default, `imresize` returns a new, optimized colormap (`newmap`) with the resized image. To return a colormap that is the same as the original colormap, use the `'Colormap'` parameter.

`___ = imresize(___, method)` specifies the interpolation method used.

`___ = imresize(___, Name, Value)` returns the resized image where `Name, Value` pairs control various aspects of the resizing operation.

`gpuarrayB = imresize(gpuarrayA, scale)` performs the resize operation on a GPU. The input image and the output image are `gpuArrays`. When used with `gpuArrays`,

`imresize` only supports cubic interpolation and always performs antialiasing. For cubic interpolation, the output image might have some values slightly outside the range of pixel values in the input image. This syntax requires the Parallel Computing Toolbox.

`gpuarrayB = imresize(gpuarrayA, [numrows numcols])` performs the resize operation on a GPU, where `[numrows numcols]` specifies the size of the output image.

Examples

Resize Image Specifying Scale Factor

Read image into the workspace.

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

Resize the image, specifying scale factor and using default interpolation method and antialiasing.

```
J = imresize(I, 0.5);
```

Display the original and the resized image.

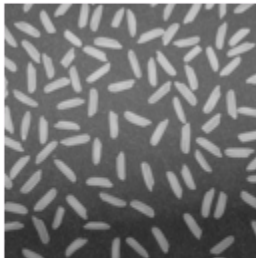
```
figure  
imshow(I)  
title('Original Image')
```

Original Image



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

Resized Image



Resize Image on GPU

Read image into the workspace in a `gpuArray`.

```
I = im2double(gpuArray(imread('rice.png')));
```

Resize the image, performing the operation on a GPU.

```
J = imresize(I, 0.5);
```

Display the original image and the resized image.

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

Resize Image Specifying Scale Factor and Interpolation Method

Read image into the workspace.

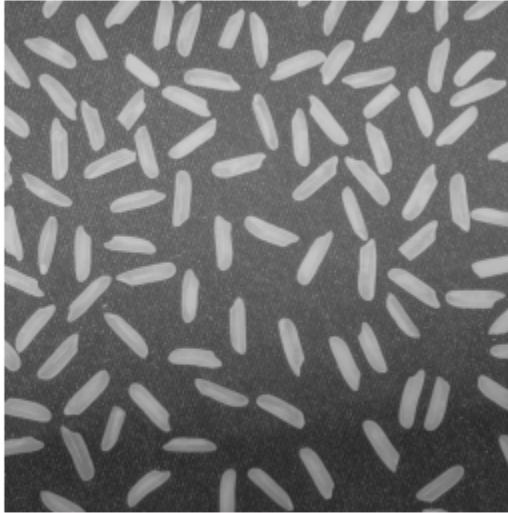
```
I = imread('rice.png');
```

Resize the image, specifying scale factor and the interpolation method.

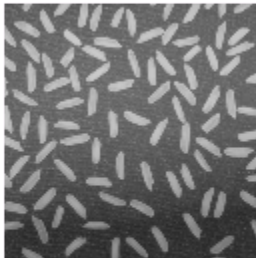
```
J = imresize(I, 0.5, 'nearest');
```

Display the original and the resized image.

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

Original Image

```
figure
imshow(J)
title('Resized Image Using Nearest-Neighbor')
```

Resized Image Using Nearest-Neighbor

Resize Indexed Image

Read image into the workspace.

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

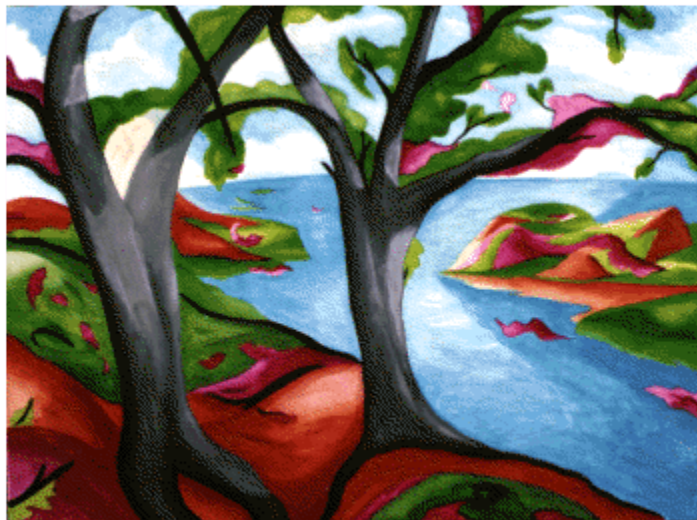
Resize the image, specifying a scale factor. By default, `imresize` returns an optimized color map with the resized indexed image.

```
[Y, newmap] = imresize(X, map, 0.5);
```

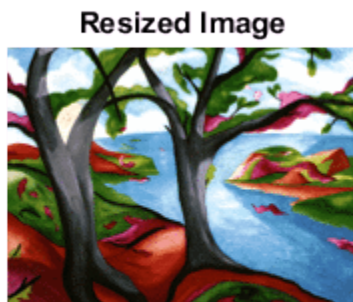
Display the original image and the resized image.

```
figure  
imshow(X, map)  
title('Original Image')
```

Original Image



```
figure  
imshow(Y,newmap)  
title('Resized Image')
```



Resize RGB Image Specifying Size of Output Image

Read image into the workspace.

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

Resize the image, specifying that the output image have 64 rows. Let `imresize` calculate the number of columns necessary to preserve the aspect ratio.

```
RGB2 = imresize(RGB, [64 NaN]);
```

Display the original image and the resized image.

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

Original Image



```
figure
imshow(IMG2)
title('Resized Image')
```


Resized Image



Resize RGB Image on GPU

Read image into the workspace in a `gpuArray`.

```
RGB = gpuArray(im2single(imread('peppers.png')));
```

Resize the image, performing the operation on a GPU.

```
RGB2 = imresize(RGB, 0.5);
```

Display the original image and the resized image.

```
figure
imshow(RGB)
title('Original')
figure
imshow(RGB2)
title('Resized Image')
```

Input Arguments

A — Image to be resized

real, nonsparse numeric or logical array

Image to be resized, specified as a real, nonsparse numeric array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

scale — Resize factor

real, numeric scalar

Resize factor, specified as a real, numeric scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

[numrows numcols] — Row and column dimensions of output image

two-element numeric vector of positive values'

Row and column dimensions of output image, specified as a two-element numeric vector of positive values. Either `numrows` or `numcols` can be NaN, in which case `imresize` computes the number of rows or columns automatically to preserve the image aspect ratio.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

X — Indexed image to be resized

real, nonsparse numeric array

Indexed image to be resized, specified as a real, nonsparse numeric array.

Example: `[X2, newmap] = imresize(X,map,0.75);`

Data Types: `double` | `uint8` | `uint16`

map — Colormap associated with indexed image

m-by-3 numeric array

Colormap associated with indexed image, *m*-by-3 numeric array.

Data Types: `double`

method — Interpolation method

'bicubic' (default) | character vector | two-element cell array

Interpolation method, specified as a character vector or two-element cell array.

The character vector identifies a particular method or named interpolation kernel, listed in the following table.

Method	Description
'nearest'	Nearest-neighbor interpolation; the output pixel is assigned the value of the pixel that the point falls within. No other pixels are considered.

Method	Description
'bilinear'	Bilinear interpolation; the output pixel value is a weighted average of pixels in the nearest 2-by-2 neighborhood
'bicubic'	Bicubic interpolation; the output pixel value is a weighted average of pixels in the nearest 4-by-4 neighborhood
	Note: Bicubic interpolation can produce pixel values outside the original range.
Interpolation Kernel	Description
'box'	Box-shaped kernel
'triangle'	Triangular kernel (equivalent to 'bilinear')
'cubic'	Cubic kernel (equivalent to 'bicubic')
'lanczos2'	Lanczos-2 kernel
'lanczos3'	Lanczos-3 kernel

The a two-element cell array defines a customer interpolation kernel of the form $\{f, w\}$. In this cell array, f is a function handle for a custom interpolation kernel and w is the width of the custom kernel. $f(x)$ must be zero outside the interval $-w/2 \leq x < w/2$. The function handle f can be called with a scalar or a vector input. For user-specified interpolation kernels, the output image can have some values slightly outside the range of pixel values in the input image.

Data Types: char | cell

gpuarrayA — Image to be resized on a GPU

gpuArray

Image to be resized on a GPU, specified as a gpuArray.

Example: `gpuarrayB = imresize(gpuarrayA,0.5);`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: `I2 = imresize(I,0.5,'Antialiasing',false);`

'Antialiasing' — Perform antialiasing when shrinking an image

true | false

Perform antialiasing when shrinking an image, specified as the comma-separated pair consisting of 'Antialiasing' and the logical Boolean value true or false. The default value depends on the interpolation method. If the method is nearest-neighbor ('nearest'), the default is false. For all other interpolation methods, the default is true.

Data Types: logical

'Colormap' — Return optimized colormap

'optimized' (default) | 'original'

Return optimized colormap, specified as the comma-separated pair consisting of 'Colormap' and the character vector 'optimized' or 'original'. (Indexed images only). If set to 'original', the output colormap (newmap) is the same as the input colormap (map). If set to 'optimized', `imresize` returns a new optimized colormap.

Data Types: char

'Dither' — Perform color dithering

true (default) | false

Perform color dithering, specified as the comma-separated pair consisting of 'Dither' and the logical Boolean value true or false. (Indexed images only).

In dithering, you apply a form of noise to the image to randomize quantization error and prevent large-scale patterns.

Data Types: logical

'Method' — Interpolation method

'bicubic' (default) | character vector | cell array

Interpolation method, specified as the comma-separated pair consisting of 'Method' and a character vector or two-element cell array. For details, see `method`.

Data Types: char | cell

'OutputSize' — Size of output image

two-element numeric vector

Size of the output image, specified as the comma-separated pair consisting of 'OutputSize' and a two-element vector of the form [numrows numcols].

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

'Scale' — Resize scale factor

positive numeric scalar | two-element vector of positive values

Resize scale factor, specified as the comma-separated pair consisting of 'Scale' and a positive numeric scalar or two-element vector of positive values.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Output Arguments

B — Resized image

real, nonsparse numeric array

Resized image, returned as a real, nonsparse numeric array, the same class as the input image.

Y — Resized indexed image

real, nonsparse numeric array

Resized indexed image, returned as a real, nonsparse numeric array, the same class as the input image.

newmap — Optimized colormap*m*-by-3 numeric array

Optimized colormap, returned as an *m*-by-3 numeric array.

gpuarrayB — Resized image

gpuArray

Resized image, returned as a gpuArray.

Tips

- The function `imresize` changed in version 5.4 (R2007a). Previous versions of the Image Processing Toolbox used a different algorithm by default. If you need the same results produced by the previous implementation, use the function `imresize_old`.
- There is a slight numerical difference between the results of `imresize` on the CPU and the GPU. These differences occur on the right and bottom borders of the image and barely noticeable to the naked eye.
- If the size of the output image is not an integer, `imresize` does not use the scale specified. `imresize` uses `ceil` when calculating the output image size.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- Syntaxes that support indexed images are not supported, including the named parameters `'Colormap'` and `'Dither'`.
- Custom interpolation kernels are not supported.
- All parameter-value pairs must be compile-time constants.

See Also

See Also

`gpuArray` | `imresize3` | `imrotate` | `imtransform` | `interp2` | `tformarray`

Introduced before R2006a

imresize3

Resize 3-D volumetric intensity image

Syntax

```
B = imresize3(V,scale)
B = imresize3(V,[numrows numcols numplanes])
B = imresize3( ___,method)
B = imresize3( ___,Name,Value)
```

Description

`B = imresize3(V,scale)` returns the volume **B** that is `scale` times the size of `V`. The input volume `V` must be a 3-D volumetric intensity image (called a *volume*). By default, `imresize3` uses cubic interpolation.

`B = imresize3(V,[numrows numcols numplanes])` returns the volume **B** that has the number of rows, columns, and planes specified by the three-element vector `[numrows numcols numplanes]`.

`B = imresize3(___,method)` returns the volume **B**, where `method` specifies the interpolation method used.

`B = imresize3(___,Name,Value)` returns a resized volume where `Name,Value` pairs control aspects of the operation.

Examples

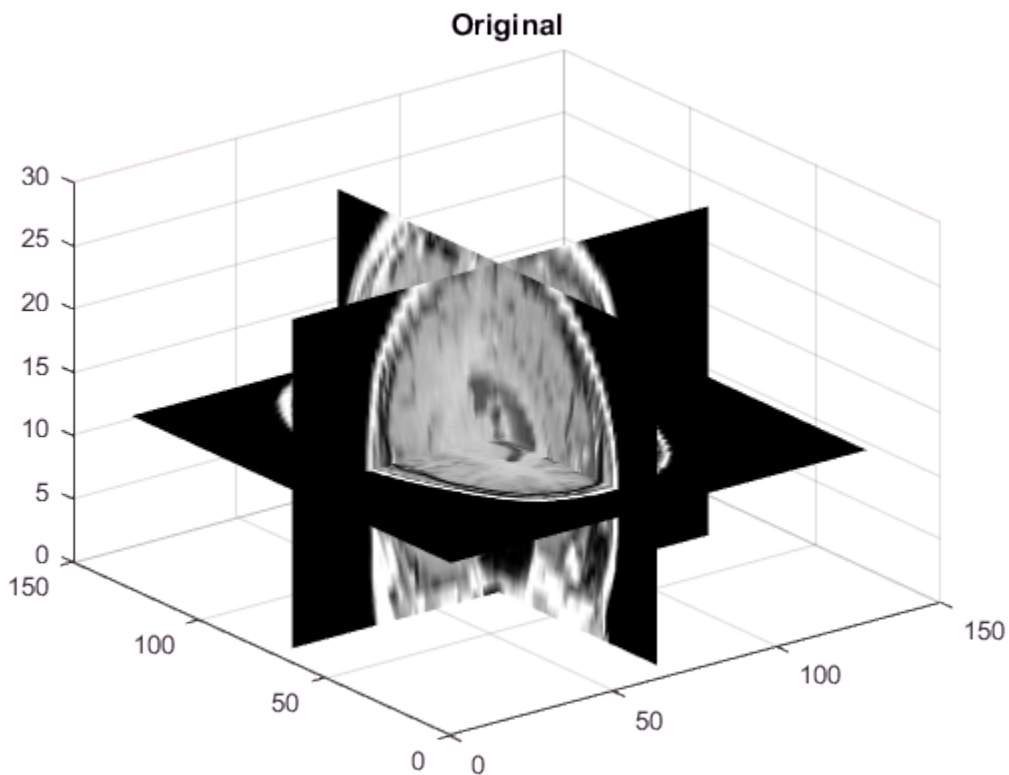
Resize 3-D Volumetric Image

Read MRI volume into the workspace.

```
s = load('mri');
mriVolumeOriginal = squeeze(s.D);
size0 = size(mriVolumeOriginal);
```

Visualize the volume.

```
figure;  
slice(double(mriVolumeOriginal),size0(2)/2,size0(1)/2,size0(3)/2);  
shading interp, colormap gray;  
title('Original');
```



Resize the volume, reducing the size all all dimensions by one-half. This example uses the default interpolation method and antialiasing.

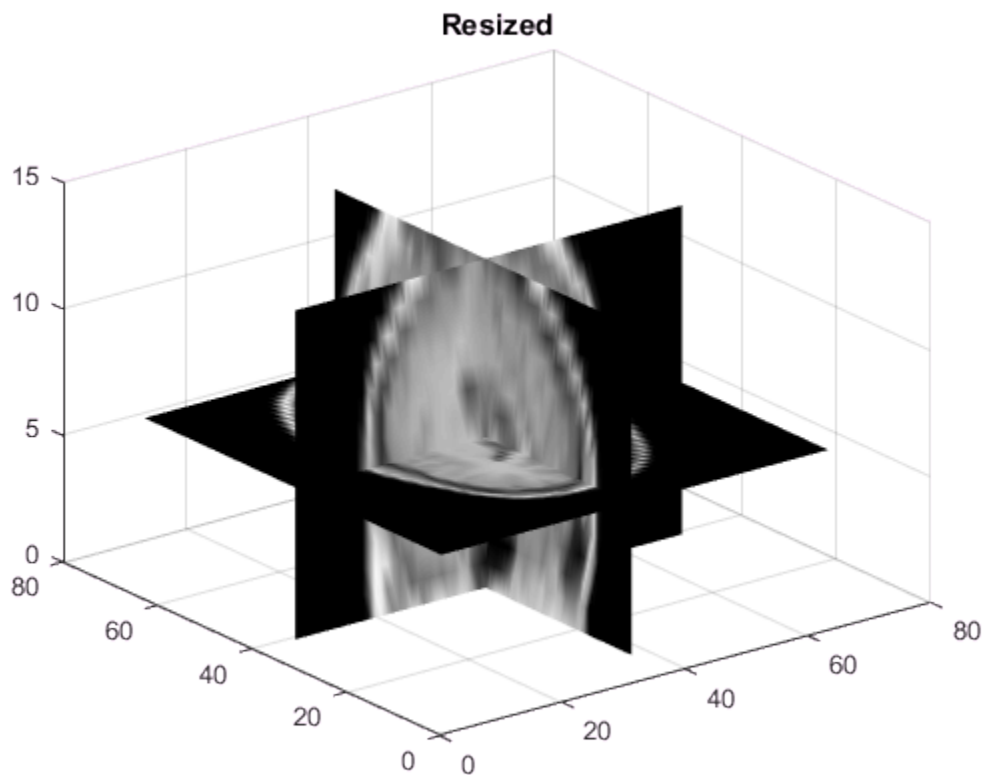
```
mriVolumeResized = imresize3(mriVolumeOriginal, 0.5);  
sizeR = size(mriVolumeResized);
```

Visualize the resized volume.

```
figure;
```



```
slice(double(mriVolumeResized),sizeR(2)/2,sizeR(1)/2,sizeR(3)/2);  
shading interp, colormap gray;  
title('Resized');
```



Input Arguments

V — Volume to be resized

3-D volumetric intensity image

Volume to be resized, specified as a 3-D volumetric intensity image. *V* is numeric or logical array with three dimensions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

scale — Scale factor

numeric scalar

Scale factor, specified as a numeric scalar. To make the resized volume smaller than the input volume, specify a `scale` value from 0 through 1.0. To make the resized volume bigger than the input volume, specify a `scale` value greater than 1.0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

[numrows numcols numplanes] — Size of output volume

three-element vector of real, positive, numeric values

Size of output image, specified as a three-element vector of real, positive, numeric values, in the form `[rows columns planes]`. If you specify one numeric value and the other two values as NaNs, `imresize3` computes the other two elements automatically to preserve the aspect ratio.

Data Types: `single` | `double`

method — Interpolation method

'cubic' (default) | 'nearest' | 'linear' | 'box' | 'triangle' | 'lanczos2' | 'lanczos3'

Interpolation method, specified as a character vector that identifies a general method or a named interpolation kernel, listed in the following table.

Method	Description
'nearest'	Nearest-neighbor interpolation
'linear'	Linear interpolation
'cubic'	Cubic interpolation
	Note: Cubic interpolation can produce pixel values outside the original range.
Interpolation Kernel	Description

Method	Description
'box'	Box-shaped kernel
'triangle'	Triangular kernel (equivalent to 'linear')
'lanczos2'	Lanczos-2 kernel
'lanczos3'	Lanczos-3 kernel

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `mriVolumeResized = imresize3(mristack, 0.5, 'Antialiasing', false);`

'Antialiasing' — Perform antialiasing when shrinking a volume

true | false

Perform antialiasing when shrinking a volume, specified as the comma-separated pair consisting of 'Antialiasing' and the logical Boolean value `true` or `false`. The default value depends on the interpolation method. If the method is nearest-neighbor ('nearest'), the default is `false`. For all other interpolation methods, the default is `true`.

Data Types: logical

'Method' — Interpolation method

'cubic' (default) | character vector

Interpolation method, specified as the comma-separated pair consisting of 'Method' and a character vector. For details, see `method`.

Data Types: char

'OutputSize' — Size of output volume

three-element numeric vector of positive valuse

Size of the output volume, specified as the comma-separated pair consisting of 'OutputSize' and a three-element numeric vector of positive values, of the form [rows cols planes]. For details, see [numrows numcols numplanes].

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

'Scale' — Resize scale factor

numeric scalar or three-element vector of positive values

Resize scale factor, specified as a numeric scalar or three-element vector of positive values. If it is a scalar, the same scale factor is applied to each dimension. If it is a vector, it contains the scale factors for the row, column, and plane dimensions, respectively.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Output Arguments

B — Resized volume

real, nonsparse numeric array

Resized volume, returned as a real, nonsparse numeric array, the same class as the input volume.

See Also

See Also

imresize | imrotate | imrotate3 | imwarp

Introduced in R2017a

imroi

Region-of-interest (ROI) base class

Description

Because the `imroi` class is abstract, creating an instance of the `imroi` class is not allowed.

Methods

`imroi` supports the following methods. Type `methods imroi` to see a complete list.

addNewPositionCallback — Add new-position callback to ROI object

`id = addNewPositionCallback(h, fcn)` adds the function handle `fcn` to the list of new-position callback functions of the ROI object `h`. Whenever the ROI object changes its position each function in the list is called with the syntax:

```
fcn(pos)
```

where `pos` is of the form returned by the object's `getPosition` method.

The return value, `id`, is used only with `removeNewPositionCallback`.

createMask — Create mask within image

`BW = createMask(h)` returns a mask, or binary image, that is the same size as the input image with 1s inside the ROI object `h` and 0s everywhere else. The input image must be contained within the same axes as the ROI.

`BW = createMask(h, h_im)` returns a mask the same size as the image `h_im` with 1s inside the ROI object `h` and 0s outside. This syntax is required when the axes that contain the ROI hold more than one image.

delete — Delete ROI object

`delete(h)` deletes the ROI object `h`

getColor — Get color used to draw ROI object.

`color = getColor(h)` gets the color used to draw the ROI object `h`. The three-element vector `color` specifies an RGB triplet.

getPosition — Return current position of ROI object

`pos = getPosition(h)` returns current position of the ROI object `h`.

getPositionConstraintFcn — Return function handle to current position constraint function

`fcn = getPositionConstraintFcn(h)` returns a function handle `fcn` to the current position constraint function of the ROI object `h`.

removeNewPositionCallback — Remove new-position callback from ROI object

`removeNewPositionCallback(h, id)` removes the corresponding function from the new-position callback list of the ROI object `h`. `id` is the identifier returned by the `addNewPositionCallback` method.

resume — Resume execution of MATLAB command line

`resume(h)` resumes execution of the MATLAB command line. When called after a call to `wait`, `resume` causes `wait` to return an accepted position. The `resume` method is useful when you need to exit `wait` from a callback function.

setColor — Set color used to draw ROI object.

`setColor(h, new_color)` sets the color used to draw the ROI object `h`. `new_color` can be a three-element vector specifying an RGB triplet, or the long or short name of a predefined color, such as `'white'` or `'w'`. See `ColorSpec` for a list of predefined colors.

setConstrainedPosition — Set ROI object to new position

`setConstrainedPosition(h, candidate_position)` sets the ROI object `h` to a new position. The candidate position is subject to the position constraint function. `candidate_position` is of the form expected by the `setPosition` method.

setPositionConstraintFcn — Set position constraint function of ROI object

`setPositionConstraintFcn(h, fcn)` sets the position constraint function of the ROI object `h` to be the specified function handle, `fcn`. Whenever the object is moved because of a mouse drag, the constraint function is called using the syntax:

```
constrained_position = fcn(new_position)
```

where `new_position` is of the form returned by the `getPosition` method. You can use the `makeConstrainToRectFcn` to create this function.

wait — Block MATLAB command line until ROI creation is finished

`accepted_pos = wait(h)` blocks execution of the MATLAB command line until you finish positioning the ROI object `h`. You indicate completion by double-clicking on the ROI object. The returned position, `accepted_pos`, is of the form returned by the object's `getPosition` method.

See Also

`makeConstrainToRectFcn`

Introduced in R2008a

imrotate

Rotate image

Syntax

```
B = imrotate(A,angle)
B = imrotate(A,angle,method)
B = imrotate(A,angle,method,bbox)
gpuarrayB = imrotate(gpuarrayA,method)
```

Description

`B = imrotate(A,angle)` rotates image `A` by `angle` degrees in a counterclockwise direction around its center point. To rotate the image clockwise, specify a negative value for `angle`. `imrotate` makes the output image `B` large enough to contain the entire rotated image. `imrotate` uses nearest neighbor interpolation, setting the values of pixels in `B` that are outside the rotated image to 0 (zero).

`B = imrotate(A,angle,method)` rotates image `A`, using the interpolation method specified by `method`.

`B = imrotate(A,angle,method,bbox)` rotates image `A`, where `bbox` specifies the size of the output image. If you specify 'cropped', `imrotate` makes the output image the same size as the input image. If you specify 'loose', `imrotate` makes the output image large enough to include the entirety of the rotated image.

`gpuarrayB = imrotate(gpuarrayA,method)` perform operation on a graphics processing unit (GPU), where `gpuarrayA` is a `gpuArray` object that contains a grayscale or binary image, and the output image is a `gpuArray` object. This syntax requires the Parallel Computing Toolbox.

Examples

Rotate Image Clockwise for Better Horizontal Alignment

Read image into the workspace.


```
I = fitsread('solarspectra.fts');
```

Convert to grayscale.

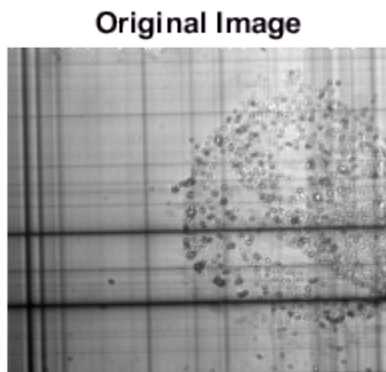
```
I = mat2gray(I);
```

Rotate the image 1 degree clockwise to bring it into better horizontal alignment. The example specified bilinear interpolation and requests that the result be cropped to be the same size as the original image.

```
J = imrotate(I,-1,'bilinear','crop');
```

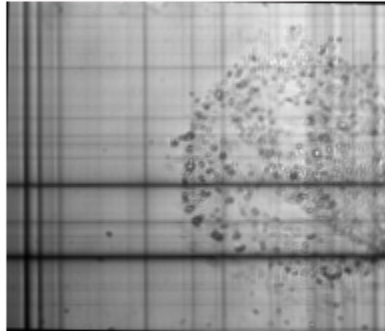
Display the original image and the rotated image.

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



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

Rotated Image



Rotate Image on GPU

Read image into a `gpuArray` object.

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

Rotate the image, performing the operation on the graphics processing unit (GPU).

```
Y = imrotate(X, 37, 'loose', 'bilinear');
```

Display the rotated image.

```
figure; imshow(Y)
```

Input Arguments

A — Image to be rotated

real, nonsparse, numeric or logical array

Image to be rotated, specified as a real, nonsparse, numeric, or logical array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

angle — Amount of rotation in degrees

numeric scalar

Amount of rotation in degrees, specified as a numeric scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

method — Interpolation method

'nearest' (default) | 'bilinear' | 'bicubic'

Interpolation method, specified as one of the following values:

Value	Description
'nearest'	Nearest-neighbor interpolation; the output pixel is assigned the value of the pixel that the point falls within. No other pixels are considered.
'bilinear'	Bilinear interpolation; the output pixel value is a weighted average of pixels in the nearest 2-by-2 neighborhood
'bicubic'	Bicubic interpolation; the output pixel value is a weighted average of pixels in the nearest 4-by-4 neighborhood Note: Bicubic interpolation can produce pixel values outside the original range.

Data Types: char

bbox — Bounding box defining size of output image

'loose' (default) | 'crop'

Bounding box that defines the size of output image, specified as either of the following values:

Value	Description
'crop'	Make output image B the same size as the input image A, cropping the rotated image to fit
'loose'	Make output image B large enough to contain the entire rotated image. B is larger than A.

Data Types: char

gpuarrayA — Image to be rotated

gpuArray

Image to be rotated, specified as a gpuArray.

Output Arguments

B — Rotated image

real, nonsparse, numeric, or logical array

Rotated image, returned as a real, nonsparse, numeric, or logical array.

gpuarrayB — Rotated image

gpuArray

Rotated image, returned as a gpuArray

Tips

- This function changed in version 9.3 (R2015b). Previous versions of the Image Processing Toolbox use different spatial conventions. If you need the same results produced by the previous implementation, use the function `imrotate_old`.
- In some instances, this function takes advantage of hardware optimization for data types `uint8`, `uint16`, `single`, and `double` to run faster.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function

generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

- The `method` and `bbox` arguments must be compile-time constants.

See Also

See Also

`gpuArray` | `imcrop` | `imresize` | `imrotate3` | `imtransform` | `tformarray`

Introduced before R2006a

imrotate3

Rotate 3-D volumetric grayscale image

Syntax

```
B = imrotate3(V,angle,W)
B = imrotate3(V,angle,W,method)
B = imrotate3(V,angle,W,method,bbox)
B = imrotate3( ____,Name,Value)
```

Description

`B = imrotate3(V,angle,W)` rotates the 3-D volumetric grayscale image `V` (referred to as a *volume*) by `angle` degrees counterclockwise around an axis passing through the origin `[0 0 0]`. `W` is a 1-by-3 vector which specifies the direction of the axis of rotation in 3-D space. By default, `imrotate3` uses nearest neighbor interpolation and sets the values of voxels in `B` that are outside the boundaries of the rotated volume to 0.

`B = imrotate3(V,angle,W,method)` rotates the volume `V`, where `method` specifies the interpolation method.

`B = imrotate3(V,angle,W,method,bbox)` rotates the volume `V`, where `bbox` specifies the size of the output volume. If you specify `'cropped'`, `imrotate3` makes the output volume the same size as the input volume. If you specify `'loose'`, `imrotate3` makes the output volume large enough to include the entirety of the rotated volume.

`B = imrotate3(____,Name,Value)` specifies additional parameters that control various aspects of the geometric transformation. Parameter names can be abbreviated.

Examples

Rotate 3-D Volumetric Grayscale Image

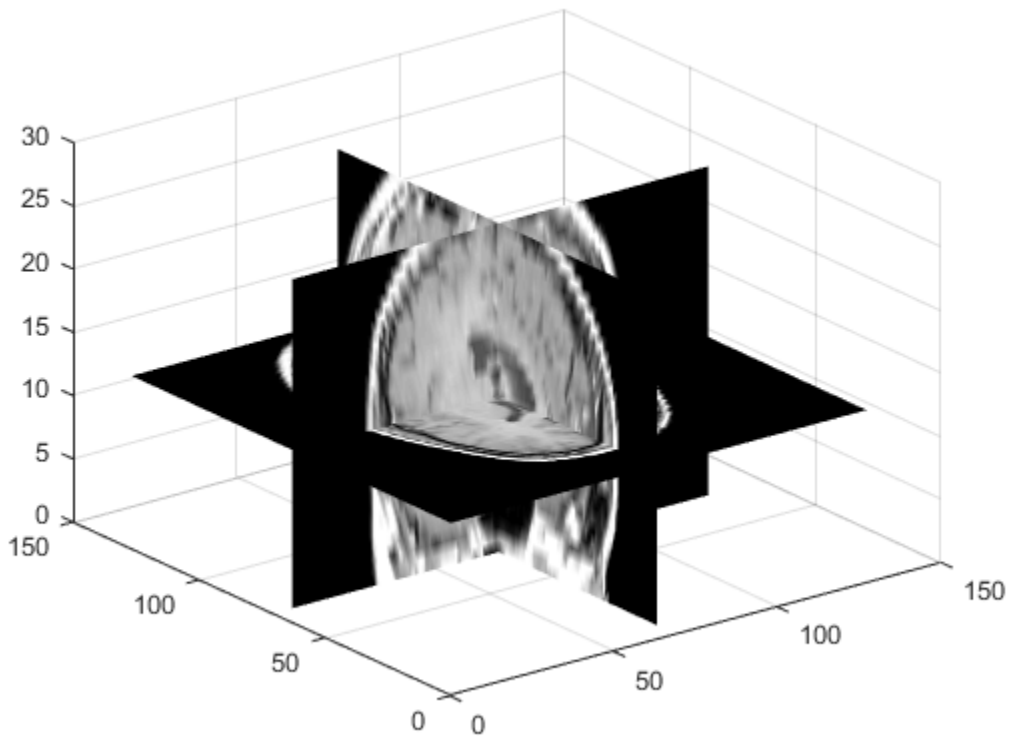
Load a 3-D volumetric grayscale image into the workspace, and display it.

```
s = load('mri');
```

```

mriVolume = squeeze(s.D);
sizeIn = size(mriVolume);
hFigOriginal = figure;
hAxOriginal = axes;
slice(double(mriVolume),sizeIn(2)/2,sizeIn(1)/2,sizeIn(3)/2);
grid on, shading interp, colormap gray

```



Rotate the volume 90 degrees around the Z axis.

```

B = imrotate3(mriVolume,90,[0 0 1], 'nearest', 'loose', 'FillValues',0);

```

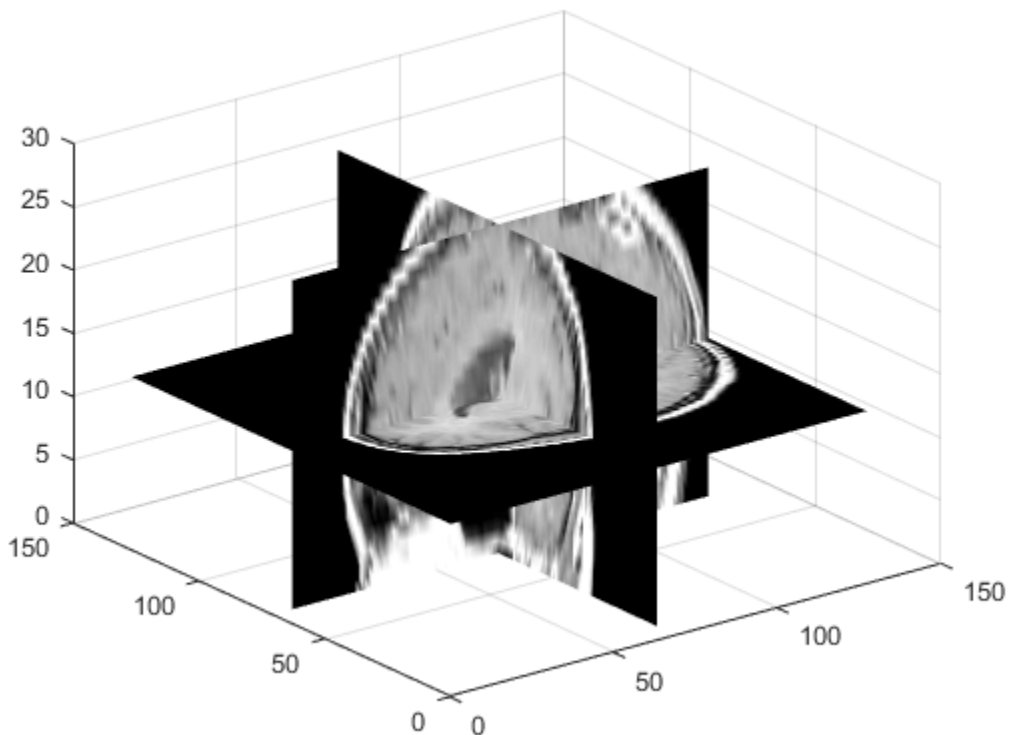
Display the rotated output.

```

figure

```

```
slice(double(B),sizeIn(2)/2,sizeIn(1)/2,sizeIn(3)/2);  
grid on, shading interp, colormap gray
```



Input Arguments

V — Volume to be rotated

3-D volumetric grayscale image

Volume to be rotated, specified as a 3-D volumetric grayscale image.

`imrotate3` assumes that the input volume `V` is centered on the origin `[0 0 0]`. If your volume is not centered on the origin, use `imtranslate` to translate the volume to `[0 0`

0] before using `imrotate3`. You can translate the output volume **B** back to the original position with the opposite translation vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

angle — Rotation angle in degrees

numeric scalar

Rotation angle in degrees, specified as numeric scalar. To rotate the volume clockwise, specify a negative value for `angle`. `imrotate3` makes the output volume **B** large enough to contain the entire rotated 3-D volume.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

W — Direction of the axis of rotation in 3-D space in Cartesian coordinates

1-by-3 vector of numeric values

Direction of the axis of rotation in 3-D space in Cartesian coordinates, specified as a 1-by-3 vector of numeric values.

If you want to specify the direction of the axis of rotation in spherical coordinates, use `sph2cart` to convert values to Cartesian coordinates before passing it to `imrotate3`.

Example: `[0 0 1]` rotate the volume around the Z axis

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

method — Interpolation method

'nearest' (default) | 'linear' | 'cubic'

Interpolation method, specified as one of the following character vectors.

Method	Description
'cubic'	Tricubic interpolation Note: Tricubic interpolation can produce pixel values outside the original range.
'linear'	Trilinear interpolation

Method	Description
'nearest'	Nearest neighbor interpolation

Data Types: char

bbox — Size of the output volume

'loose' (default) | 'crop'

Size of the output volume, specified as either of the following character vectors.

Method	Description
'crop'	Make the output volume the same size as the input volume, cropping the rotated volume to fit.
'loose'	Make the output volume large enough to contain the entire rotated volume. Usually, the rotated volume is larger than the input volume.

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example: `B = imrotate3(V,angle,W,'nearest','loose','FillValues',5);`

'FillValues' — Value used to fill voxels in the output volume that are outside the limits of the rotated volume

0 (default) | numeric scalar

Value used to fill voxels in the output volume that are outside the limits of the rotated volume, specified as the comma-separated pair consisting of 'FillValues' and a numeric scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

Output Arguments

B — Rotated volume

numeric array

Rotated volume, returned as a numeric array the same class as the input volume.

See Also

See Also

[imresize](#) | [imresize3](#) | [imrotate](#) | [imtranslate](#) | [imwarp](#)

Introduced in R2017a

imsave

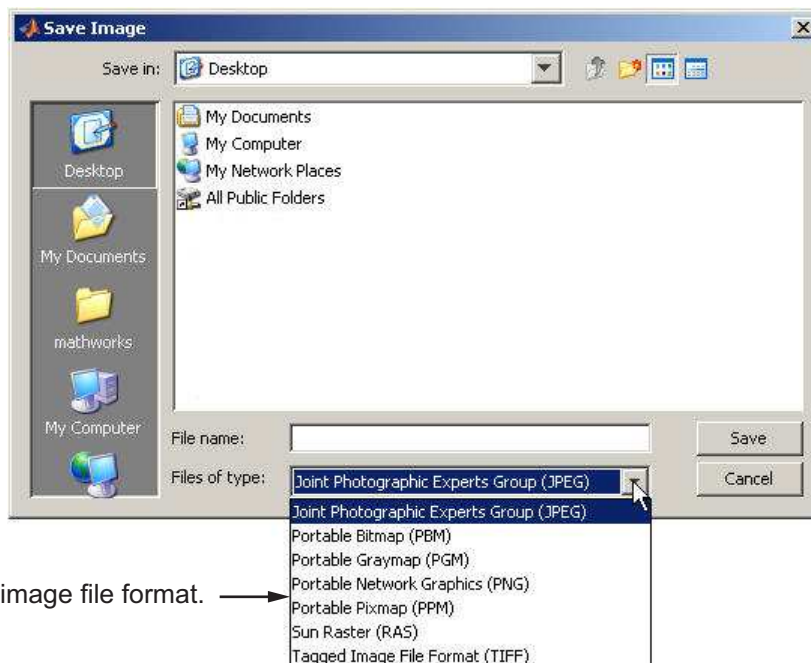
Save Image Tool

Syntax

```
imsave  
imsave(h)  
[filename, user_canceled] = imsave(...)
```

Description

`imsave` creates a Save Image tool in a separate figure that is associated with the image in the current figure, called the target image. The Save Image tool displays an interactive file chooser dialog box (shown below) in which you can specify a path and filename. When you click **Save**, the Save Image tool writes the target image to a file using the image file format you select in the Files of Type menu. `imsave` uses `imwrite` to save the image, using default options.



Select image file format. →

If you specify a filename that already exists, `imsave` displays a warning message. Select **Yes** to use the filename or **No** to return to the dialog to select another filename. If you select **Yes**, the Save Image tool attempts to overwrite the target file.

Note: The Save Image tool is modal; it blocks the MATLAB command line until you respond.

`imsave(h)` creates a Save Image tool associated with the image specified by the handle `h`. `h` can be an image, axes, uipanel, or figure handle. If `h` is an axes or figure handle, `imsave` uses the first image returned by `findobj(h, 'Type', 'image')`.

`[filename, user_canceled] = imsave(...)` returns the full path to the file selected in `filename`. If you press the **Cancel** button, `imsave` sets `user_canceled` to `true` (1); otherwise, `false` (0).

Remarks

In contrast to the **Save as** option in the figure **File** menu, the Save Image tool saves only the image displayed in the figure. The **Save as** option in the figure window File menu saves the entire figure window, not just the image.

Examples

```
imshow peppers.png  
imsave
```

See Also

```
imformats | imgetfile | imputfile | imwrite
```

Introduced in R2007b

imscrollpanel

Scroll panel for interactive image navigation

Syntax

```
hpanel = imscrollpanel(hparent, himage)
```

Description

`hpanel = imscrollpanel(hparent, himage)` creates a scroll panel containing the target image (the image to be navigated). `himage` is a handle to the target image. `hparent` is a handle to the figure or uipanel that will contain the new scroll panel. The function returns `hpanel`, a handle to the scroll panel, which is a uipanel object.

A scroll panel makes an image scrollable. If the size or magnification makes an image too large to display in a figure on the screen, the scroll panel displays a portion of the image at 100% magnification (one screen pixel represents one image pixel). The scroll panel adds horizontal and vertical scroll bars to enable navigation around the image.

`imscrollpanel` changes the object hierarchy of the target image. Instead of the familiar figure->axes->image object hierarchy, `imscrollpanel` inserts several uipanel and uicontrol objects between the figure and the axes object.

API Functions

A scroll panel contains a structure of function handles, called an API. You can use the functions in this API to manipulate the scroll panel. To retrieve this structure, use the `iptgetapi` function, as in the following example.

```
api = iptgetapi(hpanel)
```

This table lists the scroll panel API functions, in the order they appear in the structure.

Function	Description
<code>setMagnification</code>	Sets the magnification of the target image in units of screen pixels per image pixel.

Function	Description
	<p><code>mag = api.setMagnification(new_mag)</code></p> <p>where <code>new_mag</code> is a scalar magnification factor.</p>
<code>getMagnification</code>	<p>Returns the current magnification factor of the target image in units of screen pixels per image pixel.</p> <p><code>mag = api.getMagnification()</code></p> <p>Multiply <code>mag</code> by 100 to convert to percentage. For example if <code>mag=2</code>, the magnification is 200%.</p>
<code>setMagnificationAndCenter</code>	<p>Changes the magnification and makes the point <code>cx, cy</code> in the target image appear in the center of the scroll panel. This operation is equivalent to a simultaneous zoom and recenter.</p> <p><code>api.setMagnificationAndCenter(mag, cx, cy)</code></p>
<code>findFitMag</code>	<p>Returns the magnification factor that would make the target image just fit in the scroll panel.</p> <p><code>mag = api.findFitMag()</code></p>
<code>setVisibleLocation</code>	<p>Moves the target image so that the specified location is visible. Scrollbars update.</p> <p><code>api.setVisibleLocation(xmin, ymin)</code> <code>api.setVisibleLocation([xmin ymin])</code></p>
<code>getVisibleLocation</code>	<p>Returns the location of the currently visible portion of the target image.</p> <p><code>loc = api.getVisibleLocation()</code></p> <p>where <code>loc</code> is a vector <code>[xmin ymin]</code>.</p>
<code>getVisibleImageRect</code>	<p>Returns the current visible portion of the image.</p> <p><code>r = api.getVisibleImageRect()</code></p> <p>where <code>r</code> is a rectangle <code>[xmin ymin width height]</code>.</p>

Function	Description
addNewMagnificationCallback	<p>Adds the function handle <code>fcn</code> to the list of new-magnification callback functions.</p> <pre>id = api.addNewMagnificationCallback(fcn)</pre> <p>Whenever the scroll panel magnification changes, each function in the list is called with the syntax:</p> <pre>fcn(mag)</pre> <p>where <code>mag</code> is a scalar magnification factor.</p> <p>The return value, <code>id</code>, is used only with <code>removeNewMagnificationCallback</code>.</p>
removeNewMagnificationCallback	<p>Removes the corresponding function from the new-magnification callback list.</p> <pre>api.removeNewMagnificationCallback(id)</pre> <p>where <code>id</code> is the identifier returned by <code>addNewMagnificationCallback</code>.</p>
addNewLocationCallback	<p>Adds the function handle <code>fcn</code> to the list of new-location callback functions.</p> <pre>id = api.addNewLocationCallback(fcn)</pre> <p>Whenever the scroll panel location changes, each function in the list is called with the syntax:</p> <pre>fcn(loc)</pre> <p>where <code>loc</code> is <code>[xmin ymin]</code>.</p> <p>The return value, <code>id</code>, is used only with <code>removeNewLocationCallback</code>.</p>

Function	Description
removeNewLocationCallback	<p>Removes the corresponding function from the new-location callback list.</p> <pre>api.removeNewLocationCallback(id)</pre> <p>where <code>id</code> is the identifier returned by <code>addNewLocationCallback</code>.</p>
replaceImage	<pre>api.replaceImage(...,PARAM1,VAL1,PARAM2,VAL2,...)</pre> <p>replaces the image displayed in the scroll panel.</p> <pre>api.replaceImage(I) api.replaceImage(BW) api.replaceImage(RGB) api.replaceImage(I,MAP) api.replaceImage(filename)</pre> <p>By default, the new image data is displayed centered, at 100% magnification. The image handle is unchanged.</p> <p>The parameters you can specify include many of the parameters supported by <code>imshow</code>, including 'Colormap', 'DisplayRange', and 'InitialMagnification'. In addition, you can use the 'PreserveView' parameter to preserve the current magnification and centering of the image during replacement. Specify the logical scalar <code>True</code> to preserve current centering and magnification. Parameter names can be abbreviated, and case does not matter.</p>

Note

Scrollbar navigation as provided by `imscrollpanel` is incompatible with the default MATLAB figure navigation buttons (pan, zoom in, zoom out). The corresponding menu items and toolbar buttons should be removed in a custom GUI that includes a scrollable uipanel created by `imscrollpanel`.

When you run `imscrollpanel`, it appears to take over the entire figure because, by default, an `hpanel` object has `'Units'` set to `'normalized'` and `'Position'` set to `[0 0 1 1]`. If you want to see other children of `hparent` while using your new scroll panel, you must manually set the `'Position'` property of `hpanel`.

Examples

Create Scroll Panel with Magnification Box and Overview Tool

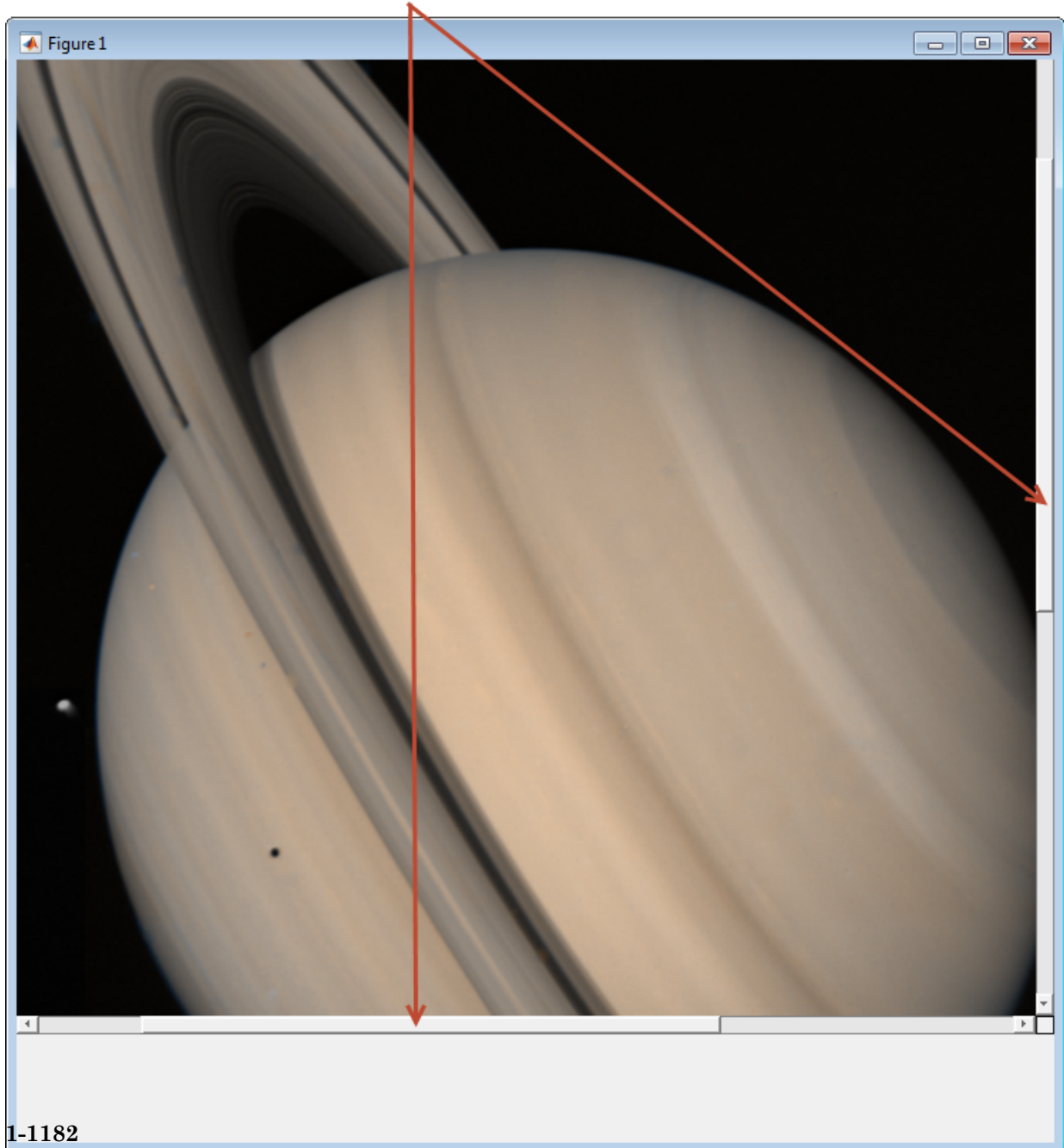
Display an image in a figure. The example suppresses the standard toolbar and menubar in the figure window because these do not work with the scroll panel.

```
hFig = figure('Toolbar','none',...  
             'Menubar','none');  
hIm = imshow('saturn.png');
```

Create a scroll panel to contain the image.

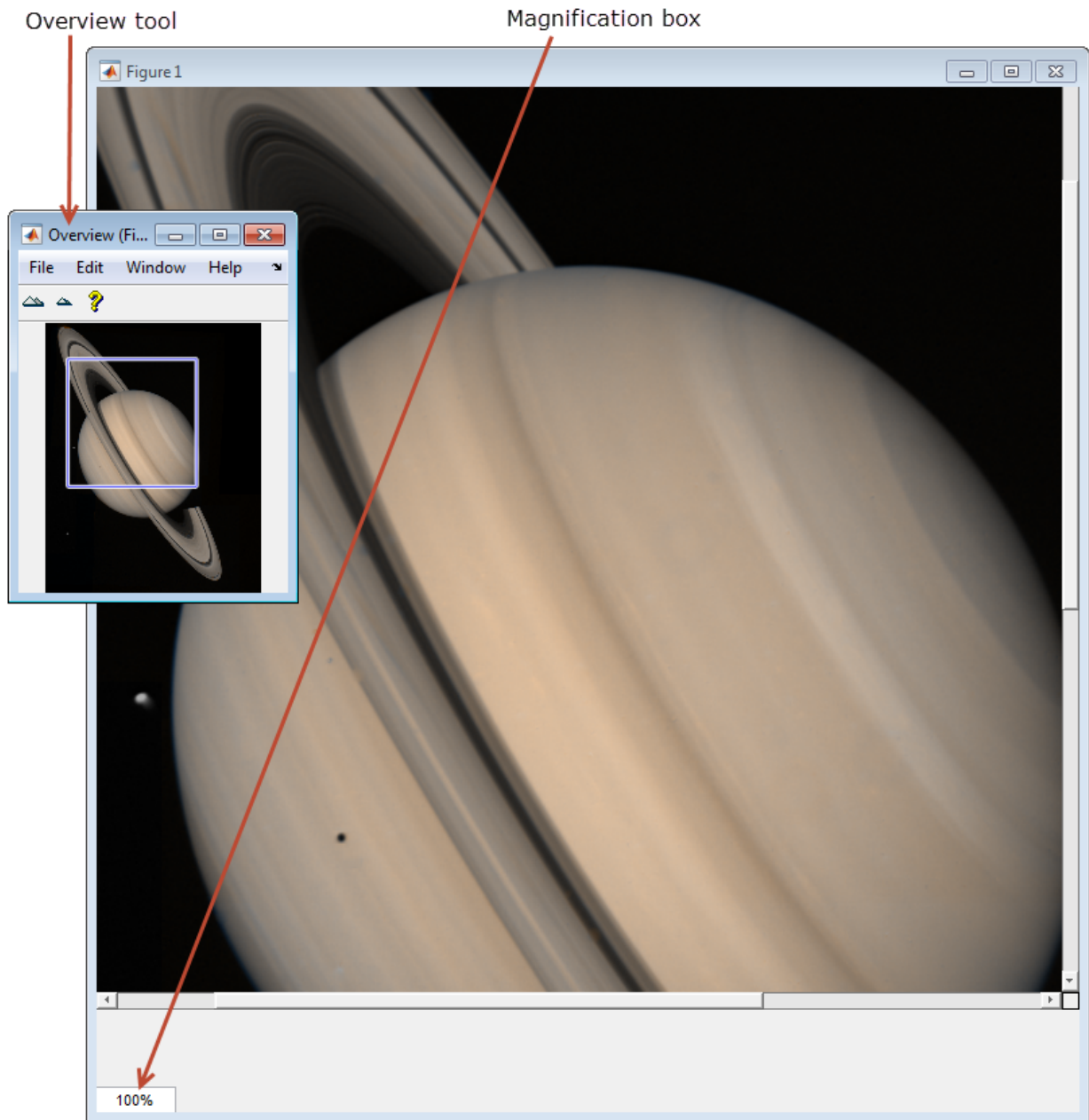
```
hSP = imscrollpanel(hFig,hIm);  
set(hSP,'Units','normalized',...  
     'Position',[0 .1 1 .9])
```

Note addition of scroll bars.



Add a Magnification Box and an Overview tool to the figure.

```
hMagBox = immagbox(hFig,hIm);  
pos = get(hMagBox,'Position');  
set(hMagBox,'Position',[0 0 pos(3) pos(4)])  
imoverview(hIm)
```



Get the scroll panel API so that you can control the view programmatically.

```
api = iptgetapi(hSP);
```

Get the current magnification and position.

```
mag = api.getMagnification()  
r = api.getVisibleImageRect()
```

```
mag =
```

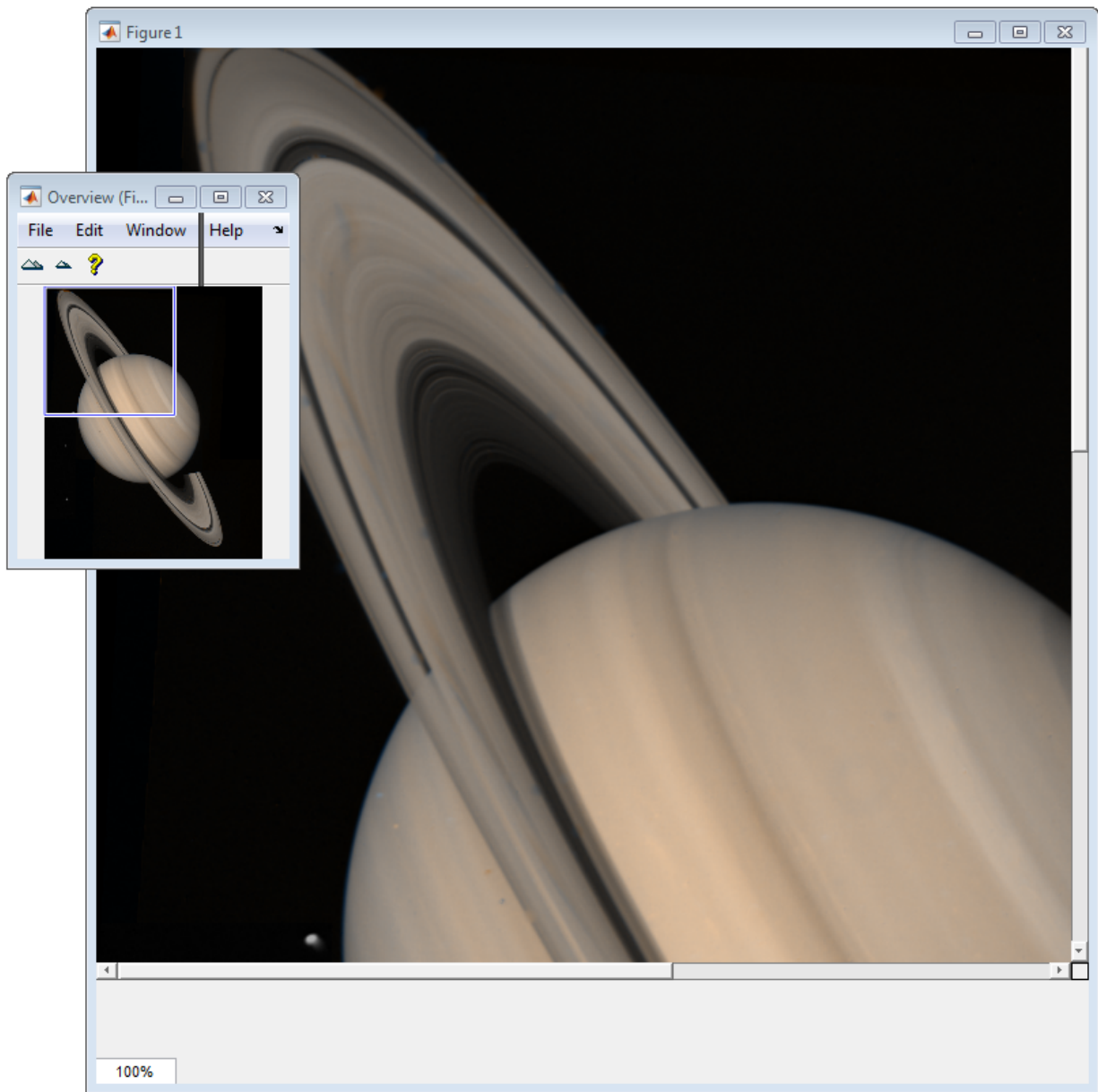
```
1
```

```
r =
```

```
125.0072 201.5646 716.0000 709.0000
```

Use the scroll panel object API function `setVisibleLocation` to view the top left corner of the image.

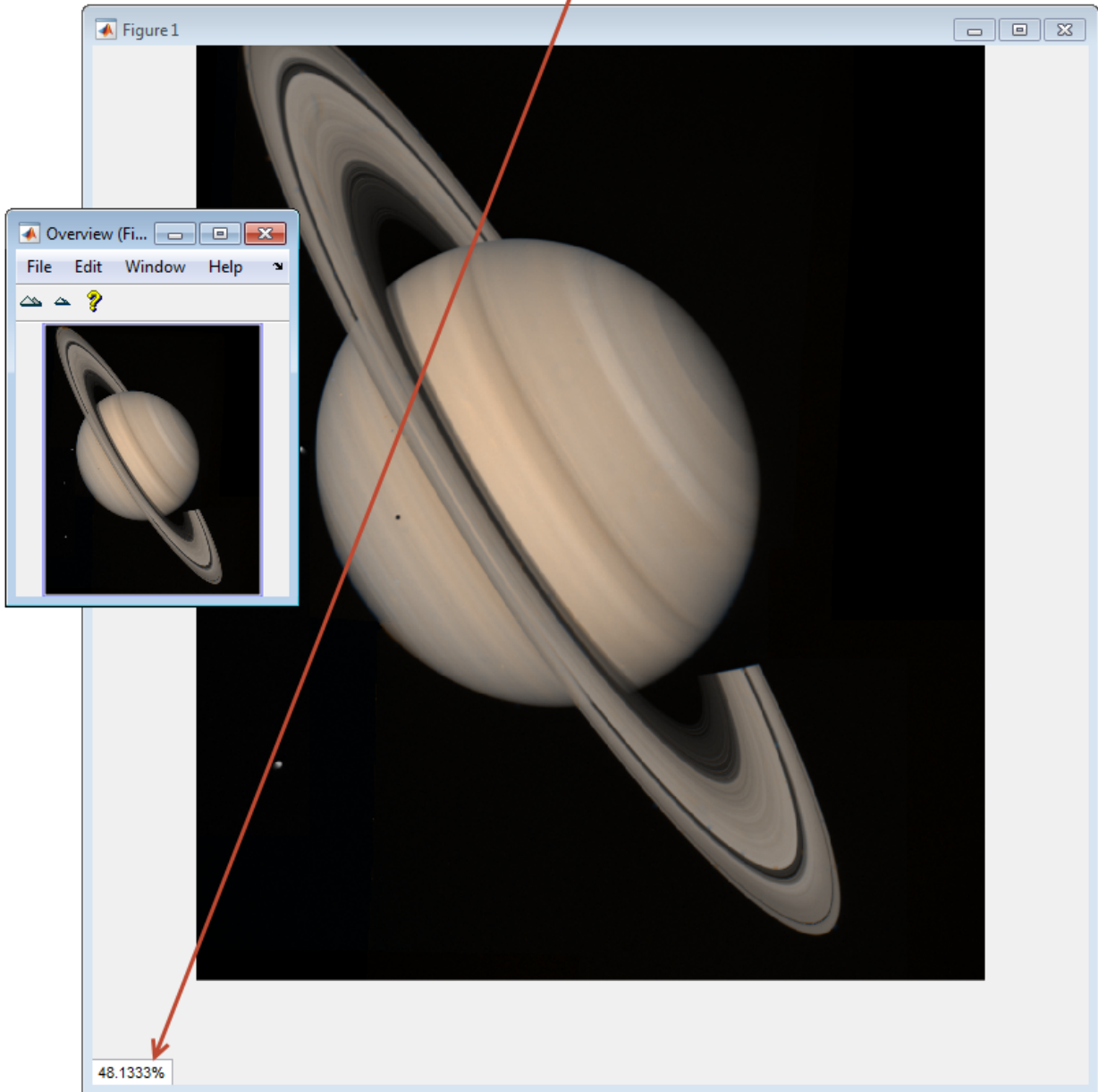
```
api.setVisibleLocation(0.5,0.5)
```



Change the magnification of the image so that the image fits entirely in the scroll panel. In the following figure, note that the scroll bars are no longer visible.

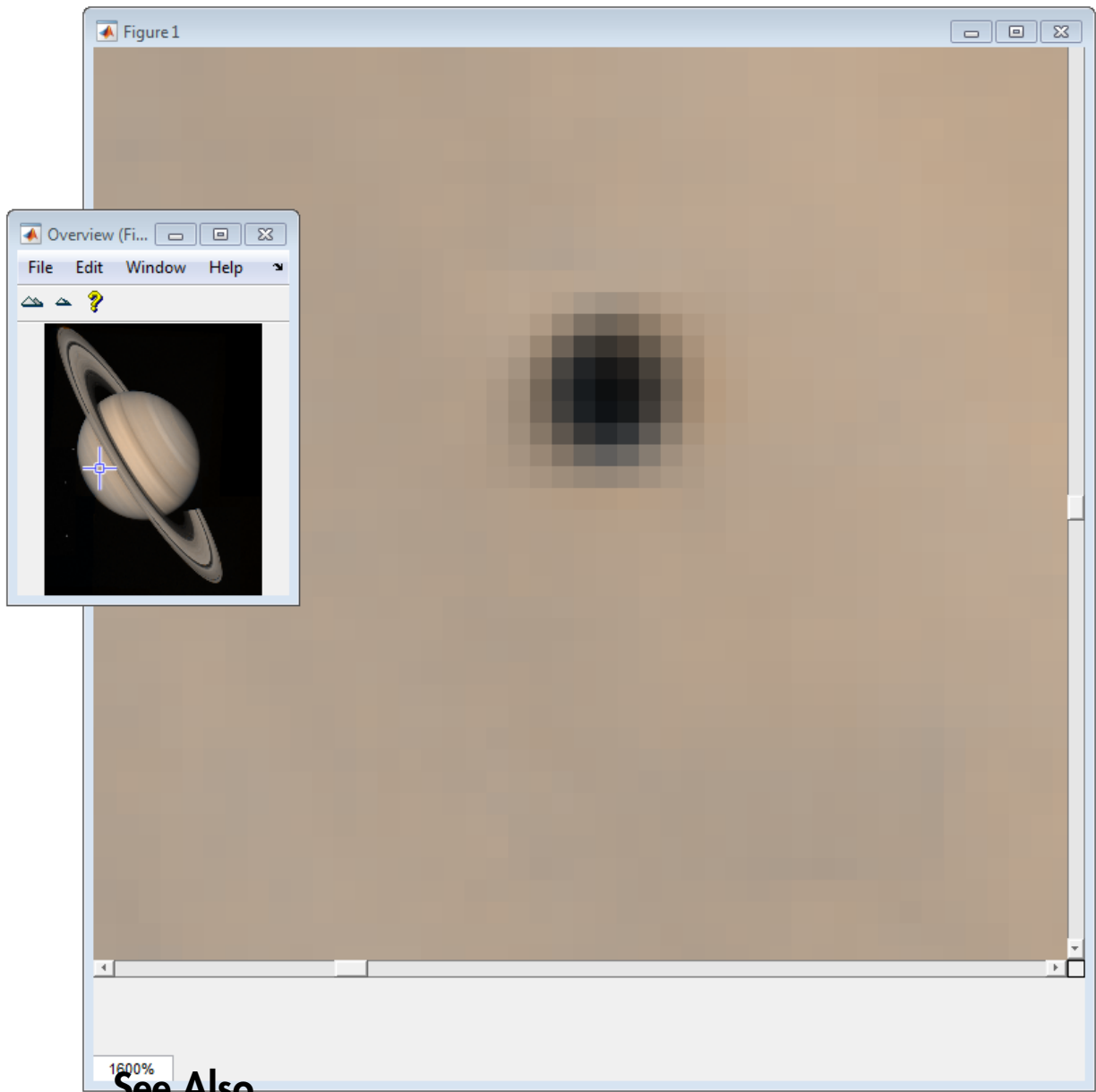
```
api.setMagnification(api.findFitMag())
```

Magnification changed to fit image completely in the figure.



Zoom in to 1600% on the dark spot.

```
api.setMagnificationAndCenter(16,306,800)
```



See Also

`immagbox` | `imoverview` | `imoverviewpanel` | `imtool` | `iptgetapi`

Topics

“Adding Navigation Aids to a GUI”

Introduced before R2006a

imsegfmm

Binary image segmentation using Fast Marching Method

Syntax

```
BW = imsegfmm(W,mask,thresh)
BW = imsegfmm(W,C,R,thresh)
BW = imsegfmm(W,C,R,P,thresh)
[BW,D] = imsegfmm( ___ )
```

Description

`BW = imsegfmm(W,mask,thresh)` returns a segmented image `BW`, which is computed using the Fast Marching Method. The array `W` specifies weights for each pixel. `mask` is a logical array that specifies seed locations. `thresh` is a non-negative scalar in the range `[0 1]` that specifies the threshold level.

`BW = imsegfmm(W,C,R,thresh)` returns a segmented image, with seed locations specified by the vectors `C` and `R`, which contain column and row indices. `C` and `R` must contain values which are valid pixel indices in `W`.

`BW = imsegfmm(W,C,R,P,thresh)` returns a segmented image, with seed locations specified by the vectors `C`, `R`, and `P`, which contain column, row, and plane indices. `C`, `R`, and `P` must contain values which are valid pixel indices in `W`.

`[BW,D] = imsegfmm(___)` returns the normalized geodesic distance map `D` computed using the Fast Marching Method. `BW` is a thresholded version of `D`, where all the pixels that have normalized geodesic distance values less than `thresh` are considered foreground pixels and set to `true`. `D` can be thresholded at different levels to obtain different segmentation results.

Examples

Segment Image Using Fast Marching Method Algorithm

This example shows how to segment an object in an image using Fast Marching Method based on differences in grayscale intensity as compared to the seed locations.

Read image.

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

Original Image



Create mask and specify seed location. You can also use `roipoly` to create the mask interactively.

```
mask = false(size(I));  
mask(170,70) = true;
```

Compute the weight array based on grayscale intensity differences.

```
W = graydiffweight(I, mask, 'GrayDifferenceCutoff', 25);
```

Segment the image using the weights.

```
thresh = 0.01;
```

```
[BW, D] = imsegfmm(W, mask, thresh);  
figure  
imshow(BW)  
title('Segmented Image')
```



You can threshold the geodesic distance matrix D using different thresholds to get different segmentation results.

```
figure  
imshow(D)  
title('Geodesic Distances')
```


Geodesic Distances



Segment Object in Volume Based on Intensity Differences

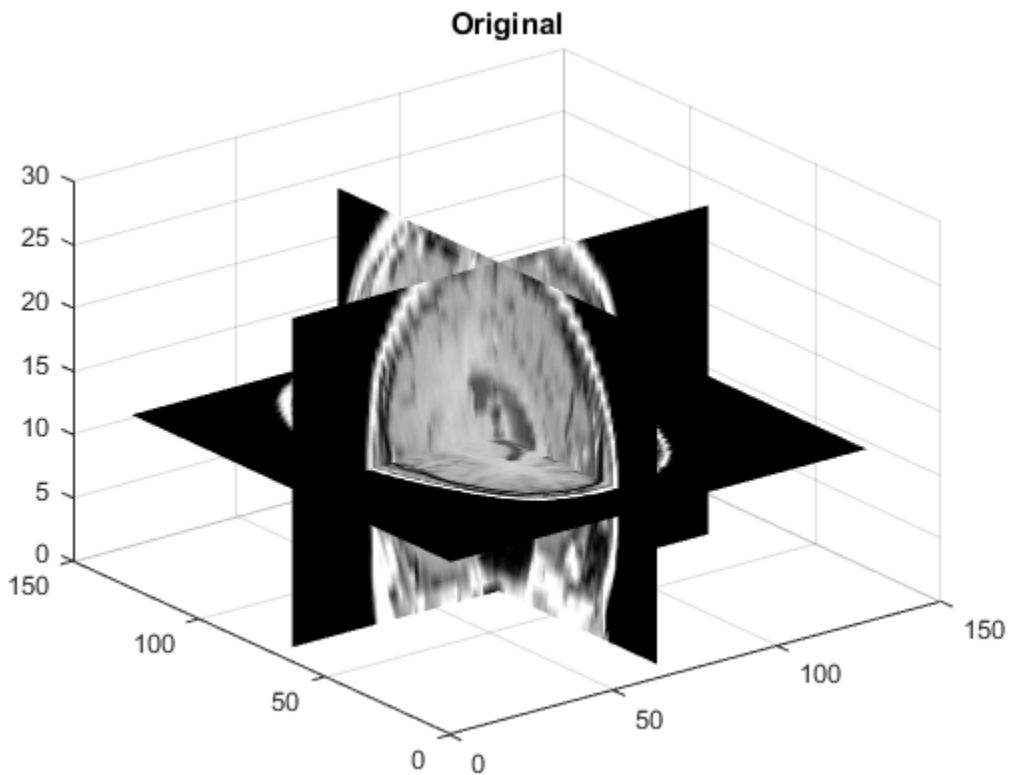
This example segments the brain from MRI data of the human head.

Load the MRI data.

```
load mri
V = squeeze(D);
```

Visualize the data.

```
size0 = size(V);
figure;
slice(double(V),size0(2)/2,size0(1)/2,size0(3)/2);
shading interp, colormap gray;
title('Original');
```



Set the seed locations.

```
seedR = 75;  
seedC = 60;  
seedP = 10;
```

Compute weights based on grayscale intensity differences.

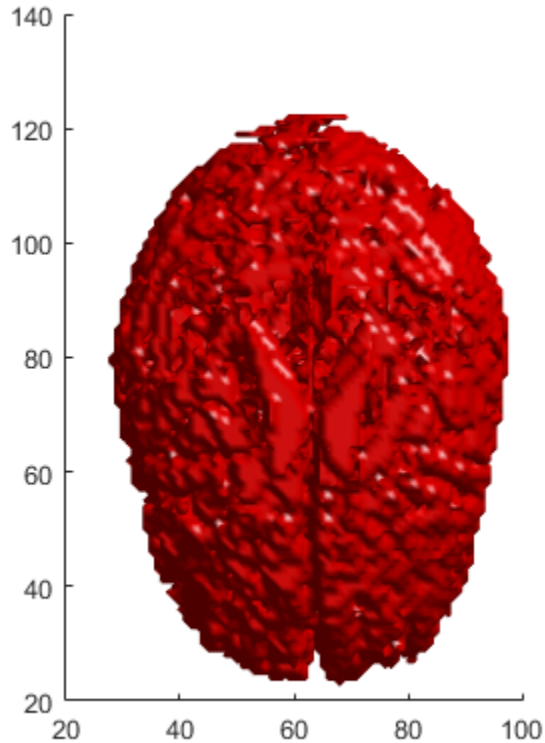
```
W = graydiffweight(V, seedC, seedR, seedP , 'GrayDifferenceCutoff', 25);
```

Segment the image using the weights.

```
thresh = 0.002;  
BW = imsegfmm(W, seedC, seedR, seedP, thresh);
```

Visualize the segmented image using an iso surface.

```
figure;  
p = patch(isosurface(double(BW)));  
p.FaceColor = 'red';  
p.EdgeColor = 'none';  
daspect([1 1 27/64]);  
camlight;  
lighting phong;
```



Input Arguments

W — Weight array

nonsparse, non-negative numeric array

Weight array, specified as a nonsparse, non-negative numeric array. Use the `graydiffweight` or `gradientweight` functions to compute this weight array. High values in `W` identify the foreground (object) and low values identify the background.

Example: `W = graydiffweight(I, mask, 'GrayDifferenceCutoff', 25);`

Data Types: `single` | `double` | `uint8` | `int8` | `int16` | `uint16` | `int32` | `uint32`

mask — Seed locations mask

logical array

Seed locations mask, specified as a logical array, the same size as *W*. Locations where *mask* is true are seed locations. If you used `graydiffweight` to create the weight matrix *W*, it is recommended that you use the same value of *mask* with `imsegfmm` that you used with `graydiffweight`.

Example: `mask = false(size(I)); mask(170,70) = true;`

Data Types: `logical`

thresh — Threshold level used to obtain the binary image

non-negative scalar in the range [0 1]

Threshold level used to obtain the binary image, specified as a non-negative scalar in the range [0 1]. Low values typically result in large foreground regions (logical true) in *BW*, and high values produce small foreground regions.

Example: 0.5

Data Types: `double`

C — Column index of reference pixels

numeric vector

Column index of reference pixels, specified as a numeric vector.

Example: [50 75 93]

Data Types: `double`

R — Row index of reference pixels

numeric vector

Row index of reference pixels, specified as a numeric vector.

Example: [48 71 89]

Data Types: `double`

P — Plane index of reference pixels

numeric vector

Plane index of reference pixels, specified as a numeric vector.

Example:]

Data Types: `double`

Output Arguments

BW — Segmented image

logical array

Segmented image, returned as a logical array of the same size as `W`.

Example:

Data Types: `logical`

D — Normalized geodesic distance map

`double` | `single`

Normalized geodesic distance map, returned as an array of `double` the same size as `W`. If `W` is of class `single`, `D` is of class `single`.

Tips

- `imsegfmm` uses double-precision floating point operations for internal computations for all classes except class `single`. If `W` is of class `single`, `imsegfmm` uses single-precision floating point operations internally.
- `imsegfmm` sets pixels with 0 or NaN weight values to `Inf` in the geodesic distance image `D`. These pixels are part of the background (logical false) in the segmented image `BW`.

References

- [1] Sethian, J. A. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*, Cambridge University Press, 2nd Edition, 1999.

See Also

See Also

`activecontour` | `gradientweight` | `graydiffweight` | `graydist` | `ImageSegmenter`

Introduced in R2014b

imseggeodesic

Segment image into two or three regions using geodesic distance-based color segmentation

Syntax

```
L = imseggeodesic(RGB,BW1,BW2)
L = imseggeodesic(RGB,BW1,BW2,BW3)
[L,P] = imseggeodesic( ___ )
[L,P] = imseggeodesic( ___,Name,Value,... )
```

Description

`L = imseggeodesic(RGB,BW1,BW2)` segments the input image `RGB`, which must be a valid RGB image, returning a segmented binary image with segment labels specified by label matrix `L`. `BW1` and `BW2` are binary images that specify the location of the initial seed regions, called *scribbles*, for the two regions (foreground and background). `imseggeodesic` uses the scribbles specified in `BW1` and `BW2` as representative samples for computing the statistics for their respective regions, which it then uses in segmentation. The scribbles specified by `BW1` and `BW2` (regions that are logical true) should not overlap. The underlying algorithm uses the statistics estimated over the regions marked by the scribbles for segmentation. The greater the number of pixels marked by scribbles, the more accurate the estimation of the region statistics, which typically leads to more accurate segmentation. Therefore, it is a good practice to provide as many scribbles as possible. Typically, provide at least a few hundred pixels as scribbles for each region.

`L = imseggeodesic(RGB,BW1,BW2,BW3)` segments the input image `RGB`, returning a segmented image with three segments (ternary segmentation) with the region labels specified by label matrix `L`. `BW1`, `BW2`, and `BW3` are binary images that specify the location of the initial seed regions or scribbles for the three regions. The scribbles specified by `BW1`, `BW2`, and `BW3` (regions that are logical true) should not overlap.

`[L,P] = imseggeodesic(___)` returns the probability for each pixel belonging to each of the labels in matrix `P`.

`[L,P] = imseggeodesic(___,Name,Value,...)` segments the image using name-value pairs to control aspects of segmentation. Parameter names can be abbreviated.

Examples

Segment Image into Two Regions Using Color Information

Read image into workspace and display it.

```
RGB = imread('yellowlily.jpg');  
imshow(RGB, 'InitialMagnification',50)  
hold on
```



Specify the initial seed regions or "scribbles" for the foreground object, in the form [left_topR left_topC bottom_rightR bottom_rightC].

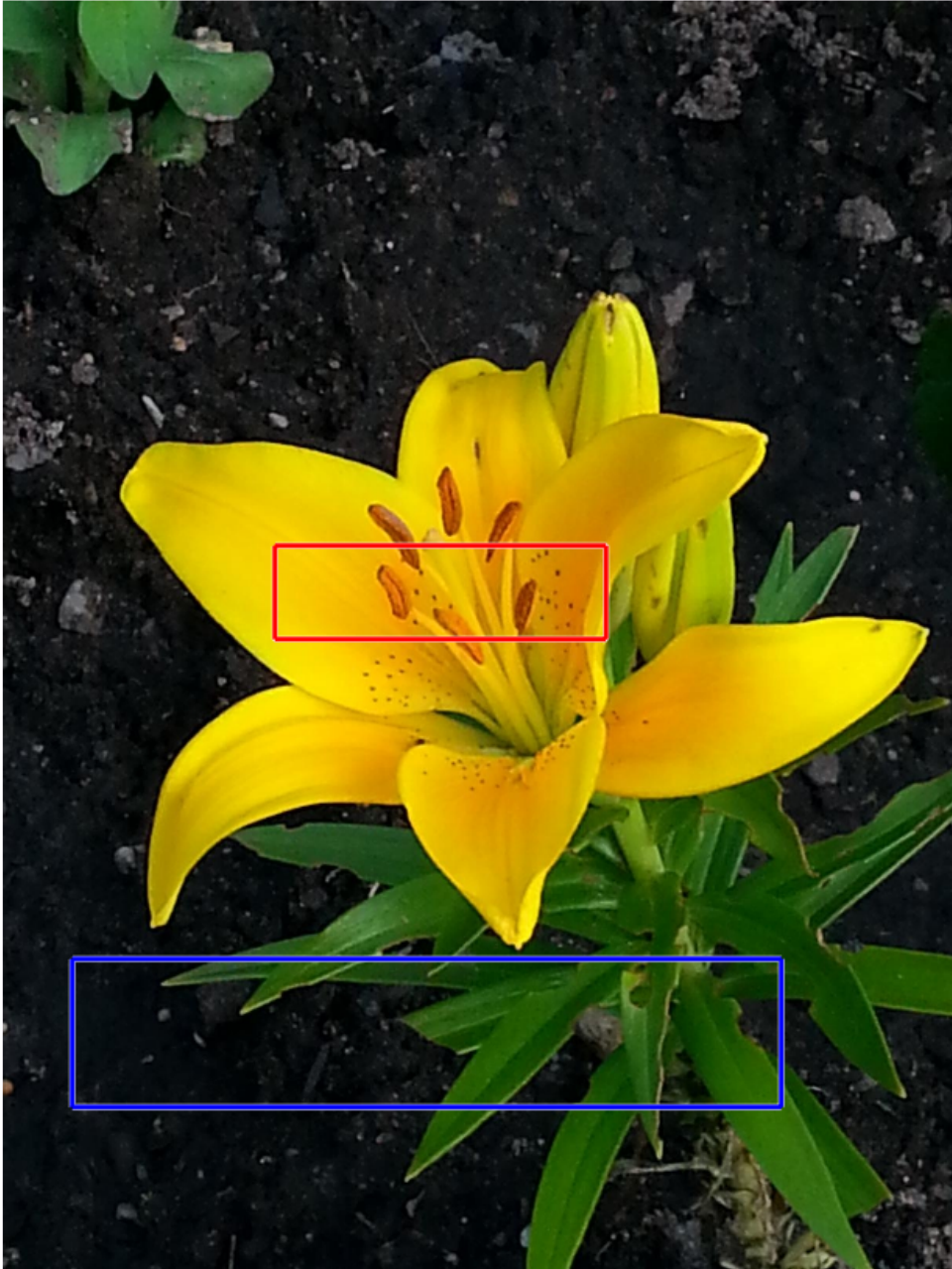
```
bbox1 = [700 350 820 775];  
BW1 = false(size(IMG,1),size(IMG,2));  
BW1(bbox1(1):bbox1(3),bbox1(2):bbox1(4)) = true;
```

Specify the initial seed regions or "scribbles" for the background.

```
bbox2 = [1230 90 1420 1000];  
BW2 = false(size(IMG,1),size(IMG,2));  
BW2(bbox2(1):bbox2(3),bbox2(2):bbox2(4)) = true;
```

Display seed regions. The foreground is in red and the background is blue.

```
visboundaries(BW1,'Color','r');  
visboundaries(BW2,'Color','b');
```



Segment the image.

```
[L,P] = imseggeodesic( RGB, BW1, BW2 );
```

Display results.

```
figure  
imshow(label2rgb(L), 'InitialMagnification', 50)  
title('Segmented image')
```

Segmented image



```
figure
imshow(P(:,:,1),'InitialMagnification', 50)
title('Probability that a pixel belongs to the foreground')
```

Probability that a pixel belongs to the foreground



Segment Image into Three Regions Using Color Information

Read image into the workspace and display it.

```
RGB = imread('yellowlily.jpg');  
imshow(RGB, 'InitialMagnification', 50)  
hold on
```



Creates scribbles for three regions. Note that you can specify the scribbles interactively using tools such as `roipoly`, `imfreehand`, `imrect`, `impoly`, and `imellipse`. Region 1 is the yellow flower. Region 2 is the green leaves. Region 3 is the background.

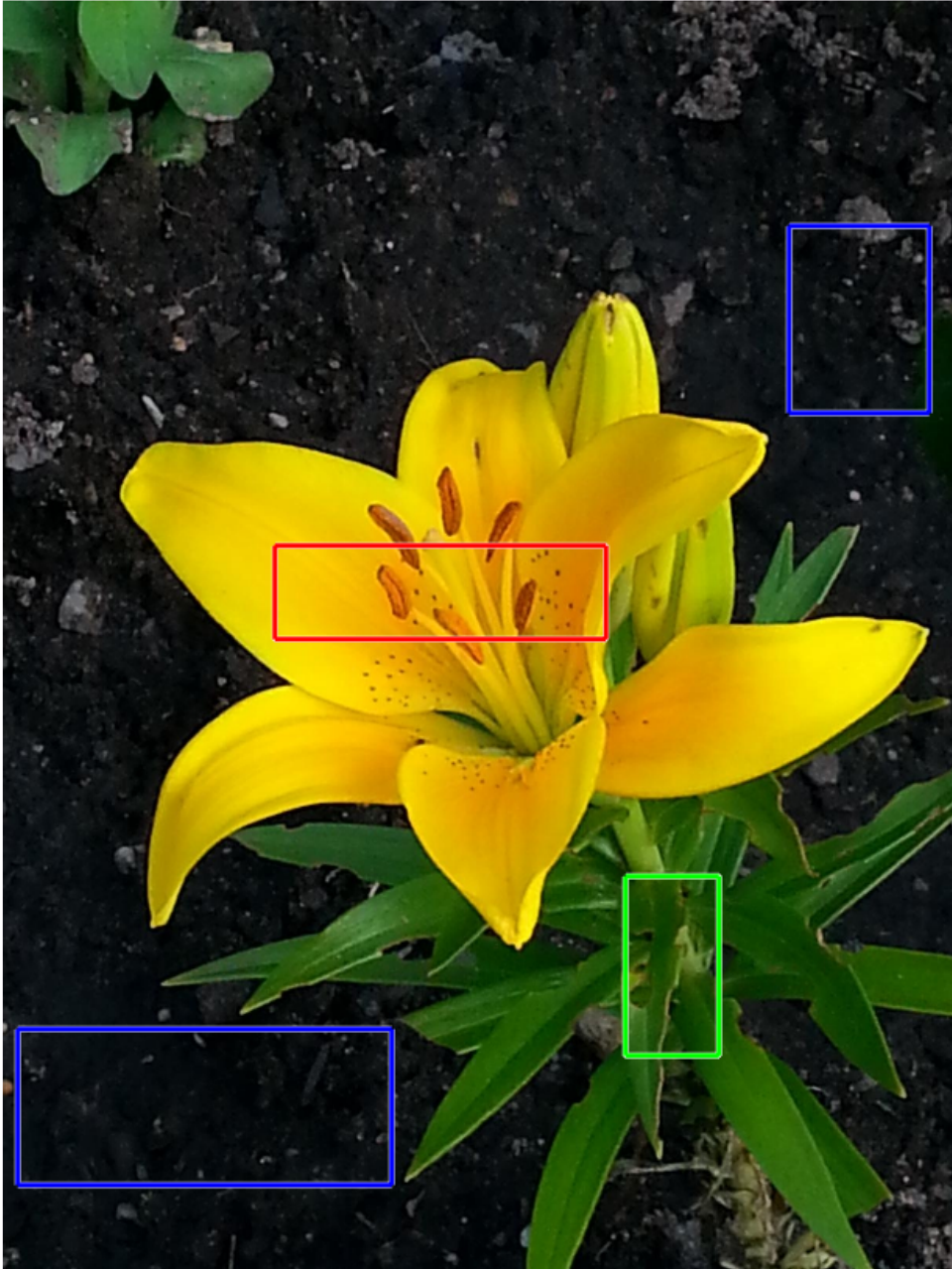
```
region1 = [350 700 425 120]; % [x y w h] format
BW1 = false(size(RGB,1),size(RGB,2));
BW1(region1(2):region1(2)+region1(4),region1(1):region1(1)+region1(3)) = true;

region2 = [800 1124 120 230];
BW2 = false(size(RGB,1),size(RGB,2));
BW2(region2(2):region2(2)+region2(4),region2(1):region2(1)+region2(3)) = true;

region3 = [20 1320 480 200; 1010 290 180 240];
BW3 = false(size(RGB,1),size(RGB,2));
BW3(region3(1,2):region3(1,2)+region3(1,4),region3(1,1):region3(1,1)+region3(1,3)) = true;
BW3(region3(2,2):region3(2,2)+region3(2,4),region3(2,1):region3(2,1)+region3(2,3)) = true;
```

Display the seed regions.

```
visboundaries(BW1,'Color','r');
visboundaries(BW2,'Color','g');
visboundaries(BW3,'Color','b');
```



Segment the image.

```
[L,P] = imseggeodesic( RGB, BW1, BW2, BW3, 'AdaptiveChannelWeighting', true);
```

Display results.

```
figure  
imshow(label2rgb(L), 'InitialMagnification', 50)  
title('Segmented image with three regions')
```

Segmented image with three regions



```
figure
imshow(P(:,:,2),'InitialMagnification', 50)
title('Probability that a pixel belongs to region/label 2')
```

Probability that a pixel belongs to region/label 2



Input Arguments

RGB — Image to be segmented

RGB image

Image to be segmented, specified as a valid RGB image. `imseggeodesic` converts the input RGB image to the YCbCr color space before performing the segmentation.

Example: `RGB = imread('peppers.png');`

Data Types: `double` | `uint8` | `uint16`

BW1 — Scribble image for first region

logical matrix

Scribble image, specified as a logical matrix. `BW1` must have the same number of rows and columns as the input image `RGB`. To specify the scribbles interactively, use `roipoly`, `imfreehand`, `imrect`, `impoly`, or `imellipse`.

Example: `bbox1 = [700 350 820 775]; BW1 = false(size(RGB,1),size(RGB,2)); BW1(bbox1(1):bbox1(3),bbox1(2):bbox1(4)) = true;`

Data Types: `logical`

BW2 — Scribble image for second region

logical matrix

Scribble image, specified as a logical matrix. `BW2` must have the same number of rows and columns as the input image `RGB`. To specify the scribbles interactively, use `roipoly`, `imfreehand`, `imrect`, `impoly`, or `imellipse`.

Example: `bbox2 = [1230 90 1420 1000]; BW2 = false(size(RGB,1),size(RGB,2)); BW2(bbox2(1):bbox2(3),bbox2(2):bbox2(4)) = true;`

Data Types: `logical`

BW3 — Scribble image for third region

logical matrix

Scribble image, specified as a logical matrix. **BW3** must have the same number of rows and columns as the input image **RGB**. To specify the scribbles interactively, use **roipoly**, **imfreehand**, **imrect**, **impoly**, or **imellipse**.

```
Example: bbox3 = [20 1320 480 200; 1010 290 180
240]; BW3 = false(size(RGB,1),size(RGB,2));
BW3(bbox3(1,2):bbox3(1,2)+bbox3(1,4),bbox3(1,1):bbox3(1,1)+bbox3(1,3))
= true;
```

Data Types: `logical`

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

```
Example: [L,P] = imseggeodesic(RGB,BW1,BW2,BW3,
'AdaptiveChannelWeighting', true);
```

'AdaptiveChannelWeighting' — Use adaptive channel weighting

`false` (default) | `true`

Use adaptive channel weighting, specified as a logical scalar. When `true`, **imseggeodesic** weights the channels proportional to the amount of discriminatory information they have that is useful for segmentation (based on the scribbles provided as input). When `false` (the default), **imseggeodesic** weights all the channels equally.

```
Example: [L,P] = imseggeodesic(RGB,BW1,BW2,BW3,
'AdaptiveChannelWeighting', true);
```

Data Types: `logical`

Output Arguments

L — Label matrix

`double` matrix

Label matrix, returned as a `double` matrix where the elements are integer values greater than or equal to 0. Pixels labeled 0 are the background and pixels labeled 1

identify a segmented region. Pixels labeled 2 identify another segmented region in trinary segmentation.

P — Probability a pixel belongs to a labeled region

M-by-*N*-by-2 double matrix (binary segmentation) | *M*-by-*N*-by-3 double matrix (trinary segmentation)

Probability a pixel belongs to a labeled region, specified as an *M*-by-*N*-by-2 matrix for binary segmentation or an *M*-by-*N*-by-3 matrix for trinary segmentation. *M* and *N* are the number of rows and columns in the input image. $P(i, j, k)$ specifies the probability of pixel at location (i, j) belonging to label *k*. **P** is of class `double`.

Tips

- The scribbles for the two (or three) regions should not overlap each other. Each scribble matrix (**BW1**, **BW2**, and **BW3**) should be nonempty, that is, there should be at least one pixel (although the more the better) marked as logical true in each of the scribbles.

Algorithms

`imseggeodesic` uses a geodesic distance-based color segmentation algorithm (similar to [1]).

References

- [1] A. Protiere and G. Sapiro, *Interactive Image Segmentation via Adaptive Weighted Distances*, IEEE Transactions on Image Processing, Volume 16, Issue 4, 2007.

See Also

See Also

`activecontour` | `Color Thresholder` | `imsegfmm` | `visboundaries`

Introduced in R2015a

imsharpen

Sharpen image using unsharp masking

Syntax

```
B = imsharpen(A)
B = imsharpen(A,Name,Value,...)
```

Description

`B = imsharpen(A)` returns an enhanced version of the grayscale or truecolor (RGB) input image `A`, where the image features, such as edges, have been sharpened using the unsharp masking method.

`B = imsharpen(A,Name,Value,...)` sharpens the image using name-value pairs to control aspects of unsharp masking. Parameter names can be abbreviated.

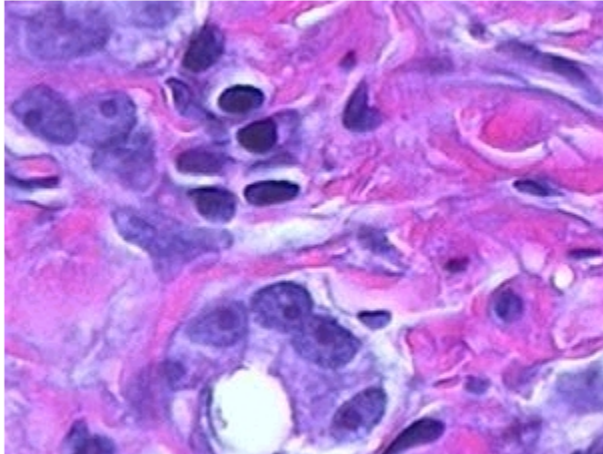
Examples

Sharpen Image

Read an image into the workspace and display it.

```
a = imread('hestain.png');
imshow(a)
title('Original Image');
```

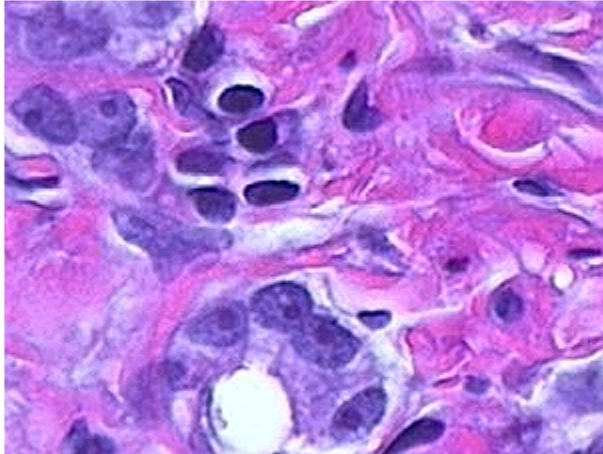
Original Image



Sharpen the image using the `imsharpen` function and display it.

```
b = imsharpen(a);  
figure, imshow(b)  
title('Sharpened Image');
```

Sharpened Image

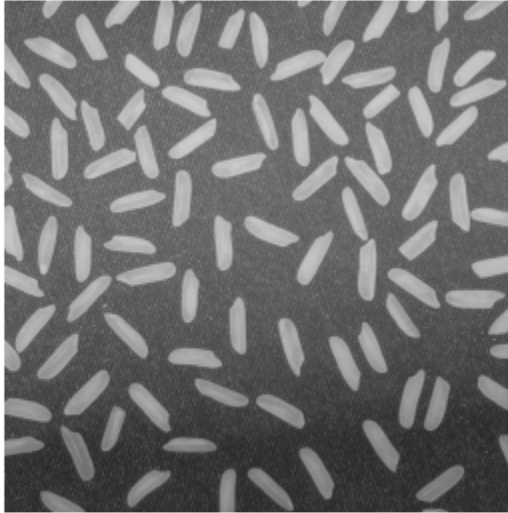


Control the Amount of Sharpening at the Edges

Read an image into the workspace and display it.

```
a = imread('rice.png');  
imshow(a), title('Original Image');
```

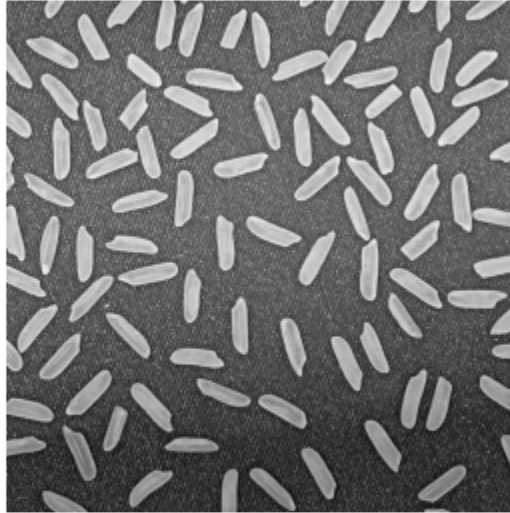
Original Image



Sharpen image, specifying the radius and amount parameters.

```
b = imsharpen(a, 'Radius', 2, 'Amount', 1);  
figure, imshow(b)  
title('Sharpened Image');
```

Sharpened Image



Input Arguments

A — Grayscale or truecolor (RGB) image to be enhanced

nonsparse numeric array

Grayscale or truecolor (RGB) image to be enhanced, specified as a nonsparse, numeric array.

If *A* is a truecolor (RGB) image, `imsharpen` converts the image to the $L^*a^*b^*$ colorspace, applies sharpening to the L^* channel only, and then converts the image back to the RGB colorspace before returning it as the output image *B*.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Radius', 1.5`

'Radius' — Standard deviation of the Gaussian lowpass filter

1 (default) | numeric

Standard deviation of the Gaussian lowpass filter, specified as a numeric value. This value controls the size of the region around the edge pixels that is affected by sharpening. A large value sharpens wider regions around the edges, whereas a small value sharpens narrower regions around edges.

Example: `'Radius', 1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'Amount' — Strength of the sharpening effect

0.8 (default) | numeric

Strength of the sharpening effect, specified as a numeric value. A higher value leads to larger increase in the contrast of the sharpened pixels. Typical values for this parameter are within the range [0 2], although values greater than 2 are allowed. Very large values for this parameter may create undesirable effects in the output image.

Example: `'Amount', 1.2`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'Threshold' — Minimum contrast required for a pixel to be considered an edge pixel

0 (default) | scalar in the range [0 1]

Minimum contrast required for a pixel to be considered an edge pixel, specified as a scalar in the range [0 1]. Higher values (closer to 1) allow sharpening only in high-contrast regions, such as strong edges, while leaving low-contrast regions unaffected. Lower values (closer to 0) additionally allow sharpening in relatively smoother regions of the image. This parameter is useful in avoiding sharpening noise in the output image.

Example: 'Threshold', 0.7

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

B — Sharpened image

nonsparse array the same size and class as the input image.

Image that has been sharpened, returned as a nonsparse array the same size and class as the input image.

Definitions

sharpening

Sharpness is actually the contrast between different colors. A quick transition from black to white looks sharp. A gradual transition from black to gray to white looks blurry. Sharpening images increases the contrast along the edges where different colors meet.

Unsharp masking

The unsharp masking technique comes from a publishing industry process in which an image is sharpened by subtracting a blurred (unsharp) version of the image from itself. Do not be confused by the name of this filter: an unsharp filter is an operator used to sharpen an image.

See Also

See Also

`fspecial` | `imadjust` | `imcontrast`

Introduced in R2013a

imshow

Display image

Syntax

```
imshow(I)
imshow(X,map)
imshow(filename)
imshow(I,[low high])
imshow(___,Name,Value)
```

```
himage = imshow( ___ )
```

```
imshow(I,RI)
imshow(X,RX,map)
imshow(gpuarrayIM, ___ )
```

Description

`imshow(I)` displays image `I` in a figure, where `I` is a grayscale, RGB (truecolor), or binary image. For binary images, `imshow` displays pixels with the value 0 (zero) as black and 1 as white. `imshow` optimizes figure, axes, and image object properties for image display.

`imshow(X,map)` displays the indexed image `X` with the colormap `map`. A colormap matrix can have any number of rows, but it must have exactly 3 columns. Each row is interpreted as a color, with the first element specifying the intensity of red light, the second green, and the third blue. Color intensity can be specified on the interval 0.0 to 1.0.

`imshow(filename)` displays the image stored in the graphics file specified by `filename`.

`imshow(I,[low high])` displays grayscale image `I`, specifying the display range as a two-element vector, `[low high]`. For more information, see the `DisplayRange` parameter.

`imshow(___, Name, Value)` displays an image, using name-value pairs to control aspects of the operation.

`himage = imshow(___)` returns the image object created by `imshow`.

`imshow(I, RI)` displays the image `I` with associated 2-D spatial referencing object `RI`.

`imshow(X, RX, map)` displays the indexed image `X` with associated 2-D spatial referencing object `RX` and colormap `map`.

`imshow(gpuarrayIM, ___)` displays the image contained in a `gpuArray`. This syntax requires the Parallel Computing Toolbox.

Examples

Display Grayscale Image

Display a grayscale image by reading an RGB image into the workspace and converting it to a grayscale image.

Read RGB image into the workspace.

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

Convert the image to grayscale.

```
I = rgb2gray(RGB);
```

Display the grayscale image.

```
imshow(I)
```



Display Image from File

Display an image stored in a file.

```
imshow('peppers.png');
```



Display Indexed Image

Read a sample indexed image, `corn.tif`, into the workspace, and then display it.

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



Change Colormap of Displayed Image

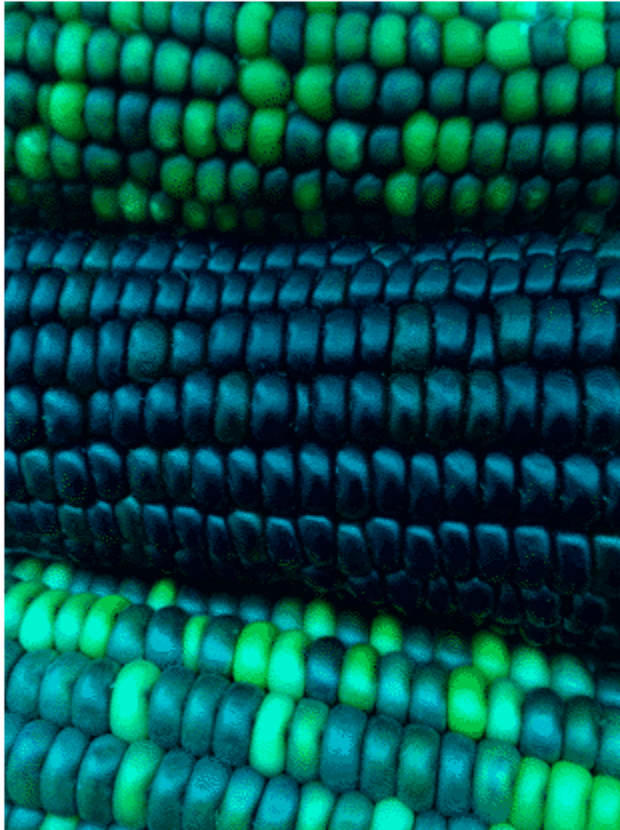
Read a sample indexed image, `corn.tif`, into the workspace, and then display it.

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



Change the colormap for the image using the `colormap` function and specifying the target axes as the first input argument. Use the original colormap without the red component.

```
newmap = map;  
newmap(:,1) = 0;  
colormap(gca,newmap)
```

Display Image Using Associated Spatial Referencing Object

Read image into the workspace.

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

Display the image. Note the axes limits reflect the size of the image.

`imshow(I)`

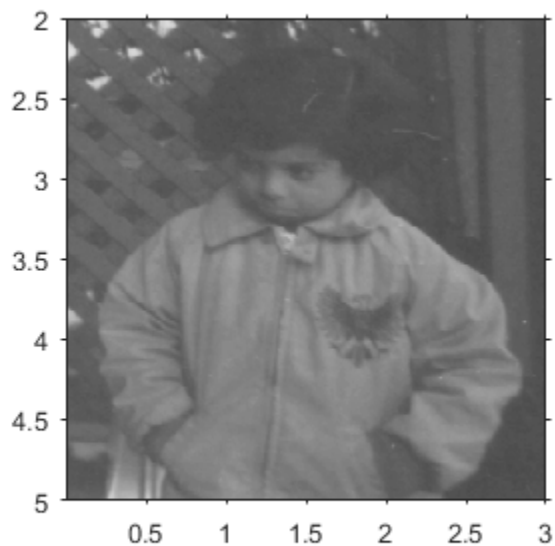


Create a spatial referencing object associated with the image. Use the referencing object to set the x- and y-axes limits in the world coordinate system.

```
RI = imref2d(size(I));  
RI.XWorldLimits = [0 3];  
RI.YWorldLimits = [2 5];
```

Display the image, specifying the spatial referencing object. Note the change to the x- and y-axes limits.

```
imshow(I,RI);
```



Display Image on a GPU

Read image into a `gpuArray`.

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

Display it.

```
figure; imshow(X)
```

Input Arguments

I — Input image

scalar | vector | matrix | m-by-n-by-3 array

Input image, specified as a scalar, vector, or matrix representing a grayscale, RGB, or binary image. Multi-plane image inputs must be RGB images of size *m*-by-*n*-by-3. An RGB image can be `uint8`, `uint16`, `single`, or `double`. A grayscale image can be any numeric data type. A binary image is of class `logical`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

X — Indexed image

2-D array of real numeric values

Indexed image, specified as a 2-D array of real numeric values. The values in *X* are indices into the colormap specified by `map`.

Data Types: `single` | `double` | `uint8` | `logical`

map — Colormap

m-by-3 array

Colormap, specified as an *m*-by-3 array of type `single` or `double` in the range [0 1], or an *m*-by-3 array of type `uint8`. Each row specifies an RGB color value.

Data Types: `single` | `double` | `uint8`

filename — File name

character vector

File name, specified as a character vector. The image must be readable by `imread`. The `imshow` function displays the image, but does not store the image data in the MATLAB workspace. If the file contains multiple images, `imshow` displays the first image in the file.

Example: `imshow('peppers.png')`

Data Types: `char`

[low high] — Grayscale image display range

two-element vector

Grayscale image display range, specified as a two-element vector. For more information, see the 'DisplayRange' name-value pair argument.

Example: `[50 250]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

RI — 2-D spatial referencing object associated with the input image

imref2d object

2-D spatial referencing object associated with input image, specified as an imref2d object.

RX — 2-D spatial referencing object associated with an indexed image

imref2d object

2-D spatial referencing object associated with an indexed image, specified as a imref2d object.

gpuarrayIM — Image to be processed on a graphics processing unit (GPU)

gpuArray object

Image to be processed on a graphics processing unit (GPU), specified as a gpuArray.

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example: `imshow('board.tif','Border','tight')`

'Border' — Figure window border space

'loose' (default) | 'tight'

Figure window border space, specified as the comma-separated pair consisting of 'Border' and either 'tight' or 'loose'. When set to 'loose', the figure window includes space around the image in the figure. When set to 'tight', the figure window does not include any space around the image in the figure.

If the image is very small or if the figure contains other objects besides an image and its axes, `imshow` might use a border regardless of how this parameter is set.

Example: `imshow('board.tif','Border','tight')`

Data Types: char

'Colormap' — Colormap

m-by-3 matrix

Colormap, specified as the comma-separated pair consisting of 'Colormap' and an m-by-3 matrix. `imshow` uses this to set the colormap for the axes. Use this parameter to view grayscale images in false color. If you specify an empty colormap (`[]`), then `imshow` ignores this parameter.

Note: Starting in R2016b, `imshow` changes the colormap for the axes that contains the image instead of the figure.

Example: `newmap = copper; imshow('board.tif','Colormap',newmap)`

Data Types: double

'DisplayRange' — Grayscale image display range

two-element vector

Display range of a grayscale image, specified as a two-element vector of the form `[low high]`. The `imshow` function displays the value `low` (and any value less than `low`) as black, and it displays the value `high` (and any value greater than `high`) as white. Values between `low` and `high` are displayed as intermediate shades of gray, using the default number of gray levels. If you specify an empty matrix (`[]`), `imshow` uses `[min(I(:)) max(I(:))]`. In other words, use the minimum value in `I` as black, and the maximum value as white.

Note: Including the parameter name is optional, except when the image is specified by a file name. The syntax `imshow(I,[low high])` is equivalent to `imshow(I,'DisplayRange',[low high])`. If you call `imshow` with a file name, then you must specify the 'DisplayRange' parameter.

Example: `h = imshow(I,'DisplayRange',[0 80]);`

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

'InitialMagnification' — Initial magnification of image display

100 (default) | numeric scalar | 'fit'

Initial magnification of image display, specified as the comma-separated pair consisting of 'InitialMagnification' and a numeric scalar or 'fit'. If set to 100, then

`imshow` displays the image at 100% magnification (one screen pixel for each image pixel). If set to `'fit'`, then `imshow` scales the entire image to fit in the window.

Initially, `imshow` always displays the entire image. If the magnification value is so large that the image is too big to display on the screen, `imshow` warns and displays the image at the largest magnification that fits on the screen.

If the image is displayed in a figure with its `'WindowStyle'` property set to `'docked'`, then `imshow` warns and displays the image at the largest magnification that fits in the figure.

Note: If you specify the axes position (using `subplot` or `axes`), `imshow` ignores any initial magnification you might have specified and defaults to the `'fit'` behavior.

When you use `imshow` with the `'Reduce'` parameter, the initial magnification must be `'fit'`.

Example: `h = imshow(I, 'InitialMagnification', 'fit');`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char`

'Parent' — Parent axes of image object

axes object

Parent axes of image object, specified as the comma-separated pair consisting of `'Parent'` and an axes object. Use the `'Parent'` name-value argument to build a UI that gives you control of the figure and axes properties.

'Reduce' — Indicator for subsampling

`true` | `false` | `1` | `0`

Indicator for subsampling image, specified as the comma-separated pair consisting of `'Reduce'` and either `true`, `false`, `1`, or `0`. This argument is valid only when you use it with the name of a TIFF file. Use the `Reduce` argument to display overviews of very large images.

Data Types: `logical`

'XData' — X-axis limits of nondefault coordinate system

two-element vector

X-axis limits of nondefault coordinate system, specified as the comma-separated pair consisting of `'XData'` and a two-element vector. This argument establishes a nondefault

spatial coordinate system by specifying the image `XData`. The value can have more than two elements, but `imshow` uses only the first and last elements.

Example: `'XData',[100 200]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'YData' — Y-axis limits of nondefault coordinate system

two-element vector

Y-axis limits of nondefault coordinate system, specified as the comma-separated pair consisting of `'YData'` and a two-element vector. The value can have more than two elements, but `imshow` uses only the first and last elements.

Example: `'YData',[100 200]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`himage` — Image created by `imshow`

image object

Image created by `imshow`, specified as an image object.

Tips

- To change the colormap after you create the image, use the `colormap` command. Specify the axes that contains the image as the first input argument and the colormap you want as the second input argument. For an example, see “Change Colormap of Displayed Image” on page 1-1233.
- You can display multiple images with different colormaps in the same figure using `imshow` with the `subplot` function.
- If you have Image Processing Toolbox, you can use the Image Viewer app as an integrated environment for displaying images and performing common image processing tasks.

- If you have Image Processing Toolbox, you can use the `iptsetpref` function to set toolbox preferences that modify the behavior of `imshow`.
- The `imshow` function is not supported when you start MATLAB with the `-nojvm` option.

See Also

See Also

`image` | `imagesc` | `imfinfo` | `imread` | `imwrite` | `iptsetpref`

Introduced before R2006a

imshowpair

Compare differences between images

Syntax

```
obj = imshowpair(A,B)
obj = imshowpair(A,RA,B,RB)
obj = imshowpair( ____,method)
obj = imshowpair( ____,Name,Value)
```

Description

`obj = imshowpair(A,B)` creates a visualization of the differences between images `A` and `B`. If `A` and `B` are different sizes, `imshowpair` pads the smaller dimensions with zeros on the bottom and right edges so that the two images are the same size. `imshowpair` returns `obj`, an image object.

`obj = imshowpair(A,RA,B,RB)` displays the differences between images `A` and `B`, using the spatial referencing information provided in `RA` and `RB`. `RA` and `RB` are spatial referencing objects.

`obj = imshowpair(____,method)` uses the visualization method specified by `method`.

`obj = imshowpair(____,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments, using any of the previous syntaxes.

Examples

Display Two Images That Differ by Rotation Offset

Display a pair of grayscale images with two different visualization methods, `'diff'` and `'blend'`.

Load an image into the workspace. Create a copy with a rotation offset applied.

```
A = imread('cameraman.tif');  
B = imrotate(A,5,'bicubic','crop');
```

Display the difference of A and B.

```
imshowpair(A,B,'diff')
```



Display a blended overlay of A and B.

```
figure  
imshowpair(A,B,'blend','Scaling','joint')
```



Display Two Spatially Referenced Images with Different Brightness Ranges

Read an image. Create a copy and apply rotation and a brightness adjustment.

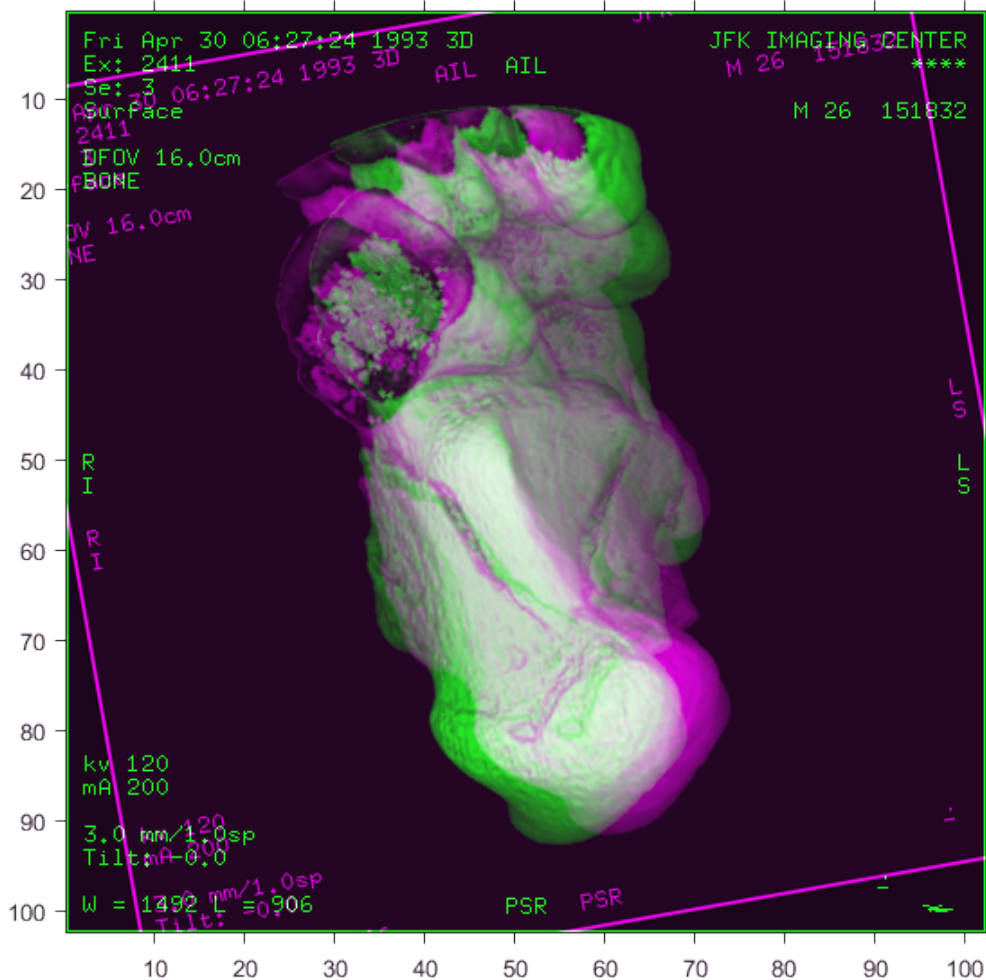
```
A = dicomread('CT-MONO2-16-ankle.dcm');  
B = imrotate(A,10,'bicubic','crop');  
B = B * 0.2;
```

In this example, we know that the resolution of images **A** and **B** is 0.2mm. Provide this information using two spatial referencing objects.

```
RA = imref2d(size(A),0.2,0.2);  
RB = imref2d(size(B),0.2,0.2);
```

Display the images with the default method ('falsecolor') and apply brightness scaling independently to each image. Specify the axes that will be the parent of the image object created by `imshowpair`.

```
figure;
hAx = axes;
imshowpair(A,RA,B,RB, 'Scaling', 'independent', 'Parent', hAx);
```



Input Arguments

A — Image to be displayed

grayscale image | truecolor image | binary image

Image to be displayed, specified as a grayscale, truecolor, or binary image.

B — Image to be displayed

grayscale image | truecolor image | binary image

Image to be displayed, specified as a grayscale, truecolor, or binary image.

RA — Spatial referencing information about an input image

spatial referencing object

Spatial referencing information about an input image, specified as spatial referencing object, of class `imref2d`.

RB — Spatial referencing information about an input image

spatial referencing object

Spatial referencing information about an input image, specified as spatial referencing object, of class `imref2d`.

method — Visualization method to display combined images

'falsecolor' (default) | 'blend' | 'diff' | 'montage'

Visualization method to display combined images, specified as one of the following values.

Value	Description
'falsecolor'	Creates a composite RGB image showing A and B overlaid in different color bands. Gray regions in the composite image show where the two images have the same intensities. Magenta and green regions show where the intensities are different. This is the default method.
'blend'	Overlays A and B using alpha blending.
'checkerboard'	Creates an image with alternating rectangular regions from A and B.

Value	Description
'diff'	Creates a difference image from A and B.
'montage'	Places A and B next to each other in the same image.

Example: `imshowpair(A,B,'montage')` displays A and B next to each other.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Scaling','joint'` scales the intensity values of A and B together as a single data set.

'ColorChannels' — Output color channel for each input image

'green-magenta' (default) | [R G B] | 'red-cyan'

Output color channel for each input image, specified as one of the following values:

[R G B]	A three element vector that specifies which image to assign to the red, green, and blue channels. The R, G, and B values must be 1 (for the first input image), 2 (for the second input image), and 0 (for neither image).
'red-cyan'	A shortcut for the vector [1 2 2], which is suitable for red/cyan stereo anaglyphs.
'green-magenta'	A shortcut for the vector [2 1 2], which is a high contrast option, ideal for people with many kinds of color blindness.

'Parent' — Parent of image object created by imshowpair

axes object

Parent of image object created by `imshowpair`, specified as an axes object.

'Scaling' — Intensity scaling option

'independent' (default) | 'joint' | 'none'

Intensity scaling option, specified as one of the following values:

'independent'	Scales the intensity values of A and B independently from each other.
'joint'	Scales the intensity values in the images jointly as if they were together in the same image. This option is useful when you want to visualize registrations of monomodal images, where one image contains fill values that are outside the dynamic range of the other image.
'none'	No additional scaling.

Output Arguments

obj — Visualization of two images

image object

Visualization of two images, returned as an image object.

Tips

- Use `imfuse` to create composite visualizations that you can save to a file. Use `imshowpair` to display composite visualizations to the screen.

See Also

See Also

`imfuse` | `imregister` | `imshow` | `imtransform`

Introduced in R2012a

imsubtract

Subtract one image from another or subtract constant from image

Syntax

```
Z = imsubtract(X,Y)
```

Description

`Z = imsubtract(X,Y)` subtracts each element in array `Y` from the corresponding element in array `X` and returns the difference in the corresponding element of the output array `Z`. `X` and `Y` are real, nonsparse numeric arrays of the same size and class, or `Y` is a double scalar. The array returned, `Z`, has the same size and class as `X` unless `X` is logical, in which case `Z` is double.

If `X` is an integer array, elements of the output that exceed the range of the integer type are truncated, and fractional values are rounded.

Examples

Subtract Two uint8 Arrays

This example shows how to subtract two `uint8` arrays. Note that negative results are rounded to 0.

```
X = uint8([ 255 0 75; 44 225 100]);  
Y = uint8([ 50 50 50; 50 50 50 ]);  
Z = imsubtract(X,Y)
```

```
Z = 2×3 uint8 matrix
```

```
    205     0    25  
     0    175    50
```

Subtract Image Background

Read a grayscale image into the workspace.

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

Estimate the background.

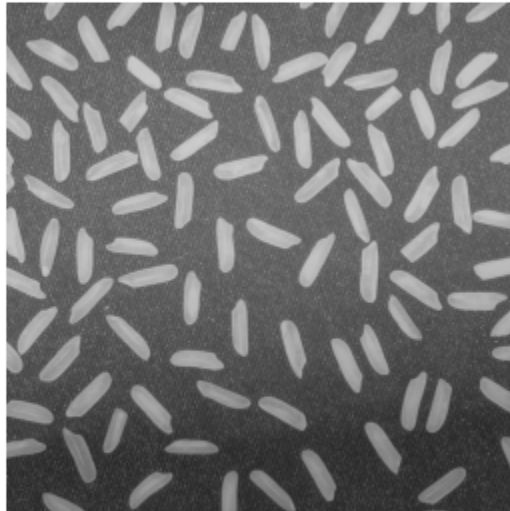
```
background = imopen(I, strel('disk', 15));
```

Subtract the background from the image.

```
J = imsubtract(I, background);
```

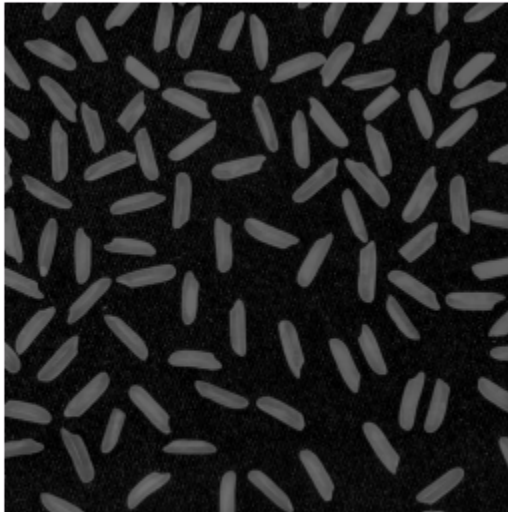
Display the original image and the processed image.

```
imshow(I)
```



figure

```
imshow(J)
```



Subtract a Constant from an Image

Read an image into the workspace.

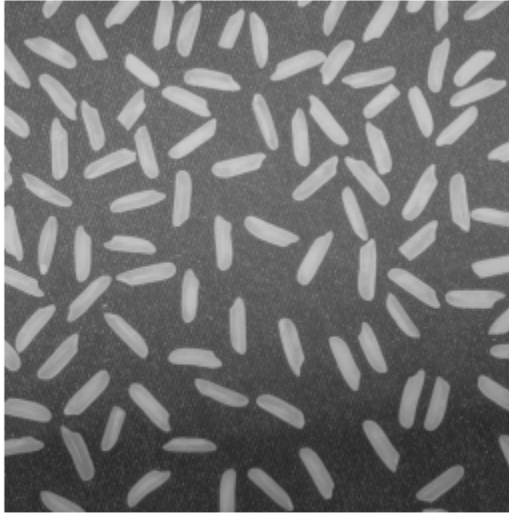
```
I = imread('rice.png');
```

Subtract a constant value from the image.

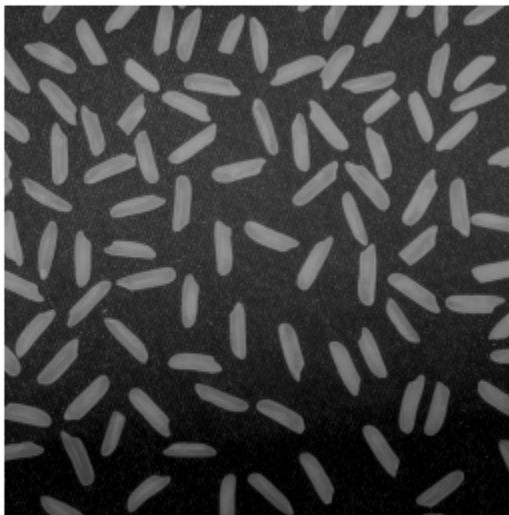
```
J = imsubtract(I,50);
```

Display the original image and the result.

```
imshow(I)
```



```
figure  
imshow(J)
```



See Also

`imabsdiff` | `imadd` | `imcomplement` | `imdivide` | `imlincomb` | `immultiply`

Introduced before R2006a

imtool

Image Viewer app

Syntax

```
imtool
imtool(I)
imtool(I,[low high])
imtool(RGB)
imtool(BW)
imtool(X,map)
imtool(filename)
hfigure = imtool(...)
imtool close all
imtool(...,param1,val1,param2,val2,...)
```

Description

`imtool` opens the Image Viewer app in an empty state. Use the **File** menu options **Open** or **Import from Workspace** to choose an image for display.

`imtool(I)` displays the grayscale image `I` in the Image Viewer.

`imtool(I,[low high])` displays the grayscale image `I` in the Image Viewer, specifying the display range for `I` in the vector `[low high]`. The value `low` (and any value less than `low`) is displayed as black, the value `high` (and any value greater than `high`) is displayed as white. Values in between are displayed as intermediate shades of gray. The Image Viewer uses the default number of gray levels. If you use an empty matrix (`[]`) for `[low high]`, the Image Viewer uses `[min(I(:)) max(I(:))]`; the minimum value in `I` is displayed as black, and the maximum value is displayed as white.

`imtool(RGB)` displays the truecolor image `RGB` in the Image Viewer.

`imtool(BW)` displays the binary image `BW` in the Image Viewer. Pixel values of 0 display as black; pixel values of 1 display as white.

`imtool(X,map)` displays the indexed image `X` with colormap `map` in the Image Viewer.

`imtool(filename)` displays the image contained in the graphics file `filename` in the Image Viewer. The file must contain an image that can be read by `imread` or `dicomread` or a reduced resolution dataset (R-Set) created by `rsetwrite`. If the file contains multiple images, the first one is displayed. The file must be in the current directory or on the MATLAB path.

`hfigure = imtool(...)` returns `hfigure`, a handle to the figure created by the Image Viewer. `close(Hfigure)` closes the Image Viewer.

`imtool close all` closes all open Image Viewers.

`imtool(...,param1,val1,param2,val2,...)` displays the image, specifying parameters and corresponding values that control various aspects of the image display. The following table lists all `imshow` parameters. Parameter names can be abbreviated, and case does not matter.

Parameter	Value
'Colormap'	2-D, real, m -by-3 matrix specifying the colormap to use for the figure's <code>colormap</code> property. Use this parameter to view grayscale images in false color. If you specify an empty colormap (<code>[]</code>), <code>imtool</code> ignores this parameter.
'DisplayRange'	Two-element vector [<code>LOW HIGH</code>] that controls the display range of a grayscale image. See the <code>imtool(I,[low high])</code> syntax for more details about how to set this parameter. Note: Including the parameter name is optional, except when the image is specified by a filename. The syntax <code>imtool(I,[LOW HIGH])</code> is equivalent to <code>imtool(I,'DisplayRange',[LOW HIGH])</code> . However, the 'DisplayRange' parameter must be specified when calling <code>imtool</code> with a filename, as in the syntax <code>imtool(filename,'DisplayRange',[LOW HIGH])</code> .
'InitialMagnification'	Initial magnification used to display the image, specified as 'adaptive', 'fit', or as a numeric scalar value. When set to 'adaptive', the entire image is visible on initial display. If the image is too large to display on the screen, the Image Viewer displays the image at the largest magnification that fits on the screen.

Parameter	Value
	<p>When set to 'fit', the Image Viewer scales the entire image to fit in the window.</p> <p>When set to a numeric value, the value specifies the magnification as a percentage. For example, if you specify 100, the Image Viewer displays the image at 100% magnification (one screen pixel for each image pixel).</p> <hr/> <p>Note: When the image aspect ratio is such that less than one pixel would be displayed in either dimension at the requested magnification, the Image Viewer issues a warning and displays the image at 100%.</p> <hr/> <p>By default, the initial magnification parameter is set to the value returned by <code>iptgetpref('ImtoolInitialMagnification')</code>.</p>

Class Support

A truecolor image can be `uint8`, `uint16`, `single`, or `double`. An indexed image can be `logical`, `uint8`, `single`, or `double`. A grayscale image can be `uint8`, `uint16`, `int16`, `single`, or `double`. A binary image must be `logical`. A binary image is of class `logical`.

For all grayscale images having integer types, the default display range is `[intmin(class(I)) intmax(class(I))]`.

For grayscale images of class `single` or `double`, the default display range is `[0 1]`. If the data range of a `single` or `double` image is much larger or smaller than the default display range, you might need to experiment with setting the display range to see features in the image that would not be visible using the default display range.

Large Data Support

To view very large TIFF or NITF images that will not fit into memory, you can use `rsetwrite` to create a reduced resolution dataset (R-Set) viewable in the Image Viewer.

R-Sets can also improve performance of the Image Viewer for large images that fit in memory.

The following tools can be used with an R-Set: Overview, Zoom, Pan, Image Information, and Distance. Other tools, however, will not work with an R-Set. You cannot use the Pixel Region, Adjust Contrast, Crop Image, and Window/Level tools. Please note that the Pixel Information tool displays only the x and y coordinates of a pixel and not the associated intensity, index, or [R G B] values.

Related Toolbox Preferences

You can use the Image Processing Preferences dialog box to set toolbox preferences that modify the behavior of the Image Viewer. To access the dialog, select **File > Preferences** in the MATLAB desktop or Image Viewer menu. Also, you can set preferences programmatically with `iptsetpref`. The Image Viewer preferences include:

- `'ImtoolInitialMagnification'` controls the initial magnification for image display. To override this toolbox preference, specify the `'InitialMagnification'` parameter when you call `imtool`, as follows:

```
imtool(...,'InitialMagnification',initial_mag).
```

- `'ImtoolStartWithOverview'` controls whether the Overview tool opens automatically when you open an image using the Image Viewer. Possible values:

`true`— Overview tool opens when you open an image.

`{false}`— Overview tool does not open when you open an image. This is the default behavior.

For more information about these preferences, see `iptprefs`.

Examples

Display an image from a file.

```
imtool('board.tif')
```

Display an indexed image.

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

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

Display a grayscale image.

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

Display a grayscale image, adjusting the display range.

```
h = imshow(I,[0 80]);  
close(h)
```

Tips

`imshow` is the toolbox's fundamental image display function, optimizing figure, axes, and image object property settings for image display. The Image Viewer provides all the image display capabilities of `imshow` but also provides access to several other tools for navigating and exploring images, such as the Pixel Region tool, Image Information tool, and the Adjust Contrast tool. The Image Viewer presents an integrated environment for displaying images and performing some common image processing tasks.

You can access the Image Viewer through the Apps tab. Navigate to the Image Processing and Computer Vision group and select Image Viewer.

See Also

[imageinfo](#) | [imcontrast](#) | [imoverview](#) | [impixelregion](#) | [imread](#) | [imshow](#) | [iptprefs](#) | [rsetwrite](#)

Introduced before R2006a

imtophat

Top-hat filtering

Syntax

```
IM2 = imtophat(IM,SE)
IM2 = imtophat(IM,NHOOD)
gpuarrayIM2 = imtophat(gpuarrayIM, ___ )
```

Description

`IM2 = imtophat(IM,SE)` performs morphological top-hat filtering on the grayscale or binary input image `IM`. Top-hat filtering computes the morphological opening of the image (using `imopen`) and then subtracts the result from the original image. `imtophat` uses the structuring element `SE`, where `SE` is returned by `strel`. `SE` must be a single structuring element object, not an array containing multiple structuring element objects.

`IM2 = imtophat(IM,NHOOD)` where `NHOOD` is an array of 0s and 1s that specifies the size and shape of the structuring element, is the same as `imtophat(IM,strel(NHOOD))`.

`gpuarrayIM2 = imtophat(gpuarrayIM, ___)` performs the operation on a GPU. `NHOOD` is the structuring element specified by `strel(NHOOD)`, if `NHOOD` is an array of 0s and 1s that specifies the structuring element neighborhood. If `NHOOD` is a `gpuArray`, `strel(gather(NHOOD))` specifies the structuring element neighborhood.

Class Support

`IM` can be numeric or logical and must be nonsparse. The output image `IM2` has the same class as the input image. If the input is binary (`logical`), the structuring element must be flat.

`gpuarrayIM` must be a `gpuArray` of type `uint8` or `logical`. When used with a `gpuArray`, the structuring element must be flat and two-dimensional.

The output has the same class as the input.

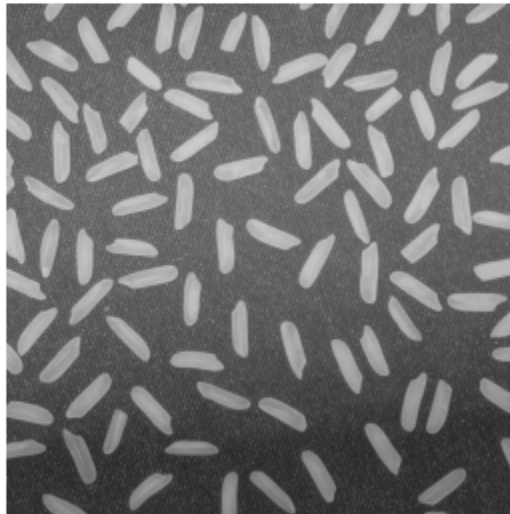
Examples

Use Top-hat Filtering to Correct Uneven Illumination

This example shows how to use top-hat filtering with a disk-shaped structuring element to remove uneven background illumination from an image with a dark background.

Read an image and display it.

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

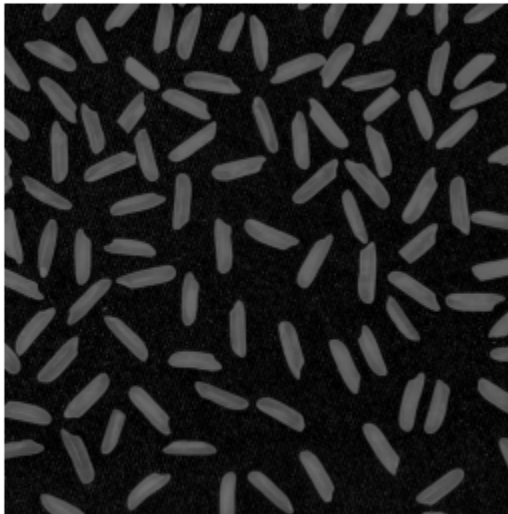


Create the structuring element.

```
se = strel('disk',12);
```

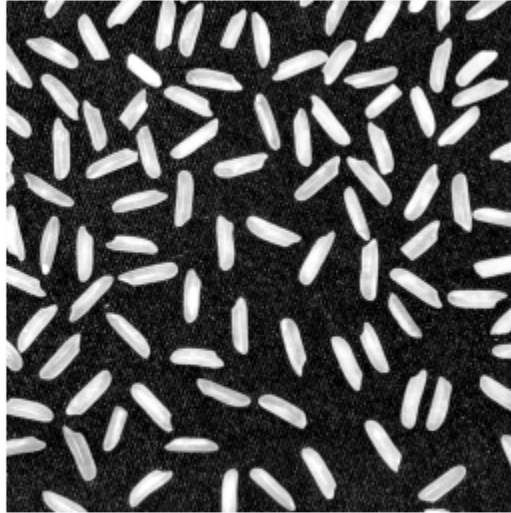
Perform the top-hat filtering and display the image.

```
tophatFiltered = imtophat(original,se);  
figure  
imshow(tophatFiltered)
```



Use `imadjust` to improve the visibility of the result.

```
contrastAdjusted = imadjust(tophatFiltered);  
figure  
imshow(contrastAdjusted)
```



Use Top-hat Filtering to Correct Uneven Illumination on the GPU

You can use top-hat filtering to correct uneven illumination when the background is dark. This example uses top-hat filtering with a disk-shaped structuring element to remove the uneven background illumination from an image.

Read an image and display it.

```
original = imread('rice.png');  
figure, imshow(original)
```

Create the structuring element.

```
se = strel('disk',12);
```

Perform the top-hat filtering and display the image. Note how the example passes the image to the `gpuArray` function before passing it to the `imtophat` function.

```
tophatFiltered = imtophat(gpuArray(original),se);  
figure, imshow(tophatFiltered)
```

Use `imadjust` to improve the visibility of the result. The `gather` function is used to retrieve the contents of the `gpuArray` from the GPU.

```
contrastAdjusted = imadjust(gather(tophatFiltered));  
figure, imshow(contrastAdjusted)
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the image input argument, **IM**, must be 2-D or 3-D and the structuring element input argument, **SE**, must be a compile-time constant.

See Also

`imbothat` | `strel` | `offsetstrel` | `gpuArray`

Introduced before R2006a

imtransform

Apply 2-D spatial transformation to image

Compatibility

`imtransform` is not recommended. Use `imwarp` instead.

Syntax

```
B = imtransform(A,tform)
B = imtransform(A,tform,interp)
[B,xdata,ydata] = imtransform(...)
[B,xdata,ydata] = imtransform(...,Name,Value)
```

Description

`B = imtransform(A,tform)` transforms the image `A` according to the 2-D spatial transformation defined by `tform`. If `ndims(A) > 2`, such as for an RGB image, then `imtransform` applies the same 2-D transformation to all 2-D planes along the higher dimensions.

`B = imtransform(A,tform,interp)` specifies the form of interpolation to use.

`[B,xdata,ydata] = imtransform(...)` returns the location of the output image `B` in the output X-Y space. By default, `imtransform` calculates `xdata` and `ydata` automatically so that `B` contains the entire transformed image `A`. However, you can override this automatic calculation by specifying values for the `'XData'` and `'YData'` arguments.

`[B,xdata,ydata] = imtransform(...,Name,Value)` transforms the image with additional options for controlling various aspects of the spatial transformation specified by one or more `Name,Value` pair arguments.

Input Arguments

A

An image of any nonsparse numeric class (real or complex) or of class `logical`.

tform

A spatial transformation structure returned by `maketform` or `cp2tform`. `imtransform` assumes spatial-coordinate conventions for the transformation `tform`. Specifically, the first dimension of the transformation is the horizontal or x -coordinate, and the second dimension is the vertical or y -coordinate. This convention is the reverse of the array subscripting convention in MATLAB.

interp

Form of interpolation to use, specified as: `'bicubic'`, `'bilinear'`, or `'nearest'` (nearest-neighbor). Alternatively, `interp` can be a `resampler` structure returned by `makeresampler`. This option allows more control over how `imtransform` performs resampling.

Default: `'bilinear'`

Name-Value Pair Arguments

Optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear within single quotes (`' '`) and is not case sensitive. You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

'UData'

A two-element, real vector that, when combined with `'VData'`, specifies the spatial location of image `A` in the 2-D input space U-V. The two elements of `'UData'` give the u -coordinates (horizontal) of the first and last columns of `A`, respectively.

Default: `[1 size(A,2)]`

'VData'

A two-element, real vector that, when combined with `'UData'`, specifies the spatial location of image `A` in the 2-D input space U-V. The two elements of `'VData'` give the v -coordinates (vertical) of the first and last rows of `A`, respectively.

Default: `[1 size(A,1)]`

'XData'

A two-element, real vector that, when combined with 'YData', specifies the spatial location of the output image **B** in the 2-D output space X-Y. The two elements of 'XData' give the *x*-coordinates (horizontal) of the first and last columns of **B**, respectively.

Default: If you do not specify 'XData' and 'YData', `imtransform` estimates values that contain the entire transformed output image. To determine these values, `imtransform` uses the `findbounds` function.

'YData'

A two-element real vector that, when combined with 'XData', specifies the spatial location of the output image **B** in the 2-D output space X-Y. The two elements of 'YData' give the *y*-coordinates (vertical) of the first and last rows of **B**, respectively.

Default: If you do not specify 'XData' and 'YData', `imtransform` estimates values that contain the entire transformed output image. To determine these values, `imtransform` uses the `findbounds` function.

'XYScale'

A one- or two-element real vector. The first element of 'XYScale' specifies the width of each output pixel in X-Y space. The second element (if present) specifies the height of each output pixel. If 'XYScale' has only one element, then the same value specifies both width and height.

Default: If you do not specify 'XYScale' but you do specify 'Size', then `imtransform` calculates 'XYScale' from 'Size', 'XData', and 'YData'. If you do not provide 'XYScale' or 'Size', then `imtransform` uses the scale of the input pixels for 'XYScale', except in cases where an excessively large output image would result.

Note: In cases where preserving the scale of the input image would result in an excessively large output image, the `imtransform` function automatically increases the 'XYScale'. To ensure that the output pixel scale matches the input pixel scale, specify the 'XYScale' parameter. For example, call `imtransform` as shown in the following syntax:

```
B = imtransform(A,T,'XYScale',1)
```

'Size'

A two-element vector of nonnegative integers that specifies the number of rows and columns of the output image **B**. For higher dimensions, `imtransform` takes the size of **B** directly from the size of **A**. Thus, `size(B,k)` equals `size(A,k)` for $k > 2$.

Default: If you do not specify `'Size'`, `imtransform` derives this value from `'XData'`, `'YData'`, and `'XYScale'`.

'FillValues'

An array containing one or several fill values. The `imtransform` function uses fill values for output pixels when the corresponding transformed location in the input image is completely outside the input image boundaries. If **A** is 2-D, `'FillValues'` requires a scalar. However, if **A**'s dimension is greater than two, then you can specify `'FillValues'` as an array whose size satisfies the following constraint: `size(fill_values,k)` must equal either `size(A,k+2)` or 1.

For example, if **A** is a `uint8` RGB image that is 200-by-200-by-3, then possibilities for `'FillValues'` include the following values.

Value	Fill
0	Fill with black
[0;0;0]	Fill with black
255	Fill with white
[255;255;255]	Fill with white
[0;0;255]	Fill with blue
[255;255;0]	Fill with yellow

If **A** is 4-D with size 200-by-200-by-3-by-10, then you can specify `'FillValues'` as a scalar, 1-by-10, 3-by-1, or 3-by-10.

Output Arguments

B

Output image of any nonsparse numeric class (real or complex) or of class `logical`.

xdata

Two-element vector that specifies the x -coordinates of the first and last columns of **B**.

Note: Sometimes the output values **xdata** and **ydata** do not exactly equal the input 'XData' and 'YData' arguments. The values differ either because of the need for an integer number of rows and columns, or because you specify values for 'XData', 'YData', 'XYScale', and 'Size' that are not entirely consistent. In either case, the first element of **xdata** and **ydata** always equals the first element of 'XData' and 'YData', respectively. Only the second elements of **xdata** and **ydata** can be different.

ydata

Two-element vector that specifies the y -coordinates of the first and last rows of **B**.

Examples

Simple Transformation. Apply a horizontal shear to an intensity image:

```
I = imread('cameraman.tif');  
tform = maketform('affine',[1 0 0; .5 1 0; 0 0 1]);  
J = imtransform(I,tform);  
imshow(I), figure, imshow(J)
```



Horizontal Shear

Projective Transformation. Map a square to a quadrilateral with a projective transformation:

```
% Set up an input coordinate system so that the input image
% fills the unit square with vertices (0 0),(1 0),(1 1),(0 1).
I = imread('cameraman.tif');
udata = [0 1]; vdata = [0 1];

% Transform to a quadrilateral with vertices (-4 2),(-8 3),
% (-3 -5),(6 3).
tform = maketform('projective',[ 0 0; 1 0; 1 1; 0 1],...
                 [-4 2; -8 -3; -3 -5; 6 3]);

% Fill with gray and use bicubic interpolation.
% Make the output size the same as the input size.

[B,xdata,ydata] = imtransform(I, tform, 'bicubic', ...
                              'udata', udata,...
                              'vdata', vdata,...
                              'size', size(I),...
                              'fill', 128);
subplot(1,2,1), imshow(I,'XData',udata,'YData',vdata), ...
    axis on
subplot(1,2,2), imshow(B,'XData',xdata,'YData',ydata), ...
    axis on
```



Projective Transformation

Image Registration. Register an aerial photo to an orthophoto.

Read an aerial photo into the MATLAB workspace and view it.

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



Aerial Photo

Read an orthophoto into the MATLAB workspace and view it.

```
figure, imshow('westconcordorthophoto.png')
```



Orthophoto

Load control points that were previously picked.

```
load westconcordpoints
```

Create a transformation structure for a projective transformation using the points.

```
t_concord = cp2tform(movingPoints,fixedPoints,'projective');
```

Get the width and height of the orthophoto, perform the transformation, and view the result.

```
info = imfinfo('westconcordorthophoto.png');  
registered = imtransform(unregistered,t_concord,...  
    'XData',[1 info.Width], 'YData',[1 info.Height]);  
figure, imshow(registered)
```



Transformed Image

Tutorials

- "Perform Simple 2-D Translation Transformation"
- Exploring Slices from a 3-Dimensional MRI Data Set

- [Padding and Shearing an Image Simultaneously](#)

Tips

- **Image Registration.** The `imtransform` function automatically shifts the origin of your output image to make as much of the transformed image visible as possible. If you use `imtransform` to do image registration, the syntax `B = imtransform(A,tform)` can produce unexpected results. To control the spatial location of the output image, set `'XData'` and `'YData'` explicitly.
- **Pure Translation.** Calling the `imtransform` function with a purely translational transformation, results in an output image that is exactly like the input image unless you specify `'XData'` and `'YData'` values in your call to `imtransform`. For example, if you want the output to be the same size as the input revealing the translation relative to the input image, call `imtransform` as shown in the following syntax:

```
B = imtransform(A,T,'XData',[1 size(A,2)],...  
    'YData',[1 size(A,1)])
```

For more information about this topic, see “Perform Simple 2-D Translation Transformation”.

- **Transformation Speed.** When you do not specify the output-space location for `B` using `'XData'` and `'YData'`, `imtransform` estimates the location automatically using the function `findbounds`. You can use `findbounds` as a quick forward-mapping option for some commonly used transformations, such as affine or projective. For transformations that do not have a forward mapping, such as the polynomial ones computed by `fitgeotrans`, `findbounds` can take much longer. If you can specify `'XData'` and `'YData'` directly for such transformations, `imtransform` may run noticeably faster.
- **Clipping.** The automatic estimate of `'XData'` and `'YData'` using `findbounds` sometimes clips the output image. To avoid clipping, set `'XData'` and `'YData'` directly.
- **Arbitrary Dimensional Transformations.** Use a 2-D transformation for `tform` when using `imtransform`. For arbitrary-dimensional array transformations, see `tformarray`.

See Also

[checkerboard](#) | [cp2tform](#) | [imresize](#) | [imrotate](#) | [maketform](#) | [makesampler](#) | [tformarray](#)

Topics

[“Perform Simple 2-D Translation Transformation”](#)
[Exploring Slices from a 3-Dimensional MRI Data Set](#)
[Padding and Shearing an Image Simultaneously](#)

Introduced before R2006a

imtranslate

Translate image

Syntax

```
B = imtranslate(A,translation)
[B,RB] = imtranslate(A,RA,translation)
___ = imtranslate( ___,method)
___ = imtranslate( ___,Name,Value)
```

Description

`B = imtranslate(A,translation)` translates image `A` by the translation vector specified in `translation`. If `A` has more than two dimensions and `translation` is a two-element vector, `imtranslate` applies a 2-D translation to `A`, one plane at a time.

`[B,RB] = imtranslate(A,RA,translation)` translates the spatially referenced image `A` with its associated spatial referencing object `RA`. The translation vector, `translation`, is in the world coordinate system. The function returns the translated spatially referenced image `B`, with its associated spatial referencing object, `RB`.

`___ = imtranslate(___,method)` translates image `A`, using the interpolation method specified by `method`.

`___ = imtranslate(___,Name,Value)` translates the input image using name-value pairs to control various aspects of the translation.

Examples

Translate 2-D Image

Read image into the workspace.

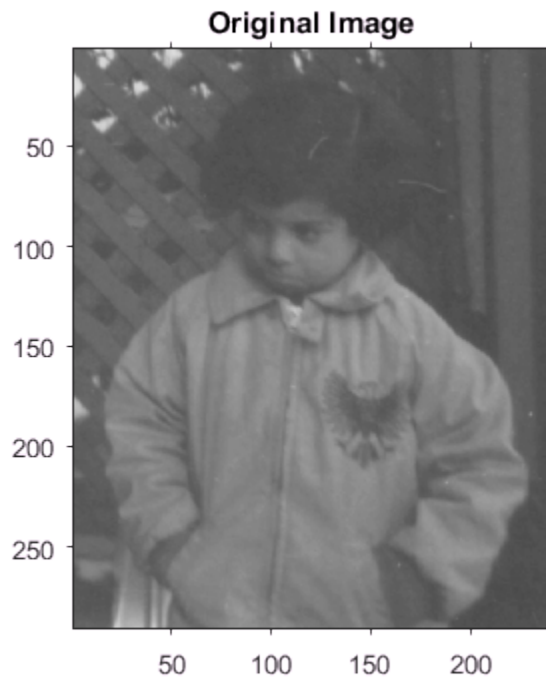
```
I = imread('pout.tif');
```

Translate the image.

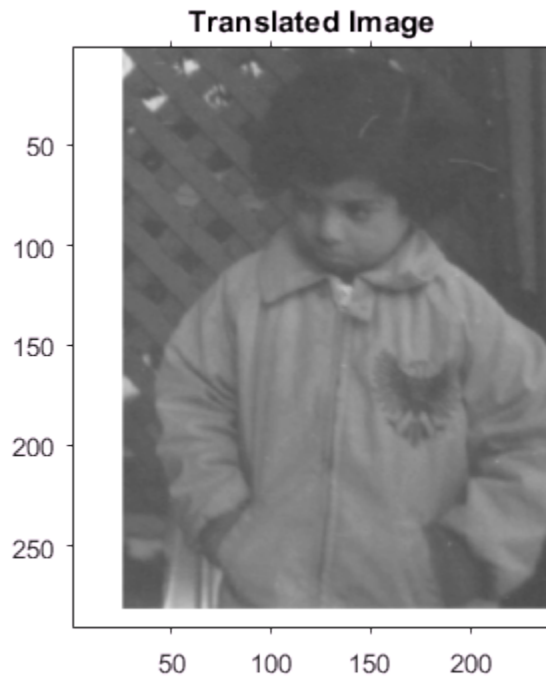
```
J = imtranslate(I,[25.3, -10.1], 'FillValues',255);
```

Display the original image and the translated image.

```
figure  
imshow(I);  
title('Original Image');  
set(gca, 'Visible', 'on');
```



```
figure  
imshow(J);  
title('Translated Image');  
set(gca, 'Visible', 'on');
```



Translate 2-D Image and View Entire Translated Image

Read image into the workspace.

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

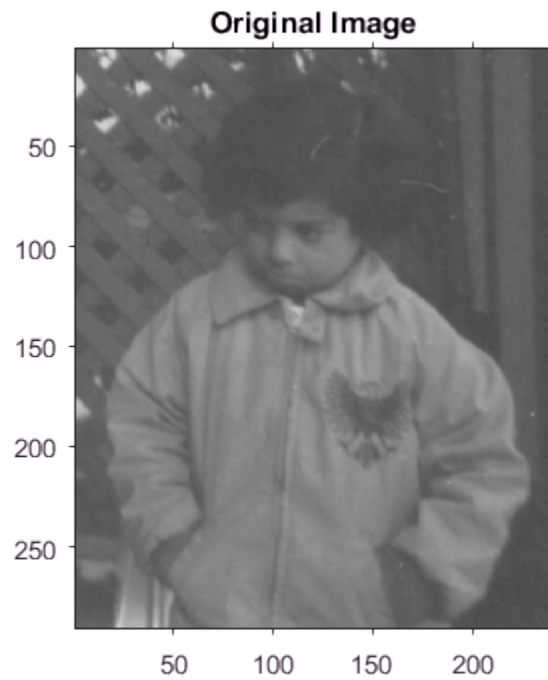
Translate the image. Use the `OutputView` parameter to specify that you want the entire translated image to be visible.

```
J = imtranslate(I,[25.3, -10.1],'FillValues',255,'OutputView','full');
```

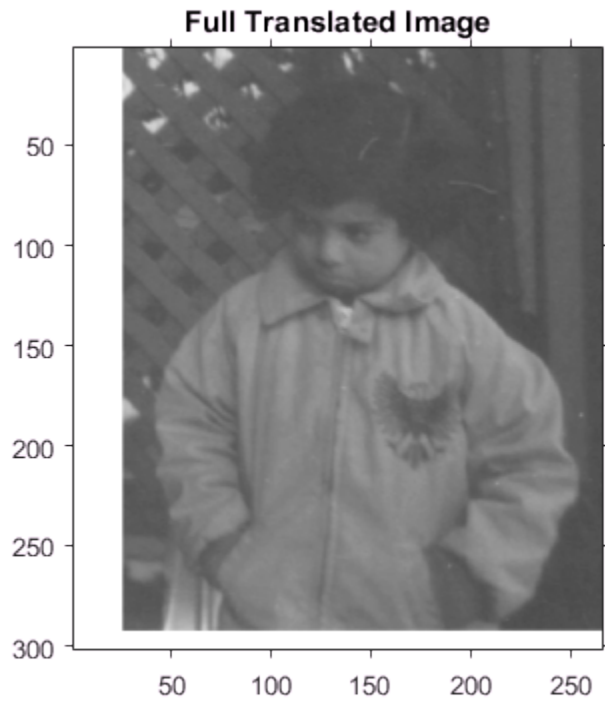
Display the original image and the translated image.

```
figure
```

```
imshow(I);  
title('Original Image');  
set(gca,'Visible','on');
```



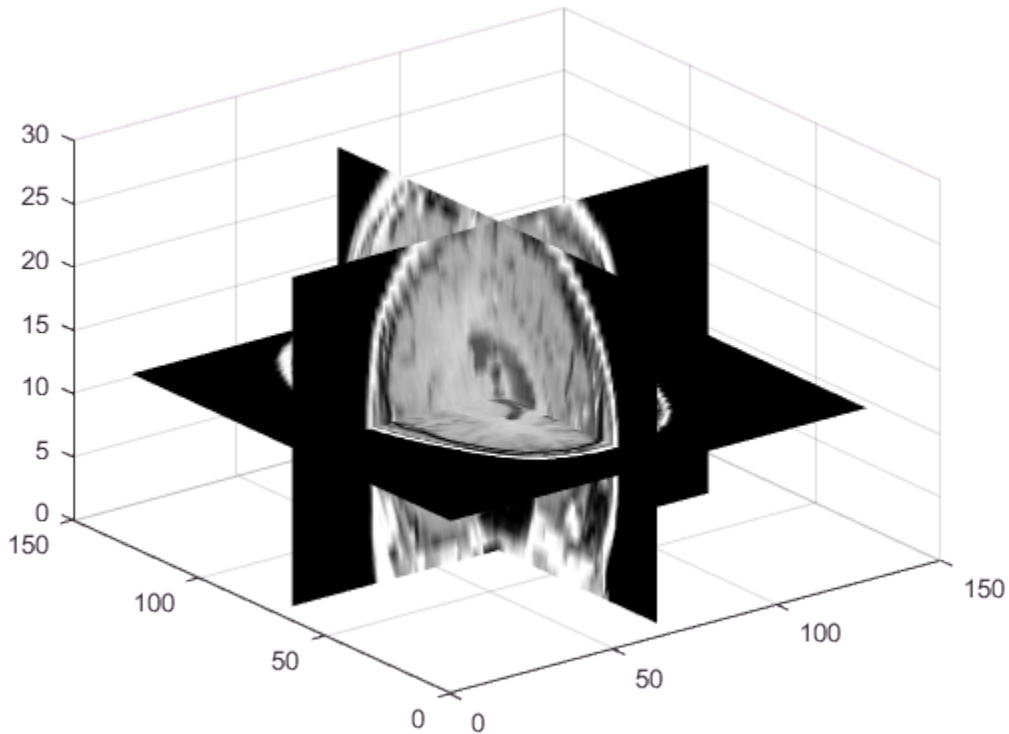
```
figure  
imshow(J);  
title('Full Translated Image');  
set(gca,'Visible','on');
```



Translate 3-D MRI Dataset

Load MRI data into the workspace and display it.

```
s = load('mri');  
mriVolume = squeeze(s.D);  
sizeIn = size(mriVolume);  
hFigOriginal = figure;  
hAxOriginal = axes;  
slice(double(mriVolume),sizeIn(2)/2,sizeIn(1)/2,sizeIn(3)/2);  
grid on, shading interp, colormap gray
```

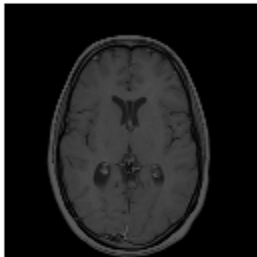


Apply a translation in the X,Y direction.

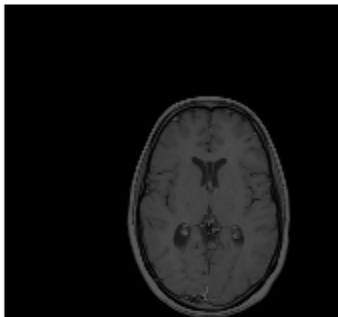
```
mriVolumeTranslated = imtranslate(mriVolume,[40,30,0], 'OutputView', 'full');
```

Visualize the translation by viewing an axial slice plane taken through center of the volume. Note the shift in the X and Y directions.

```
sliceIndex = round(sizeIn(3)/2);  
axialSliceOriginal = mriVolume(:,:,sliceIndex);  
axialSliceTranslated = mriVolumeTranslated(:,:,sliceIndex);  
  
imshow(axialSliceOriginal);
```



```
imshow(axialSliceTranslated);
```



- “Translate an Image using imtranslate Function”

Input Arguments

A — Image to be translated

nonsparse, numeric array | logical array

Image to be translated, specified as a nonsparse, numeric array of any class, except `uint64` and `int64`, or a logical array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

RA — Spatial referencing information associated with the input image A

spatial referencing object

Spatial referencing information associated with the input image A, specified as a spatial referencing object, `imref2d` or `imref3d`.

translation — Translation vector

two-element or three-element, nonsparse, real-valued, numeric vector

Translation vector, specified as a two-element or three-element, nonsparse, real-valued, numeric vector, such as `[Tx Ty]`, for 2-D inputs, and `[Tx Ty Tz]`, for 3-D inputs. Values can be fractional.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

method — Interpolation method

'linear' (default) | 'nearest' | 'cubic'

Interpolation method, specified by one of the following values:

Value	Description
'cubic'	Cubic interpolation. Note: Cubic interpolation can produce pixel values outside the original range.
'linear'	Linear interpolation
'nearest'	Nearest-neighbor interpolation; the output pixel is assigned the value of the pixel that the point falls within. No other pixels are considered.

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

```
Example: mriVolumeTranslated = imtranslate(mriVolume,
[40,30,0], 'OutputView', 'full');
```

'OutputView' — Output world limits

'same' (default) | 'full'

Output world limits, specified as the comma-separated pair consisting of 'OutputView' and one of the following values:

Value	Description
'same'	Output world limits are the same as the input image.
'full'	Output world limits are the bounding rectangle that includes both the input image and the translated output image.

Data Types: char

'FillValues' — Fill values used for output pixels outside the input image

0 (default) | numeric array

Fill values used for output pixels outside the input image, specified as the comma-separated pair consisting of 'FillValues' and a numeric array containing one or several fill values. `imtranslate` uses fill values for output pixels when the corresponding inverse transformed location in the input image is completely outside the input image boundaries.

- If `A` is 2-D, `FillValues` must be a scalar.
- If `A` is 3-D and `translation` is a three-element vector, `FillValues` must be a scalar.
- If `A` is N-D and `translation` is a two-element vector, `FillValues` can be either scalar or an array whose size matches dimensions 3-to-N of `A`. For example, if `A` is a `uint8` RGB image that is 200-by-200-by-3, `FillValues` can be a scalar or a 3-by-1 array.

- If **A** is 4-D, **FillValues** can be a scalar or an array. For example, if **A** is 200-by-200-by-3-by-10, then **FillValues** can be a scalar or a 3-by-10 array.

Some example fill values:

Fill Value	Description
0	Fill with black
[0;0;0]	Fill with black
255	Fill with white
[255;255;255]	Fill with white
[0;0;255]	Fill with blue
[255;255;0]	Fill with yellow

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output Arguments

B — Translated image

nonsparse, real-valued, numeric array | logical array

Translated image, returned as a nonsparse, real-valued, numeric array or logical array. The class of **B** is the same as the class of **A**.

RB — Spatial referencing information associated with the output image

spatial referencing object

Spatial referencing information associated with the output image, returned as a spatial referencing object, `imref2d` or `imref3d`.

Tips

- `imtranslate` is optimized for integrally valued translation vectors.
- When 'OutputView' is 'full' and translation is a fractional number of pixels, `imtranslate` expands the world limits of the output spatial referencing object to the nearest full pixel increment. `imtranslate` does this so that it contains both the original and translated images at the same resolution as the input image.

The additional image extent in each is added on one side of the image, in the direction that the translation vector points. For example, when `translation` is fractional and positive in both *X* and *Y*, then `imtranslate` expands the maximum of `XWorldLimits` and `YWorldLimits` to enclose the 'full' bounding rectangle at the resolution of the input image.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- The function supports only 2-D translation vectors, `translation`. 3-D translations are not supported.

See Also

See Also

`imref2d` | `imref3d` | `imresize` | `imrotate` | `imwarp`

Topics

“Translate an Image using `imtranslate` Function”

Introduced in R2014a

imview

Display image in image tool

Note: `imview` has been removed. Use `imtool` instead.

Introduced before R2006a

imwarp

Apply geometric transformation to image

Syntax

```
B = imwarp(A,tform)
B = imwarp(A,D)
[B,RB] = imwarp(A,RA,tform)
B = imwarp( ____,Interp)
[B,RB] = imwarp( ____,Name,Value)
```

Description

`B = imwarp(A,tform)` transforms the image `A` according to the geometric transformation defined by `tform`, which is a geometric transformation object. `B` is the transformed image.

`B = imwarp(A,D)` transforms the input image `A` according to the displacement field defined by `D`.

`[B,RB] = imwarp(A,RA,tform)` transforms the spatially referenced image, specified by the image data `A` and the associated spatial referencing object `RA`. The output is a spatially referenced image specified by the image data `B` and the associated spatial referencing object `RB`.

`B = imwarp(____,Interp)` specifies the form of interpolation to use

`[B,RB] = imwarp(____,Name,Value)` specifies parameters that control various aspects of the geometric transformation. Parameter names can be abbreviated, and case does not matter.

Examples

Apply Horizontal Shear to Image

Read grayscale image into workspace and display it.

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



Create a 2-D geometric transformation object.

```
tform = affine2d([1 0 0; .5 1 0; 0 0 1])
```

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

Apply the transformation to the image.

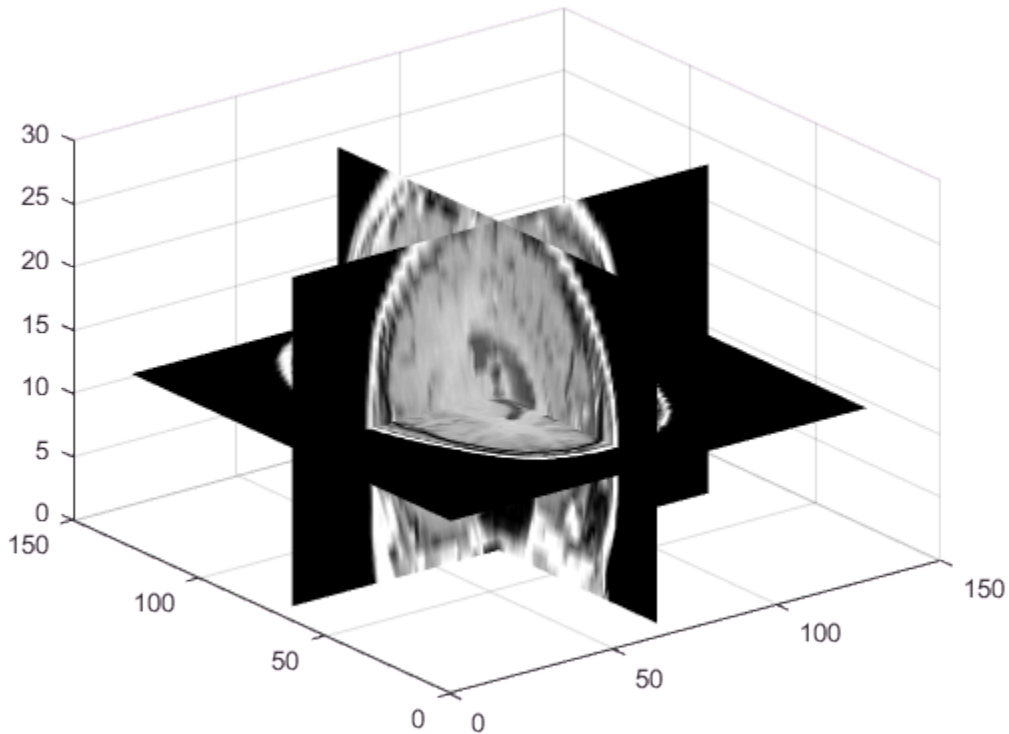
```
J = imwarp(I,tform);  
figure  
imshow(J)
```



Apply Rotation Transformation to 3-D MRI Dataset

Read 3-D MRI data into the workspace and visualize it.

```
s = load('mri');  
mriVolume = squeeze(s.D);  
sizeIn = size(mriVolume);  
hFigOriginal = figure;  
hAxOriginal = axes;  
slice(double(mriVolume),sizeIn(2)/2,sizeIn(1)/2,sizeIn(3)/2);  
grid on, shading interp, colormap gray
```

Create a 3-D geometric transformation object. First create a transformation matrix that rotates the image around the y-axis. Then pass the matrix to the `affine3d` object constructor.

```
theta = pi/8;
t = [cos(theta) 0 -sin(theta) 0
     0 1 0 0
     sin(theta) 0 cos(theta) 0
     0 0 0 1]

t =

    0.9239    0   -0.3827    0
         0    1.0000    0    0
```

```
0.3827    0    0.9239    0
         0    0         0    1.0000
```

```
tform = affine3d(t)
```

```
tform =  
  affine3d with properties:
```

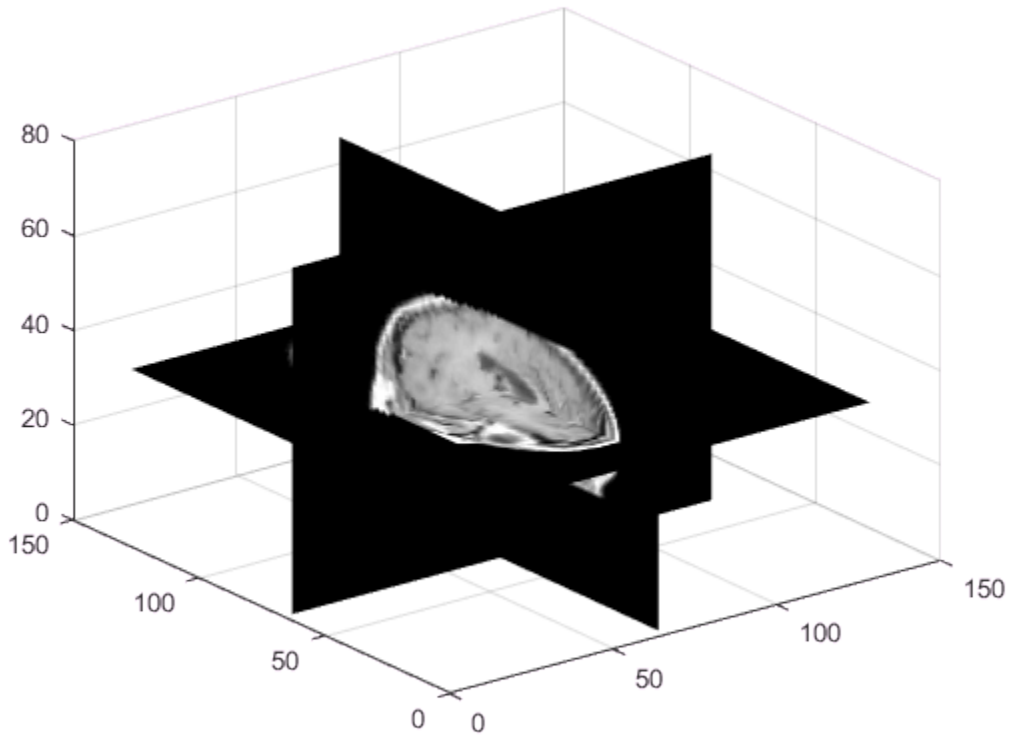
```
          T: [4x4 double]  
  Dimensionality: 3
```

Apply the transformation to the image.

```
mriVolumeRotated = imwarp(mriVolume,tform);
```

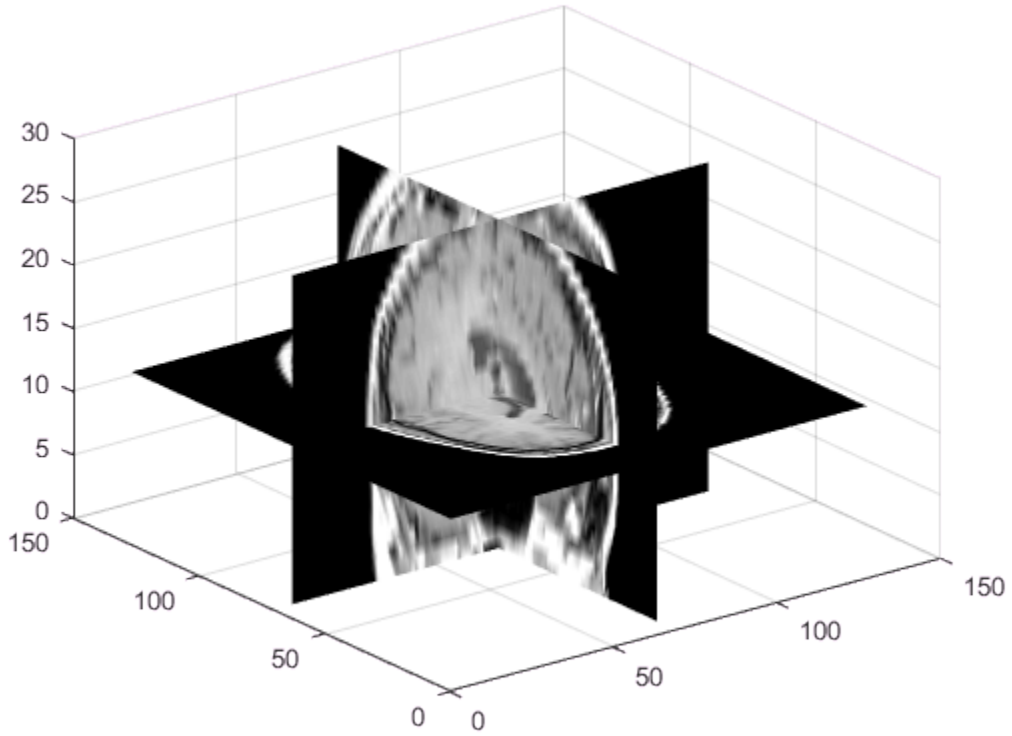
Visualize three slice planes through the center of the transformed volumes.

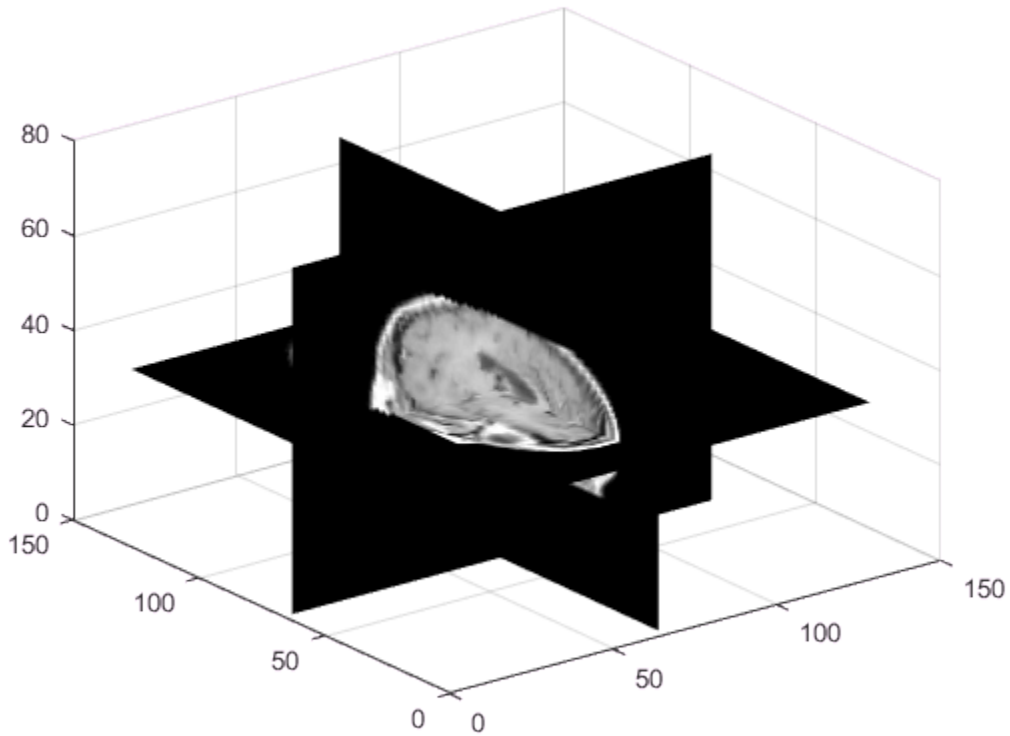
```
sizeOut = size(mriVolumeRotated);  
hFigRotated = figure;  
hAxRotated = axes;  
slice(double(mriVolumeRotated),sizeOut(2)/2,sizeOut(1)/2,sizeOut(3)/2);  
grid on, shading interp, colormap gray
```



Link views of both axes together.

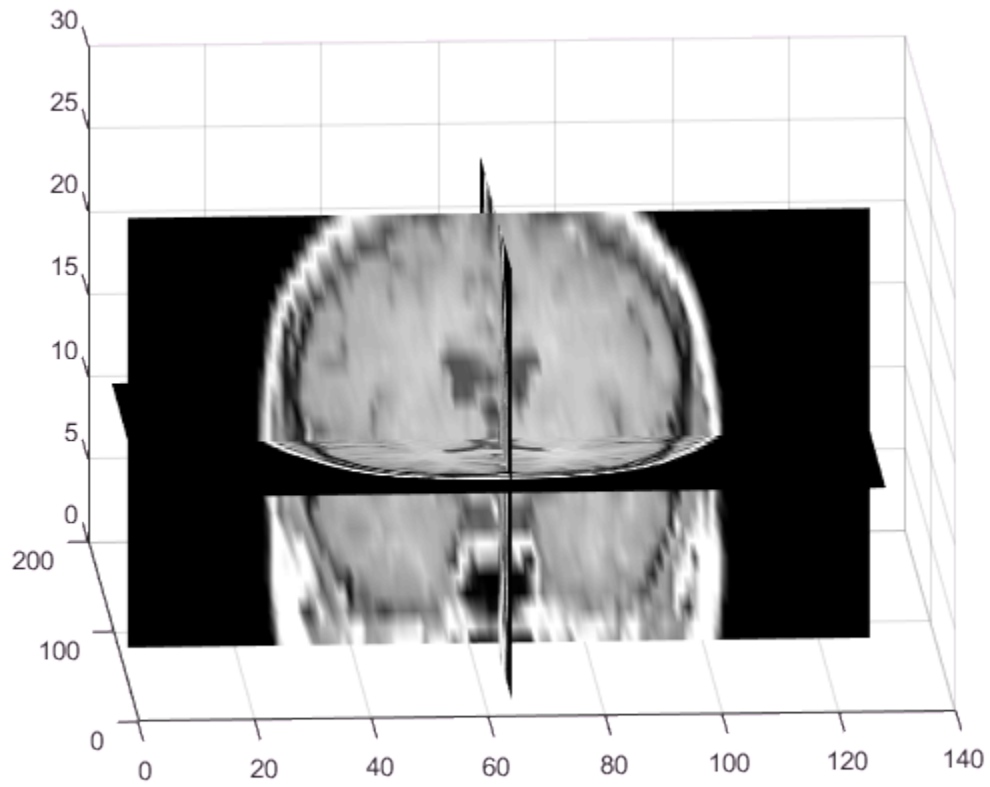
```
linkprop([hAxOriginal,hAxRotated],'View');
```

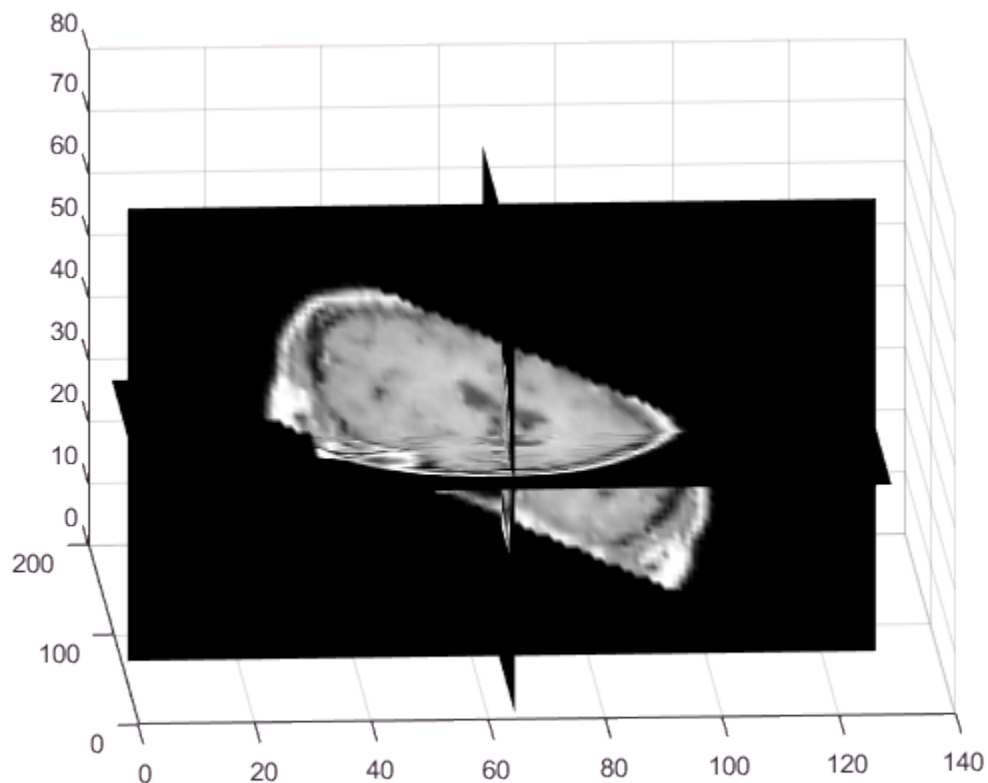




Set view to see effect of rotation.

```
set(hAxRotated, 'View', [-3.5 20.0])
```





Input Arguments

A — Image to be transformed

nonsparse, real-valued array

Image to be transformed, specified as a nonsparse, real-valued array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

tform — 2-D or 3-D geometric transformation to perform

geometric transformation object

2-D or 3-D geometric transformation to perform, specified as a geometric transformation object. There are three types of geometric transformation objects: `affine2d`, `projective2d` or `affine3d`.

- If `tform` is 2-D and `ndims(A) > 2`, such as for a truecolor image, `imwarp` applies the same 2-D transformation to all 2-D planes along the higher dimensions.
- If `tform` is 3-D, `A` must be a 3-D image volume.

D — Displacement field

nonsparse numeric array

Displacement field, specified as nonsparse numeric array. The displacement field defines the grid size and location of the output image. Displacement values are in units of pixels. `imwarp` assumes that `D` is referenced to the default intrinsic coordinate system. To estimate the displacement field, use `imregdemons`.

- If `A` is a 2-D grayscale image of size m -by- n , then `D` is m -by- n -by-2. The first plane of the displacement field, `D(:, :, 1)`, describes the x -component of additive displacement. `imwarp` adds these values to column and row locations in `D` to produce remapped locations in `A`. Similarly, the second plane of the displacement field, `D(:, :, 2)`, describes the y -component of additive displacement values.
- If `A` is a 2-D truecolor or 3-D grayscale image of size m -by- n -by- p , then `D` is:
 - m -by- n -by- p -by-3. `D(:, :, :, 1)` contains displacements along the x -axis, `D(:, :, :, 2)` contains displacements along the y -axis, and `D(:, :, :, 3)` contains displacements along the z -axis
 - m -by- n -by-2, then `imwarp` applies the displacement field to one plane at a time.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

RA — Spatial referencing information associated with the image to be transformed

spatial referencing object

Spatial referencing information associated with the image to be transformed, specified as a spatial referencing object.

- If `tform` is a 2-D geometric transformation, `RA` must be a 2-D spatial referencing object (`imref2d`).
- If `tform` is a 3-D geometric transformation, `RA` must be a 3-D spatial referencing object (`imref3d`).

Interp — Form of interpolation used

'linear' (default) | 'nearest' | 'cubic'

Form of interpolation used, specified as one of the following values:

Interpolation Method	Description
'linear'	Linear interpolation
'nearest'	Nearest-neighbor interpolation—the output pixel is assigned the value of the pixel that the point falls within. No other pixels are considered.
'cubic'	Cubic interpolation

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, . . . , **NameN**, **ValueN**.

Example: `J = imwarp(I,tform,'FillValues',255)` uses white pixels as fill values.

'OutputView' — Size and location of output image in world coordinate system

imref2d or imref3d spatial referencing object

Size and location of output image in world coordinate system, specified as the comma-separated pair consisting of 'OutputView' and an imref2d or imref3d spatial referencing object. The **ImageSize**, **XWorldLimits**, and **YWorldLimits** properties of the spatial referencing object define the size of the output image and the location of the output image in the world coordinate system. The use of 'OutputView' is not available when applying displacement fields.

'FillValues' — Value used for output pixels outside image boundaries

numeric scalar or array

Value used for output pixels outside the input image boundaries, specified as the comma-separated pair consisting of 'FillValues' and a numeric array. Fill values are used for output pixels when the corresponding inverse transformed location in the input image is completely outside the input image boundaries.

- If the input image is 2-D, **FillValues** must be a scalar.
- If the input image is 3-D and the geometric transformation is 3-D, **FillValues** must be a scalar.
- If the input image is N-D and the geometric transformation is 2-D, **FillValues** can be a scalar or an array. If you specify an array, the array size must match the higher dimensions of the input image. For example, if the input image is a **uint8** RGB image that is 200-by-200-by-3, **FillValues** can be a scalar or a 3-by-1 array. In another example, if the input image is 4-D with size 200-by-200-by-3-by-10, **FillValues** can be a scalar or a 3-by-10 array.

In this RGB image example, possibilities for **FillValues** include:

FillValue	Effect
0	Fill with black
[0;0;0]	Fill with black
255	Fill with white
[255;255;255]	Fill with white
[0;0;255]	Fill with blue
[255;255;0]	Fill with yellow

'SmoothEdges' — Pad image to create smooth edges

false (default) | true

Pad image to create smooth edges, specified as the logical value **true** or **false**. When set to **true**, **imwarp** pads the input image (with values specified by **FillValues**) to create a smoother edge in the output image. When set to **false**, **imwarp** does not pad the image. Choosing **false** (not padding) the input image can result in a sharper edge in the output image. This sharper edge can be useful to minimize seam distortions when registering two images side by side.

Output Arguments

B — Transformed image

nonsparse real-valued array

Transformed image, returned as a nonsparse real-valued array. **B** is the same class as **A**.

RB — Spatial referencing information associated with the transformed image

`imref2d` or `imref3d` spatial referencing object

Spatial referencing information associated with the transformed image, returned as an `imref2d` or `imref3d` spatial referencing object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- The geometric transformation object input, `tform`, must be either an `affine2d` or `projective2d` object.
- The interpolation method and optional parameter names must be constants.

See Also

See Also

Apps

Registration Estimator

Functions

`imregdemons` | `imregister` | `imregtform` | `imtranslate`

Classes

`affine2d` | `projective2d`

Introduced in R2013a

ind2gray

Convert indexed image to grayscale image

Syntax

```
I = ind2gray(X,map)
```

Description

`I = ind2gray(X,map)` converts the image `X` with colormap `map` to a grayscale image `I`. `ind2gray` removes the hue and saturation information from the input image while retaining the luminance.

Note: A grayscale image is also called a gray-scale, gray scale, or gray-level image.

Class Support

`X` can be of class `uint8`, `uint16`, `single`, or `double`. `map` is `double`. `I` is of the same class as `X`.

Examples

Convert Indexed Image to Grayscale

Load an indexed image into the workspace.

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

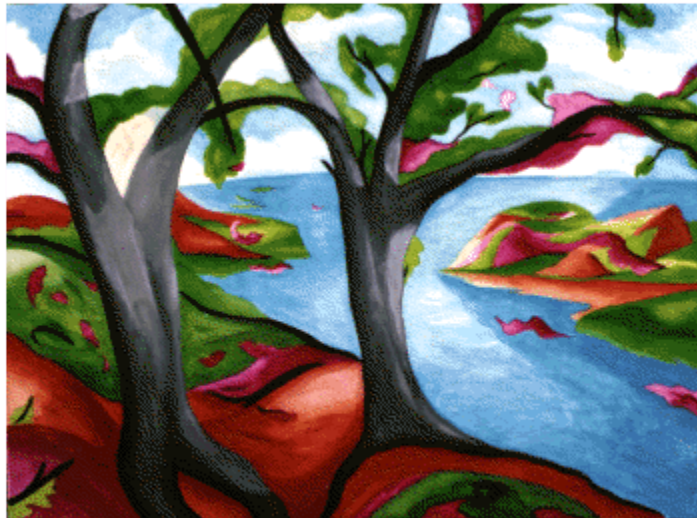
Convert the image to grayscale using `ind2gray`.

```
I = ind2gray(X,map);
```

Display the indexed image and the converted grayscale image.

```
imshow(X,map)
title('Indexed Image')
```

Indexed Image



```
figure
imshow(I)
title('Converted Grayscale Image')
```

Converted Grayscale Image

Algorithms

`ind2gray` converts the colormap to NTSC coordinates using `rgb2ntsc`, and sets the hue and saturation components (I and Q) to zero, creating a gray colormap. `ind2gray` then replaces the indices in the image X with the corresponding grayscale intensity values in the gray colormap.

See Also

`gray2ind` | `imshow` | `imtool` | `mat2gray` | `rgb2gray` | `rgb2ntsc`

Introduced before R2006a

ind2rgb

Convert indexed image to RGB image

Syntax

```
RGB = ind2rgb(X,map)
```

Description

`RGB = ind2rgb(X,map)` converts the matrix `X` and corresponding colormap `map` to RGB (truecolor) format.

Class Support

`X` can be of class `uint8`, `uint16`, or `double`. `RGB` is an `m-by-n-by-3` array of class `double`.

See Also

`ind2gray` | `rgb2ind`

Introduced before R2006a

integralBoxFilter

2-D box filtering of integral images

Syntax

```
B = integralBoxFilter(intA)
B = integralBoxFilter(intA,filterSize)
B = integralBoxFilter( ____,Name,Value)
```

Description

`B = integralBoxFilter(intA)` filters the integral image `intA` with a 3-by-3 box filter. Returns the filtered image, `B`.

`B = integralBoxFilter(intA,filterSize)` filters the integral image `intA` with a 2-D box filter with size specified by `filterSize`.

`B = integralBoxFilter(____,Name,Value)` filters integral image `intA` with Name-Value pairs to control various aspects of the filtering.

Examples

Filter Integral Image

Read image into the workspace.

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

Pad the image by the radius of the filter neighborhood. This example uses an 11-by-11 filter.

```
filterSize = [11 11];
padSize = (filterSize-1)/2;
Apad = padarray(A, padSize, 'replicate', 'both');
```

Compute the integral image of the padded input image.

```
intA = integralImage(Apad);
```

Filter the integral image.

```
B = integralBoxFilter(intA, filterSize);
```

Display original image and filtered image.

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



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

Filtered Image**Filter Image with Horizontal and Vertical Motion Blur**

Read image into the workspace.

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

Pad the image by radius of the filter neighborhood, calculated $(11 - 1) / 2$.

```
padSize = [5 5];  
Apad = padarray(A, padSize, 'replicate', 'both');
```

Calculate the integral image of the padded input.

```
intA = integralImage(Apad);
```

Filter the integral image with a vertical $[11\ 1]$ filter.

```
Bvert = integralBoxFilter(intA, [11 1]);
```

Crop the output to retain input image size and display it.

```
Bvert = Bvert(:,6:end-5);
```

Filter the integral image with a horizontal [1 11] filter.

```
Bhorz = integralBoxFilter(intA, [1 11]);
```

Crop the output to retain input image size.

```
Bhorz = Bhorz(6:end-5,:);
```

Display the original image and the filtered images.

```
figure,  
imshow(A)  
title('Original Image')
```

Original Image



```
figure,  
imshow(Bvert,[])  
title('Filtered with Vertical Filter')
```

Filtered with Vertical Filter



```
figure,  
imshow(Bhorz,[])  
title('Filtered with Horizontal Filter')
```

Filtered with Horizontal Filter



Input Arguments

intA — Integral image to be filtered

real, nonsparse double matrix of any dimension

Integral image to be filtered, specified as a real, nonsparse matrix of any dimension. The integral image must be upright—`integralBoxFilter` does not support rotated integral images. The first row and column of the integral image is assumed to be zero-padded, as returned by `integralImage`.

Example: `B = integralBoxFilter(A);`

Data Types: double

filterSize — Size of box filter

3-by-3 box filter (default) | scalar or 2-element vector of positive, odd integers

Size of box filter, specified as a scalar or 2-element vector of positive, odd integers. If `filterSize` is scalar, `integralBoxFilter` uses a square box filter.

Example: `B = integralBoxFilter(A,5);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `B = integralBoxFilter(A,5,'NormalizationFactor',1);`

'NormalizationFactor' — Normalization factor applied to box filter

$1/\text{filterSize}.^2$, if scalar, and $1/\text{prod}(\text{filterSize})$, if vector (default) | numeric scalar

Normalization factor applied to box filter, specified as a numeric scalar or vector.

The default 'NormalizationFactor' has the effect of a mean filter—the pixels in the output image are the local means of the image. To get local area sums, set 'NormalizationFactor' to 1. To avoid overflow in such circumstances, consider using double precision images by converting the input image to class `double`.

Example: `B = integralBoxFilter(A,5,'NormalizationFactor',1);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

B — Filtered image

real, nonsparse `double` matrix

Filtered image, returned as a real, nonsparse matrix of class `double`.

`integralBoxFilter` returns only the parts of the filtering that are computed without padding.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- The 'NormalizationFactor' parameter must be a compile-time constant.

See Also

See Also

`imboxfilt` | `integralImage`

Topics

“Integral Image”

Introduced in R2015b

integralBoxFilter3

3-D box filtering of 3-D integral images

Syntax

```
B = integralBoxFilter3(intA)
B = integralBoxFilter3(intA,filterSize)
B = integralBoxFilter3( ____,Name,Value)
```

Description

`B = integralBoxFilter3(intA)` filters integral image `intA` with a 3-by-3-by-3 box filter. `B` is a 3-D image of class `double` containing the filtered output.

`B = integralBoxFilter3(intA,filterSize)` filters integral image `intA` with a 3-D box filter with size specified by `filterSize`.

`B = integralBoxFilter3(____,Name,Value)` filters integral image `intA` with a 3-D box filter with `Name-Value` pairs to control various aspects of the filtering.

Examples

Filter 3-D MRI Volume with Box Filter

Load 3-D MRI data.

```
volData = load('mri');
vol = squeeze(volData.D);
```

Pad the image volume by the radius of the filter neighborhood.

```
filterSize = [5 5 3];
padSize = (filterSize-1)/2;
volPad = padarray(vol, padSize, 'replicate', 'both');
```

Calculate the 3-D integral image of the padded input.

```
intVol = integralImage3(volPad);
```

Filter the 3-D integral image with a [5 5 3] filter.

```
volFilt = integralBoxFilter3(intVol, filterSize);
```

Input Arguments

intA — Integral image to be filtered

real, non-sparse 3-D double array

Integral image to be filtered, specified as a real, non-sparse 3-D array of class `double`.

`integralBoxFilter3` expects the input integral image, `intA`, to be an upright integral image computed using `integralImage3`. `integralBoxFilter3` does not support rotated integral images. The first row, column and page of the integral image is assumed to be padded, as returned by `integralImage3`.

Example: `B = integralBoxFilter3(A);`

Data Types: `double`

filterSize — Size of box filter

3-by-3-by-3 box filter (default) | scalar or 3-element vector of positive, odd integers

Size of box filter, specified as a scalar or 3-element vector of positive, odd integers. If `filterSize` is scalar, `integralBoxFilter3` uses a cube box filter.

Example: `B = integralBoxFilter3(A,5);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `B = integralBoxFilter3(A,5,'NormalizationFactor',1);`

'NormalizationFactor' — Normalization factor applied to box filter

`1/filterSize.^3`, if scalar, and `1/prod(filterSize)`, if vector (default) | numeric scalar

Normalization factor applied to box filter, specified as a numeric scalar.

The default 'NormalizationFactor' has the effect of a mean filter—the pixels in the output image are the local means of the image. To get local area sums, set 'NormalizationFactor' to 1. To avoid overflow in such circumstances, consider using double precision images by converting the input image to class `double`.

```
Example: B = integralBoxFilter3(A,5,'NormalizationFactor',1);
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

B — Filtered image

real, nonsparse 3-D array

Filtered image, returned as a real, nonsparse 3-D array of class `double`.

`integralBoxFilter3` returns only the parts of the filtering that are computed without padding.

See Also

See Also

`imboxfilt3` | `integralimage3`

Topics

“Integral Image”

Introduced in R2015b

integralImage

Calculate integral image

Syntax

```
J = integralImage(I)
J = integralImage(I,orientation)
```

Description

`J = integralImage(I)` calculates the “Integral Image” on page 1-1321 , `J`, from the intensity image, `I`.

`J = integralImage(I,orientation)` calculates the integral image with the orientation specified by `orientation`.

Examples

Use Integral Image to Compute Region Sums

Create a simple sample image. For this example, sum the 2-by-2 rectangular region starting at row 1, column 3 (value 1) and extending to row 2, column 4 (value 14). In this trivial example, it's easy to calculate the sum of the pixels in the region: $1 + 7 + 8 + 14 = 30$.

```
I = magic(5)
```

```
I =
```

```
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Create an integral image of the sample image. The value of each pixel in the integral image is the sum of the pixel above it and the pixel to its left. Note how `integralImage` adds a padding row to the top and left sides of the image.

```
J = integralImage(I)
```

```
J =
```

```

0     0     0     0     0     0
0    17    41    42    50    65
0    40    69    77    99   130
0    44    79   100   142   195
0    54   101   141   204   260
0    65   130   195   260   325

```

Sum the rectangular region in the integral image. In this calculation, you extend the rectangular region you sum. The coordinates of the four corners are: (startRow,startCol), (startRow,endCol+1),(endRow,startCol), and (endRow+1,endCol+1)

```
regionSum = (J(1,3) + J(3,5)) - (J(1,5) + J(3,3))
```

```
regionSum = 30
```

Compute Integral Image with Rotated Orientation

Create sample image.

```
I = magic(5)
```

```
I =
```

```

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

```

```
% Define rotated rectangular region as [x, y, width, height] where x, y
% denote the indices of the top corner of the rectangle. Width and height
% are along 45 degree lines from the top corner.
```

```
[x, y, w, h] = deal(3, 1, 3, 2);
```

Create integral image.

```
J = integralImage(I, 'rotated');
```

Compute the sum over the region using the integral image.

```
regionSum = J(y+w+h,x+w-h+1) + J(y,x+1) - J(y+h,x-h+1) - J(y+w,x+w+1);
```

Input Arguments

I — Input grayscale image

real, nonsparse 2-D matrix

Input grayscale image, specified as a real, nonsparse 2-D matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

orientation — Image orientation

'upright' (default) | 'rotated'

Image orientation, specified as 'upright' or 'rotated'. If you set the orientation to 'rotated', `integralImage` returns the integral image for computing sums over rectangles rotated by 45 degrees. To facilitate computation of pixel sums along all image boundaries, the `integralImage` pads the output integral images as follows:

Integral Image	Description
Upright integral image	Zero-padded on top and left, resulting in <code>size(J) = size(I)+1</code>
Rotated integral image	Zero-padded at the top, left, and right, resulting in <code>size(J) = size(I)+[1 2]</code>

If the input image has more than two dimensions (`ndims(I)>2`), such as for an RGB image, the `integralImage` function computes the integral image for all 2-D planes along the higher dimensions.

Data Types: `char`

Output Arguments

J — Integral image

real, nonsparse matrix of class `double`

Integral image, returned as a real, nonsparse matrix of class `double`. The function zero-pads the top and left side of the integral image so the size of the output integral image is the size as the input image, plus 1, `size(J) = size(I)+1`. Such sizing facilitates easy computation of pixel sums along all image boundaries. The integral image, `J`, is essentially a padded version of the value `cumsum(cumsum(I,2))`.

Definitions

Integral Image

In an *integral image*, every pixel is the summation of the pixels above and to the left of it. Using an integral image, you can rapidly calculate summations over image subregions. Use of integral images was popularized by the Viola-Jones algorithm. Integral images facilitate summation of pixels and can be performed in constant time, regardless of the neighborhood size.

References

- [1] Viola, Paul and Michael J. Jones, “Rapid Object Detection using a Boosted Cascade of Simple Features”, *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2001. Volume: 1, pp.511–518.

See Also

See Also

`cumsum` | `integralBoxFilter`

Topics

“Integral Image”

Introduced in R2015b

integralImage3

Calculate 3-D integral image

Syntax

```
J = integralImage3(I)
```

Description

`J = integralImage3(I)` calculates the integral image, `J`, from the input intensity image, `I`.

Examples

Compute Integral Image of 3-D Input Image

Create a 3-D input image.

```
I = reshape(1:125,5,5,5);
```

Define a 3-by-3-by-3 sub-volume as [`startRow`, `startCol`, `startPlane`, `endRow`, `endCol`, `endPlane`].

```
[sR, sC, sP, eR, eC, eP] = deal(2, 2, 2, 4, 4, 4);
```

Create an integral image from the input image and compute the sum over a 3-by-3-by-3 sub-volume of `I`.

```
J = integralImage3(I);  
regionSum = J(eR+1,eC+1,eP+1) - J(eR+1,eC+1,sP) - J(eR+1,sC,eP+1) ...  
           - J(sR,eC+1,eP+1) + J(sR,sC,eP+1) + J(sR,eC+1,sP) ...  
           + J(eR+1,sC,sP) - J(sR,sC,sP)
```

```
regionSum = 1701
```

Verify that the sum of pixels is accurate.

```
sum(sum(sum(I(sR:eR, sC:eC, sP:eP))))
```



```
ans = 1701
```

Input Arguments

I — Input intensity image

real, nonsparse 3-D array

Input intensity image, specified as a real, nonsparse 3-D array of any numeric class.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output Arguments

J — Integral image

real, nonsparse matrix of class `double`

Integral image, returned as a real, nonsparse matrix of class `double`. The function zero-pads the top, left and along the first plane, resulting in `size(J) = size(I) + 1`. side of the integral image. The class of the output is `double`. The resulting size of the output integral image equals: `size(J) = size(I) + 1`. Such sizing facilitates easy computation of pixel sums along all image boundaries. The integral image, `J`, is essentially a padded version of the value `cumsum(cumsum(cumsum(I),2),3)`.

See Also

See Also

`integralBoxFilter3` | `integralImage`

Introduced in R2015b

interfileinfo

Read metadata from Interfile file

Syntax

```
info = interfileinfo(filename)
```

Description

`info = interfileinfo(filename)` returns a structure whose fields contain information about an image in a Interfile file. `filename` is a character vector that specifies the name of the file. The file must be in the current directory or in a directory on the MATLAB path.

The Interfile file format was developed for the exchange of nuclear medicine data. In Interfile 3.3, metadata is stored in a header file, separate from the image data. The two files have the same name with different file extensions. The header file has the file extension `.hdr` and the image file has the file extension `.img`. For more information about this file format, visit the Interfile Archive, maintained by the Department of Medical Physics and Bioengineering, University College, London, UK.

Examples

Read metadata from an Interfile file.

```
info = interfileinfo('MyFile.hdr');
```

See Also

`interfileread`

Introduced before R2006a

interfileread

Read images in Interfile format

Syntax

```
A = interfileread(filename)
A = interfileread(filename, window)
```

Description

`A = interfileread(filename)` reads the images in the first energy window of `filename` into `A`, where `A` is an `M`-by-`N` array for a single image and an `M`-by-`N`-by-`P` array for multiple images. The file must be in the current directory or in a directory on the MATLAB path.

`A = interfileread(filename, window)` reads the images in the energy window specified by `window` of `filename` into `A`.

The images in the energy window must be of the same size.

The Interfile file format was developed for the exchange of nuclear medicine data. In Interfile 3.3, metadata is stored in a header file, separate from the image data. The two files have the same name with different file extensions. The header file has the file extension `.hdr` and the image file has the file extension `.img`. For more information about this file format, visit the Interfile Archive, maintained by the Department of Medical Physics and Bioengineering, University College, London, UK.

Examples

Read image data from an Interfile file.

```
img = interfileread('MyFile.hdr');
```

See also

`interfileinfo`

Introduced before R2006a

intlut

Convert integer values using lookup table

Syntax

```
B = intlut(A, LUT)
```

Description

`B = intlut(A, LUT)` converts values in array `A` based on lookup table `LUT` and returns these new values in array `B`. For example, if `A` is a vector whose k th element is equal to `alpha`, then `B(k)` is equal to the `LUT` value corresponding to `alpha`, i.e., `LUT(alpha+1)`.

Class Support

`A` can be `uint8`, `uint16`, or `int16`. If `A` is `uint8`, `LUT` must be a `uint8` vector with 256 elements. If `A` is `uint16` or `int16`, `LUT` must be a vector with 65536 elements that has the same class as `A`. `B` has the same size and class as `A`.

Examples

Convert Integer Values using Lookup Table

Create an array of integers.

```
A = uint8([1 2 3 4; 5 6 7 8; 9 10 0 1])
```

```
A = 3x4 uint8 matrix
```

```
 1   2   3   4
 5   6   7   8
 9  10   0   1
```

Create a lookup table. In this example, the lookup table is created by following the vector `[2 4 8 16]` with repeated copies of the vector `[0 150 200 250]`.

```
LUT = [2 4 8 16 repmat(uint8([0 150 200 255]),1,63)];
```

Convert the values of *A* by referring to the lookup table. Note that the first index of the lookup table is 0.

```
B = intlut(A, LUT)
```

```
B = 3×4 uint8 matrix
```

```
     4     8    16     0
    150    200    255     0
    150    200     2     4
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

See Also

[ind2gray](#) | [rgb2ind](#)

Introduced before R2006a

iptaddcallback

Add function handle to callback list

Syntax

```
ID = iptaddcallback(obj,callback,func_handle)
```

Description

`ID = iptaddcallback(obj,callback,func_handle)` adds the function handle `func_handle` to the list of functions to be called when the callback specified by `callback` executes. `callback` specifies the name of a callback property of the graphics object specified.

`iptaddcallback` returns a unique callback identifier, `ID`, that can be used with `iptremovecallback` to remove the function from the callback list.

`iptaddcallback` can be useful when you need to notify more than one tool about the same callback event for a single object.

Note

Callback functions that have already been added to an object using the `set` command continue to work after you call `iptaddcallback`. The first time you call `iptaddcallback` for a given object and callback, the function checks to see if a different callback function is already installed. If a callback is already installed, `iptaddcallback` replaces that callback function with the `iptaddcallback` callback processor, and then adds the preexisting callback function to the `iptaddcallback` list.

Examples

Create a figure and register two callback functions. Whenever MATLAB detects mouse motion over the figure, function handles `f1` and `f2` are called in the order in which they were added to the list.

```
figobj = figure;  
f1 = @(varargin) disp('Callback 1');  
f2 = @(varargin) disp('Callback 2');  
iptaddcallback(figobj, 'WindowButtonMotionFcn', f1);  
iptaddcallback(figobj, 'WindowButtonMotionFcn', f2);
```

See Also

`iptremovecallback`

Introduced before R2006a

iptcheckconn

Check validity of connectivity argument

Syntax

```
iptcheckconn(conn, func_name, var_name, arg_pos)
```

Description

`iptcheckconn(conn, func_name, var_name, arg_pos)` checks whether `conn` is a valid connectivity argument. If it is invalid, the function issues a formatted error message.

A connectivity argument can be one of the following scalar values: 1, 4, 6, 8, 18, or 26. A connectivity argument can also be a 3-by-3-by- ... -by-3 array of 0's and 1s. The central element of a connectivity array must be nonzero and the array must be symmetric about its center.

`func_name` specifies the name used in the formatted error message to identify the function checking the connectivity argument.

`var_name` specifies the name used in the formatted error message to identify the argument being checked.

`arg_pos` is a positive integer that indicates the position of the argument being checked in the function argument list. `iptcheckconn` includes this information in the formatted error message.

Class Support

`conn` must be of class `double` or `logical` and must be real and nonsparse.

Examples

Create a 4-by-4 array and pass it as the connectivity argument.

```
iptcheckconn(eye(4), 'func_name', 'var_name', 2)
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, all input arguments must be compile-time constants.

Introduced before R2006a

iptcheckhandle

Check validity of handle

Syntax

```
iptcheckhandle(obj, valid_types, func_name, var_name, arg_pos)
```

Description

`iptcheckhandle(obj, valid_types, func_name, var_name, arg_pos)` checks the validity of the object `obj` and issues a formatted error message if the handle is invalid. `obj` must be a single figure, uipanel, hggroup, axes, or image object.

`valid_types` is a cell array of character vectors specifying the set of MATLAB graphics object types to which `obj` is expected to belong. For example, if you specify `{'uipanel', 'figure'}`, `obj` can be a handle to either a uipanel object or a figure object.

`func_name` specifies the name used in the formatted error message to identify the function performing the check.

`var_name` specifies the name used in the formatted error message to identify the argument being checked.

`arg_pos` is a positive integer that indicates the position of the argument being checked in the function argument list. `iptcheckhandle` converts this value to an ordinal number and includes this information in the formatted error message.

Examples

To trigger the error message, create a figure that does not contain an axes object and then check for a valid axes handle.

```
fig = figure; % create figure without an axes
iptcheckhandle(fig,{'axes'},'my_function','my_variable',2)
```

The following shows the format of the error message and indicates which parts you can customize using `iptcheckhandle` arguments.

```
      func_name      arg_pos      var_name
      ↓             ↓             ↓
Function MY_FUNCTION expected its second input argument, my_variable,
to be a handle of one of these types:
axes ←————— valid_types
Instead, its type was: figure.
```

See Also

`iptcheckinput` | `iptcheckmap` | `iptchecknargin` | `iptcheckstrs` |
`iptnum2ordinal`

Introduced before R2006a

iptcheckinput

Check validity of array

Compatibility

iptcheckinput will be removed in a future release. Use `validateattributes` instead.

Syntax

```
iptcheckinput(A, classes, attributes, func_name, var_name, arg_pos)
```

Description

`iptcheckinput(A, classes, attributes, func_name, var_name, arg_pos)` checks the validity of the array `A` and issues a formatted error message if it is invalid.

`classes` is a cell array of character vectors specifying the set of classes to which `A` is expected to belong. For example, if you specify `classes` as `{'logical' 'cell'}`, `A` is required to be either a logical array or a cell array. Use `'numeric'` as an abbreviation for the classes `uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`, `single`, and `double`.

`attributes` is a cell array of character vectors specifying the set of attributes that `A` must satisfy. For example, if `attributes` is `{'real' 'nonempty' 'finite'}`, `A` must be real and nonempty, and it must contain only finite values. The following table lists the supported attributes in alphabetical order.

2d	nonempty	odd	twod
column	nonnan	positive	vector
even	nonnegative	real	
finite	nonsparse	row	
integer	nonzero	scalar	

`func_name` specifies the name used in the formatted error message to identify the function checking the input.

`var_name` specifies the name used in the formatted error message to identify the argument being checked.

`arg_pos` is a positive integer that indicates the position of the argument being checked in the function argument list. `iptcheckinput` converts this value to an ordinal number and includes this information in the formatted error message.

Examples

To trigger this error message, create a three-dimensional array and then check for the attribute '2d'.

```
A = [ 1 2 3; 4 5 6 ];
B = [ 7 8 9; 10 11 12];
C = cat(3,A,B);
iptcheckinput(C,{'numeric'},{'2d'}, 'func_name', 'var_name', 2)
```

The following shows the format of the error message and indicates which parts you can customize using `iptcheckinput` arguments.

```
          func_name          arg_pos          var_name
          ↓                  ↓                ↓
Function FUNC_NAME expected its second input, var_name, to be
two-dimensional.
          ↑
        attributes
```

See Also

`iptcheckhandle` | `iptcheckmap` | `iptchecknargin` | `iptcheckstrs` | `iptnum2ordinal`

Introduced before R2006a

iptcheckmap

Check validity of colormap

Syntax

```
iptcheckmap(map, func_name, var_name, arg_pos)
```

Description

`iptcheckmap(map, func_name, var_name, arg_pos)` checks the validity of the MATLAB colormap and issues a formatted error message if it is invalid.

`func_name` specifies the name used in the formatted error message to identify the function checking the colormap.

`var_name` specifies the name used in the formatted error message to identify the argument being checked.

`arg_pos` is a positive integer that indicates the position of the argument being checked in the function argument list. `iptcheckmap` includes this information in the formatted error message.

Examples

```
bad_map = ones(10);  
iptcheckmap(bad_map, 'func_name', 'var_name', 2)
```

The following shows the format of the error message and indicates which parts you can customize using `iptcheckmap` arguments.

```
func_name          arg_pos  var_name  
  ↓                ↓          ↓  
Function FUNC_NAME expected input number 2, var_name, to be a valid colormap.  
Valid colormaps must be nonempty, double, 2-D matrices with 3 columns.
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.

See Also

`iptcheckhandle` | `iptcheckinput` | `iptchecknargin` | `iptcheckstrs`

Introduced before R2006a

iptchecknargin

Check number of input arguments

Compatibility

`iptchecknargin` will be removed in a future release. Use `narginchk` instead.

Syntax

```
iptchecknargin(low, high, num_inputs, func_name)
```

Description

`iptchecknargin(low, high, num_inputs, func_name)` checks whether `num_inputs` is in the range indicated by `low` and `high`. If not, `iptchecknargin` issues a formatted error message.

`low` should be a scalar nonnegative integer.

`high` should be a scalar nonnegative integer or `Inf`.

`func_name` specifies the name used in the formatted error message to identify the function checking the handle.

Examples

Create a function and use `iptchecknargin` to check that the number of arguments passed to the function is within the expected range.

```
function test_function(varargin)
iptchecknargin(1,3,nargin,mfilename);
```

Trigger the error message by executing the function at the MATLAB command line, specifying more than the expected number of arguments.

```
test_function(1,2,3,4)
```

See Also

`iptcheckhandle` | `iptcheckinput` | `iptcheckmap` | `iptcheckstrs` |
`iptnum2ordinal`

Introduced before R2006a

iptcheckstrs

Check validity of option

Compatibility

iptcheckstrs will be removed in a future release. Use `validatestring` instead.

Syntax

```
out = iptcheckstrs(in, valid_strs, func_name, var_name, arg_pos)
```

Description

`out = iptcheckstrs(in, valid_strs, func_name, var_name, arg_pos)` checks the validity of the option `in`. If the character vector matches one of the character vectors in the cell array `valid_strs`, `iptcheckstrs` returns the valid character vector in `out`. If the character vector does not match, `iptcheckstrs` issues a formatted error message. `iptcheckstrs` looks for a case-insensitive, nonambiguous match between `in` and the values in `valid_strs`.

`valid_strs` is a cell array of character vectors.

`func_name` specifies the name of the function doing the checking in the formatted error message.

`var_name` specifies the name used in the formatted error message to identify the argument being checked.

`arg_pos` is a positive integer that indicates the position of the argument being checked in the function argument list. `iptcheckstrs` converts this value to an ordinal number and includes this information in the formatted error message.

Examples

To trigger this error message, define a cell array of character vectors and pass in a character vector that isn't in the cell array.

```
iptcheckstrs('option3',{ 'option1','option2'},...  
            'func_name','var_name',2)
```

The following shows the format of the error message and indicates which parts you can customize using `iptcheckhandle` arguments.

```
      func_name          arg_pos          var_name  
      ↓                ↓                ↓  
Function FUNC_NAME expected its second input argument, var_name,  
to match one of these strings:  
  
option1, option2 ← valid_strs
```

The input, 'option3', did not match any of the valid strings.

See Also

`iptcheckhandle` | `iptcheckinput` | `iptcheckmap` | `iptchecknargin` | `iptnum2ordinal`

Introduced before R2006a

iptdemos

Index of Image Processing Toolbox examples

Syntax

iptdemos

Description

iptdemos displays the HTML page that lists all the Image Processing examples.

iptdemos displays the page in the MATLAB Help browser.

Introduced before R2006a

iptgetapi

Get Application Programmer Interface (API) for handle

Syntax

```
API = iptgetapi(h)
```

Description

`API = iptgetapi(h)` returns the API structure associated with handle `h` if there is one. Otherwise, `iptgetapi` returns an empty array.

To view functions that use this type of API, see `immagbox`, `imdistline`, or `imscrollpanel`.

Examples

```
hFig = figure('Toolbar','none',...  
             'Menubar','none');  
hIm = imshow('tape.png');  
hSP = imscrollpanel(hFig,hIm);  
api = iptgetapi(hSP);  
api.setMagnification(2) % 2X = 200%
```

See Also

`immagbox` | `imdistline` | `imscrollpanel`

Introduced before R2006a

iptGetPointerBehavior

Retrieve pointer behavior from graphics object

Syntax

```
pointerBehavior = iptGetPointerBehavior(obj)
```

Description

`pointerBehavior = iptGetPointerBehavior(obj)` returns the pointer behavior structure associated with the graphics object `obj`. A pointer behavior structure contains function handles that interact with a figure's pointer manager (see `iptPointerManager`) to control what happens when the figure's mouse pointer moves over and then exits the object. See `iptSetPointerBehavior` for details.

If `obj` does not contain a pointer behavior structure, `iptGetPointerBehavior` returns `[]`.

See Also

`iptPointerManager` | `iptSetPointerBehavior`

Introduced in R2006a

iptgetpref

Get values of Image Processing Toolbox preferences

Syntax

```
prefs = iptgetpref  
value = iptgetpref(prefname)
```

Description

`prefs = iptgetpref` returns a structure containing all the Image Processing Toolbox preferences with their current values. Each field in the structure has the name of an Image Processing Toolbox preference.

`value = iptgetpref(prefname)` returns the value of the Image Processing Toolbox preference specified by the character vector `prefname`. See `iptprefs` for a complete list of valid preference names or access the Image Processing preferences dialog box from the **File** menu in the MATLAB desktop. Preference names are not case sensitive and can be abbreviated.

Examples

```
value = iptgetpref('ImshowAxesVisible')
```

```
value =
```

```
off
```

See Also

`imshow` | `iptprefs` | `iptsetpref`

Introduced before R2006a

ipticondir

Directories containing IPT and MATLAB icons

Syntax

```
[D1, D2] = ipticondir
```

Description

[D1, D2] = `ipticondir` returns the names of the directories containing the Image Processing Toolbox icons (D1) and the MATLAB icons (D2).

Examples

```
[iptdir, MATLABdir] = ipticondir  
dir(iptdir)
```

See Also

`imtool`

Introduced before R2006a

iptnum2ordinal

Convert positive integer to ordinal character vector

Syntax

```
ordstr = iptnum2ordinal(number)
```

Description

`ordstr = iptnum2ordinal(number)` converts the positive integer `number` to the ordinal character vector `ordstr`.

Examples

The following example returns the character vector 'fourth'.

```
str = iptnum2ordinal(4)
```

The following example returns the character vector '23rd'.

```
str = iptnum2ordinal(23)
```

Introduced before R2006a

iptPointerManager

Create pointer manager in figure

Syntax

```
iptPointerManager(hFigure)  
iptPointerManager(hFigure, 'disable')  
iptPointerManager(hFigure, 'enable')
```

Description

`iptPointerManager(hFigure)` creates a pointer manager in the specified figure. The pointer manager controls pointer behavior for graphics objects in the figure that contain pointer behavior structures. Use `iptSetPointerBehavior` to associate a pointer behavior structure with a particular object to define specific actions that occur when the mouse pointer moves over and then leaves the object. See `iptSetPointerBehavior` for more information.

`iptPointerManager(hFigure, 'disable')` disables the figure's pointer manager.

`iptPointerManager(hFigure, 'enable')` enables and updates the figure's pointer manager.

Note: If the figure already contains a pointer manager, `iptPointerManager(hFigure)` does not create a new one. It has the same effect as `iptPointerManager(hFigure, 'enable')`.

Examples

Plot a line. Create a pointer manager in the figure. Then, associate a pointer behavior structure with the line object in the figure that changes the mouse pointer into a fleur whenever the pointer is over it.

```
h = plot(1:10);
```

```
iptPointerManager(gcf);  
enterFcn = @(hFigure, currentPoint)...  
    set(hFigure, 'Pointer', 'fleur');  
iptSetPointerBehavior(h, enterFcn);
```

Tips

`iptPointerManager` considers not just the object the pointer is over, but all objects in the figure. `iptPointerManager` searches the HG hierarchy to find the first object that contains a pointer behavior structure. The `iptPointerManager` then executes that object's pointer behavior function. For example, you could set the pointer to be a fleur and associate that pointer with the axes. Then, when you slide the pointer into the figure window, it will initially be the default pointer, then change to a fleur when you cross into the axes, and remain a fleur when you slide over the objects parented to the axes.

See Also

`iptGetPointerBehavior` | `iptSetPointerBehavior`

Introduced in R2006a

iptprefs

Display Image Processing Toolbox Preferences dialog box

Syntax

`iptprefs`

Description

`iptprefs` opens the Image Processing Toolbox Preferences dialog box, part of the MATLAB Preferences dialog box. You can also open this dialog box by clicking **Preferences** on the Home tab, in the Environment section.

The Image Processing Toolbox Preferences dialog box contains display preferences for `imshow`, `imshowinfo`, and provides an option for enabling hardware optimizations. For a list of all supported preferences with information about how to set them at the command line, see `iptsetpref`. The following figure shows how the preferences relate to options in the Preferences dialog box.

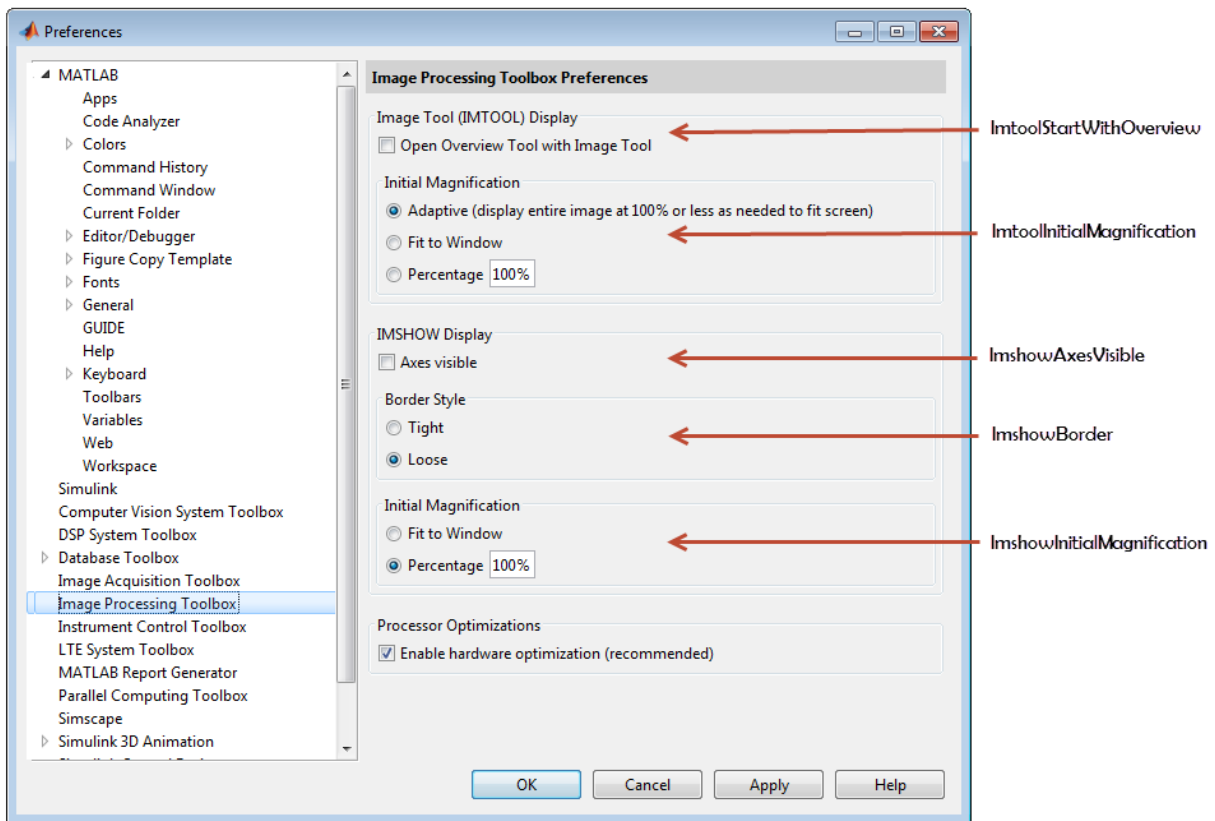


Image Processing Toolbox Preferences Dialog Box

See Also

`imshow` | `imtool` | `iptgetpref` | `iptsetpref` | `iptsetpref`

Introduced in R2009a

iptremovecallback

Delete function handle from callback list

Syntax

```
iptremovecallback(h,callback,ID)
```

Description

`iptremovecallback(h,callback,ID)` deletes a callback from the list of callbacks created by `imaddcallback` for the object with handle `h` and the associated callback, specified by the character vector `callback`. `ID` is the identifier of the callback to be deleted. This `ID` is returned by `iptaddcallback` when you add the function handle to the callback list.

Examples

Register three callbacks and try them interactively.

```
h = figure;  
f1 = @(varargin) disp('Callback 1');  
f2 = @(varargin) disp('Callback 2');  
f3 = @(varargin) disp('Callback 3');  
id1 = iptaddcallback(h, 'WindowButtonMotionFcn', f1);  
id2 = iptaddcallback(h, 'WindowButtonMotionFcn', f2);  
id3 = iptaddcallback(h, 'WindowButtonMotionFcn', f3);
```

Remove one of the callbacks and then move the mouse over the figure again. Whenever MATLAB detects mouse motion over the figure, function handles `f1` and `f3` are called in that order.

```
iptremovecallback(h, 'WindowButtonMotionFcn', id2);
```

See Also

`iptaddcallback`

Introduced before R2006a

iptSetPointerBehavior

Store pointer behavior structure in graphics object

Syntax

```
iptSetPointerBehavior(obj, pointerBehavior)
iptSetPointerBehavior(obj, [])
iptSetPointerBehavior(obj, enterFcn)
```

Description

`iptSetPointerBehavior(obj, pointerBehavior)` stores the specified pointer behavior structure in the specified graphics object, `obj`. If `obj` is an array of objects, `iptSetPointerBehavior` stores the same structure in each object.

When used with a figure's pointer manager (see `iptPointerManager`), a pointer behavior structure controls what happens when the figure's mouse pointer moves over and then exits an object in the figure. For details about this structure, see "Pointer Behavior Structure" on page 1-1355.

`iptSetPointerBehavior(obj, [])` clears the pointer behavior from the graphics object or objects.

`iptSetPointerBehavior(obj, enterFcn)` creates a pointer behavior structure, setting the `enterFcn` field to the function handle specified, and setting the `traverseFcn` and `exitFcn` fields to `[]`. See "Pointer Behavior Structure" on page 1-1355 for details about these fields. This syntax is provided as a convenience because, for most common uses, only the `enterFcn` is necessary.

Pointer Behavior Structure

A pointer behavior structure contains three fields: `enterFcn`, `traverseFcn`, and `exitFcn`. You set the value of these fields to function handles and use the `iptSetPointerBehavior` function to associate this structure with a graphics object in a figure. If the figure has a pointer manager installed, the pointer manager calls these functions when the following events occur. If you set a field to `[]`, no action is taken.

Function Handle	When Called
<code>enterFcn</code>	Called when the mouse pointer moves over the object.
<code>traverseFcn</code>	Called once when the mouse pointer moves over the object, and called again each time the mouse moves within the object.
<code>exitFcn</code>	Called when the mouse pointer leaves the object.

When the pointer manager calls the functions you create, it passes two arguments: the figure object and the current position of the pointer.

Examples

Example 1

Change the mouse pointer to a fleur whenever it is over a specific object and restore the original pointer when the mouse pointer moves off the object. The example creates a patch object and associates a pointer behavior structure with the object. Because this scenario requires only an `enterFcn`, the example uses the `iptSetPointerBehavior(obj, enterFcn)` syntax. The example then creates a pointer manager in the figure. Note that the pointer manager takes care of restoring the original figure pointer.

```
patchobj = patch([.25 .75 .75 .25 .25],...  
                [.25 .25 .75 .75 .25], 'r');  
xlim([0 1]);  
ylim([0 1]);  
  
enterFcn = @(fig, currentPoint)...  
           set(fig, 'Pointer', 'fleur');  
iptSetPointerBehavior(patchobj, enterFcn);  
iptPointerManager(gcf);
```

Example 2

Change the appearance of the mouse pointer, depending on where it is within the object. This example sets up the pointer behavior structure, setting the `enterFcn` and `exitFcn` fields to `[]`, and setting `traverseFcn` to a function named `overMe` that handles the position-specific behavior. `overMe` is an example function (in `\toolbox\images`

\imdemos) that varies the mouse pointer depending on the location of the mouse within the object. For more information, edit `overMe`.

```
patchobj = patch([.25 .75 .75 .25 .25],...
                 [.25 .25 .75 .75 .25], 'r');
xlim([0 1])
ylim([0 1])

pointerBehavior.enterFcn = [];
pointerBehavior.exitFcn = [];
pointerBehavior.traverseFcn = @overMe;

iptSetPointerBehavior(patchobj, pointerBehavior);
iptPointerManager(gcf);
```

Example 3

Change the figure's title when the mouse pointer is over the object. In this scenario, `enterFcn` and `exitFcn` are used to achieve the desired effect, and `traverseFcn` is `[]`.

```
patchobj = patch([.25 .75 .75 .25 .25],...
                 [.25 .25 .75 .75 .25], 'r');
xlim([0 1])
ylim([0 1])

pointerBehavior.enterFcn = ...
    @(fig, currentPoint)...
        set(fig, 'Name', 'Over patch');
pointerBehavior.exitFcn = ...
    @(fig, currentPoint) set(fig, 'Name', '');
pointerBehavior.traverseFcn = [];

iptSetPointerBehavior(patchobj, pointerBehavior);
iptPointerManager(gcf)
```

See Also

`iptGetPointerBehavior` | `iptPointerManager`

Introduced in R2006a

iptsetpref

Set Image Processing Toolbox preferences or display valid values

Syntax

```
iptsetpref(prefname)  
iptsetpref(prefname,prefvalue)
```

Description

`iptsetpref(prefname)` displays the valid values for the Image Processing Toolbox preference specified by `prefname`.

`iptsetpref(prefname,prefvalue)` sets the Image Processing Toolbox preference specified by the `prefname` to the value specified by `prefvalue`. The setting persists until you change it.

You can also use the Image Processing Toolbox Preferences dialog box to set the preferences. To access the dialog box, click **Preferences** on the **Home** tab in the MATLAB desktop, or call the `iptprefs` function.

Examples

Set Image Processing Toolbox Preference

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

Input Arguments

prefname — Name of an Image Processing Toolbox preference

character vector

Name of an Image Processing Toolbox preference, specified as one of the following character vectors.

The following table details the available preferences and their syntaxes. Note that preference names are case insensitive and you can abbreviate them. The default value appears enclosed in braces ({}).

Image Processing Toolbox Preferences

Preference Name	Description
'ImshowAxesVisible'	<p>Controls whether <code>imshow</code> displays images with the axes box and tick labels. Possible values:</p> <p>'on' — Include axes box and tick labels.</p> <p>{ 'off' } — Do not include axes box and tick labels.</p>
'ImshowBorder'	<p>Controls whether <code>imshow</code> includes a border around the image in the figure window. Possible values:</p> <p>{ 'loose' } — Include a border between the image and the edges of the figure window, thus leaving room for axes labels, titles, etc.</p> <p>'tight' — Adjust the figure size so that the image entirely fills the figure.</p> <hr/> <p>Note: There still can be a border if the image is very small, or if there are other objects besides the image and its axes in the figure.</p> <hr/> <p>You can override this preference by specifying the 'Border' parameter when you call <code>imshow</code>.</p>
'ImshowInitialMagnification'	<p>Controls the initial magnification of the image displayed by <code>imshow</code>. Possible values:</p> <p>Any numeric value — <code>imshow</code> interprets numeric values as a percentage. The default value is 100. A magnification of 100% means that there should be one screen pixel for every image pixel.</p>

Preference Name	Description
	<p>'fit' — Scale the image so that it fits into the window in its entirety.</p> <p>You can override this preference by specifying the 'InitialMagnification' parameter when you call <code>imshow</code>, or by calling the <code>truesize</code> function manually after displaying the image.</p>
'ImtoolInitialMagnification'	<p>Controls the initial magnification of the image displayed by <code>imtool</code>. Possible values:</p> <p>{ 'adaptive' } — Display the entire image. If the image is too large to display on the screen at 100% magnification, display the image at the largest magnification that fits on the screen. This is the default.</p> <p>Any numeric value — Specify the magnification as a percentage. A magnification of 100% means that there should be one screen pixel for every image pixel.</p> <p>'fit' — Scale the image so that it fits into the window in its entirety.</p> <p>You can override this preference by specifying the 'InitialMagnification' parameter when you call <code>imtool</code>.</p>
'ImtoolStartWithOverview'	<p>Controls whether the Overview tool opens automatically when you open an image using the Image Tool (<code>imtool</code>). Possible values:</p> <p><code>true</code> — Overview tool opens when you open an image.</p> <p>{<code>false</code>} — Overview tool does not open when you open an image. This is the default behavior.</p>

Preference Name	Description
'VolumeViewerUseHardware'	<p>Controls whether the <code>volumeViewer</code> app uses OpenGL shaders on the local graphics hardware to optimize volume rendering. Possible values:</p> <p><code>{true}</code> — Enable hardware optimization.</p> <p><code>false</code> — Disable hardware optimization.</p> <p>NOTE: Setting this preference to <code>false</code> has the side effect of removing certain functionality from the app and will drastically slow down rendering performance. This preference should only be set to <code>false</code> in technical support scenarios to resolve problems with graphics drivers.</p>
'UseIPPL'	<p>Controls whether some toolbox functions use hardware optimization or not. Possible values:</p> <p><code>{true}</code> — Enable hardware optimization</p> <p><code>false</code> — Disable hardware optimization</p> <p>Note: Setting this preference value clears all loaded MEX-files.</p>

Data Types: char

prefvalue — Value you want to assign to an Image Processing Toolbox preference

character vector

Value you want to assign to an Image Processing Toolbox preference, specified as one of the values listed in the table in `prefname`.

Example: `iptsetpref('ImshowBorder','tight')`

Data Types: char

See Also

See Also

`imshow` | `imshow` | `iptgetpref` | `iptprefs` | Volume Viewer

Introduced before R2006a

iptwindowalign

Align figure windows

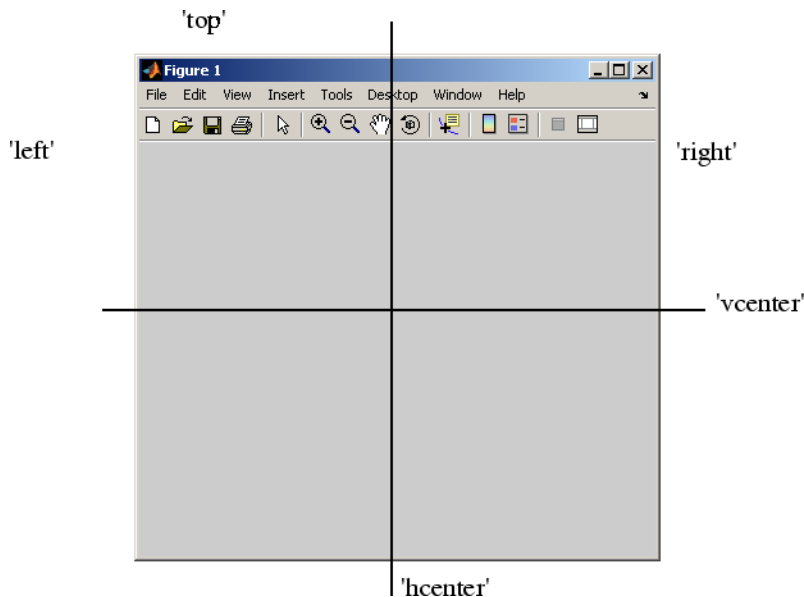
Syntax

```
iptwindowalign(fixed_fig, fixed_fig_edge, moving_fig,  
moving_fig_edge)
```

Description

`iptwindowalign(fixed_fig, fixed_fig_edge, moving_fig, moving_fig_edge)` moves the figure `moving_fig` to align it with the figure `fixed_fig`. `moving_fig` and `fixed_fig` are handles to figure objects.

`fixed_fig_edge` and `moving_fig_edge` describe the alignment of the figures in relation to their edges and can take any of the following values: 'left', 'right', 'hcenter', 'top', 'bottom', or 'vcenter'. 'hcenter' means center horizontally and 'vcenter' means center vertically. The following figure shows these alignments.



Notes

The two specified locations must be consistent in terms of their direction. For example, you cannot specify 'left' for `fixed_fig_edge` and 'bottom' for `moving_fig_edge`.

`iptwindowalign` constrains the position adjustment of `moving_fig` to keep it entirely visible on the screen.

`iptwindowalign` has no effect if either figure window is docked.

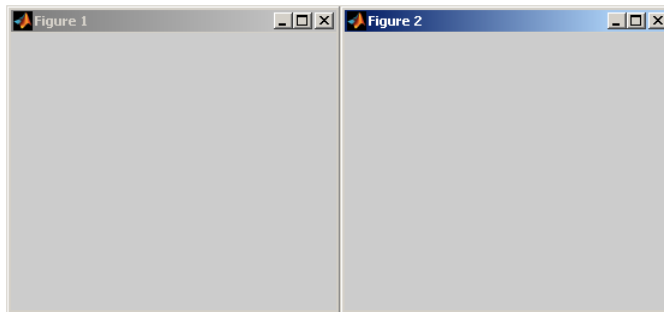
Examples

To illustrate some possible figure window alignments, first create two figures: `fig1` and `fig2`. Initially, `fig2` overlays `fig1` on the screen.

```
fig1 = figure;  
fig2 = figure;
```

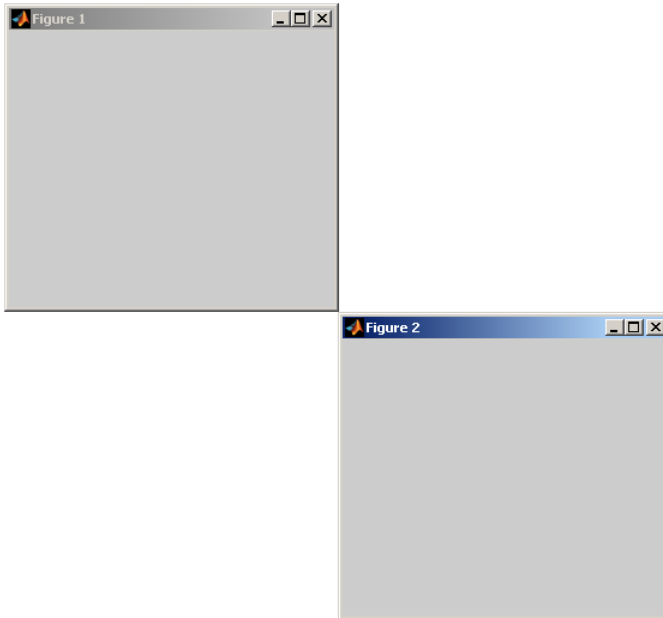
Use `iptwindowalign` to move `fig2` so its left edge is aligned with the right edge of `fig1`.

```
iptwindowalign(fig1, 'right', fig2, 'left');
```



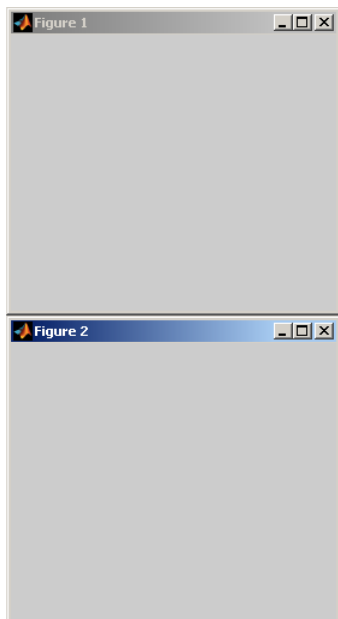
Now move `fig2` so its top edge is aligned with the bottom edge of `fig1`.

```
iptwindowalign(fig1, 'bottom', fig2, 'top');
```



Now move `fig2` so the two figures are centered horizontally.

```
iptwindowalign(fig1, 'hcenter', fig2, 'hcenter');
```



See Also

`imshow`

Introduced before R2006a

iradon

Inverse Radon transform

Syntax

```
I = iradon(R, theta)
I = iradon(R, theta, interp, filter, frequency_scaling, output_size)
[I, H] = iradon(...)
[ ___ ] = iradon(gpuarrayR, ___)
```

Description

`I = iradon(R, theta)` reconstructs the image `I` from projection data in the two-dimensional array `R`. The columns of `R` are parallel beam projection data. `iradon` assumes that the center of rotation is the center point of the projections, which is defined as `ceil(size(R,1)/2)`.

`theta` describes the angles (in degrees) at which the projections were taken. It can be either a vector containing the angles or a scalar specifying `D_theta`, the incremental angle between projections. If `theta` is a vector, it must contain angles with equal spacing between them. If `theta` is a scalar specifying `D_theta`, the projections were taken at angles `theta = m*D_theta`, where `m = 0, 1, 2, ..., size(R,2) - 1`. If the input is the empty matrix (`[]`), `D_theta` defaults to `180/size(R,2)`.

`iradon` uses the filtered back-projection algorithm to perform the inverse Radon transform. The filter is designed directly in the frequency domain and then multiplied by the FFT of the projections. The projections are zero-padded to a power of 2 before filtering to prevent spatial domain aliasing and to speed up the FFT.

`I = iradon(R, theta, interp, filter, frequency_scaling, output_size)` specifies parameters to use in the inverse Radon transform. You can specify any combination of the last four arguments. `iradon` uses default values for any of these arguments that you omit.

`interp` specifies the type of interpolation to use in the back projection. The available options are listed in order of increasing accuracy and computational complexity.

Value	Description
'nearest'	Nearest-neighbor interpolation
'linear'	Linear interpolation (the default)
'spline'	Spline interpolation
'pchip'	Shape-preserving piecewise cubic interpolation
'v5cubic'	Cubic interpolation from MATLAB 5. This method does not extrapolate, and it issues a warning and uses 'spline' if X is not equally spaced.

`filter` specifies the filter to use for frequency domain filtering. `filter` can be any of the following values:

Value	Description
'Ram-Lak'	Cropped Ram-Lak or ramp filter. This is the default. The frequency response of this filter is $ f $. Because this filter is sensitive to noise in the projections, one of the filters listed below might be preferable. These filters multiply the Ram-Lak filter by a window that deemphasizes high frequencies.
'Shepp-Logan'	Multiplies the Ram-Lak filter by a <code>sinc</code> function
'Cosine'	Multiplies the Ram-Lak filter by a <code>cosine</code> function
'Hamming'	Multiplies the Ram-Lak filter by a Hamming window
'Hann'	Multiplies the Ram-Lak filter by a Hann window
'None'	No filtering. When you specify this value, <code>iradon</code> returns unfiltered backprojection data.

`frequency_scaling` is a scalar in the range (0,1] that modifies the filter by rescaling its frequency axis. The default is 1. If `frequency_scaling` is less than 1, the filter is compressed to fit into the frequency range $[0, \text{frequency_scaling}]$, in normalized frequencies; all frequencies above `frequency_scaling` are set to 0.

`output_size` is a scalar that specifies the number of rows and columns in the reconstructed image. If `output_size` is not specified, the size is determined from the length of the projections.

```
output_size = 2*floor(size(R,1)/(2*sqrt(2)))
```

If you specify `output_size`, `iradon` reconstructs a smaller or larger portion of the image but does not change the scaling of the data. If the projections were calculated with the `radon` function, the reconstructed image might not be the same size as the original image.

`[I,H] = iradon(...)` returns the frequency response of the filter in the vector `H`.

`[___]= iradon(gpuarrayR, ___)` reconstructs the image `gpuarrayI` from projection data in the `gpuArray R`. The input image and the return values are 2-D `gpuArrays`. All other numeric arguments must be a `double` or a `gpuArray` of underlying class `double`. This syntax requires the Parallel Computing Toolbox.

Note: The GPU implementation of this function supports only nearest-neighbor and linear interpolation methods for the back projection.

Class Support

`R` can be `double` or `single`. All other numeric input arguments must be of class `double`. `I` has the same class as `R`. `H` is `double`.

`R` can be a `gpuArray` of underlying class `double` or `single`. All other numeric input arguments must be `double` or `gpuArray` of underlying class `double`. `I` has the same class as `R`. `H` is a `gpuArray` of underlying class `double`.

Examples

Compare Filtered and Unfiltered Backprojection

Create an image of the phantom. Display the image.

```
P = phantom(128);  
imshow(P)  
title('Original image')
```

Original image

Perform a Radon transform of the image.

```
R = radon(P,0:179);
```

Perform filtered backprojection.

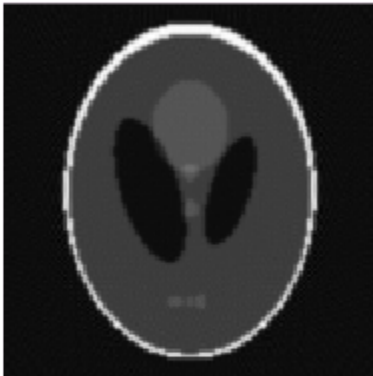
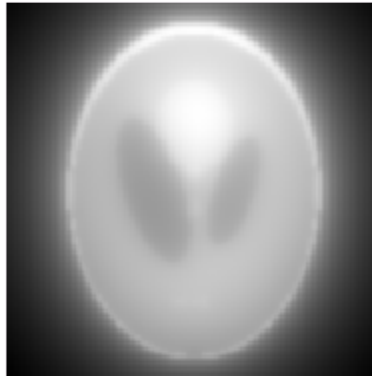
```
I1 = iradon(R,0:179);
```

Perform unfiltered backprojection.

```
I2 = iradon(R,0:179, 'linear', 'none');
```

Display the reconstructed images.

```
figure  
subplot(1,2,1)  
imshow(I1,[])  
title('Filtered Backprojection')  
subplot(1,2,2)  
imshow(I2,[])  
title('Unfiltered Backprojection')
```


Filtered Backprojection**Unfiltered Backprojection**

Examine Backprojection at a Single Angle

Create an image of the phantom.

```
P = phantom(128);
```

Perform a Radon transform of the image, then get the projection vector corresponding to a projection at a 45 degree angle.

```
R = radon(P,0:179);  
r45 = R(:,46);
```

Perform the inverse Radon transform of this single projection vector. The `iradon` syntax does not allow you to do this directly, because if `theta` is a scalar it is treated as an

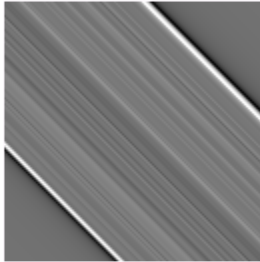
increment. You can accomplish the task by passing in two copies of the projection vector and then dividing the result by 2.

```
I = iradon([r45 r45], [45 45])/2;
```

Display the result.

```
imshow(I, [])  
title('Backprojection from 45 degrees')
```

Backprojection from 45 degrees



Calculate the inverse Radon transform on a GPU

Calculate the inverse Radon transform on a GPU.

```
P = gpuArray(phantom(128));  
R = radon(P,0:179);  
I1 = iradon(R,0:179);  
I2 = iradon(R,0:179,'linear','none');  
subplot(1,3,1), imshow(P), title('Original')  
subplot(1,3,2), imshow(I1), title('Filtered backprojection')  
subplot(1,3,3), imshow(I2,[]), title('Unfiltered backprojection')
```

Algorithms

`iradon` uses the filtered back projection algorithm to perform the inverse Radon transform. The filter is designed directly in the frequency domain and then multiplied

by the FFT of the projections. The projections are zero-padded to a power of 2 before filtering to prevent spatial domain aliasing and to speed up the FFT.

References

- [1] Kak, A. C., and M. Slaney, *Principles of Computerized Tomographic Imaging*, New York, NY, IEEE Press, 1988.

See Also

fan2para | fanbeam | ifanbeam | para2fan | phantom | radon

Introduced before R2006a

isbw

True for binary image

Syntax

```
flag = isbw(A)
```

Note: `isbw` has been removed.

Description

`flag = isbw(A)` returns 1 if `A` is a binary image and 0 otherwise.

The input image `A` is considered to be a binary image if it is a nonsparse logical array.

Class Support

The input image `A` can be any MATLAB array.

See Also

`isind` | `isgray` | `isrgb`

Introduced before R2006a

isflat

True for flat structuring element

Note: `isflat` will be removed in a future release. See `strel` for the current list of methods.

Syntax

TF = `isflat`(SE)

Description

TF = `isflat`(SE) returns true (1) if the structuring element SE is flat; otherwise it returns false (0). If SE is an array of STREL objects, then TF is the same size as SE.

Class Support

SE is a STREL object. TF is a double-precision value.

Introduced before R2006a

isgray

True for grayscale image

Syntax

```
flag = isgray(A)
```

Note: `isgray` has been removed.

Description

`flag = isgray(A)` returns 1 if `A` is a grayscale intensity image and 0 otherwise.

`isgray` uses these criteria to decide whether `A` is an intensity image:

- If `A` is of class `double`, all values must be in the range `[0,1]`, and the number of dimensions of `A` must be 2.
- If `A` is of class `uint16` or `uint8`, the number of dimensions of `A` must be 2.

Note A four-dimensional array that contains multiple grayscale images returns 0, not 1.

Class Support

The input image `A` can be of class `logical`, `uint8`, `uint16`, or `double`.

See Also

`isbw` | `isind` | `isrgb`

Introduced before R2006a

isicc

True for valid ICC color profile

Syntax

```
TF = isicc(P)
```

Description

`TF = isicc(P)` returns `True` if structure `P` is a valid ICC color profile; otherwise `False`.

`isicc` checks if `P` has a complete set of the tags required for an ICC profile. `P` must contain a `Header` field, which in turn must contain a `Version` field and a `DeviceClass` field. These fields, and others, are used to determine the set of required tags according to the ICC Profile Specification, either Version 2 (ICC.1:2001-04) or Version 4 (ICC.1:2001-12), which are available at www.color.org. The set of required tags is given in Section 6.3 in either version.

Examples

Read in an ICC profile and `isicc` returns `True`.

```
P = iccread('sRGB.icm');
```

```
TF = isicc(P)
```

```
TF =
```

```
1
```

This example creates a MATLAB structure and uses `isicc` to test if it's a valid ICC profile. `isicc` returns `False`.

```
S.name = 'Any Student';  
S.score = 83;
```

```
S.grade = 'B+'
```

```
TF = isicc(S)
```

```
TF =
```

```
    0
```

See Also

`applycform` | `iccread` | `iccwrite` | `makecform`

Introduced before R2006a

isind

True for indexed image

Syntax

```
flag = isind(A)
```

Note: `isind` has been removed.

Description

`flag = isind(A)` returns 1 if `A` is an indexed image and 0 otherwise.

`isind` uses these criteria to determine if `A` is an indexed image:

- If `A` is of class `double`, all values in `A` must be integers greater than or equal to 1, and the number of dimensions of `A` must be 2.
- If `A` is of class `uint8` or `uint16`, the number of dimensions of `A` must be 2.

Note A four-dimensional array that contains multiple indexed images returns 0, not 1.

Class Support

`A` can be of class `logical`, `uint8`, `uint16`, or `double`.

See Also

`isbw` | `isgray` | `isrgb`

Introduced before R2006a

isnif

Check if file is National Imagery Transmission Format (NITF) file

Syntax

```
[tf, NITF_version] = isnif(filename)
```

Description

`[tf, NITF_version] = isnif(filename)` returns **True** (1) if the file specified by `filename` is a National Imagery Transmission Format (NITF) file, otherwise **False** (0). If the file is a NITF file, `isnif` returns a character vector identifying the NITF version in `NITF_version`, such as `'2.1'`. If the file is not a NITF file, `NITF_version` contains the character vector `'UNK'`.

See Also

`nitfinfo` | `nitfread`

Introduced in R2007b

isrgb

True for RGB image

Syntax

```
flag = isrgb(A)
```

Note: `isrgb` has been removed.

Description

`flag = isrgb(A)` returns 1 if `A` is an RGB truecolor image and 0 otherwise.

`isrgb` uses these criteria to determine whether `A` is an RGB image:

- If `A` is of class `double`, all values must be in the range `[0,1]`, and `A` must be `m-by-n-by-3`.
- If `A` is of class `uint16` or `uint8`, `A` must be `m-by-n-by-3`.

Note A four-dimensional array that contains multiple RGB images returns 0, not 1.

Class Support

`A` can be of class `logical`, `uint8`, `uint16`, or `double`.

See Also

`isbw` | `isgray` | `isind`

Introduced before R2006a

isrset

Check if file is R-Set

Syntax

```
[tf, supported] = isrset(filename)
```

Description

`[tf, supported] = isrset(filename)` sets `tf` to true if the file `filename` is a reduced resolution dataset (R-Set) created by `rsetwrite` and false if it is not. The value of `supported` is true if the R-Set file is compatible with the R-Set tools (such as `imtool`) in the version of the Image Processing Toolbox you are using. If `supported` is false, the R-Set file was probably created by a newer version of `rsetwrite` than the one in the version of the Image Processing Toolbox you are using.

See Also

`rsetwrite`

Introduced in R2009a

lab2double

Convert $L^*a^*b^*$ data to double

Syntax

```
labd = lab2double(lab)
```

Description

`labd = lab2double(lab)` converts an M-by-3 or M-by-N-by-3 array of $L^*a^*b^*$ color values to class `double`. The output array `labd` has the same size as `lab`.

The Image Processing Toolbox software follows the convention that double-precision $L^*a^*b^*$ arrays contain 1976 CIE $L^*a^*b^*$ values. $L^*a^*b^*$ arrays that are `uint8` or `uint16` follow the convention in the ICC profile specification (ICC.1:2001-4, www.color.org) for representing $L^*a^*b^*$ values as unsigned 8-bit or 16-bit integers. The ICC encoding convention is illustrated by these tables.

Value (L^*)	uint8 Value	uint16 Value
0.0	0	0
100.0	255	65280
$100.0 + (25500/65280)$	None	65535

Value (a^* or b^*)	uint8 Value	uint16 Value
-128.0	0	0
0.0	128	32768
127.0	255	65280
$127.0 + (255/256)$	None	65535

Class Support

`lab` is a `uint8`, `uint16`, or `double` array that must be real and nonsparse. `labd` is `double`.

Examples

Convert L*a*b* Color Values to double

This example shows how to convert `uint8` L*a*b* values to `double`.

Create a `uint8` vector specifying the color white in L*a*b* colorspace.

```
w = uint8([255 128 128]);
```

Convert the L*a*b* color value to `double`.

```
lab2double(w)
```

```
ans =
```

```
    100     0     0
```

See Also

[applycform](#) | [lab2uint8](#) | [lab2uint16](#) | [makecform](#) | [whitepoint](#) | [xyz2double](#)
| [xyz2uint16](#)

Introduced before R2006a

lab2rgb

Convert CIE 1976 L*a*b* to RGB

Syntax

```
rgb = lab2rgb(lab)
rgb = lab2rgb(lab, Name, Value)
```

Description

`rgb = lab2rgb(lab)` converts CIE 1976 L*a*b* values to RGB values.

`rgb = lab2rgb(lab, Name, Value)` specifies additional options with one or more Name, Value pair arguments.

Examples

Convert L*a*b* Color to RGB

Convert a color value in the L*a*b* color space to standard RGB color space.

```
lab2rgb([70 5 10])
ans =
    0.7359    0.6566    0.6010
```

Convert L*a*b* Color to Adobe RGB

Convert a color value in L*a*b* color space to the Adobe RGB (1998) color space.

```
lab2rgb([70 5 10], 'ColorSpace', 'adobe-rgb-1998')
ans =
    0.7086    0.6507    0.5978
```

Convert L*a*b* Color to RGB Specifying Whitepoint

Convert an L*a*b* color value to standard RGB specifying the D50 whitepoint.

```
lab2rgb([70 5 10], 'WhitePoint', 'd50')
```

```
ans =
```

```
    0.7282    0.6573    0.6007
```

Convert L*a*b* Color to 8-bit-encoded RGB Color

Convert an L*a*b* color value to an 8-bit encoded RGB color value.

```
lab2rgb([70 5 10], 'OutputType', 'uint8')
```

```
ans = 1×3 uint8 row vector
```

```
    188    167    153
```

Input Arguments

lab — Color values to convert

p-by-3 matrix | m-by-n-by-3 image array | m-by-n-by-3-by-f image stack

Color values to convert, specified as a p-by-3 matrix of color values (one color per row), an m-by-n-by-3 image array, or an m-by-n-by-3-by-f image stack.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `lab2rgb([70 5 10], 'WhitePoint', 'd50')`

'ColorSpace' — Color space of the output RGB values

'srgb' (default) | 'adobe-rgb-1998' | 'linear-rgb'

Color space of the input RGB values, specified as 'srgb', 'adobe-rgb-1998', or 'linear-rgb'.

Data Types: char

'WhitePoint' — Reference white point

'd65' (default) | 'a' | 'c' | 'e' | 'd50' | 'd55' | 'icc' | 1-by-3 vector

Reference white point, specified as a 1-by-3 vector or one of the CIE standard illuminants, listed in the following table.

Value	White Point
'a'	CIE standard illuminant A, [1.0985, 1.0000, 0.3558]. Simulates typical, domestic, tungsten-filament lighting with correlated color temperature of 2856 K.
'c'	CIE standard illuminant C, [0.9807, 1.0000, 1.1822]. Simulates average or north sky daylight with correlated color temperature of 6774 K. Deprecated by CIE.
'e'	Equal-energy radiator. Useful as a theoretical reference. [1.000, 1.000, 1.000].
'd50'	CIE standard illuminant D50, [0.9642, 1.0000, 0.8251]. Simulates warm daylight at sunrise or sunset with correlated color temperature of 5003 K. Also known as <i>horizon light</i> .
'd55'	CIE standard illuminant D55, [0.9568, 1.0000, 0.9214]. Simulates mid-morning or mid-afternoon daylight with correlated color temperature of 5500 K.
'd65'	CIE standard illuminant D65, [0.9504, 1.0000, 1.0888]. Simulates noon daylight with correlated color temperature of 6504 K.
'icc'	Profile Connection Space (PCS) illuminant used in ICC profiles. Approximation of [0.962, 1.000, 0.8249] using fixed-point, signed, 32-bit numbers with 16 fractional bits. Actual value: [31595, 32768, 27030]/32768.

Data Types: single | double | char

'OutputType' — Data type of returned RGB values`'double' | 'single' | 'uint8' | 'uint16'`

Data type of returned RGB values, specified as one of the following values: `'double'`, `'single'`, `'uint8'`, or `'uint16'`. If you do not specify `OutputType`, the output type is the same type as the input.

Data Types: char

Output Arguments

rgb — Converted color values

array the same shape as the input

Converted color values, returned as an array the same shape as the input. The output type is the same as the input class unless you specify the type using the `'OutputType'` parameter.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, all character vector input arguments must be compile-time constants.

See Also

See Also

`lab2xyz` | `rgb2lab` | `rgb2xyz` | `xyz2lab` | `xyz2rgb`

Introduced in R2014b

lab2uint16

Convert $L^*a^*b^*$ data to `uint16`

Syntax

```
lab16 = lab2uint16(lab)
```

Description

`lab16 = lab2uint16(lab)` converts an M-by-3 or M-by-N-by-3 array of $L^*a^*b^*$ color values to `uint16`. `lab16` has the same size as `lab`.

The Image Processing Toolbox software follows the convention that double-precision $L^*a^*b^*$ arrays contain 1976 CIE $L^*a^*b^*$ values. $L^*a^*b^*$ arrays that are `uint8` or `uint16` follow the convention in the ICC profile specification (ICC.1:2001-4, www.color.org) for representing $L^*a^*b^*$ values as unsigned 8-bit or 16-bit integers. The ICC encoding convention is illustrated by these tables.

Value (L^*)	<code>uint8</code> Value	<code>uint16</code> Value
0.0	0	0
100.0	255	65280
100.0 + (25500/65280)	None	65535

Value (a^* or b^*)	<code>uint8</code> Value	<code>uint16</code> Value
-128.0	0	0
0.0	128	32768
127.0	255	65280
127.0 + (255/256)	None	65535

Class Support

`lab` can be a `uint8`, `uint16`, or `double` array that must be real and nonsparse. `lab16` is of class `uint16`.

Examples

Convert L*a*b* Color Values to uint16

This example shows how to convert L*a*b* color values from `double` to `uint16`.

Create a `double` vector specifying the color white in L*a*b* colorspace.

```
w = [100 0 0];
```

Convert the L*a*b* color value to `uint16`.

```
lab2uint16(w)
```

```
ans = 1×3 uint16 row vector
```

```
    65280    32768    32768
```

See Also

[applycform](#) | [lab2double](#) | [lab2uint8](#) | [makecform](#) | [whitepoint](#) | [xyz2double](#)
| [xyz2uint16](#)

Introduced before R2006a

lab2uint8

Convert $L^*a^*b^*$ data to uint8

Syntax

```
lab8 = lab2uint8(lab)
```

Description

`lab8 = lab2uint8(lab)` converts an M-by-3 or M-by-N-by-3 array of $L^*a^*b^*$ color values to `uint8`. `lab8` has the same size as `lab`.

The Image Processing Toolbox software follows the convention that double-precision $L^*a^*b^*$ arrays contain 1976 CIE $L^*a^*b^*$ values. $L^*a^*b^*$ arrays that are `uint8` or `uint16` follow the convention in the ICC profile specification (ICC.1:2001-4, www.color.org) for representing $L^*a^*b^*$ values as unsigned 8-bit or 16-bit integers. The ICC encoding convention is illustrated by these tables.

Value (L^*)	uint8 Value	uint16 Value
0.0	0	0
100.0	255	65280
$100.0 + (25500/65280)$	None	65535

Value (a^* or b^*)	uint8 Value	uint16 Value
-128.0	0	0
0.0	128	32768
127.0	255	65280
$127.0 + (255/256)$	None	65535

Class Support

`lab` is a `uint8`, `uint16`, or `double` array that must be real and nonsparse. `lab8` is `uint8`.

Examples

Convert L*a*b* Color Values to uint8

This example shows how to convert L*a*b* color values from `double` to `uint8`.

Create a `double` vector specifying the color white in L*a*b* colorspace.

```
w = [100 0 0];
```

Convert the L*a*b* color value to `uint8`.

```
lab2uint8(w)
```

```
ans = 1×3 uint8 row vector
```

```
    255    128    128
```

See Also

[applycform](#) | [lab2double](#) | [lab2uint16](#) | [makecform](#) | [whitepoint](#) | [xyz2double](#)
| [xyz2uint16](#)

Introduced before R2006a

lab2xyz

Convert CIE 1976 L*a*b* to CIE 1931 XYZ

Syntax

```
xyz = lab2xyz(lab)
xyz = lab2xyz(lab, Name, Value)
```

Description

`xyz = lab2xyz(lab)` converts CIE 1976 L*a*b* values to CIE 1931 XYZ values.

`xyz = lab2xyz(lab, Name, Value)` specifies additional options with one or more Name, Value pair arguments.

Examples

Convert L*a*b* Color to XYZ

Convert an L*a*b* color value to XYZ using the default reference white point, D65.

```
lab2xyz([50 10 -5])
ans =
    0.1942    0.1842    0.2282
```

Convert L*a*b* Color to XYZ Specifying Whitepoint

Convert an L*a*b* color value to XYZ specifying the D50 whitepoint.

```
lab2xyz([50 10 -5], 'WhitePoint', 'd50')
ans =
    0.1970    0.1842    0.1729
```


Input Arguments

lab — Color values to convert

P-by-3 matrix | M-by-N-by-3 image array | M-by-N-by-3-by-F image stack

Color values to convert, specified as a P-by-3 matrix of color values (one color per row), an M-by-N-by-3 image array, or an M-by-N-by-3-by-F image stack.

Example: `lab2xyz([0.25 0.40 0.10])`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `lab2xyz([0.25 0.40 0.10],'WhitePoint','d50')`

'WhitePoint' — Reference white point

'd65' (default) | 'a' | 'c' | 'e' | 'd50' | 'd55' | 'icc' | 1-by-3 vector

Reference white point, specified as a 1-by-3 vector or one of the CIE standard illuminants, listed in the following table.

Value	White Point
'a'	CIE standard illuminant A, [1.0985, 1.0000, 0.3558]. Simulates typical, domestic, tungsten-filament lighting with correlated color temperature of 2856 K.
'c'	CIE standard illuminant C, [0.9807, 1.0000, 1.1822]. Simulates average or north sky daylight with correlated color temperature of 6774 K. Deprecated by CIE.
'e'	Equal-energy radiator. Useful as a theoretical reference. [1.000, 1.000, 1.000].

Value	White Point
'd50'	CIE standard illuminant D50, [0.9642, 1.0000, 0.8251]. Simulates warm daylight at sunrise or sunset with correlated color temperature of 5003 K. Also known as <i>horizon light</i> .
'd55'	CIE standard illuminant D55, [0.9568, 1.0000, 0.9214]. Simulates mid-morning or mid-afternoon daylight with correlated color temperature of 5500 K.
'd65'	CIE standard illuminant D65, [0.9504, 1.0000, 1.0888]. Simulates noon daylight with correlated color temperature of 6504 K.
'icc'	Profile Connection Space (PCS) illuminant used in ICC profiles. Approximation of [0.962, 1.000, 0.8249] using fixed-point, signed, 32-bit numbers with 16 fractional bits. Actual value: [31595, 32768, 27030] / 32768.

Data Types: single | double | char

Output Arguments

xyz — Converted color values

array the same shape and type as the input

Converted color values, returned as an array the same shape and type as the input.

See Also

See Also

rgb2lab | rgb2xyz | xyz2lab | xyz2rgb

Introduced in R2014b

label2idx

Convert label matrix to cell array of linear indices

Syntax

```
pixelIndexList = label2idx(L)
```

Description

`pixelIndexList = label2idx(L)` converts the regions described by the label matrix `L` into the 1-by- N cell array of linear indices `pixelIndexList`.

Examples

Calculate Pixel Index List for Small Label Matrix

Create a small sample matrix containing three regions.

```
BW = logical([1 1 1 0 0 0 0 0
              1 1 1 0 1 1 0 0
              1 1 1 0 1 1 0 0
              1 1 1 0 0 0 0 0
              1 1 1 0 0 0 1 0
              1 1 1 0 0 0 1 0
              1 1 1 0 0 1 1 0
              1 1 1 0 0 0 0 0]);
```

Create a label matrix from this sample image.

```
L = bwlabel(BW)
```

L =

```

1     1     1     0     0     0     0     0
1     1     1     0     2     2     0     0
1     1     1     0     2     2     0     0
1     1     1     0     0     0     0     0
1     1     1     0     0     0     3     0
1     1     1     0     0     0     3     0
```

```
1     1     1     0     0     3     3     0
1     1     1     0     0     0     0     0
```

Get a linear index list of all the pixels in each region. The function returns a cell array with an element for each region it finds in the label matrix.

```
pixelIndexList = label2idx(L)

pixelIndexList = 1×3 cell array
    [24×1 double]    [4×1 double]    [4×1 double]
```

Examine one of the pixel index lists returned. For example, look at the second cell in the returned cell array. It contains the linear indices for all the pixels in the region labeled "2". The upper left corner of the region is pixel BW(2,5), which is the 34th pixel in linear indexing.

```
pixelIndexList{2}

ans =

    34
    35
    42
    43
```

Input Arguments

L — Label matrix

real, nonsparse, nonnegative, finite numeric N-D matrix

Label matrix, specified as a real, nonsparse, nonnegative, finite numeric N-D matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output Arguments

pixelIndexList — Linear indices of pixels in regions

1-by-*N* cell array

Linear indices of pixels in regions, returned as a 1-by- N cell array. Each element of the output, `pixelIndexList{N}`, is a vector that contains all the linear indices in L where L is equal to N .

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.

See Also

See Also

`labelmatrix` | `superpixels`

Introduced in R2016a

label2rgb

Convert label matrix into RGB image

Syntax

```
RGB = label2rgb(L)
RGB = label2rgb(L, map)
RGB = label2rgb(L, map, zerocolor)
RGB = label2rgb(L, map, zerocolor, order)
```

Description

`RGB = label2rgb(L)` converts a label matrix, `L`, such as those returned by `labelmatrix`, `bwlabel`, `bwlabeln`, or `watershed`, into an RGB color image for the purpose of visualizing the labeled regions. The `label2rgb` function determines the color to assign to each object based on the number of objects in the label matrix and range of colors in the colormap. The `label2rgb` function picks colors from the entire range.

`RGB = label2rgb(L, map)` specifies the colormap `map` to be used in the RGB image. `map` can have any of the following values:

- *n*-by-3 colormap matrix
- Name of a MATLAB colormap function, such as 'jet' or 'gray' (See `colormap` for a list of supported colormaps.)
- Function handle of a colormap function, such as `@jet` or `@gray`

If you do not specify `map`, the default value is 'jet'.

`RGB = label2rgb(L, map, zerocolor)` specifies the RGB color of the elements labeled 0 (zero) in the input label matrix `L`. As the value of `zerocolor`, specify an RGB triple or one of the colors listed in this table.

Value	Color
'b'	Blue

Value	Color
'c'	Cyan
'g'	Green
'k'	Black
'm'	Magenta
'r'	Red
'w'	White
'y'	Yellow

If you do not specify `zerocolor`, the default value for zero-labeled elements is `[1 1 1]` (white).

`RGB = label2rgb(L, map, zerocolor, order)` controls how `label2rgb` assigns colormap colors to regions in the label matrix. If `order` is `'noshuffle'` (the default), `label2rgb` assigns colormap colors to label matrix regions in numerical order. If `order` is `'shuffle'`, `label2rgb` assigns colormap colors pseudorandomly.

Class Support

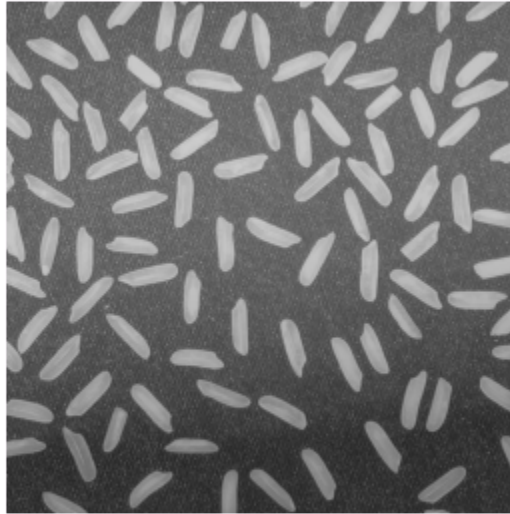
The input label matrix `L` can have any numeric class. It must contain finite, nonnegative integers. The output of `label2rgb` is of class `uint8`.

Examples

Use Color to Highlight Elements in a Label Matrix

Read an image and display it.

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



Create a label matrix from the image.

```
BW = im2bw(I, graythresh(I));  
CC = bwconncomp(BW);  
L = labelmatrix(CC);
```

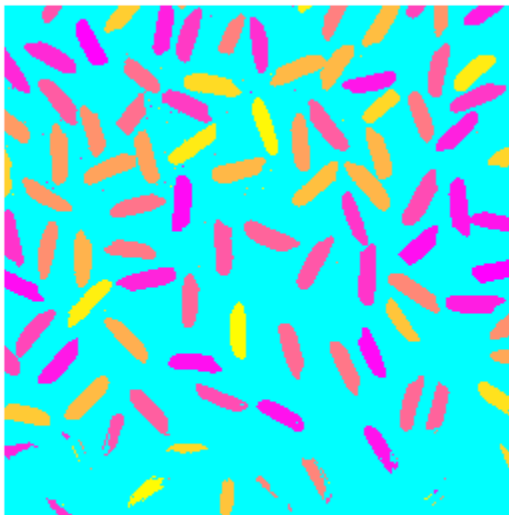
Convert the label matrix into RGB image, using default settings.

```
RGB = label2rgb(L);  
figure, imshow(RGB)
```




Convert label matrix into RGB image, specifying optional parameters.

```
RGB2 = label2rgb(L, 'spring', 'c', 'shuffle');  
figure, imshow(RGB2)
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, for best results when using the standard syntax `RGB = label2rgb(L, map, zerocolor, order)`:

- Submit at least two input arguments: the label matrix, `L`, and the colormap matrix, `map`.
- `map` must be an `n-by-3, double`, colormap matrix. You cannot use the name of a MATLAB colormap function or a function handle of a colormap function.
- If you set the boundary color `zerocolor` to the same color as one of the regions, `label2rgb` will not issue a warning.
- If you supply a value for `order`, it must be `'noshuffle'`.

See Also

`bwconncomp` | `bwlabel` | `colormap` | `ismember` | `labelmatrix` | `watershed`

Introduced before R2006a

labelmatrix

Create label matrix from `bwconncomp` structure

Syntax

```
L = labelmatrix(CC)
```

Description

`L = labelmatrix(CC)` creates a label matrix from the connected components structure `CC` returned by `bwconncomp`. The size of `L` is `CC.ImageSize`. The elements of `L` are integer values greater than or equal to 0. The pixels labeled 0 are the background. The pixels labeled 1 make up one object; the pixels labeled 2 make up a second object; and so on. The class of `L` depends on `CC.NumObjects`, as shown in the following table.

Class	Range
'uint8'	$CC.NumObjects \leq 255$
'uint16'	$256 \leq CC.NumObjects \leq 65535$
'uint32'	$65536 \leq CC.NumObjects \leq 2^{32} - 1$
'double'	$CC.NumObjects \geq 2^{32}$

`labelmatrix` is more memory efficient than `bwlabel` and `bwlabeln` because it returns its label matrix in the smallest numeric class necessary for the number of objects.

Class Support

`CC` is a structure returned by `bwconncomp`. The label matrix `L` is `uint8`, `uint16`, `uint32`, or `double`.

Examples

Calculate connected components and display results

Read binary image into the workspace.

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

Calculate the connected components, using `bwconncomp`.

```
CC = bwconncomp(BW);
```

Create a label matrix, using `labelmatrix`.

```
L = labelmatrix(CC);
```

For comparison, create a second label matrix, using `bwlabel`.

```
L2 = bwlabel(BW);
```

View both label matrices in the workspace. Note that `labelmatrix` is more memory efficient than `bwlabel`, using the smallest numeric class necessary for the number of objects.

```
whos L L2
```

Name	Size	Bytes	Class	Attributes
L	256x256	65536	uint8	
L2	256x256	524288	double	

Display the label matrix as an RGB image, using `label2rgb`.

```
figure  
imshow(label2rgb(L));
```

The term watershed
refers to a ridge that ...

... divides areas
drained by different
river systems.

See Also

`bwconncomp` | `bwlabel` | `bwlabeln` | `label2rgb` | `regionprops`

Introduced in R2009a

lazysnapping

Segment image into foreground and background using graph-based segmentation

Syntax

```
BW = lazysnapping(A,L,foremask,backmask)
```

```
BW = lazysnapping(A,L,foreind,backind)
```

```
BW = lazysnapping(V, ___)
```

```
BW = lazysnapping( ___,Name,Value)
```

Description

`BW = lazysnapping(A,L,foremask,backmask)` segments the image `A` into foreground and background regions using lazy snapping. The label matrix `L` specifies the subregions of the image. `foremask` and `backmask` are masks designating pixels in the image as foreground and background, respectively.

`BW = lazysnapping(A,L,foreind,backind)` segments the image `A` into foreground and background regions. `foreind` and `backind` specify the linear indices of the pixels in the image marked as foreground and background, respectively

`BW = lazysnapping(V, ___)` segments the volume `V` into foreground and background regions.

`BW = lazysnapping(___,Name,Value)` segments the image or volume using name-value pairs to control aspects of the segmentation.

Examples

Segment Image into Foreground and Background

Read an image into the workspace.

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

Mark locations on image as foreground.

```
figure;  
imshow(RGB)  
h1 = impoly(gca,[34,298;114,140;195,135;...  
    259,200;392,205;467,283;483,104], 'Closed',false);
```



Convert the locations into linear indices.

```
foresub = getPosition(h1);  
foregroundInd = sub2ind(size(RGB),foresub(:,2),foresub(:,1));
```

Mark locations on image as background.

```
figure;  
imshow(RGB)
```



```
h2 = impoly(gca,[130,52;170,32], 'Closed', false);
```



Convert the locations into linear indices.

```
backsub = getPosition(h2);  
backgroundInd = sub2ind(size(RGB),backsub(:,2),backsub(:,1));
```

Generate label matrix.

```
L = superpixels(RGB,500);
```

Perform lazy snapping.

```
BW = lazysnapping(RGB,L,foregroundInd,backgroundInd);
```

Create masked image.

```
maskedImage = RGB;  
maskedImage(repmat(~BW,[1 1 3])) = 0;  
figure;  
imshow(maskedImage)
```



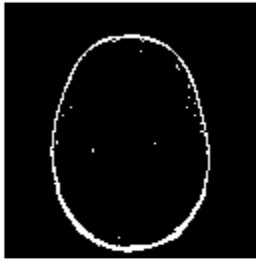
Segment Volume in Foreground and Background

Load 3-D volumetric image into the workspace.

```
D = load('mri.mat');  
V = squeeze(D,D);
```

Create a 2-D mask identifying initial foreground and background seed points.

```
seedLevel = 10;  
fseed = V(:,:,seedLevel) > 75;  
bseed = V(:,:,seedLevel) == 0;  
figure;  
imshow(fseed)
```



```
figure;  
imshow(bseed)
```



Place seed points into empty 3-D mask.

```
fmask = zeros(size(V));  
bmask = fmask;  
fmask(:,:,seedLevel) = fseed;  
bmask(:,:,seedLevel) = bseed;
```

Generate a 3-D label matrix.

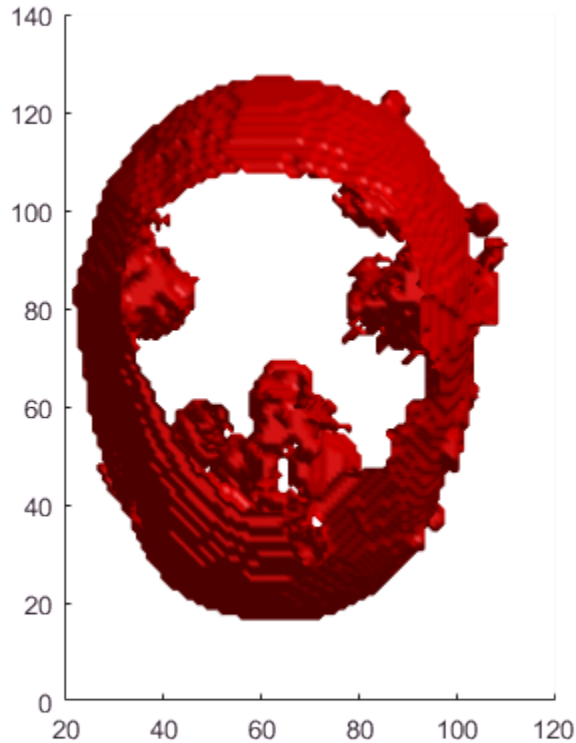
```
L = superpixels3(V,500);
```

Segment the image into foreground and background using Lazy Snapping.

```
bw = lazysnapping(V,L,fmask,bmask);
```

Display the 3-D segmented image.

```
figure;  
p = patch(isosurface(double(bw)));  
p.FaceColor = 'red';  
p.EdgeColor = 'none';  
daspect([1 1 27/128]);  
camlight; lighting phong
```



- “Segment Image Using Graph Cut”

Input Arguments

A — Input image

real, finite, nonsparse numeric array

Input image, specified as a real, finite, nonsparse numeric array. For double and single images, `lazysnapping` assumes the range of the image to be $[0 \ 1]$. For `uint16`, `int16`, and `uint8` images, `lazysnapping` assumes the range to be the full range for the given data type. If the values do not match the expected range based on the

data type, scale the image to the expected range or adjust `EdgeWeightScaleFactor` to improve results.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

V — Input volume

real, finite, nonsparse numeric array

Input volume, specified as a real, finite, nonsparse numeric array.

Example:

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

L — Label matrix of the input image or volume

numeric array

Label matrix of the input image or volume, specified as numeric array. For 2-D grayscale images and 3-D volumetric grayscale images, the size of `L` must match the size of the input image `A`. For color images and multichannel images, `L` must be a 2-D array where the first two dimensions match the first two dimensions of the input image `A`.

Do not mark a given subregion of the label matrix as belonging to both the foreground mask and the background mask. If a region of the label matrix contains pixels belonging to both the foreground mask and background mask, `lazysnapping` segments the region as background.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

foremask — Mask image that defines the foreground

logical array

Mask image that defines the foreground, specified as a logical array. For 2-D grayscale images and 3-D volumetric grayscale images, the size of `foremask` must match the size of the input image `A`. For color images and multichannel images, `foremask` must be a 2-D array where the first two dimensions match the first two dimensions of the input image `A`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

backmask — Mask image that defines the background

logical array

Mask image that defines the background, specified as a logical array. For 2-D grayscale images and 3-D volumetric grayscale images, the size of `backmask` must match the size of the input image `A`. For color images and multichannel images, `backmask` must be a 2-D array where the first two dimensions match the first two dimensions of the input image `A`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

foreind — Linear index of foreground pixels in the label matrix

numeric vector

Linear index of pixels in the label matrix, specified as a numeric vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

backind — Linear index of background pixels in the label matrix

numeric vector

Linear index of pixels that define the background, specified as a numeric vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

'Connectivity' — Connectivity of connected components

8 for 2-D images and 26 for 3-D images (default) | 4 | 6 | 18

Connectivity of connected components, specified as 4 or 8, for 2-D images, and 6, 18, or 26 for 3-D images (volumes).

Example:

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'EdgeWeightScaleFactor' — Scale factor for edge weights between the subregions of the label matrix

500 (default) | positive scalar

Scale factor for edge weights between the subregions of the label matrix, specified as a positive scalar. Typical values range from [10 1000]. Increasing this value increases the likelihood that `lazysnapping` labels neighboring subregions together as either foreground or background.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

BW — Output image

numeric array

Output image, returned as a numeric array the same size as the label matrix, L.

Algorithms

The Lazy Snapping algorithm developed by Li et al. clusters foreground and background values using the K-means method. This implementation of the Lazy Snapping algorithm does not cluster similar foreground or background pixels. To improve performance, reduce the number of pixels with similar values that are identified as foreground or background.

References

- [1] Y. Li, S. Jian, C. Tang, H. Shum, *Lazy Snapping* In Proceedings from the 31st International Conference on Computer Graphics and Interactive Techniques, 2004.

See Also

See Also

Image Segmenter | `labelmatrix` | `superpixels` | `superpixels3` | `watershed`

Topics

“Segment Image Using Graph Cut”

Introduced in R2017a

localcontrast

Edge-aware local contrast manipulation of images

Syntax

```
B = localcontrast(A)
B = localcontrast(A,edgeThreshold,amount)
```

Description

`B = localcontrast(A)` enhances the local contrast of the grayscale or RGB image `A`.

`B = localcontrast(A,edgeThreshold,amount)` enhances or flattens the local contrast of `A` by increasing or smoothing details while leaving strong edges unchanged. `edgeThreshold` defines the minimum intensity amplitude of strong edges to leave intact. `amount` is the amount of enhancement or smoothing desired.

Examples

Increase or Reduce Local Contrast of Image

Import an RGB image.

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

Increase the local contrast of the input image.

```
edgeThreshold = 0.4;
amount = 0.5;
B = localcontrast(A, edgeThreshold, amount);
```

Display the results compared to the original image

```
imshowpair(A, B, 'montage')
```



Reduce the local contrast of the input image.

```
amount = -0.5;  
B = localcontrast(A, edgeThreshold, amount);
```

Display the new results again, compared to the original image.

```
imshowpair(A, B, 'montage')
```



Input Arguments

A — Grayscale or RGB image to be filtered

real, non-sparse, m -by- n or m -by- n -by-3 matrix

Grayscale or RGB image to be filtered, specified as a real, non-sparse, m -by- n or m -by- n -by-3 matrix.

Data Types: `single` | `int8` | `int16` | `uint8` | `uint16`

`edgeThreshold` — Amplitude of strong edges to leave intact

0.3 (default) | numeric scalar in the range $[0, 1]$

Amplitude of strong edges to leave intact, specified as a numeric scalar in the range $[0, 1]$.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`amount` — Amount of enhancement or smoothing desired

0.25 (default) | numeric scalar in the range $[-1, 1]$

Amount of enhancement or smoothing desired, specified as a numeric scalar in the range $[-1, 1]$. Negative values specify edge-aware smoothing. Positive values specify edge-aware enhancement.

Value	Description
0	Leave input image unchanged.
1	Strongly enhance the local contrast of the input image
-1	Strongly smooth the details of the input image

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

B — Filtered image

numeric array

Filtered image, returned as a numeric array the same size and class as the input image.

See Also

See Also

`imadjust` | `imcontrast` | `imsharpen` | `locallapfilt`

Introduced in R2016b

locallapfilt

Fast Local Laplacian Filtering of images

Syntax

```
B = locallapfilt(A,sigma,alpha)
B = locallapfilt(A,sigma,alpha,beta)
B = locallapfilt( ____,Name,Value,...)
```

Description

`B = locallapfilt(A,sigma,alpha)` filters the grayscale or RGB image `A` with an edge-aware, fast local Laplacian filter. `sigma` characterizes the amplitude of edges in `A`. `alpha` controls smoothing of details.

`B = locallapfilt(A,sigma,alpha,beta)` filters the image using `beta` to control the dynamic range of `A`.

`B = locallapfilt(____,Name,Value,...)` filters the image using name-value pairs to control advanced aspects of the filter. Parameter names can be abbreviated.

Examples

Increase Local Contrast of RGB Image Using Local Laplacian Filtering

Import an RGB image

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

Set parameters of the filter to increase details smaller than 0.4.

```
sigma = 0.4;
alpha = 0.5;
```

Use fast local Laplacian filtering

```
B = locallapfilt(A, sigma, alpha);
```

Display the original and filtered images side-by-side.

```
imshowpair(A, B, 'montage')
```



Increase Local Contrast, Balancing Speed and Quality

Local Laplacian filtering is a computationally intensive algorithm. To speed up processing, `locallapfilt` approximates the algorithm by discretizing the intensity range into a number of samples defined by the `'NumIntensityLevels'` parameter. This parameter can be used to balance speed and quality.

Import an RGB image and display it.

```
A = imread('peppers.png');  
figure  
imshow(A)  
title('Original Image')
```

Original Image



Use a `sigma` value to process the details and an `alpha` value to increase the contrast, effectively enhancing the local contrast of the image.

```
sigma = 0.2;  
alpha = 0.3;
```

Using fewer samples increases the execution speed, but can produce visible artifacts, especially in areas of flat contrast. Time the function using only 20 intensity levels.

```
t_speed = timeit(@() locallapfilt(A, sigma, alpha, 'NumIntensityLevels', 20))  
t_speed = 0.2234
```

Now, process the image and display it.


```
B_speed = locallapfilt(A, sigma, alpha, 'NumIntensityLevels', 20);  
figure  
imshow(B_speed)  
title(['Enhanced with 20 intensity levels in ' num2str(t_speed) ' sec'])
```

Enhanced with 20 intensity levels in 0.22336 sec



A larger number of samples yields better looking results at the expense of more processing time. Time the function using 100 intensity levels.

```
t_quality = timeit(@() locallapfilt(A, sigma, alpha, 'NumIntensityLevels', 100))  
t_quality = 1.0359
```

Process the image with 100 intensity levels and display it:

```
B_quality = locallapfilt(A, sigma, alpha, 'NumIntensityLevels', 100);
```

```
figure
imshow(B_quality)
title(['Enhancement with 100 intensity levels in ' num2str(t_quality) ' sec'])
```

Enhancement with 100 intensity levels in 1.0359 sec



Try varying the number of intensity levels on your own images. Try also flattening the contrast (with $\alpha > 1$). You will see that the optimal number of intensity levels is different for every image and varies with α . By default, `localappfilt` uses a heuristic to balance speed and quality, but it cannot predict the best value for every image.

Boost Local Color Contrast Using 'ColorMode'

Import a color image, reduce its size, and display it.

```
A = imread('car2.jpg');  
A = imresize(A, 0.25);  
figure  
imshow(A)  
title('Original Image')
```



Set the parameters of the filter to dramatically increase details smaller than 0.3 (out of a normalized range of 0 to 1).

```
sigma = 0.3;  
alpha = 0.1;
```

Let's compare the two different modes of color filtering. Process the image by filtering its intensity and by filtering each color channel separately:

```
B_luminance = locallapfilt(A, sigma, alpha);
```

```
B_separate = locallapfilt(A, sigma, alpha, 'ColorMode', 'separate');
```

Display the filtered images.

```
figure  
imshow(B_luminance)  
title('Enhanced by boosting the local luminance contrast')
```



```
figure  
imshow(B_separate)  
title('Enhanced by boosting the local color contrast')
```

Enhanced by boosting the local color contrast



An equal amount of contrast enhancement has been applied to each image, but colors are more saturated when setting 'ColorMode' to 'separate'.

Perform Edge-Aware Noise Reduction

Import an image. Convert the image to floating point so that we can add artificial noise more easily.

```
A = imread('pout.tif');  
A = im2single(A);
```

Add Gaussian noise with zero mean and 0.001 variance.

```
A_noisy = imnoise(A, 'gaussian', 0, 0.001);
```

```
psnr_noisy = psnr(A_noisy, A);  
fprintf('The peak signal-to-noise ratio of the noisy image is %0.4f\n', psnr_noisy);
```

The peak signal-to-noise ratio of the noisy image is 30.0234

Set the amplitude of the details to smooth, then set the amount of smoothing to apply.

```
sigma = 0.1;  
alpha = 4.0;
```

Apply the edge-aware filter.

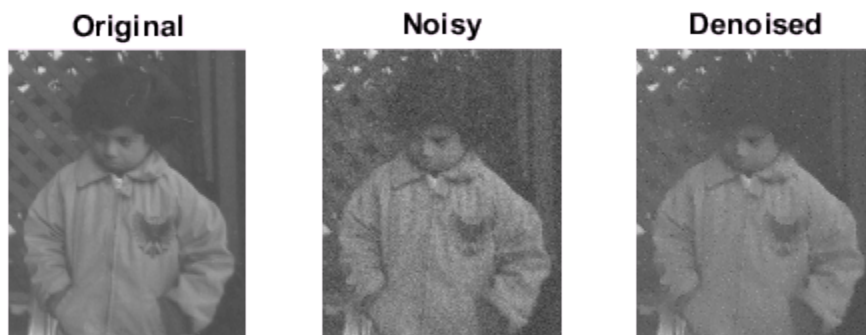
```
B = locallapfilt(A_noisy, sigma, alpha);  
psnr_denoised = psnr(B, A);  
fprintf('The peak signal-to-noise ratio of the denoised image is %0.4f\n', psnr_denoised);
```

The peak signal-to-noise ratio of the denoised image is 32.3065

Note an improvement in the PSNR of the image.

Display all three images side by side. Observe that details are smoothed and sharp intensity variations along edges are unchanged.

```
figure  
subplot(1,3,1), imshow(A), title('Original')  
subplot(1,3,2), imshow(A_noisy), title('Noisy')  
subplot(1,3,3), imshow(B), title('Denoised')
```



Smooth Image Details Without Affecting Edge Sharpness

Import the image, resize it and display it

```
A = imread('car1.jpg');  
A = imresize(A, 0.25);  
figure  
imshow(A)  
title('Original Image')
```

Original Image



The car is dirty and covered in markings. Let's try to erase the dust and markings on the body. Set the amplitude of the details to smooth, and set a large amount of smoothing to apply.

```
sigma = 0.2;  
alpha = 5.0;
```

When smoothing ($\alpha > 1$), the filter produces high quality results with a small number of intensity levels. Set a small number of intensity levels to process the image faster.

```
numLevels = 16;
```

Apply the filter.

```
B = locallapfilt(A, sigma, alpha, 'NumIntensityLevels', numLevels);
```


Display the "clean" car.

```
figure
imshow(B)
title('After smoothing details')
```

After smoothing details



Input Arguments

A — Grayscale or RGB image to be filtered

real, non-sparse, m -by- n or m -by- n -by-3 matrix

Grayscale or RGB image to be filtered, specified as a real, non-sparse, m -by- n or m -by- n -by-3 matrix.

Data Types: `single` | `int8` | `int16` | `uint8` | `uint16`

sigma — Amplitude of edges

non-negative scalar

Amplitude of edges, specified as a non-negative scalar. `sigma` should be in the range [0, 1] for integer images and for single images defined over the range [0, 1]. For single images defined over a different range [a, b], `sigma` should also be in the range [a, b].

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

alpha — Smoothing of details

positive scalar

Smoothing of details, specified as a positive scalar. Typical values of `alpha` are in the range [0.01, 10].

Value	Description
alpha less than 1	Increases the details of the input image, effectively enhancing the local contrast of the image without affecting edges or introducing halos.
alpha greater than 1	Smooths details in the input image while preserving crisp edges
alpha equal to 1	The details of the input image are left unchanged.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

beta — Dynamic range

1 (default) | non-negative scalar

Dynamic range, specified as a non-negative scalar. Typical values of `beta` are in the range [0, 5]. `beta` affects the dynamic range of `A`.

Value	Description
beta less than 1	Reduces the amplitude of edges in the image, effectively compressing the dynamic range without affecting details.

Value	Description
beta greater than 1	Expands the dynamic range of the image.
beta equal to 1	Dynamic range of the image is left unchanged. This is the default value.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:

'ColorMode' — Method used to filter RGB images

'luminance' (default) | 'separate'

Method used to filter RGB images, specified as one of the following values. This parameter has no effect on grayscale images.

Value	Description
'luminance'	<code>locallapfilt</code> converts the input RGB image to grayscale before filtering and reintroduces color after filtering, which changes the contrast of the input image without affecting colors.
'separate'	<code>locallapfilt</code> filters each color channel independently.

Data Types: char

'NumIntensityLevels' — Number of intensity samples in the dynamic range of the input image

'auto' (default) | positive integer

Number of intensity samples in the dynamic range of the input image, specified as a character vector or positive integer. A higher number of samples gives results closer to exact local Laplacian filtering. A lower number increases the execution speed. Typical

values are in the range [10, 100]. If set to 'auto', `locallapfilt` chooses the number of intensity levels automatically to balance quality and speed based on other parameters of the filter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char`

Output Arguments

B — Filtered image

numeric array

Filtered image, returned as a numeric array the same size and class as the input image.

References

- [1] Paris, Sylvain, Samuel W. Hasinoff, and Jan Kautz. *Local Laplacian filters: edge-aware image processing with a Laplacian pyramid*, ACM Trans. Graph. 30.4 (2011): 68.
- [2] Aubry, Mathieu, et al. *Fast local laplacian filters: Theory and applications*. ACM Transactions on Graphics (TOG) 33.5 (2014): 167.

See Also

See Also

`localcontrast` | `localtonemap`

Introduced in R2016b

localtonemap

Render HDR image for viewing while enhancing local contrast

Syntax

```
rgb = localtonemap(hdr)  
rgb = localtonemap(hdr,Name,Value,...)
```

Description

`rgb = localtonemap(hdr)` converts the high dynamic range image `hdr` to a lower dynamic range image, `rgb`, suitable for display. `localtonemap` uses a process called tone mapping while preserving its local contrast.

`rgb = localtonemap(hdr,Name,Value,...)` performs tone mapping where parameters control various aspects of the operation. Parameter names can be abbreviated.

Examples

Compress Dynamic Range of HDR Image for Viewing

Load a high dynamic range image.

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

Apply local tone mapping with a small amount of dynamic range compression.

```
RGB = localtonemap(HDR, 'RangeCompression', 0.1);
```

Display the resulting tone-mapped image.

```
imshow(RGB)
```



Repeat the operation but, this time, accentuate the details in the image.

```
RGB = localtonemap(HDR, ...  
                  'RangeCompression', 0.1, ...  
                  'EnhanceContrast', 0.5);
```

Display the resulting tone-mapped image with increased details.

```
imshow(RGB)
```



Input Arguments

hdr — High dynamic range image

real, nonsparse, m -by- n or m -by- n -by-3 matrix

High dynamic range image, specified as a real, nonsparse, m -by- n or m -by- n -by-3 matrix of class `single`.

Data Types: `single`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

'RangeCompression' — Amount of compression applied to the dynamic range of the HDR image

1 (default) | numeric scalar in the range [0,1]

Amount of compression applied to the dynamic range of the HDR image, specified as a numeric scalar in the range [0,1].

Value	Description
0	Minimum compression, which consists in only remapping the middle 99% intensities to a dynamic range of 100:1 followed by gamma correction with an exponent of 1/2.2.
1	Maximum compression using local Laplacian filtering.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'EnhanceContrast' — Amount of local contrast enhancement applied

0 (default) | numeric scalar in the range [0,1]

Amount of local contrast enhancement applied, specified as a numeric scalar. Value must be in the range [0,1].

Value	Description
0	No change to local contrast
1	Maximum local contrast enhancement

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

rgb — Tone-mapped image

RGB image

Tone-mapped image, returned as an RGB image.

Algorithms

`localtonemap` uses local Laplacian filtering in logarithmic space to compress the dynamic range of HDR while preserving or enhancing its local contrast. The 99% middle intensities of the compressed image are then remapped to a fixed 100:1 dynamic range to give the output image a consistent look. `localtonemap` then applies gamma correction to produce the final image for display.

See Also

See Also

`locallapfilt` | `tonemap`

Introduced in R2016b

images.geotrans.LocalWeightedMeanTransformation2D class

Package: images.geotrans

2-D local weighted mean geometric transformation

Description

A `LocalWeightedMeanTransformation2D` object encapsulates a 2-D local weighted mean geometric transformation.

You can create a `LocalWeightedMeanTransformation2D` object using the following methods:

- `fitgeotrans` — Returns a `LocalWeightedMeanTransformation2D` object that maps control point pairs using a local weighted mean transformation
- The `LocalWeightedMeanTransformation2D` class constructor

Construction

`tform = images.geotrans.LocalWeightedMeanTransformation2D(movingPoints, fixedPoints, n)` constructs a `LocalWeightedMeanTransformation2D` object given m -by-2 matrices `movingPoints` and `fixedPoints`, which define matched control points in the moving and fixed images, respectively. The local weighted mean transformation creates a mapping, by inferring a polynomial at each control point using neighboring control points. The mapping at any location depends on a weighted average of these polynomials. The n closest points are used to infer a second degree polynomial transformation for each control point pair. n can be as small as 6, but making it small risks generating ill-conditioned polynomials.

Input Arguments

movingPoints — x - and y -coordinates of control points in the moving image
 m -by-2 matrix

x - and y -coordinates of control points in the moving image, specified as an m -by-2 matrix. The number of control points m must be greater than or equal to n .

Data Types: `double` | `single`

fixedPoints — x - and y -coordinates of control points in the fixed image

m -by-2 matrix

x - and y -coordinates of control points in the fixed image, specified as an m -by-2 matrix. The number of control points m must be greater than or equal to n .

Data Types: `double` | `single`

n — Number of points to use in local weighted mean calculation

numeric value

Number of points to use in local weighted mean calculation, specified as a numeric value. n can be as small as 6, but making n small risks generating ill-conditioned polynomials

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

Properties

Dimensionality — Describes the dimensionality of the geometric transformation

2

Describes the dimensionality of the geometric transformation for both input and output points.

Methods

`transformPointsInverse`
`outputLimits`

Apply inverse geometric transformation
Find output limits of geometric transformation

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB documentation.

Examples

Fit set of fixed and moving control points using second degree polynomial

Fit a local weighted mean transformation to a set of fixed and moving control points that are actually related by a global second degree polynomial transformation across the entire plane.

Set up variables.

```
x = [10, 12, 17, 14, 7, 10];  
y = [8, 2, 6, 10, 20, 4];
```

```
a = [1 2 3 4 5 6];  
b = [2.3 3 4 5 6 7.5];
```

```
u = a(1) + a(2).*x + a(3).*y + a(4) .*x.*y + a(5).*x.^2 + a(6).*y.^2;  
v = b(1) + b(2).*x + b(3).*y + b(4) .*x.*y + b(5).*x.^2 + b(6).*y.^2;
```

```
movingPoints = [u',v'];  
fixedPoints = [x',y'];
```

Fit local weighted mean transformation to points.

```
tformLocalWeightedMean = images.geotrans.LocalWeightedMeanTransformation2D(movingPoints,
```

Verify the fit of our `LocalWeightedMeanTransformation2D` object at the control points.

```
movingPointsComputed = transformPointsInverse(tformLocalWeightedMean, fixedPoints);
```

```
errorInFit = hypot(movingPointsComputed(:,1)-movingPoints(:,1), ...  
                  movingPointsComputed(:,2)-movingPoints(:,2))
```

See Also

See Also

Functions

fitgeotrans | imwarp

Classes

affine2d | projective2d | images.geotrans.PiecewiseLinearTransformation2d |
images.geotrans.PolynomialTransformation2d

images.geotrans.LocalWeightedMeanTransformation2d.trans

Apply inverse geometric transformation

Syntax

```
[u,v] = transformPointsInverse(tform,x,y)  
U = transformPointsInverse(tform,X)
```

Description

`[u,v] = transformPointsInverse(tform,x,y)` applies the inverse transformation of `tform` to the input 2-D point arrays `x` and `y` and outputs the point arrays `u` and `v`. The input point arrays `x` and `y` must be of the same size.

`U = transformPointsInverse(tform,X)` applies the inverse transformation of `tform` to the input n -by-2 point matrix `X` and outputs the n -by-2 point matrix `U`. `transformPointsFoward` maps the point `X(k, :)` to the point `U(k, :)`.

Input Arguments

tform

Geometric transformation, specified as a `images.geotrans.LocalWeightedMeanTransformation2D` object.

x

Coordinates in X dimension of points to be transformed, specified as a array.

y

Coordinates in Y dimension of points to be transformed, specified as a array.

X

X and Y coordinates of points to be transformed, specified as an n -by-2 matrix

Output Arguments

u

Transformed coordinates in X dimension, returned as an array.

v

Transformed coordinates in Y dimension, returned as an array.

U

Transformed X and Y coordinates, returned as an n -by-2 matrix

See Also

images.geotrans.LocalWeightedMeanTransformation2d.outp

Find output limits of geometric transformation

Syntax

```
[xLimitsOut,yLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn)
[xLimitsOut,yLimitsOut,zLimitsOut] =
outputLimits(tform,xLimitsIn,yLimitsIn,zLimitsIn)
```

Description

`[xLimitsOut,yLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn)` estimates the output spatial limits corresponding to a given 2D geometric transformation and a set of input spatial limits.

`[xLimitsOut,yLimitsOut,zLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn,zLimitsIn)` estimates the output spatial limits corresponding to a given 3D geometric transformation and a set of input spatial limits.

Input Arguments

tform

Geometric transformation, specified as an `images.geotrans.LocalWeightedMeanTransformation2D` object.

xLimitsIn

Limits along the *X* axes, specified as a two-element vector, such as `[]`.

yLimitsIn

Limits along the *Y* axes, specified as a two-element vector, such as `[]`.

zLimitsIn

Limits along the Z axes, specified as a two-element vector, such as [].

Output Arguments

xLimitsOut

Actual limits along the X dimension, returned as an array.

yLimitsOut

Actual limits along the Y dimension, returned as an array.

zLimitsOut

Actual limits along the Z dimension, returned as an array.

See Also

makecform

Create color transformation structure

Syntax

```
C = makecform(type)
C = makecform(type, 'WhitePoint', WP)
C = makecform(type, 'AdaptedWhitePoint', WP)
C = makecform('srgb2cmyk', 'RenderingIntent', intent)
C = makecform('cmyk2srgb', 'RenderingIntent', intent)
C = makecform('adapt', 'WhiteStart', WPS, 'WhiteEnd', WPE,
'AdaptModel', modelName)
C = makecform('icc', src_profile, dest_profile)
C = makecform('icc', src_profile, dest_profile,
'SourceRenderingIntent', src_intent, 'DestRenderingIntent',
dest_intent)
C = makecform('clut', profile, LUTtype)
C = makecform('mattrc', MatTrc, 'Direction', direction)
C = makecform('mattrc', profile, 'Direction', direction)
C = makecform('mattrc', profile, 'Direction', direction,
'RenderingIntent', intent)
C = makecform('graytrc', profile, 'Direction', direction)
C = makecform('graytrc', profile, 'Direction', direction,
'RenderingIntent', intent)
C = makecform('named', profile, space)
```

Description

`C = makecform(type)` creates the color transformation structure `C` that defines the color space conversion specified by *type*. To perform the transformation, pass the color transformation structure as an argument to the `applycform` function.

The *type* argument specifies one of the conversions listed in the following table. `makecform` supports conversions between members of the family of device-independent color spaces defined by the CIE, *Commission Internationale de l'Éclairage* (International Commission on Illumination). In addition, `makecform` also supports conversions to

and from the *sRGB* and *CMYK* color spaces. For a list of the abbreviations used by the Image Processing Toolbox software for each color space, see the Remarks section of this reference page.

Type	Description
'cmyk2srgb'	Convert from the <i>CMYK</i> color space to the <i>sRGB</i> color space.
'lab2lch'	Convert from the $L^*a^*b^*$ to the L^*ch color space.
'lab2srgb'	Use lab2rgb instead.
'lab2xyz'	Use lab2xyz instead.
'lch2lab'	Convert from the L^*ch to the $L^*a^*b^*$ color space.
'srgb2cmyk'	Convert from the <i>sRGB</i> to the <i>CMYK</i> color space.
'srgb2lab'	Use rgb2lab instead.
'srgb2xyz'	Use rgb2xyz instead.
'upvp12xyz'	Convert from the u^*v^*L to the <i>XYZ</i> color space.
'uvl2xyz'	Convert from the uvL to the <i>XYZ</i> color space.
'xyz12xyz'	Convert from the xyY to the <i>XYZ</i> color space.
'xyz2lab'	Use xyz2lab instead.
'xyz2srgb'	Use xyz2rgb instead.
'xyz2upvp1'	Convert from the <i>XYZ</i> to the u^*v^*L color space.
'xyz2uvl'	Convert from the <i>XYZ</i> to the uvL color space.
'xyz2xy1'	Convert from the <i>XYZ</i> to the xyY color space.

$C = \text{makeform}(\text{type}, \text{'WhitePoint'}, \text{WP})$ specifies the value of the reference white point. *type* can be either 'xyz2lab' or 'lab2xyz'. *WP* is a 1-by-3 vector of *XYZ* values scaled so that $Y = 1$. The default is `whitepoint('ICC')`. Use the `whitepoint` function to create the *WP* vector.

$C = \text{makeform}(\text{type}, \text{'AdaptedWhitePoint'}, \text{WP})$ specifies the adapted white point. *type* can be either 'srgb2lab', 'lab2srgb', 'srgb2xyz', or 'xyz2srgb'. As above, *WP* is a row vector of *XYZ* values scaled so that $Y = 1$. If not specified, the default adapted white point is `whitepoint('ICC')`. To get answers consistent with some published *sRGB* equations, specify `whitepoint('D65')` for the adapted white point.

$C = \text{makeform}(\text{'srgb2cmyk'}, \text{'RenderingIntent'}, \text{intent})$ and

`C = makecform('cmyk2srgb', 'RenderingIntent', intent)` specify the rendering intent for transforms of type `srgb2cmyk` and `cmyk2srgb`. These transforms convert data between *sRGB* IEC61966-2.1 and "Specifications for Web Offset Publications" (SWOP) *CMYK*. *intent* must be one of these values: 'AbsoluteColorimetric', 'Perceptual', 'RelativeColorimetric', or 'Saturation'. For more information, see the table Rendering Intent.

`C = makecform('adapt', 'WhiteStart', WPS, 'WhiteEnd', WPE, 'AdaptModel', modelName)` creates a linear chromatic-adaptation transform. WPS and WPE are row vectors of XYZ values, scaled so that $Y = 1$, specifying the starting and ending white points. *modelName* is either 'vonKries' or 'Bradford' and specifies the type of chromatic-adaptation model to be employed. If 'AdaptModel' is not specified, it defaults to 'Bradford'.

`C = makecform('icc', src_profile, dest_profile)` creates a color transform based on two ICC profiles. `src_profile` and `dest_profile` are ICC profile structures returned by `iccread`.

`C = makecform('icc', src_profile, dest_profile, 'SourceRenderingIntent', src_intent, 'DestRenderingIntent', dest_intent)` creates a color transform based on two ICC color profiles, `src_profile` and `dest_profile`, specifying rendering intent arguments for the source, *src_intent*, and the destination, *dest_intent*, profiles.

Rendering intents specify the style of reproduction that should be used when these profiles are combined. For most devices, the range of reproducible colors is much smaller than the range of colors represented by the PCS. Rendering intents define gamut mapping techniques. Possible values for these rendering intents are listed below. Each rendering intent has distinct aesthetic and color-accuracy trade-offs.

Rendering Intent

Value	Description
'AbsoluteColorimetric'	Maps all out-of-gamut colors to the nearest gamut surface while maintaining the relationship of all in-gamut colors. This absolute rendering contains color data that is relative to a perfectly reflecting diffuser.
'Perceptual' (default)	Employs vendor-specific gamut mapping techniques for optimizing the range of producible colors of a given device. The objective is to provide the most aesthetically pleasing result even though the relationship of the in-gamut colors

Value	Description
	might not be maintained. This media-relative rendering contains color data that is relative to the device's white point.
'RelativeColorimetric'	Maps all out-of-gamut colors to the nearest gamut surface while maintaining the relationship of all in-gamut colors. This media-relative rendering contains color data that is relative to the device's white point.
'Saturation'	Employs vendor-specific gamut mapping techniques for maximizing the saturation of device colors. This rendering is generally used for simple business graphics such as bar graphs and pie charts. This media-relative rendering contains color data that is relative to the device's white point.

`C = makecform('clut', profile, LUTtype)` creates the color transformation structure `C` based on a color lookup table (CLUT) contained in an ICC color profile. `profile` is an ICC profile structure returned by `iccread`. `LUTtype` specifies which `clut` in the `profile` structure is to be used. Each `LUTtype` listed in the table below contains the components of an 8-bit or 16-bit LUTtag that performs a transformation between device colors and PCS colors using a particular rendering. For more information about 'clut' transformations, see Section 6.5.7 of the International Color Consortium specification ICC.1:2001-04 (Version 2) or Section 6.5.9 of ICC.1:2001-12 (Version 4), available at www.color.org.

LUT Type	Description
'AToB0' (default)	Device to PCS: perceptual rendering intent
'AToB1'	Device to PCS: media-relative colorimetric rendering intent
'AToB2'	Device to PCS: saturation rendering intent
'AToB3'	Device to PCS: ICC-absolute rendering intent
'BToA0'	PCS to device: perceptual rendering intent
'BToA1'	PCS to device: media-relative colorimetric rendering intent
'BToA2'	PCS to device: saturation rendering intent
'BToA3'	PCS to device: ICC-absolute rendering intent
'Gamut'	Determines which PCS colors are out of gamut for a given device

LUT Type	Description
'Preview0'	PCS colors to the PCS colors available for soft proofing using the perceptual rendering
'Preview1'	PCS colors available for soft proofing using the media-relative colorimetric rendering.
'Preview2'	PCS colors to the PCS colors available for soft proofing using the saturation rendering.

`C = makecform('mattrc', MatTrc, 'Direction', direction)` creates the color transformation structure `C` based on a Matrix/Tone Reproduction Curve (MatTRC) model, containing an *RGB*-to-*XYZ* matrix and *RGB* Tone Reproduction Curves. `MatTRC` is typically the 'MatTRC' field of an ICC profile structure returned by `iccread`, based on tags contained in an ICC color profile. `direction` can be either 'forward' or 'inverse' and specifies whether the `MatTRC` is to be applied in the forward (*RGB* to *XYZ*) or inverse (*XYZ* to *RGB*) direction. For more information, see section 6.3.1.2 of the International Color Consortium specification ICC.1:2001-04 or ICC.1:2001-12, available at www.color.org.

`C = makecform('mattrc', profile, 'Direction', direction)` creates a color transform based on the `MatTRC` field of the given ICC profile structure `profile`. `direction` is either 'forward' or 'inverse' and specifies whether the `MatTRC` is applied in the forward (*RGB* to *XYZ*) or inverse (*XYZ* to *RGB*) direction.

`C = makecform('mattrc', profile, 'Direction', direction, 'RenderingIntent', intent)` is similar, but adds the option of specifying the rendering intent. `intent` must be either 'RelativeColorimetric' (the default) or 'AbsoluteColorimetric'. When 'AbsoluteColorimetric' is specified, the colorimetry is referenced to a perfect diffuser, rather than to the Media White Point of the profile.

`C = makecform('graytrc', profile, 'Direction', direction)` creates the color transformation structure `C` that specifies a monochrome transform based on a single-channel Tone Reproduction Curve (GrayTRC) contained in an ICC color profile. `direction` can be either 'forward' or 'inverse' and specifies whether the `grayTRC` transform is to be applied in the forward (device to PCS) or inverse (PCS to device) direction. ("Device" here refers to the grayscale signal communicating with the monochrome device. "PCS" is the Profile Connection Space of the ICC profile and can be either *XYZ* or *L*a*b**, depending on the 'ConnectionSpace' field in `profile.Header`.)

`C = makecform('graytrc', profile, 'Direction', direction, 'RenderingIntent', intent)` is similar but adds the option of specifying the rendering intent. *intent* must be either 'RelativeColorimetric' (the default) or 'AbsoluteColorimetric'. When 'AbsoluteColorimetric' is specified, the colorimetry is referenced to a perfect diffuser, rather than to the Media White Point of the profile.

`C = makecform('named', profile, space)` creates the color transformation structure `C` that specifies the transformation from color names to color-space coordinates. *profile* must be a profile structure for a Named Color profile (with a `NamedColor2` field). *space* is either 'PCS' or 'Device'. The 'PCS' option is always available and will return $L^*a^*b^*$ or XYZ coordinates, depending on the 'ConnectionSpace' field in `profile.Header`, in 'double' format. The 'Device' option, when active, returns device coordinates, the dimension depending on the 'ColorSpace' field in `profile.Header`, also in 'double' format.

Examples

Convert RGB image to $L^*a^*b^*$, assuming input image is *sRGB*.

```
rgb = imread('peppers.png');
cform = makecform('srgb2lab');
lab = applycform(rgb,cform);
```

Convert from a non-standard RGB color profile to the device-independent XYZ profile connection space. Note that the ICC input profile must include a `MatTRC` value.

```
InputProfile = iccread('myRGB.icc');
C = makecform('mattrc',InputProfile.MatTRC, ...
    'direction', 'forward');
```

Tips

The Image Processing Toolbox software uses the following abbreviations to represent color spaces.

Abbreviation	Description
xyz	1931 CIE XYZ tristimulus values (2° observer)

Abbreviation	Description
xy1	1931 CIE xyY chromaticity values (2° observer), where x and y refer to the xy -coordinates of the associated CIE chromaticity diagram, and 1 refers to Y (luminance).
uv1	1960 CIE uvY values, where u and v refer to the uv -coordinates, and 1 refers to Y (luminance).
upvp1	1976 CIE $u'v'Y$ values, where up and vp refer to the $u'v'$ -coordinates and 1 refers to Y (luminance).
lab	1976 CIE $L^*a^*b^*$ values Note: 1 or L refers to L^* (CIE 1976 psychometric lightness) rather than luminance (Y).
lch	Polar transformation of CIE $L^*a^*b^*$ values, where c = chroma and h = hue
cmyk	Standard values used by printers
srgb	Standard computer monitor RGB values, (IEC 61966-2-1)

See Also

lab2rgb | lab2xyz | rgb2lab | rgb2xyz | xyz2lab | xyz2rgb | applycform
 | iccread | iccwrite | isicc | lab2double | lab2uint8 | lab2uint16 |
 whitepoint | xyz2double | xyz2uint16

Introduced before R2006a

makeConstrainToRectFcn

Create rectangularly bounded drag constraint function

Syntax

```
fcn = makeConstrainToRectFcn(type, xlim, ylim)
```

Description

`fcn = makeConstrainToRectFcn(type, xlim, ylim)` creates a position constraint function for draggable tools of a given type, where `type` is one of the following values: 'imellipse', 'imfreehand', 'imline', 'impoint', 'impoly', or 'imrect'. The rectangular boundaries of the position constraint function are described by the vectors `xlim` and `ylim` where `xlim = [xmin xmax]` and `ylim = [ymin ymax]`.

Examples

Constrain drag of `impoint` within axes limits.

```
figure, plot(1:10);  
h = impoint(gca,2,6);  
api = iptgetapi(h);  
fcn = makeConstrainToRectFcn('impoint',get(gca,'XLim'),...  
    get(gca,'YLim'));  
api.setPositionConstraintFcn(fcn);
```

See Also

`imdistline` | `imellipse` | `imfreehand` | `imline` | `impoint` | `impoly` | `imrect`

Introduced in R2006a

makehdr

Create high dynamic range image

Syntax

```
HDR = makehdr(files)
HDR = makehdr(files,Name,Value,...)
```

Description

`HDR = makehdr(files)` creates the single-precision, high dynamic range image HDR from the set of spatially registered, low dynamic-range images listed in the `files` cell array. These files must contain EXIF exposure metadata. `makehdr` uses the middle exposure between the brightest and darkest images as the base exposure for the high dynamic-range calculations. This value does not need to appear in any particular file. For more information about calculating this middle exposure value, see .)

`HDR = makehdr(files,Name,Value,...)` creates a high dynamic- range image, where you can use parameters and corresponding values to control various aspects of the image creation. Parameter names can be abbreviated and case does not matter.

Note: You can use only one of the `BaseFile`, `ExposureValues`, and `RelativeExposure` parameters at a time.

Examples

Make High Dynamic Range Image from Series of Low Dynamic Range Images

Make a high dynamic range (HDR) image from a series of low dynamic range images that share the same `f/stop` number and have different exposure times.

Load six low dynamic range images. Create a vector of their respective exposure times.

```
files = {'office_1.jpg', 'office_2.jpg', 'office_3.jpg', ...
        'office_4.jpg', 'office_5.jpg', 'office_6.jpg'};
expTimes = [0.0333, 0.1000, 0.3333, 0.6250, 1.3000, 4.0000];
```

Combine the images into an HDR image.

```
hdr = makehdr(files, 'RelativeExposure', expTimes ./ expTimes(1));
```

Visualize the HDR image.

```
rgb = tonemap(hdr);  
figure  
imshow(rgb)
```



Input Arguments

files — Set of spatially registered, low dynamic-range images
cell array of character vectors

Set of spatially registered, low dynamic-range images, specified as a cell array of character vectors. The input images can be color or grayscale, and can have a bit-depth of 8 or 16.

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example:

'BaseFile' — Name of the file to use as the base exposure

character vector

Name of the file to use as the base exposure, specified as a character vector.

Data Types: `char`

'ExposureValues' — Exposure value of each file in the set

numeric vector of positive values

Exposure value of each file in the set, specified as a numeric vector of positive values, with one element for each low dynamic-range image in `files`. An increase of one exposure value (EV) corresponds to a doubling of exposure, while a decrease of one EV corresponds to a halving of exposure. Any positive value is allowed. Values specified using this parameter override EXIF exposure metadata.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

'RelativeExposure' — Relative exposure value of each low dynamic range image in the set

(default) | numeric vector

Relative exposure value of each low dynamic range image in the set, specified as a numeric vector with one element for each low dynamic-range image in `files`. An image with a relative exposure (RE) of 0.5 has half as much exposure as an image with an RE of 1.0. An RE value of 3 has three times the exposure of an image with an RE of 1. Values specified using this parameter override EXIF exposure metadata.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

'MinimumLimit' — Minimum correctly exposed value

numeric scalar

Minimum correctly exposed value, specified as a numeric scalar. For each low dynamic-range image, pixels with smaller values are considered underexposed and do not contribute to the final high dynamic-range image. If the value of this parameter is omitted, it is assumed to be 2% of the maximum intensity allowed by the image data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

'MaximumLimit' — Maximum correctly exposed value

numeric scalar

Maximum correctly exposed value, specified as a numeric scalar. For each low dynamic-range image, pixels with larger values are considered overexposed and do not contribute to the final high dynamic-range image. If the value of this parameter is omitted, it is assumed to be 98% of the maximum intensity allowed by the image data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Output Arguments

HDR — High dynamic-range image

numeric array of class `single`

High dynamic-range image, returned as a numeric array of class `single`.

Algorithms

The `makehdr` function calculates the middle exposure by computing the exposure values (EV) for each image, based on the aperture and shutter speed metadata stored in the files or specified using the `'Exposurevalues'` parameter. The EV of a middle image (hypothetical or actual) is used as the base exposure. Thus, the middle exposure is an

average of the highest and lowest EV, not of actual brightness (because of the nonlinear, geometric nature of EV).

References

[1] Reinhard, et al. *High Dynamic Range Imaging* 2006. Ch. 4.

See Also

See Also

hdrread | tonemap

Introduced in R2008a

makelut

Create lookup table for use with `bwlookup`

Syntax

```
lut = makelut(fun,n)
```

Description

`lut = makelut(fun,n)` returns a lookup table for use with `bwlookup`. `fun` is a function that accepts an n -by- n matrix of 1's and 0's as input and return a scalar. `n` can be either 2 or 3. `makelut` creates `lut` by passing all possible 2-by-2 or 3-by-3 neighborhoods to `fun`, one at a time, and constructing either a 16-element vector (for 2-by-2 neighborhoods) or a 512-element vector (for 3-by-3 neighborhoods). The vector consists of the output from `fun` for each possible neighborhood. `fun` must be a function handle. Parameterizing Functions, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`.

Class Support

`lut` is returned as a vector of class `double`.

Examples

Construct a lookup table for 2-by-2 neighborhoods. In this example, the function passed to `makelut` returns `TRUE` if the number of 1's in the neighborhood is 2 or greater, and returns `FALSE` otherwise.

```
f = @(x) (sum(x(:)) >= 2);  
lut = makelut(f,2)  
lut =  
    0  
    0  
    0
```

1
0
1
1
1
0
1
1
1
1
1
1
1
1

See Also

`bwlookup`

Topics

“Anonymous Functions” (MATLAB)

“Parameterizing Functions” (MATLAB)

Introduced before R2006a

makesampler

Create resampling structure

Syntax

```
R = makesampler(interpolant,padmethod)
R = makesampler(Name,Value,...)
```

Description

`R = makesampler(interpolant,padmethod)` creates a separable resampler structure for use with `tformarray`. The `interpolant` argument specifies the interpolating kernel that the separable resampler uses. The `padmethod` argument controls how the resampler interpolates or assigns values to output elements that map close to or outside the edge of the input array.

`R = makesampler(Name,Value,...)` create a resampler structure that uses a user-written resampler using parameter value pairs.

Examples

Use Separable Resampler to Stretch an Image in the Y Direction

Read an image into the workspace and display it.

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



Create a separable resampler.

```
resamp = makesampler({'nearest', 'cubic'}, 'fill');
```

Create a spatial transformation structure (TFORM) that defines an affine transformation.

```
stretch = maketform('affine', [1 0; 0 1.3; 0 0]);
```

Apply the transformation, specifying the custom resampler.

```
B = imtransform(A, stretch, resamp);
```

Display the transformed image.

```
imshow(B)
```



Input Arguments

interpolant — Interpolating kernel

character vector | cell array

Interpolating kernel, specified as character vector or cell array. When you specify a character vector, **interpolant** can have any of the following values:

Interpolant	Description
'cubic'	Cubic interpolation
'linear'	Linear interpolation
'nearest'	Nearest-neighbor interpolation

If you are using a custom interpolating kernel, you can specify **interpolant** as a cell array in either of these forms:

{half_width, positive_half}	half_width is a positive scalar designating the half width of a symmetric interpolating kernel. positive_half is a vector of values regularly sampling the kernel on the closed interval [0 positive_half].
{half_width, interp_fcn}	interp_fcn is a function handle that returns interpolating kernel values, given an array of input values in the interval [0 positive_half].

To specify the interpolation method independently along each dimension, combine both types of interpolant specifications. The number of elements in the cell array must equal the number of transform dimensions. For example, consider the following example of an **interpolant** value:

```
{'nearest', 'linear', {2 KERNEL_TABLE}}
```

In this example, the resampler uses nearest-neighbor interpolation along the first transform dimension, linear interpolation along the second dimension, and custom table-based interpolation along the third.

Data Types: char | cell

padmethod — Method used to assign values to output elements that map outside the input array

character vector

Method used to assign values to output elements that map outside the input array, specified as one of the following character vectors.

Pad Method	Description
'bound'	Assigns values from the fill value array to points that map outside the input array. Repeats border elements of the array for points that map inside the array (same as 'replicate'). When <code>interpolant</code> is 'nearest', this pad method produces the same results as 'fill'. 'bound' is like 'fill', but avoids mixing fill values and input image values.
'circular'	Pads array with circular repetition of elements within the dimension. Same as <code>padarray</code> .
'fill'	Generates an output array with smooth-looking edges (except when using nearest-neighbor interpolation). For output points that map near the edge of the input array (either inside or outside), it combines input image and fill values. When <code>interpolant</code> is 'nearest', this pad method produces the same results as 'bound'.
'replicate'	Pads array by repeating border elements of array. Same as <code>padarray</code> .
'symmetric'	Pads array with mirror reflections of itself. Same as <code>padarray</code> .

For 'fill', 'replicate', 'circular', or 'symmetric', the resampling performed by `tformarray` occurs in two logical steps:

- 1 Pad the array `A` infinitely to fill the entire input transform space.
- 2 Evaluate the convolution of the padded `A` with the resampling kernel at the output points specified by the geometric map.

Each nontransform dimension is handled separately. The padding is virtual (accomplished by remapping array subscripts) for performance and memory efficiency. If you implement a custom resampler, you can implement these behaviors.

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `resamp = makeresampler('Type', 'separable', 'Interpolant', 'linear', 'PadMethod', 'fill');`

'Type' — Resampler type
 'separable' | 'custom'

Resampler type, specified as one of the following character vectors.

Type	Description
'separable'	Create a separable resampler. If you specify this value, the only other properties that you can specify are 'Interpolant' and 'PadMethod'. The result is equivalent to using the <code>makeresampler(interpolant, padmethod)</code> syntax.
'custom'	Create a customer resampler. If you specify this value, you must specify the 'NDims' and 'ResampleFcn' properties and, optionally, the 'CustomData' property.

Data Types: char

'PadMethod' — Method used to assign values to output elements that map close to or outside the edge of the input array

character vector

See the `padmethod` argument for more information.

Data Types: char

'Interpolant' — Interpolating kernel

character vector | cell array

See the `interpolant` argument for more information.

Data Types: char | cell

'NDims' — Dimensionality custom resampler can handle

positive integer

Dimensionality custom resampler can handle, specified as a positive integer. Use a value of `Inf` to indicate that the custom resampler can handle any dimension. If `'Type'` is `'custom'`, you must specify `NDims`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

'ResampleFcn' — Function that performs the resampling

function handle

Function that performs the resampling, specified as a function handle. You call this function with the following interface:

```
B = resample_fcn(A,M,TDIMS_A,TDIMS_B,FSIZE_A,FSIZE_B,F,R)
```

For more information about the input arguments to this function, see the help for `tformarray`. The argument `M` is an array that maps the transform subscript space of `B` to the transform subscript space of `A`. If `A` has `N` transform dimensions (`N = length(TDIMS_A)`) and `B` has `P` transform dimensions (`P = length(TDIMS_B)`), then `ndims(M) = P + 1`, if `N > 1` and `P` if `N == 1`, and `size(M,P + 1) = N`.

The first `P` dimensions of `M` correspond to the output transform space, permuted according to the order in which the output transform dimensions are listed in `TDIMS_B`. (In general `TDIMS_A` and `TDIMS_B` need not be sorted in ascending order, although some resamplers can impose such a limitation.) Thus, the first `P` elements of `size(M)` determine the sizes of the transform dimensions of `B`. The input transform coordinates to which each point is mapped are arrayed across the final dimension of `M`, following the order given in `TDIMS_A`. `M` must be `double`. `FSIZE_A` and `FSIZE_B` are the full sizes of `A` and `B`, padded with 1's as necessary to be consistent with `TDIMS_A`, `TDIMS_B`, and `size(A)`.

Data Types: `function_handle`

'CustomData' — User-define data

numeric array or character vector

User-defined data, specified using a numeric array or character vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Output Arguments

R — Resampler

structure

Resampler, returned as a structure.

See Also

See Also

tformarray

Introduced before R2006a

maketform

Create spatial transformation structure (TFORM)

Compatibility

`maketform` is not recommended. Use `fitgeotrans`, `affine2d`, `affine3d`, or `projective2d` instead.

Syntax

```
T = maketform('affine',A)
T = maketform('affine',U,X)
T = maketform('projective',A)
T = maketform('projective',U,X)
T = maketform('custom',NDIMS_IN,NDIMS_OUT,FORWARD_FCN,INVERSE_FCN,
TDATA)
T = maketform('box',tsize,LOW,HIGH)
T = maketform('box',INBOUNDS, OUTBOUNDS)
T = maketform('composite',T1,T2,...,TL)
T = maketform('composite', [T1 T2 ... TL])
```

Description

`T = maketform('affine',A)` creates a multidimensional spatial transformation structure `T` for an N -dimensional affine transformation. `A` is a nonsingular real $(N+1)$ -by- $(N+1)$ or $(N+1)$ -by- N matrix. If `A` is $(N+1)$ -by- $(N+1)$, the last column of `A` must be `[zeros(N,1);1]`. Otherwise, `A` is augmented automatically, such that its last column is `[zeros(N,1);1]`. The matrix `A` defines a forward transformation such that `tformfwd(U,T)`, where `U` is a 1-by- N vector, returns a 1-by- N vector `X`, such that $X = U * A(1:N,1:N) + A(N+1,1:N)$. `T` has both forward and inverse transformations.

A spatial transformation structure (called a TFORM struct) that can be used with the `tformfwd`, `tforminv`, `fliptform`, `imtransform`, or `tformarray` functions.

`T = maketform('affine',U,X)` creates a TFORM struct `T` for a two-dimensional affine transformation that maps each row of `U` to the corresponding row of `X`. The `U` and

X arguments are each 3-by-2 and define the corners of input and output triangles. The corners cannot be collinear.

`T = maketform('projective',A)` creates a TFORM struct for an N-dimensional projective transformation. A is a nonsingular real (N+1)-by-(N+1) matrix. A(N+1,N+1) cannot be 0. The matrix A defines a forward transformation such that `tformfwd(U,T)`, where U is a 1-by-N vector, returns a 1-by-N vector X, such that $X = W(1:N)/W(N+1)$, where $W = [U \ 1] * A$. The transformation structure T has both forward and inverse transformations.

`T = maketform('projective',U,X)` creates a TFORM struct T for a two-dimensional projective transformation that maps each row of U to the corresponding row of X. The U and X arguments are each 4-by-2 and define the corners of input and output quadrilaterals. No three corners can be collinear.

`T = maketform('custom',NDIMS_IN,NDIMS_OUT,FORWARD_FCN,INVERSE_FCN,TDATA)` creates a custom TFORM struct T based on user-provided function handles and parameters. NDIMS_IN and NDIMS_OUT are the numbers of input and output dimensions. FORWARD_FCN and INVERSE_FCN are function handles to forward and inverse functions. The forward function must support the following syntax: $X = \text{FORWARD_FCN}(U, T)$. The inverse function must support the following syntax: $U = \text{INVERSE_FCN}(X, T)$. In these syntaxes, U is a P-by-NDIMS_IN matrix whose rows are points in the transformation input space. X is a P-by-NDIMS_OUT matrix whose rows are points in the transformation output space. The TDATA argument can be any MATLAB array and is typically used to store parameters of the custom transformation. It is accessible to FORWARD_FCN and INVERSE_FCN via the `tdata` field of T. Either FORWARD_FCN or INVERSE_FCN can be empty, although at least INVERSE_FCN must be defined to use T with `tformarray` or `imtransform`.

`T = maketform('box',tsize,LOW,HIGH)` or

`T = maketform('box',INBOUNDS, OUTBOUNDS)` builds an N-dimensional affine TFORM struct T. The `tsize` argument is an N-element vector of positive integers. LOW and HIGH are also N-element vectors. The transformation maps an input box defined by the opposite corners `ones(1,N)` and `tsize`, or by corners `INBOUNDS(1,:)` and `INBOUND(2,:)`, to an output box defined by the opposite corners LOW and HIGH or `OUTBOUNDS(1,:)` and `OUTBOUNDS(2,:)`. `LOW(K)` and `HIGH(K)` must be different unless `tsize(K)` is 1, in which case the affine scale factor along the Kth dimension is assumed to be 1.0. Similarly, `INBOUNDS(1,K)` and `INBOUNDS(2,K)` must be different unless `OUTBOUNDS(1,K)` and `OUTBOUNDS(2,K)` are the same, and conversely. The 'box' TFORM is typically used to register the row and column subscripts of an image or array to some world coordinate system.

```
T = maketform('composite',T1,T2,...,TL) or  
T = maketform('composite', [T1 T2 ... TL]) builds a TFORM struct T whose  
forward and inverse functions are the functional compositions of the forward and inverse  
functions of T1, T2, ..., TL.
```

The inputs T1, T2, ..., TL are ordered just as they would be when using the standard notation for function composition: $T = T1 \circ T2 \circ \dots \circ TL$ and note also that composition is associative, but not commutative. This means that to apply T to the input U, one must apply TL first and T1 last. Thus if L = 3, for example, then `tformfwd(U,T)` is the same as `tformfwd(tformfwd(tformfwd(U,T3),T2),T1)`. The components T1 through TL must be compatible in terms of the numbers of input and output dimensions. T has a defined forward transform function only if all the component transforms have defined forward transform functions. T has a defined inverse transform function only if all the component functions have defined inverse transform functions.

Examples

Make TFORM and Apply Transformation to Image

Create a transformation structure (TFORM) that defines an affine transformation.

```
T = maketform('affine',[.5 0 0; .5 2 0; 0 0 1])
```

```
T =
```

```
struct with fields:
```

```
    ndims_in: 2  
    ndims_out: 2  
    forward_fcn: @fwd_affine  
    inverse_fcn: @inv_affine  
    tdata: [1x1 struct]
```

Apply the forward transformation.

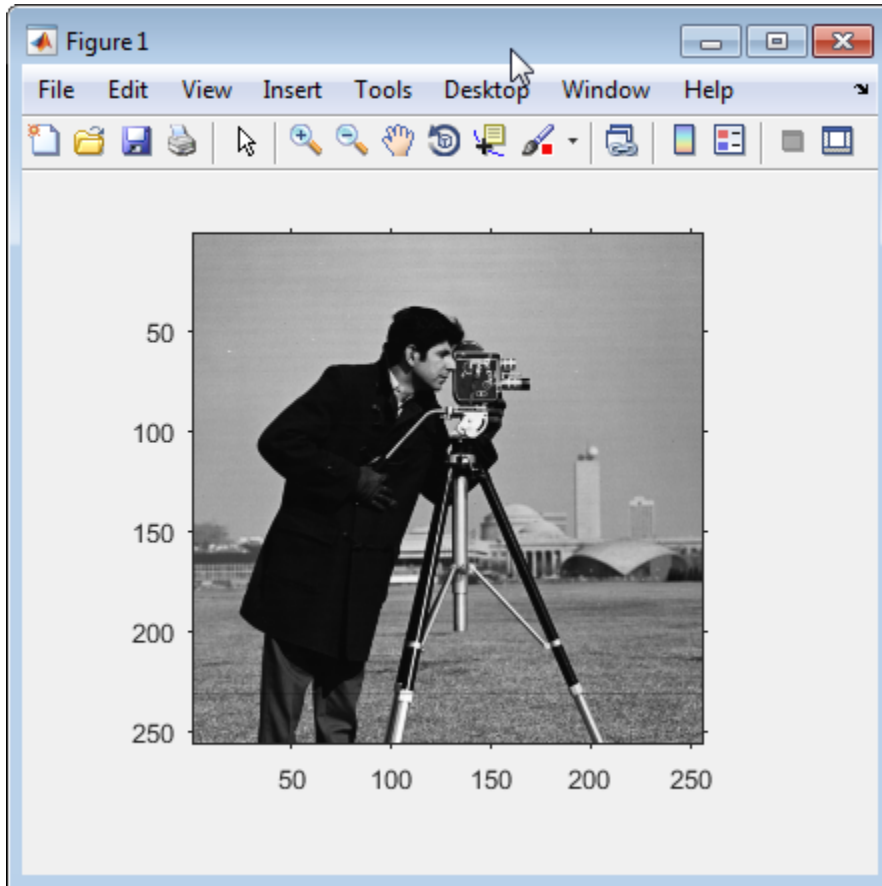
```
tformfwd([10 20],T)
```

```
ans =
```

```
    15    40
```

Read an image into the workspace and display it.

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

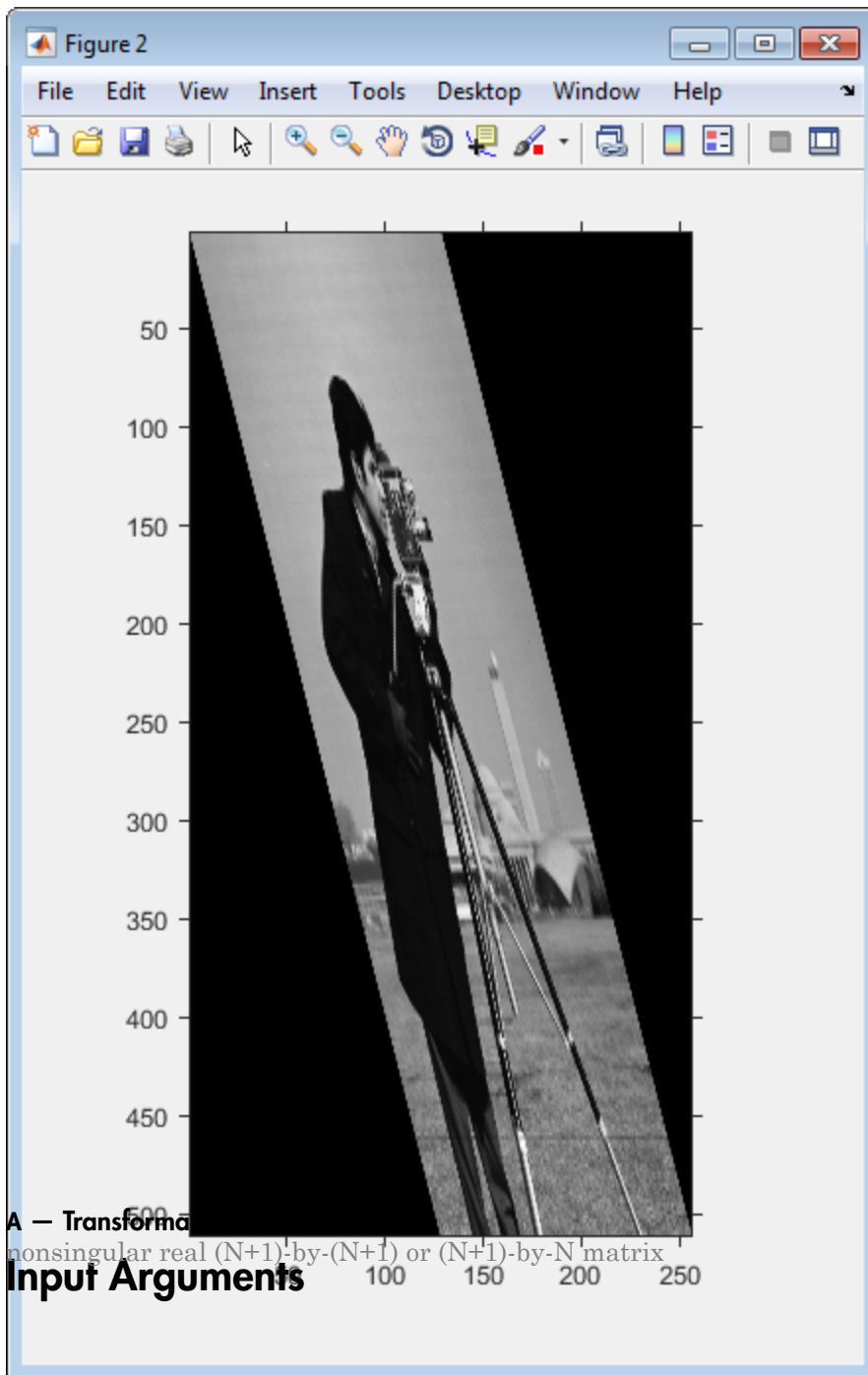


Apply the transformation to the image.

```
I2 = imtransform(I,T);
```

Display the original image and the transformed image.

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



A — Transformation matrix
nonsingular real $(N+1)$ -by- $(N+1)$ or $(N+1)$ -by- N matrix
Input Arguments

Transformation matrix, specified as a nonsingular, real (N+1)-by-(N+1) or (N+1)-by-N matrix.

Data Types: double

U, X — Corners

4-by-2 matrix for projective transformations | 3-by-2 matrix for affine transformations

Corners, specified as a 3-by-2 matrix (for affine transformations) or 4-by-2 matrix (for projective transformations). The matrices define the corners of triangles (for affine transformations) or quadrangles (for projective transformations).

Data Types: double

NDIMS_IN, NDIMS_OUT — Number of input and output dimensions

scalar

Number of input and output dimensions, specified as a scalar.

Example:

Data Types: double

FORWARD_FCN, INVERSE_FCN — Forward and inverse functions

function handle

Forward and inverse functions, specified as function handles.

Data Types: function_handle

TDATA — Parameters of custom transformation

array

Parameters of custom transformation, specified as an array.

Example:

Data Types: double

tsize — Size of input box

n-element vector of positive integers

Size of input box, specified as an n-element vector of positive integers.

Example:

Data Types: double

LOW, HIGH — Corners of output box

n-element vectors.

Corners of output box, specified as an n-element vector.

Example:

Data Types: double

T1, T2, . . . , TL — Forward and inverse functions

function handles

Function and inverse functions, specified as function handles.

Example:

Data Types: function_handle

Output Arguments

T — Multidimensional spatial transformation structure

transformation structure (TFORM)

Multidimensional spatial transformation structure, returned as a transformation structure (TFORM).

Tips

- An affine or projective transformation can also be expressed like this equation, for a 3-by-2 A:

$$[X \ Y]' = A' * [U \ V \ 1]'$$

Or, like this equation, for a 3-by-3 A:

$$[X \ Y \ 1]' = A' * [U \ V \ 1]'$$

See Also

See Also

fliptform | imtransform | tformarray | tformfwd | tforminv

Introduced before R2006a

mat2gray

Convert matrix to grayscale image

Syntax

```
I = mat2gray(A, [amin amax])  
I = mat2gray(A)  
gpuarrayI = mat2gray(gpuarrayA, ___ )
```

Description

`I = mat2gray(A, [amin amax])` converts the matrix `A` to the intensity image `I`. The returned matrix `I` contains values in the range 0.0 (black) to 1.0 (full intensity or white). `amin` and `amax` are the values in `A` that correspond to 0.0 and 1.0 in `I`. Values less than `amin` become 0.0, and values greater than `amax` become 1.0.

`I = mat2gray(A)` sets the values of `amin` and `amax` to the minimum and maximum values in `A`.

`gpuarrayI = mat2gray(gpuarrayA, ___)` performs the operation on a GPU. This syntax requires the Parallel Computing Toolbox.

Class Support

The input array `A` can be `logical` or numeric. The output image `I` is `double`.

The input gpuArray `gpuarrayA` can be `logical` or numeric. The output gpuArray image `gpuarrayI` is `double`.

Examples

Convert a Matrix into an Image

Read an image and, for this example, turn it into a numeric matrix.

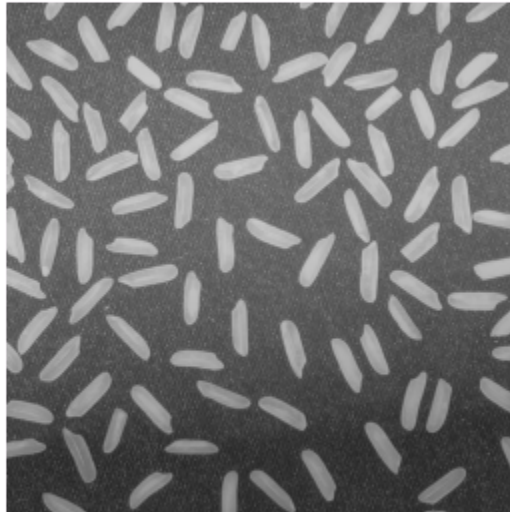
```
I = imread('rice.png');  
J = filter2(fspecial('sobel'),I);
```

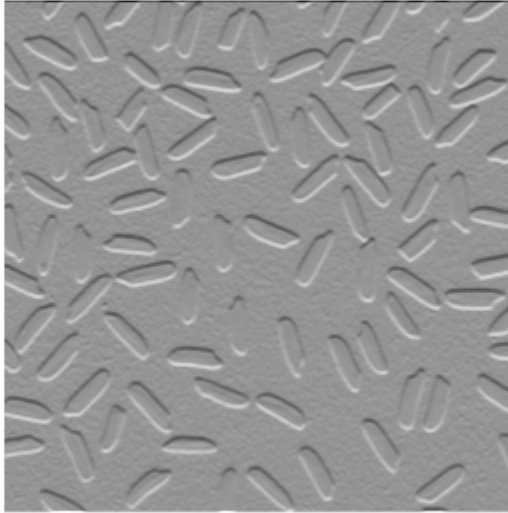
Convert the matrix into an image.

```
K = mat2gray(J);
```

Display the original image and the result of the conversion.

```
imshow(I), figure, imshow(K)
```





See Also

`gray2ind` | `ind2gray` | `rgb2gray` | `gpuArray`

Introduced before R2006a

mean2

Average or mean of matrix elements

Syntax

```
B = mean2(A)
gpuarrayB = mean2(gpuarrayA)
```

Description

`B = mean2(A)` computes the mean of the values in `A`.

`gpuarrayB = mean2(gpuarrayA)` computes the mean of the values in `gpuarrayA`, performing the operation on a GPU. This syntax requires the Parallel Computing Toolbox.

Class Support

The input image `A` can be `numeric` or `logical`. The output image `B` is a scalar of class `double`.

The input image `gpuarrayA` is a `gpuArray` whose underlying class is `numeric` or `logical`. The output image `gpuarrayB` is a `gpuArray` scalar with the underlying class `double`.

Examples

Compute Mean of an Image

Read an image into the workspace.

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

Compute the mean.

```
meanval = mean2(I)
```

```
meanval = 140.2991
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.

See Also

See Also

corr2 | gpuArray | mean | std | std2

Introduced before R2006a

medfilt2

2-D median filtering

Syntax

```
B = medfilt2(A)
B = medfilt2(A, [m n])
B = medfilt2( ____, padopt)
gpuarrayB = medfilt2(gpuarrayA, ____)
```

Description

`B = medfilt2(A)` performs median filtering of the matrix `A` in two dimensions. Each output pixel contains the median value in a 3-by-3 neighborhood around the corresponding pixel in the input image. `medfilt2` pads the image with 0's on the edges, so the median values for points within one-half the width of the neighborhood ($([m\ n])/2$) of the edges might appear distorted.

`B = medfilt2(A, [m n])` performs median filtering, where each output pixel contains the median value in the `m`-by-`n` neighborhood around the corresponding pixel in the input image.

`B = medfilt2(____, padopt)` controls how `medfilt2` pads the matrix boundaries.

`gpuarrayB = medfilt2(gpuarrayA, ____)` performs the median filtering operation on a GPU. The input image and the output image are `gpuArrays`. When working with `gpuArrays`, `medfilt2` only supports square neighborhoods with odd-length sides between 3 and 15. This syntax requires the Parallel Computing Toolbox.

Examples

Remove Salt and Pepper Noise from Image

Read image into workspace and display it.

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



Add salt and pepper noise.

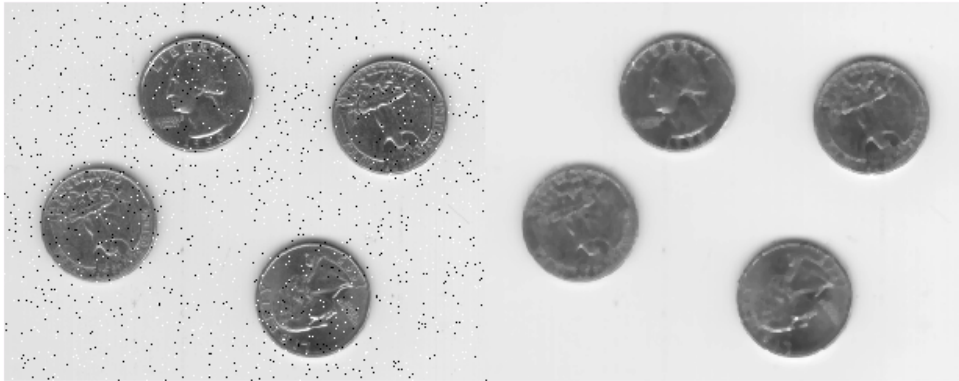
```
J = imnoise(I, 'salt & pepper', 0.02);
```

Use a median filter to filter out the noise.

```
K = medfilt2(J);
```

Display results, side-by-side.

```
imshowpair(J,K, 'montage')
```

Remove Salt and Pepper Noise from Image Using a GPU

Read the image into a gpuArray.

```
I = gpuArray(imread('eight.tif'));
```

Read the image into a gpuArray.

```
J = imnoise(I,'salt & pepper',0.02);  
K = medfilt2(J);  
figure, imshow(J), figure, imshow(K)
```

Input Arguments

A — Input matrix

2-D, real, nonsparse, numeric or logical matrix

Input matrix, specified as a 2-D, real, nonsparse, numeric or logical matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

[m n] — Neighborhood size

3-by-3 (default) | two-element numeric vector

Neighborhood size, specified as a two-element numeric vector, [m n], of real positive integers.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

gpuarrayA — Input matrix when run on a GPU

gpuArray

Input matrix when run on a GPU, specified as a gpuArray.

padopt — Padding option

'zeros' (default) | 'symmetric' | 'indexed'

Padding option, specified as one of the following values:

Value	Description
'zeros'	Padded with 0s. This is the default.
'symmetric'	Symmetrically extended at the boundaries.
'indexed'	Padded with 1s, if the class of A is double; otherwise, padded with 0s.

Data Types: char

Output Arguments

B — Output image

2-D array

Output image, returned as a 2-D array of the same class as the input image A.

gpuarrayB — Output image when run on a GPU

gpuArray

Output image when run on a GPU, returned as a gpuArray.

Tips

- Median filtering is a nonlinear operation often used in image processing to reduce "salt and pepper" noise. A median filter is more effective than convolution when the goal is to simultaneously reduce noise and preserve edges. For information about performance considerations, see `ordfilt2`.
- If the input image `A` is of an integer class, all the output values are returned as integers. If the number of pixels in the neighborhood (i.e., $m \times n$) is even, some of the median values might not be integers. In these cases, the fractional parts are discarded. Logical input is treated similarly. For example, the true median for the following 2-by-2 neighborhood in a `uint8` array is 4.5, but `medfilt2` discards the fractional part and returns 4.

```
1 5
4 8
```

Algorithms

On the CPU, `medfilt2` uses `ordfilt2` to perform the filtering.

References

- [1] Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, pp. 469-476.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a

shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

- When generating code, the `padopt` argument must be a compile-time constant.

See Also

See Also

`filter2` | `gpuArray` | `ordfilt2` | `wiener2`

Introduced before R2006a

medfilt3

3-D median filtering

Syntax

```
B = medfilt3(A)
B = medfilt3(A,[m n p])
B = medfilt3( ____,padopt)
```

Description

`B = medfilt3(A)` filters the 3-D image `A` with a 3-by-3-by-3 filter. By default, `medfilt3` pads the image by replicating the values in a mirrored way at the borders.

`B = medfilt3(A,[m n p])` performs median filtering of the 3-D image `A` in three dimensions. Each output voxel in `B` contains the median value in the *m*-by-*n*-by-*p* neighborhood around the corresponding voxel in `A`.

`B = medfilt3(____,padopt)` controls how `medfilt3` pads the array boundaries.

Examples

Use Median Filtering to Remove Outliers in 3-D Data

Create a noisy 3-D surface.

```
[x,y,z,V] = flow(50);
noisyV = V + 0.1*double(rand(size(V))>0.95) - 0.1*double(rand(size(V))<0.05);
```

Apply median filtering.

```
filteredV = medfilt3(noisyV);
```

Display the noisy and filtered surfaces together.

```
subplot(1,2,1)
hpatch1 = patch(isosurface(x,y,z,noisyV,0));
isonormals(x,y,z,noisyV,hpatch1)
set(hpatch1,'FaceColor','red','EdgeColor','none')
```

```
daspect([1,4,4])
view([-65,20])
axis tight off
camlight left
lighting phong

subplot(1,2,2)
hpatch2 = patch(isosurface(x,y,z,filteredV,0));
isonormals(x,y,z,filteredV,hpatch2)
set(hpatch2,'FaceColor','red','EdgeColor','none')
daspect([1,4,4])
view([-65,20])
axis tight off
camlight left
lighting phong
```



Input Argument

A — Input image

3-D, real, nonsparse, numeric or logical array

Input image, specified as a 3-D, real, nonsparse, numeric, or logical array. If the input image is an integer class, all the output values are also integers.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

[m n p] — Neighborhood size

3-by-3-by-3 (default) | three-element numeric vector

Neighborhood size, specified as a three-element numeric vector, [m n p], of real positive integers. The values of *m*, *n*, and *p* must be odd integers.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

padopt — Padding option

'symmetric' (default) | 'zeros' | 'replicate'

Padding option, specified as one of the following values:

Value	Description
'symmetric'	Pad array with mirror reflections of itself
'replicate'	Pad array by repeating border elements
'zeros'	Pad array with 0s

Data Types: char

Output Arguments

B — Output image

3-D array

Output image, returned as a 3-D array of the same class and size as the input image A.

See Also

See Also

medfilt2

Introduced in R2016b

montage

Display multiple image frames as rectangular montage

Syntax

```
montage(filenamees)
montage(I)
montage(X, map)
montage(..., param1, value1, param2, value2, ...)
img = montage(...)
```

Description

`montage(filenamees)` displays a montage of the images specified in `filenamees`. `filenamees` is an N -by-1 or 1-by- N cell array of filenames. If the files are not in the current directory or in a directory on the MATLAB path, you must specify the full pathname. See the `imread` command for more information. If one or more of the image files contains an indexed image, `montage` uses the colormap from the first indexed image file. `montage` arranges the frames so that they roughly form a square.

`montage(I)` displays all the frames of a multiframe image array `I` in a single image object. `I` can be a sequence of binary, grayscale, or truecolor images. A binary or grayscale image sequence must be an M -by- N -by-1-by- K array. A truecolor image sequence must be an M -by- N -by-3-by- K array.

`montage(X, map)` displays all the frames of the indexed image array `X`, using the colormap `map` for all frames. `X` is an M -by- N -by-1-by- K array.

`montage(..., param1, value1, param2, value2, ...)` returns a customized display of an image montage, depending on the values of the optional parameter/value pairs. See “Parameters” on page 1-1500 for a list of available parameters.

`img = montage(...)` returns a single image object which contains all the frames displayed.

Parameters

The following table lists the parameters available, alphabetically by name. Parameter names can be abbreviated, and case does not matter.

Parameter	Value
'DisplayRange'	1-by-2 vector, [LOW HIGH] that controls the display range of each image in a grayscale sequence. The value LOW (and any value less than LOW) displays as black; the value HIGH (and any value greater than HIGH) displays as white. If you specify an empty matrix ([]), <code>montage</code> uses the minimum and maximum values of the images to be displayed in the montage as specified by 'Indices'. For example, if 'Indices' is 1:K and the 'DisplayRange' is set to [], the minimum value in <code>I (min(I(:)))</code> is displayed as black, and the maximum value (<code>max(I(:))</code>) is displayed as white. Default: Range of the data type of I.
'Indices'	Numeric array specifying which frames to display in the montage. <code>montage</code> interprets the values as indices into array I or cell array <code>filenames</code> . For example, to create a montage of the first four frames in I, enter <code>montage(I, 'Indices', 1:4);</code> . You can use this parameter to specify individual frames or skip frames. For example, the value 1:2:20 displays every other frame. Default: 1:K, where K is the total number of frames or image files.
'Parent'	Axes object that is the parent of the image object created by <code>montage</code> .
'Size'	Two-element vector, [NROWS NCOLS], specifying the number of rows and number of columns in the montage. Use NaNs in the size vector to indicate that <code>montage</code> should calculate size in a particular dimension in a way that includes all the images in the montage. For example, if 'Size' is [2 NaN], the montage will have two rows, and the number of columns will be computed automatically to include all of the images in the montage. <code>montage</code> displays the images horizontally across columns. Default: Calculated so the images in the montage roughly form a square. <code>montage</code> considers the aspect ratio when calculating the number of images to display horizontally and vertically.

Class Support

A grayscale image array can be `logical`, `uint8`, `uint16`, `int16`, `single`, or `double`. An indexed image can be `logical`, `uint8`, `uint16`, `single`, or `double`. The colormap must be `double`. A truecolor image can be `uint8`, `uint16`, `single`, or `double`. The output is an image object produced by `montage`.

Examples

Create Montage from Series of Files

Create a list of filenames.

```
fileFolder = fullfile(matlabroot, 'toolbox', 'images', 'imdata');  
dirOutput = dir(fullfile(fileFolder, 'AT3_1m4_*.tif'));  
fileNames = {dirOutput.name}'
```

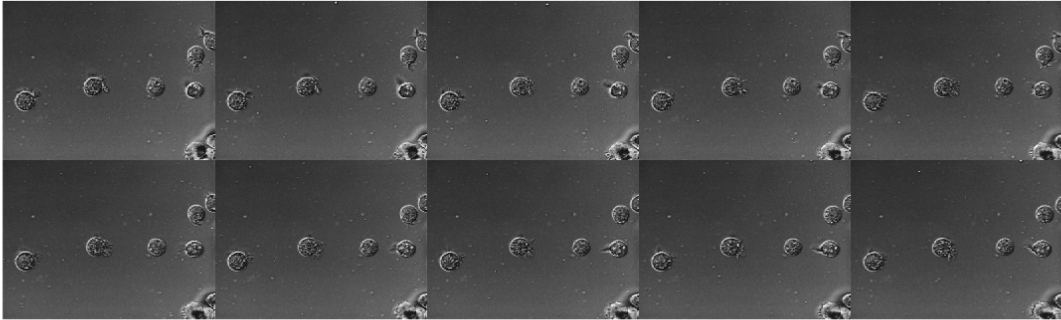
```
fileNames =
```

```
10×1 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'
```

Display the sequence of images.

```
montage(fileNames, 'Size', [2 5]);
```



Use DisplayRange Parameter to Highlight Image Structures

Create a list of file names.

```
fileFolder = fullfile(matlabroot, 'toolbox', 'images', 'imdata');  
dirOutput = dir(fullfile(fileFolder, 'AT3_1m4_*.tif'));  
fileNames = {dirOutput.name}'
```

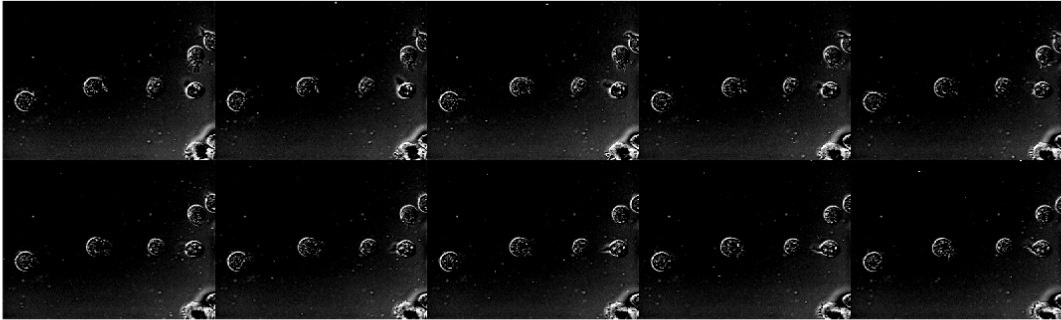
```
fileNames =
```

```
10×1 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'
```

Display the sequence of images as a montage, using the `DisplayRange` parameter to highlight structures in the images.

```
montage(fileNames, 'Size', [2 5], 'DisplayRange', [75 200]);
```



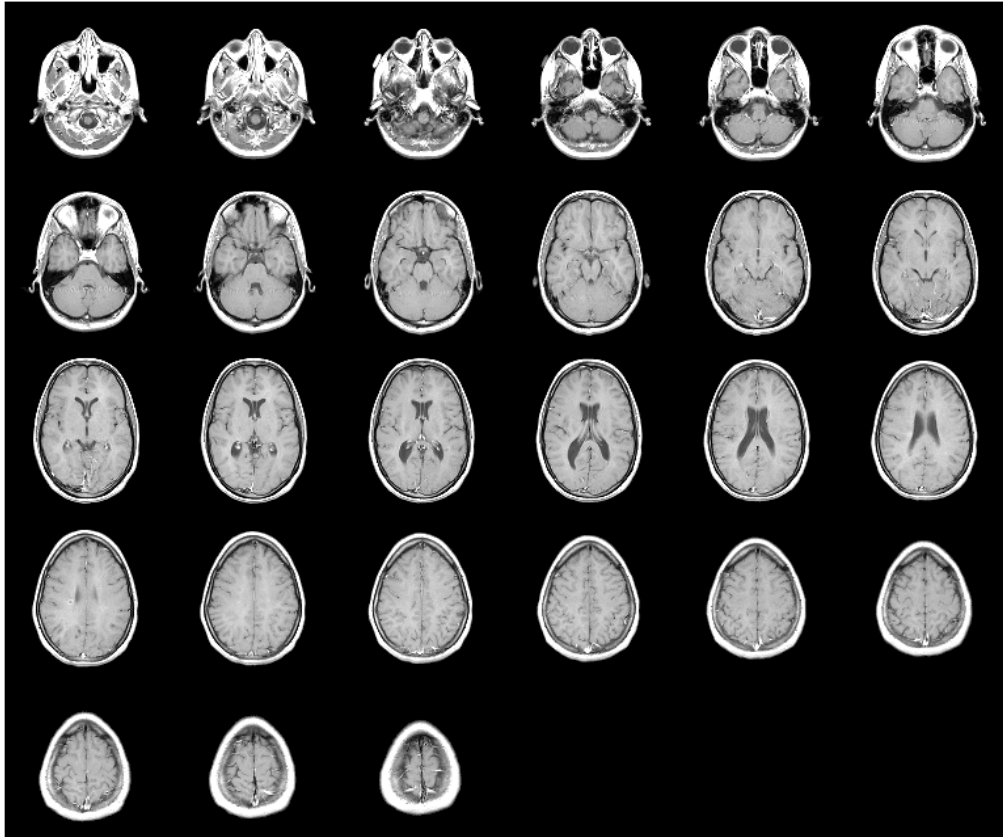
Customize Number of Images in Montage

Load images.

```
load mri
```

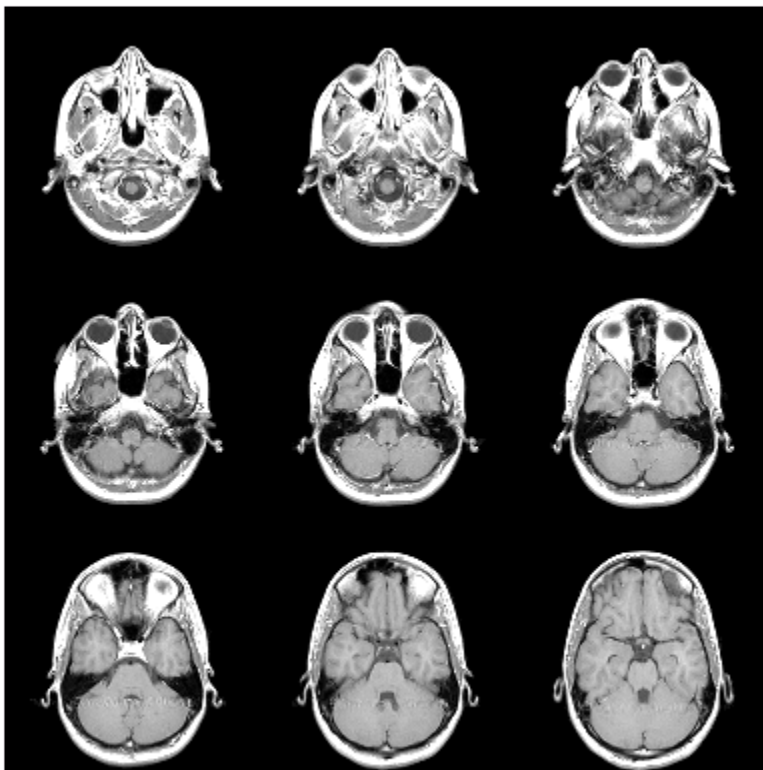
Display as montage.

```
montage(D,map)
```



Create new montage containing only the first nine images.

```
figure  
montage(D, map, 'Indices', 1:9);
```



See Also

`immovie` | `imshow` | `imshowarray`

Introduced before R2006a

multithresh

Multilevel image thresholds using Otsu's method

Syntax

```
thresh = multithresh(A)  
thresh = multithresh(A,N)  
[thresh,metric] = multithresh( ___ )
```

Description

`thresh = multithresh(A)` returns the single threshold value `thresh` computed for image `A` using Otsu's method. You can use `thresh` as an input argument to `imquantize` to convert an image into a two-level image.

`thresh = multithresh(A,N)` returns `thresh` a 1-by-`N` vector containing `N` threshold values using Otsu's method. You can use `thresh` as an input argument to `imquantize` to convert image `A` into an image with `N + 1` discrete levels.

`[thresh,metric] = multithresh(___)` returns `metric`, a measure of the effectiveness of the computed thresholds. `metric` is in the range `[0 1]` and a higher value indicates greater effectiveness of the thresholds in separating the input image into `N + 1` regions based on Otsu's objective criterion.

Examples

Segment Image Into Two Regions

Read image and display it.

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

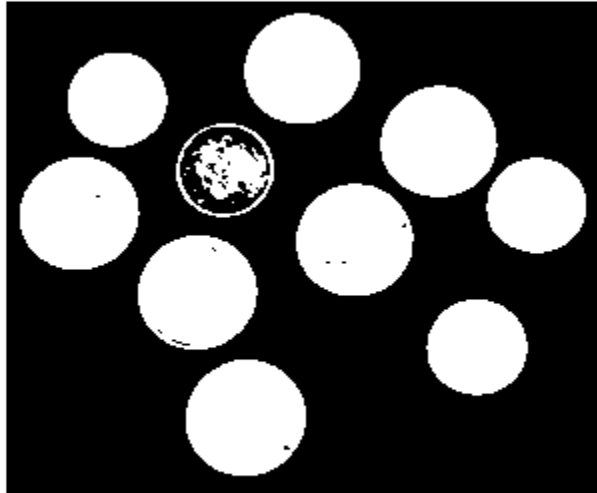



Calculate a single threshold value for the image.

```
level = multithresh(I);
```

Segment the image into two regions using `imquantize` , specifying the threshold level returned by `multithresh` .

```
seg_I = imquantize(I,level);  
figure  
imshow(seg_I,[])
```

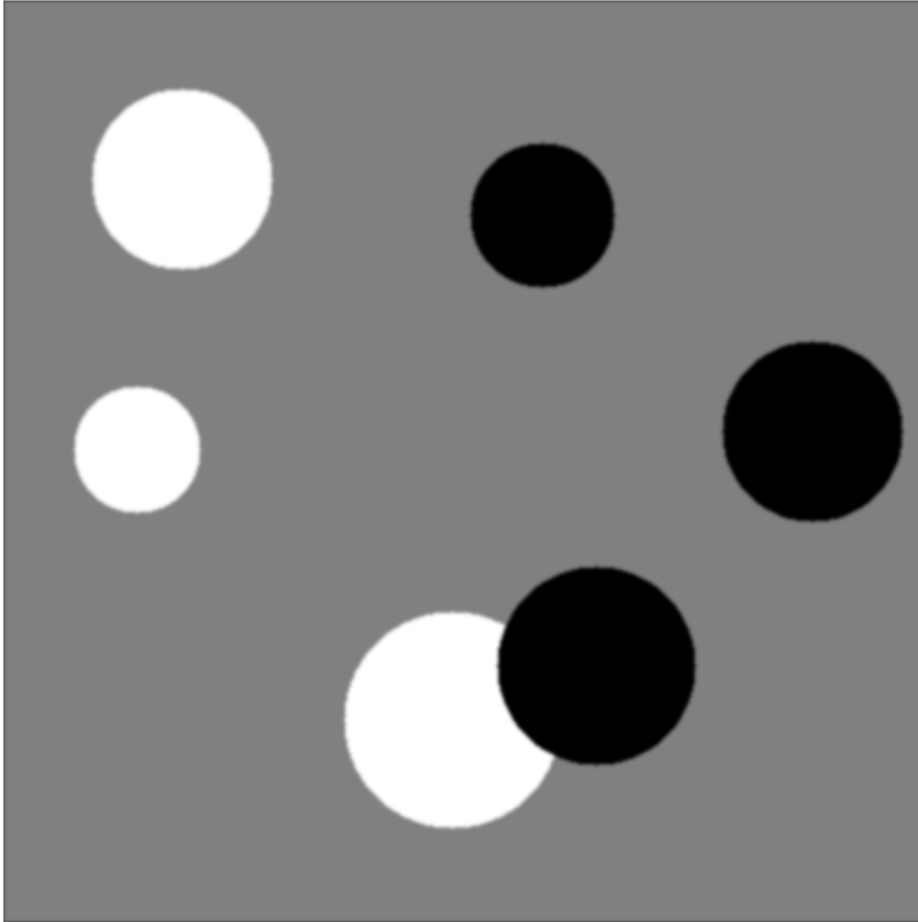


Segment Image into Three Levels Using Two Thresholds

Read image and display it.

```
I = imread('circlesBrightDark.png');  
imshow(I)  
axis off  
title('Original Image')
```

Original Image



Calculate two threshold levels.

```
thresh = multithresh(I,2);
```

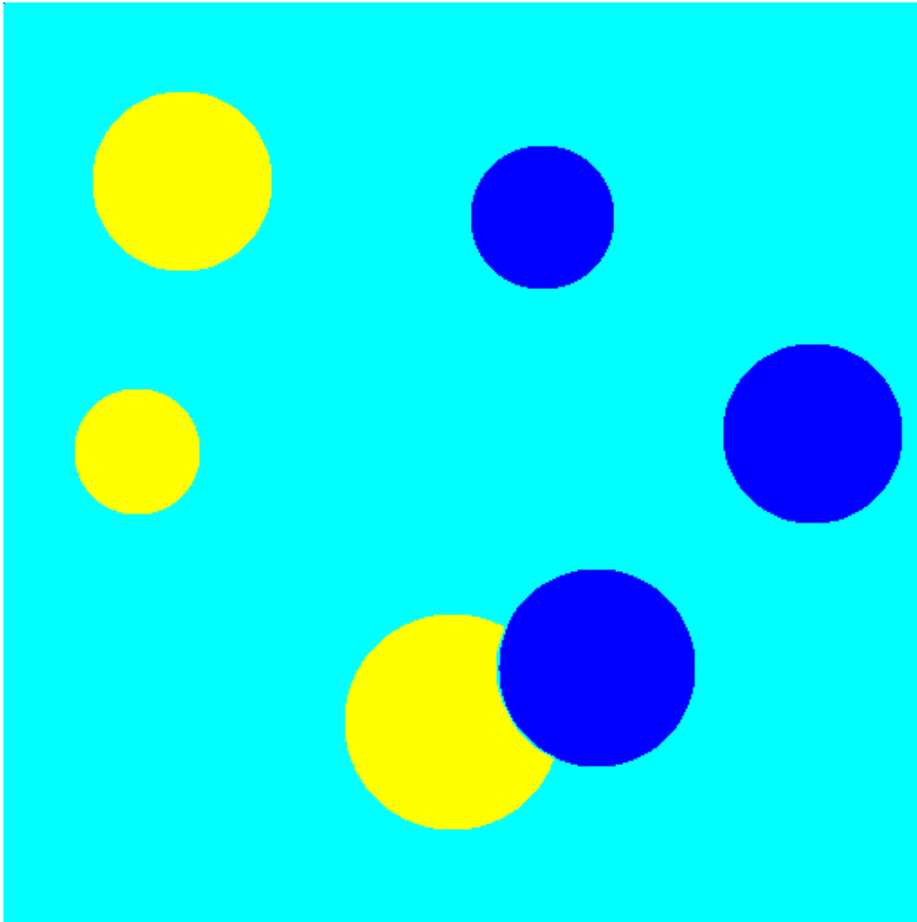
Segment the image into three levels using `imquantize` .

```
seg_I = imquantize(I,thresh);
```

Convert segmented image into color image using `label2rgb` and display it.

```
RGB = label2rgb(seg_I);  
figure;  
imshow(RGB)  
axis off  
title('RGB Segmented Image')
```

RGB Segmented Image



Compare Thresholding Entire Image Versus Plane-by-Plane Thresholding

Read truecolor (RGB) image and display it.

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

```
imshow(I)
axis off
title('RGB Image');
```

RGB Image



Generate thresholds for seven levels from the entire RGB image.

```
threshRGB = multithresh(I,7);
```

Generate thresholds for each plane of the RGB image.

```
threshForPlanes = zeros(3,7);
```

```
for i = 1:3
```

```

    threshForPlanes(i,:) = multithresh(I(:,:,i),7);
end

```

Process the entire image with the set of threshold values computed from entire image.

```

value = [0 threshRGB(2:end) 255];
quantRGB = imquantize(I, threshRGB, value);

```

Process each RGB plane separately using the threshold vector computed from the given plane. Quantize each RGB plane using threshold vector generated for that plane.

```

quantPlane = zeros( size(I) );

for i = 1:3
    value = [0 threshForPlanes(i,2:end) 255];
    quantPlane(:,:,i) = imquantize(I(:,:,i),threshForPlanes(i,:),value);
end

```

```

quantPlane = uint8(quantPlane);

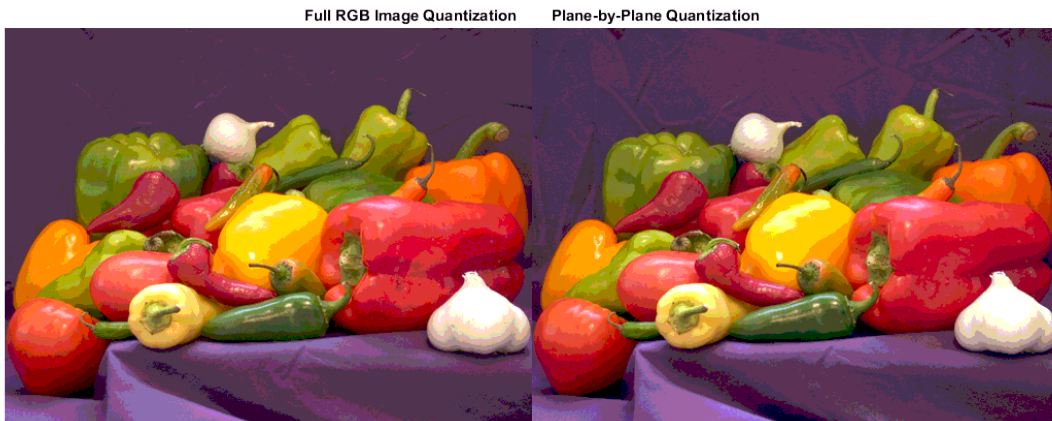
```

Display both posterized images and note the visual differences in the two thresholding schemes.

```

imshowpair(quantRGB,quantPlane,'montage')
axis off
title('Full RGB Image Quantization           Plane-by-Plane Quantization')

```



To compare the results, calculate the number of unique RGB pixel vectors in each output image. Note that the plane-by-plane thresholding scheme yields about 23% more colors than the full RGB image scheme.

```
dim = size( quantRGB );
quantRGBmx3 = reshape(quantRGB, prod(dim(1:2)), 3);
quantPlanemx3 = reshape(quantPlane, prod(dim(1:2)), 3);

colorsRGB = unique(quantRGBmx3, 'rows' );
colorsPlane = unique(quantPlanemx3, 'rows' );

disp(['Unique colors in RGB image          : ' int2str(length(colorsRGB))]);
Unique colors in RGB image          : 188
disp(['Unique colors in Plane-by-Plane image : ' int2str(length(colorsPlane))]);
Unique colors in Plane-by-Plane image : 231
```

Check Results Using the Metric Output Argument

Read image.

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

Find all unique grayscale values in image.

```
uniqLevels = unique(I(:));

disp(['Number of unique levels = ' int2str( length(uniqLevels) )]);
Number of unique levels = 148
```

Compute a series of thresholds at monotonically increasing values of N.

```
Nvals = [1 2 4 8];
for i = 1:length(Nvals)
    [thresh, metric] = multithresh(I, Nvals(i) );
    disp(['N = ' int2str(Nvals(i)) ' | metric = ' num2str(metric)]);
end

N = 1 | metric = 0.54767
N = 2 | metric = 0.98715
N = 4 | metric = 0.99648
```



```
N = 8 | metric = 0.99902
```

Apply the set of 8 threshold values to obtain a 9-level segmentation using `imquantize` .

```
seg_Neq8 = imquantize(I,thresh);
uniqLevels = unique( seg_Neq8(:) )
```

```
uniqLevels =
```

```
1
2
3
4
5
6
7
8
9
```

Threshold the image using `seg_Neq8` as an input to `multithresh`. Set `N` equal to 8, which is 1 less than the number of levels in this segmented image. `multithresh` returns a `metric` value of 1.

```
[thresh, metric] = multithresh(seg_Neq8,8)
```

```
thresh =
```

```
1.8784 2.7882 3.6667 4.5451 5.4549 6.3333 7.2118 8.1216
```

```
metric = 1
```

Threshold the image again, this time increasing the value of `N` by 1. This value now equals the number of levels in the image. Note how the input is degenerate because the number of levels in the image is too few for the number of requested thresholds. Hence, `multithresh` returns a `metric` value of 0.

```
[thresh, metric] = multithresh(seg_Neq8,9)
```

```
Warning: No solution exists because the number of unique levels in the image are too fe
```

```
thresh =
```

```
1 2 3 4 5 6 7 8 9
```

```
metric = 0
```

Input Arguments

A — Image to be thresholded

real, nonsparse numeric array of any dimension

Image to be thresholded, specified as a real, nonsparse numeric array of any dimension. `multithresh` finds the thresholds based on the aggregate histogram of the entire array. `multithresh` considers an RGB image as a 3-D numeric array and computes the thresholds for the combined data from all three color planes.

`multithresh` uses the range of the input image `A`, `[min(A(:)) max(A(:))]`, as the limits for computing the histogram used in subsequent computations. `multithresh` ignores any NaNs in computation. Any `Inf`s and `-Inf`s are counted in the first and last bin of the histogram, respectively.

For degenerate inputs where the number of unique values in `A` is less than or equal to `N`, there is no viable solution using Otsu's method. For such inputs, the return value `thresh` contains all the unique values from `A` and possibly some extra values that are chosen arbitrarily.

```
Example: I = imread('cameraman.tif'); thresh = multithresh(I);
```

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

N — Number of threshold values

1 (default) | positive integer scalar

Number of threshold values, specified as a positive integer scalar value. For `N > 2`, `multithresh` uses search-based optimization of Otsu's criterion to find the thresholds. The search-based optimization guarantees only locally optimal results. Since the chance of converging to local optimum increases with `N`, it is preferable to use smaller values of `N`, typically `N < 10`. The maximum allowed value for `N` is 20.

```
Example: thresh = multithresh(I,4);
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

thresh — Set of threshold values used to quantize an image

1×N vector

Set of threshold values used to quantize an image, returned as a 1-by-N vector, whose data type is the same as image A.

These thresholds are in the same range as the input image A, unlike the `graythresh` function, which returns a normalized threshold in the range [0, 1].

metric — Measure of the effectiveness of the thresholds

scalar

Measure of the effectiveness of the thresholds, returned as a scalar value. Higher values indicates greater effectiveness of the thresholds in separating the input image into N+1 classes based on Otsu's objective criterion. For degenerate inputs where the number of unique values in A is less than or equal to N, `metric` equals 0.

Data Types: `double`

References

- [1] Otsu, N., "A Threshold Selection Method from Gray-Level Histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 9, No. 1, 1979, pp. 62-66.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms

for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

- The input argument `N` must be a compile-time constant.

See Also

See Also

`graythresh` | `im2bw` | `imquantize` | `rgb2ind`

Introduced in R2012b

nitfinfo

Read metadata from National Imagery Transmission Format (NITF) file

Syntax

```
metadata = nitfinfo(filename)
```

Description

`metadata = nitfinfo(filename)` returns a structure whose fields contain file-level metadata about the images, annotations, and graphics in a National Imagery Transmission Format (NITF) file. NITF is an image format used by the U.S. government and military for transmitting documents. A NITF file can contain multiple images and include text and graphic layers. *filename* is a character array that specifies the name of the NITF file, which must be in the current directory, in a directory on the MATLAB path, or contain the full path to the file.

`nitfinfo` supports version 2.0 and 2.1 NITF files, at all Joint Interoperability Test Command (JITC) compliance levels, as well as the NATO Secondary Image Format (NSIF) 1.0. `nitfinfo` does not support NITF 1.1 files.

See Also

`isnitf` | `nitfread`

Introduced in R2007b

nitfread

Read image from NITF file

Syntax

```
X = nitfread(filename)
X = nitfread(filename,idx)
X = nitfread( ___ Name,Value)
```

Description

`X = nitfread(filename)` reads the first image from the National Imagery Transmission Format (NITF) file specified by the character array `filename`. The `filename` array must be in the current folder or in a folder on the MATLAB path, or it must contain the full path to the file.

`X = nitfread(filename,idx)` reads the image with index number `idx` from a NITF file that contains multiple images.

`X = nitfread(___ Name,Value)` reads an image from a NITF image, where optional parameters control aspects of the operation.

Examples

Read Image Data from NITF File

To run this example, replace the name of the file with the name of a NITF file on your system. You can find sample NITF files on the web.

Read the second image from a NITF file containing multiple images. The example reads a subset of the image data, starting at row, column location (100,200), reading every other value to (105,205).

```
subsec = {[100 2 105],[200 2 205]}
```

```
ntfdata = nitfread('your_file.ntf',2,'PixelRegion',subsec);
```

Input Arguments

filename — Name of NITF file

character vector

Name of NITF file, specified as a character vector.

Data Types: char

idx — Index number of image in NITF file

numeric scalar

Index number of image in NITF file, specified as a numeric scalar of class **double**.

Data Types: double

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

```
Example: ntfdata = nitfread('your_file.ntf',2,'PixelRegion',{[100 2
150],[200 2 250]});
```

'PixelRegion' — Row and column indices of pixels to be read from file

two-element cell array containing vectors of positive integers

Row and column indices of pixels to be read from file, specified as a two-element cell array containing vectors of positive integers. Each element is a two-element vector of the form [**start stop**] or a three-element vector of the form [**start increment stop**], where the first vector specifies the row index and the second vector specifies the column index.

Example: {[100 150],[200 250]} — read pixels starting at row/column location (100,200) ending at location (150,250)

{[100 2 150],[200 2 250]} — read every other pixel starting at row/column location (100,200) ending at location (150,250)

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

Output Arguments

X — Image data from NITF file

numeric array

Image data from NITF file, returned as a numeric array.

Tips

- This function supports version 2.0 and 2.1 NITF files, and NSIF 1.0 files. Image submasks and NITF 1.1 files are not supported.

See Also

See Also

`isnitf` | `nitfinfo`

Introduced in R2007b

nlfiter

General sliding-neighborhood operations

Syntax

```
B = nlfiter(A, [m n], fun)
B = nlfiter(A, 'indexed',...)
```

Description

`B = nlfiter(A, [m n], fun)` applies the function `fun` to each `m`-by-`n` sliding block of the grayscale image `A`. `fun` is a function that accepts an `m`-by-`n` matrix as input and returns a scalar result.

```
c = fun(x)
```

`fun` must be a function handle.

`c` is the output value for the center pixel in the `m`-by-`n` block `x`. `nlfiter` calls `fun` for each pixel in `A`. `nlfiter` zero-pads the `m`-by-`n` block at the edges, if necessary.

`B = nlfiter(A, 'indexed',...)` processes `A` as an indexed image, padding with 1's if `A` is of class `single` or `double` and 0's if `A` is of class `logical`, `uint8`, or `uint16`.

Class Support

The input image `A` can be of any class supported by `fun`. The class of `B` depends on the class of the output from `fun`. When `A` is grayscale, it can be any numeric type or `logical`. When `A` is indexed, it can be `logical`, `uint8`, `uint16`, `single`, or `double`.

Examples

Apply median filter to image

This example shows how apply a median filter to an image using `nlfiter`. This example produces the same result as calling `medfilt2` with a 3-by-3 neighborhood.

```
A = imread('cameraman.tif');  
A = im2double(A);  
fun = @(x) median(x(:));  
B = nlfilter(A,[3 3],fun);  
imshow(A), figure, imshow(B)
```





Tips

`nfilter` can take a long time to process large images. In some cases, the `colfilt` function can perform the same operation much faster.

See Also

`blockproc` | `colfilt`

Topics

“Anonymous Functions” (MATLAB)
“Parameterizing Functions” (MATLAB)
“Create Function Handle” (MATLAB)

Introduced before R2006a

normxcorr2

Normalized 2-D cross-correlation

Syntax

```
C = normxcorr2(template,A)
gpuarrayC = normxcorr2(gpuarrayTemplate,gpuarrayA)
```

Description

`C = normxcorr2(template,A)` computes the normalized cross-correlation of the matrices `template` and `A`. The resulting matrix `C` contains the correlation coefficients.

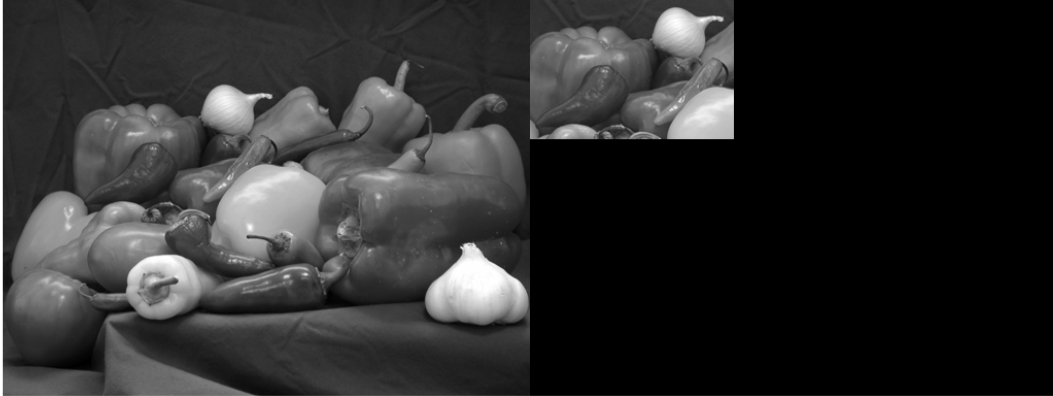
`gpuarrayC = normxcorr2(gpuarrayTemplate,gpuarrayA)` performs the normalized cross-correlation operation on a GPU.

Examples

Use Cross-Correlation to Find Template in Image

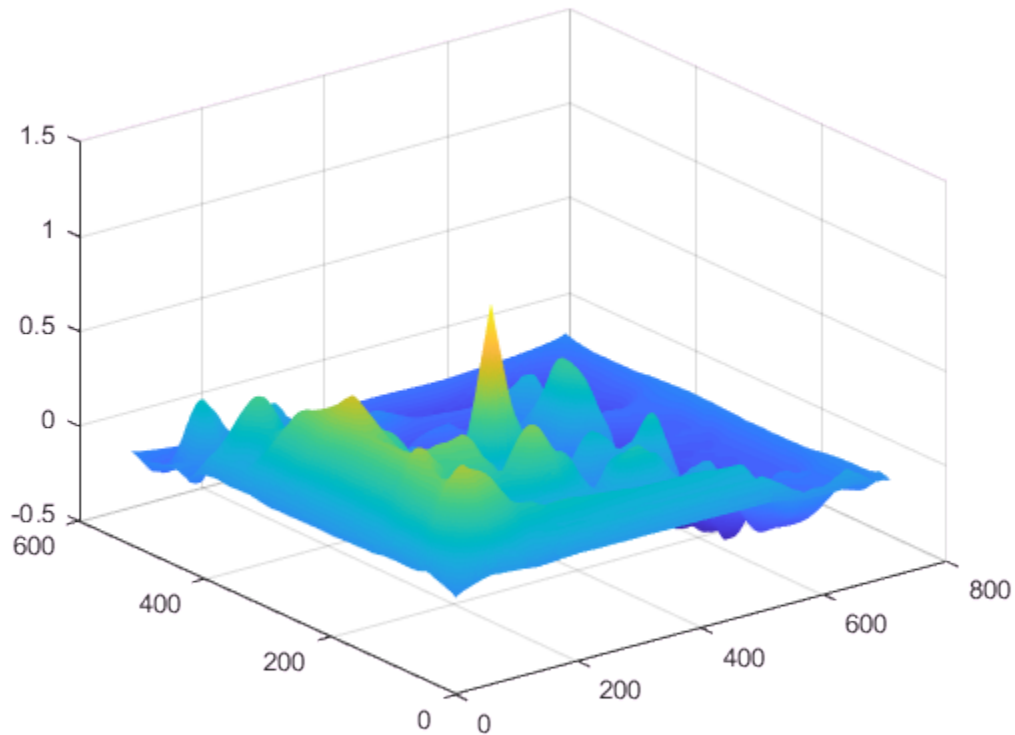
Read images into the workspace and display them side-by-side.

```
onion    = rgb2gray(imread('onion.png'));
peppers = rgb2gray(imread('peppers.png'));
imshowpair(peppers,onion,'montage')
```



Perform cross-correlation and display result as surface.

```
c = normxcorr2(onion,peppers);  
figure, surf(c), shading flat
```



Find peak in cross-correlation.

```
[ypeak, xpeak] = find(c==max(c(:)));
```

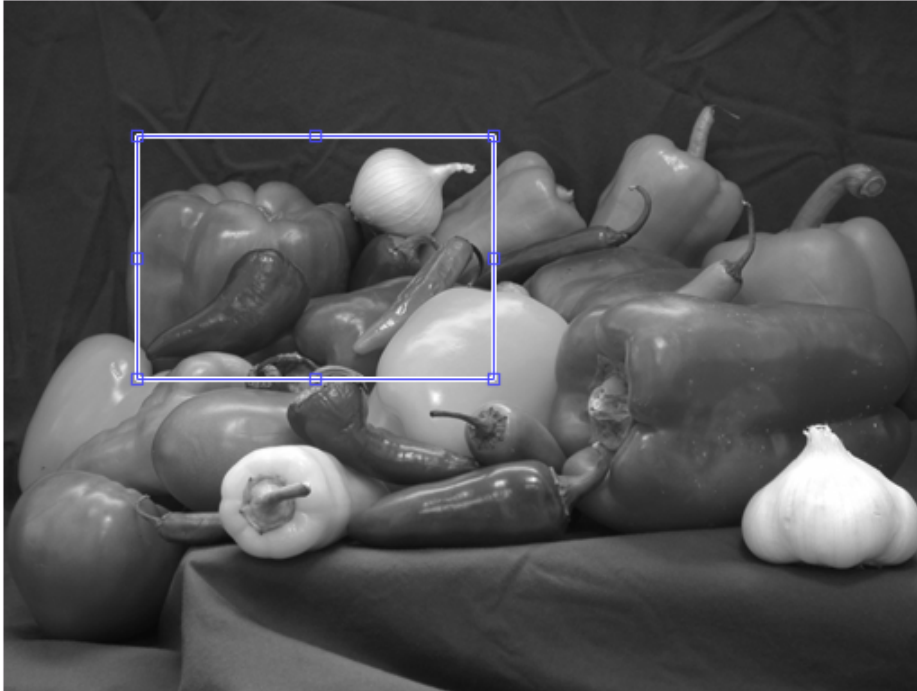
Account for the padding that normxcorr2 adds.

```
yoffset = ypeak-size(onion,1);  
xoffset = xpeak-size(onion,2);
```

Display matched area.

```
hFig = figure;  
hAx = axes;  
imshow(peppers, 'Parent', hAx);
```

```
imrect(hAx, [xoffSet+1, yoffSet+1, size(onion,2), size(onion,1)]);
```



Use Cross-Correlation to Find Template in Image on a GPU

Read images into `gpuArrays`.

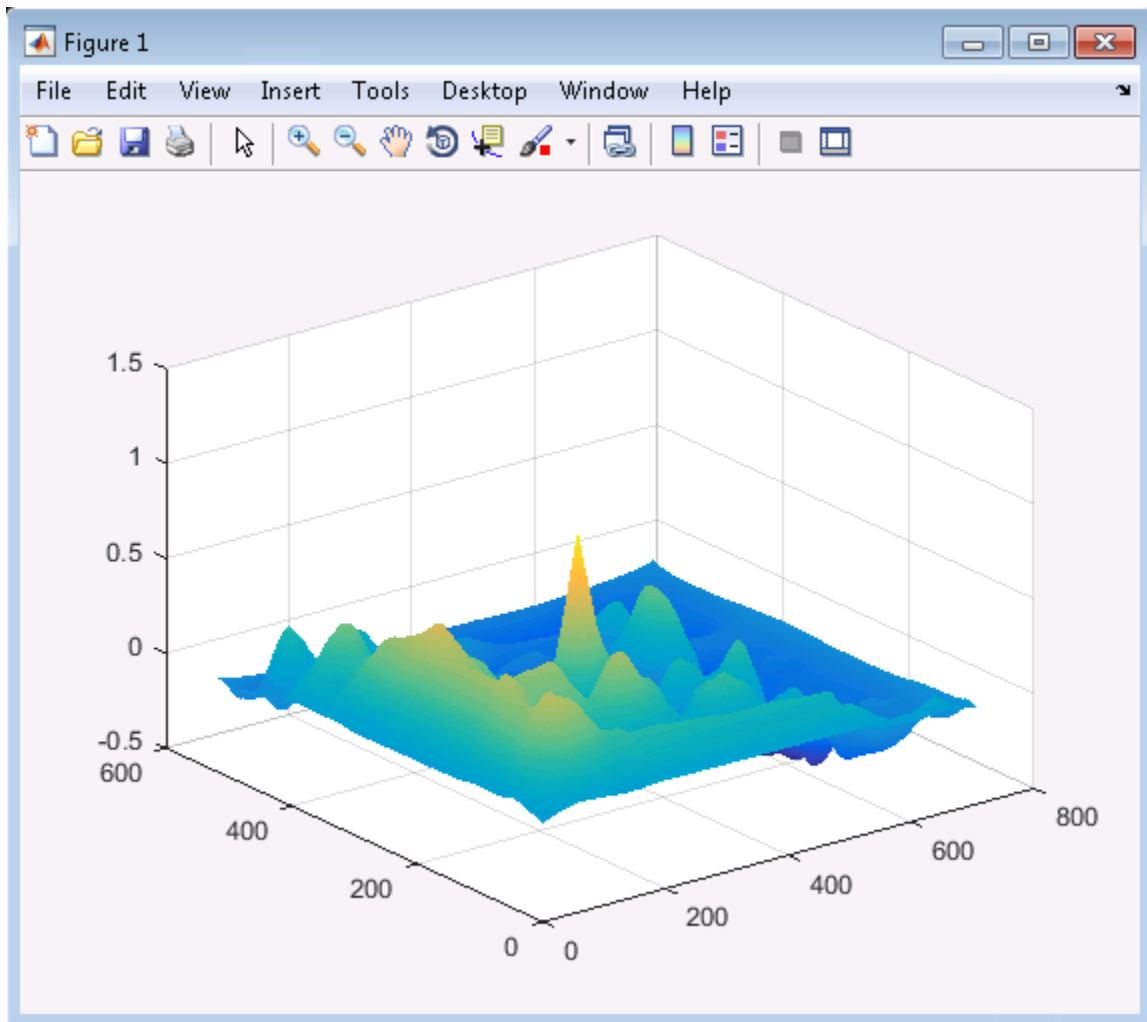
```
onion    = gpuArray(imread('onion.png'));  
peppers = gpuArray(imread('peppers.png'));
```

Convert the color images to 2-D. The `rgb2gray` function accepts `gpuArrays`.


```
onion  = rgb2gray(onion);  
peppers = rgb2gray(peppers);
```

Perform cross-correlation and display result as surface.

```
c = normxcorr2(onion,peppers);  
figure, surf(c), shading flat
```



Find peak in cross-correlation.

```
[ypeak, xpeak] = find(c==max(c(:)));
```

Account for the padding that `normxcorr2` adds.

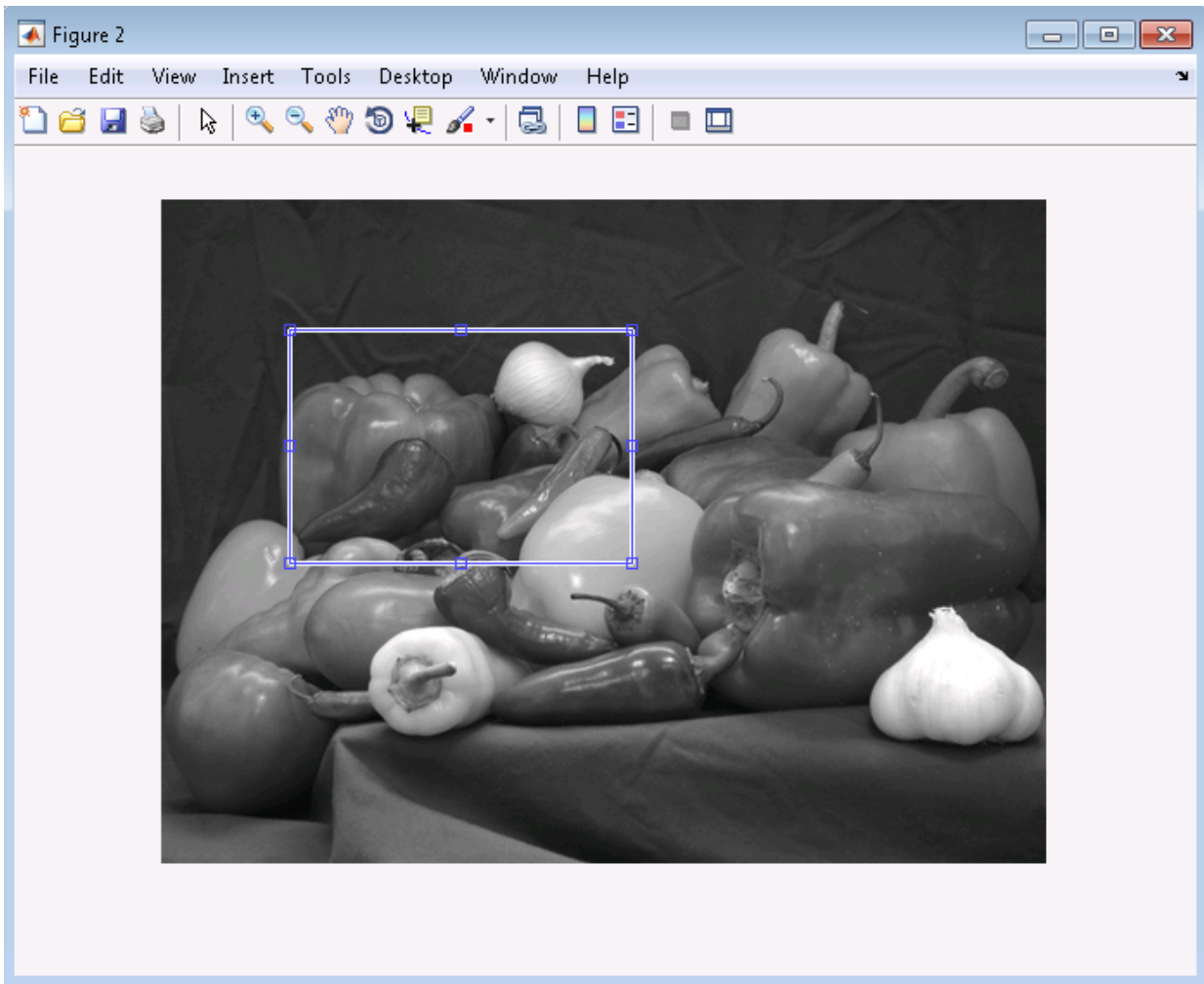
```
yoffSet = ypeak-size(onion,1);  
xoffSet = xpeak-size(onion,2);
```

Move data back to CPU for display.

```
yoffSet = gather(ypeak-size(onion,1));  
xoffSet = gather(xpeak-size(onion,2));
```

Display matched area.

```
hFig = figure;  
hAx = axes;  
imshow(peppers, 'Parent', hAx);  
imrect(hAx, [xoffSet+1, yoffSet+1, size(onion,2), size(onion,1)]);
```



Input Arguments

template — Input template
numeric matrix

Input template, specified as a numeric matrix. The values of `template` cannot all be the same.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

A — Input image

numeric matrix

Input image, specified as a numeric image. `A` must be larger than the matrix `template` for the normalization to be meaningful.

Normalized cross-correlation is an undefined operation in regions where `A` has zero variance over the full extent of the template. In these regions, `normxcorr2` assigns correlation coefficients of zero to the output `C`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

gpuarrayTemplate — Input template

`gpuArray`

Input template, specified as a `gpuArray`.

gpuarrayA — Input image

`gpuArray`

Input image, specified as a `gpuArray`.

Output Arguments

C — Correlation coefficients

numeric matrix

Correlation coefficients, returned as a numeric matrix of class `double`. The correlation coefficients can range in value from -1.0 to 1.0.

gpuarrayC — Correlation coefficients in gpuArray

`gpuArray` whose underlying class must be of class `double`

Correlation coefficients, returned as a `gpuArray` whose underlying type must be of class `double`.

Algorithms

normxcorr2 uses the following general procedure , :

- 1 Calculate cross-correlation in the spatial or the frequency domain, depending on size of images.
- 2 Calculate local sums by precomputing running sums.
- 3 Use local sums to normalize the cross-correlation to get correlation coefficients.

The implementation closely follows the formula from :

$$\gamma(u,v) = \frac{\sum_{x,y} [f(x,y) - \bar{f}_{u,v}] [t(x-u, y-v) - \bar{t}]}{\left\{ \sum_{x,y} [f(x,y) - \bar{f}_{u,v}]^2 \sum_{x,y} [t(x-u, y-v) - \bar{t}]^2 \right\}^{0.5}}$$

where

- f is the image.
- \bar{t} is the mean of the template
- $\bar{f}_{u,v}$ is the mean of $f(x,y)$ in the region under the template.

References

- [1] Lewis, J. P., "Fast Normalized Cross-Correlation," *Industrial Light & Magic*
- [2] Haralick, Robert M., and Linda G. Shapiro, *Computer and Robot Vision*, Volume II, Addison-Wesley, 1992, pp. 316-317.

See Also

See Also

corrcoef

Introduced before R2006a

ntsc2rgb

Convert NTSC values to RGB color space

Syntax

```
rgbmap = ntsc2rgb(yiqmap)  
RGB = ntsc2rgb(YIQ)
```

Description

`rgbmap = ntsc2rgb(yiqmap)` converts the *m*-by-3 NTSC (television) color values in `yiqmap` to RGB color space. If `yiqmap` is *m*-by-3 and contains the NTSC luminance (*Y*) and chrominance (*I* and *Q*) color components as columns, then `rgbmap` is an *m*-by-3 matrix that contains the red, green, and blue values equivalent to those colors. Both `rgbmap` and `yiqmap` contain intensities in the range 0 to 1.0. The intensity 0 corresponds to the absence of the component, while the intensity 1.0 corresponds to full saturation of the component.

`RGB = ntsc2rgb(YIQ)` converts the NTSC image `YIQ` to the equivalent truecolor image `RGB`.

`ntsc2rgb` computes the RGB values from the NTSC components using

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.621 \\ 1.000 & -0.272 & -0.647 \\ 1.000 & -1.106 & 1.703 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}.$$

Class Support

The input image or colormap must be of class `double`. The output is of class `double`.

Examples

Convert Image from YIQ to RGB

This example shows how to convert an image from RGB to NTSC color space and back.

Read an RGB image into the workspace.

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

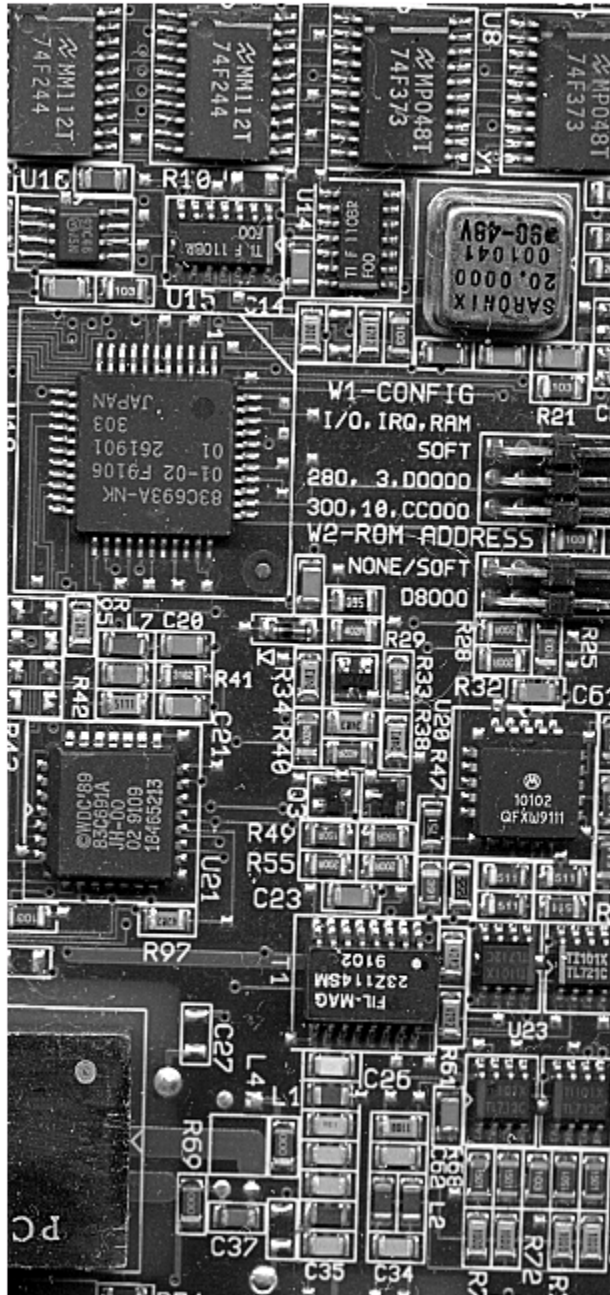
Convert the image to YIQ color space.

```
YIQ = rgb2ntsc(RGB);
```

Display the NTSC luminance, represented by the first color channel in the YIQ image.

```
imshow(YIQ(:,:,1))  
title('Luminance in YIQ Color Space')
```

Luminance in YIQ Color Space



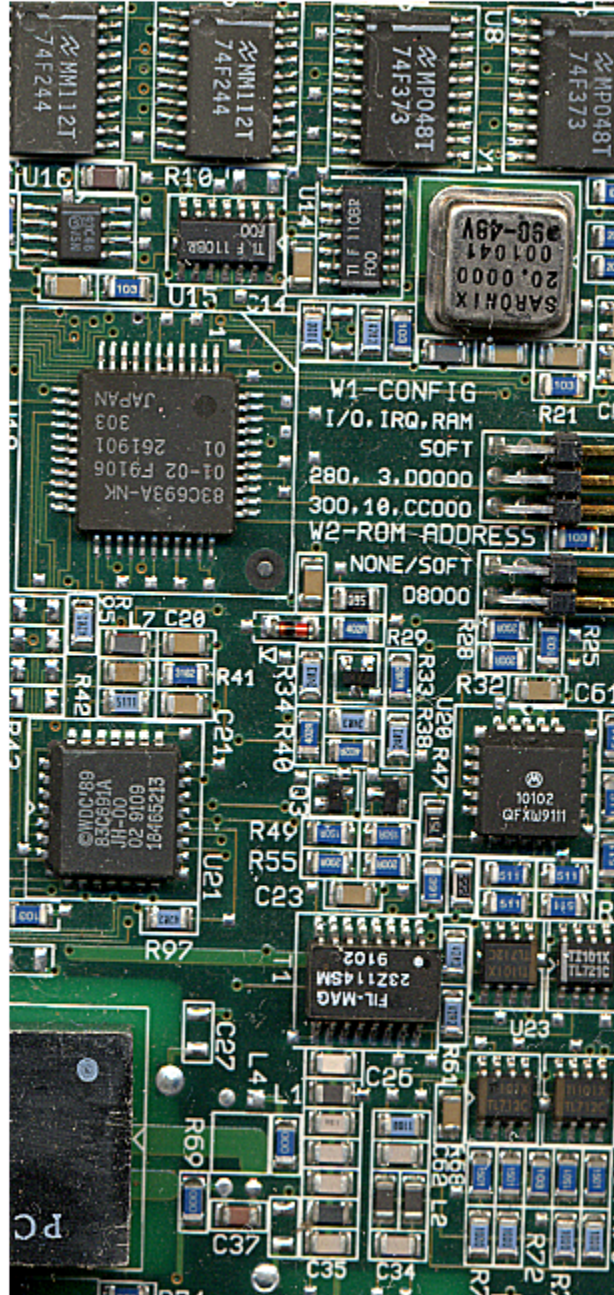
Convert the YIQ image back to RGB color space.

```
RGB2 = ntsc2rgb(YIQ);
```

Display the image that was converted from YIQ to RGB color space.

```
figure  
imshow(RGB2)  
title('Image Converted from YIQ to RGB Color Space')
```

Image Converted from YIQ to RGB Color Space



See Also

[rgb2ntsc](#) | [rgb2ind](#) | [ind2rgb](#) | [ind2gray](#)

Introduced before R2006a

offsetstrel class

Morphological offset structuring element

Description

An `offsetstrel` object represents a nonflat morphological *structuring element*, which is an essential part of morphological dilation and erosion operations. (To create a flat structuring element, use `strel`.)

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.

You can only use `offsetstrel` objects with grayscale images.

Construction

`SE = offsetstrel('ball',R,H)` creates a nonflat, ball-shaped structuring element whose radius in the X-Y plane is `R` and whose maximum offset height is `H`. For improved performance, `offsetstrel` approximates this shape by a sequence of eight nonflat line-shaped structuring elements.

`SE = offsetstrel('ball',R,H,N)` creates a nonflat ball-shaped structuring element, where `N` specifies the number of nonflat, line-shaped structuring elements that `offsetstrel` uses to approximate the shape. Morphological operations using ball approximations run much faster when you specify a value for `N` greater than 0.

`SE = offsetstrel(offset)` creates a nonflat structuring element with the additive offset specified in the matrix `offset`.

Input Arguments

R — Radius of the ball-shaped structuring element
nonnegative integer

Radius of the ball-shaped structuring element in the X-Y plane, specified as a nonnegative integer.

Data Types: double

H — Maximum offset height

real scalar

Maximum offset height, specified as a real scalar.

Data Types: double

N — Number of nonflat line-shaped structuring elements used to approximate the shape

8 (default) | even number

Number of nonflat line-shaped structuring elements used to approximate the shape, specified as a nonnegative integer.

Value of N	Behavior
N > 0	offsetstrel uses a sequence of N nonflat, line-shaped structuring elements to approximate the shape. N must be an even number.
N = 0	offsetstrel does not use any approximation. The structuring element members comprise all pixels whose centers are no greater than R away from the origin. The corresponding height values are determined from the formula of the ellipsoid specified by R and H.

Data Types: double

offset — Values to be added to each pixel location in the neighborhood

matrix

Values to be added to each pixel location in the neighborhood when performing the morphological operation, specified as a matrix. Values that are -Inf are not considered in the computation.

Data Types: double

Properties

Offset — Structuring element neighborhood with offsets

double matrix

Structuring element neighborhood with offsets, specified as a double matrix.

Data Types: logical

Dimensionality — Dimensions of structuring element

nonnegative scalar

Dimensions of structuring element, specified as a nonnegative scalar of class double.

Data Types: double

Methods

decompose	Return sequence of decomposed structuring elements
reflect	Reflect offset structuring element
translate	Translate structuring element

Examples

Create Ball-shaped Structuring Element

Create a ball-shaped structuring element.

```
SE = offsetstrel('ball',5, 6)
```

```
SE =
```

```
offsetstrel is a ball shaped offset structuring element with properties:
```

```
    Offset: [11×11 double]  
 Dimensionality: 2
```

View the structuring element.

SE.Offset

ans =

-Inf	-Inf	0	0.7498	1.4996	2.2494	1.4996	0.7498		
-Inf	0.7498	1.4996	2.2494	2.9992	2.9992	2.9992	2.2494	1.4	
0	1.4996	2.2494	2.9992	3.7491	3.7491	3.7491	2.9992	2.2	
0.7498	2.2494	2.9992	3.7491	4.4989	4.4989	4.4989	3.7491	2.9	
1.4996	2.9992	3.7491	4.4989	5.2487	5.2487	5.2487	4.4989	3.7	
2.2494	2.9992	3.7491	4.4989	5.2487	5.9985	5.2487	4.4989	3.7	
1.4996	2.9992	3.7491	4.4989	5.2487	5.2487	5.2487	4.4989	3.7	
0.7498	2.2494	2.9992	3.7491	4.4989	4.4989	4.4989	3.7491	2.9	
0	1.4996	2.2494	2.9992	3.7491	3.7491	3.7491	2.9992	2.2	
-Inf	0.7498	1.4996	2.2494	2.9992	2.9992	2.9992	2.2494	1.4	

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- The 'ball' input argument and all other input arguments must be compile-time constants.
- The methods associated with `offsetstrel` objects are not supported in code generation.

See Also

See Also

`strel`

Topics

“Structuring Elements”

Property Attributes (MATLAB)

Introduced in R2016a

decompose

Class: `offsetstrel`

Return sequence of decomposed structuring elements

Syntax

`SEQ = decompose(SE)`

Description

`SEQ = decompose(SE)` returns an array of structuring elements, `SEQ`, that are the decomposition of the input structuring element `SE`. `SEQ` is equivalent to `SE`, but the elements of `SEQ` cannot be decomposed further.

Input Arguments

SE — Offset structuring element

`offsetstrel` object

Offset structuring element, specified as an `offsetstrel` object.

Output Arguments

SEQ — Sequence of structuring elements that approximate an `offsetstrel` object

array of `offsetstrel` objects

Sequence of structuring elements that approximate an offset structuring element, returned as an array of `offsetstrel` objects.

Examples

Extract Decomposition of Structuring Element

Create a ball-shaped structuring element.

```
se = offsetstrel('ball',5, 6.5)
```

```
se =  
offsetstrel is a ball shaped offset structuring element with properties:
```

```
    Offset: [11×11 double]  
Dimensionality: 2
```

Obtain the decomposition of the structuring element.

```
seq = decompose(se)
```

```
seq =  
1×8 offsetstrel array with properties:
```

```
    Offset  
Dimensionality
```

See Also

Introduced in R2016a

reflect

Class: `offsetstrel`

Reflect offset structuring element

Syntax

```
SE2 = reflect(SE)
```

Description

`SE2 = reflect(SE)` reflects the structuring element (or structuring elements) specified by `SE`. This method reflects the structuring element through its center. The effect is the same as if you rotated the structuring element's domain 180 degrees around its center (for a 2-D structuring element).

Input Arguments

SE — **Offset structuring element**

`offsetstrel` object or array of objects

Offset structuring element, specified as an `offsetstrel` object or array of objects. If `SE` is an array of structuring element objects, then `reflect(SE)` reflects each element of `SE`, and `SE2` has the same size as `SE`.

Output Arguments

SE2 — **Reflected structuring element or structuring elements**

`offsetstrel` object or array of `offsetstrel` objects

Reflected structuring element or structuring elements, returned as an `offsetstrel` object or array of `offsetstrel` objects.

Examples

Reflect Offset Structuring Element

Create offset strel structuring element.

```
se = offsetstrel('ball', 5, 6.5)
```

```
se =  
offsetstrel is a ball shaped offset structuring element with properties:
```

```
    Offset: [11×11 double]  
 Dimensionality: 2
```

Reflect the structuring element.

```
se2 = se.reflect()
```

```
se2 =  
offsetstrel is a ball shaped offset structuring element with properties:
```

```
    Offset: [11×11 double]  
 Dimensionality: 2
```

See Also

Introduced in R2016a

translate

Class: `offsetstrel`

Translate structuring element

Syntax

`SE2 = translate(SE,V)`

Description

`SE2 = translate(SE,V)` translates an offset structuring element `SE` in N-D space. `V` is an N-element vector containing the offsets of the desired translation in each dimension.

Input Arguments

SE — Offset structuring element

`offsetstrel` object

Offset structuring element, specified as an `offsetstrel` object.

V — Translation offsets

numeric vector

Translation offsets, specified as a numeric vector.

Output Arguments

SE2 — Translated structuring element

`offsetstrel` object

Translated structuring element, returned as an `offsetstrel` object.

Examples

Translate Offset Structuring Element

Create an offset structuring element.

```
SE = offsetstrel('ball', 5, 6.5)
```

```
SE =
```

offsetstrel is a ball shaped offset structuring element with properties:

```
        Offset: [11×11 double]
Dimensionality: 2
```

```
SE.Offset
```

```
ans =
```

```

-Inf    -Inf         0    0.8123    1.6246    2.4369    1.6246    0.8123
-Inf    0.8123    1.6246    2.4369    3.2492    3.2492    3.2492    2.4369    1.6246
      0    1.6246    2.4369    3.2492    4.0615    4.0615    4.0615    3.2492    2.4369
0.8123    2.4369    3.2492    4.0615    4.8738    4.8738    4.8738    4.0615    3.2492
1.6246    3.2492    4.0615    4.8738    5.6861    5.6861    5.6861    4.8738    4.0615
2.4369    3.2492    4.0615    4.8738    5.6861    6.4984    5.6861    4.8738    4.0615
1.6246    3.2492    4.0615    4.8738    5.6861    5.6861    5.6861    4.8738    4.0615
0.8123    2.4369    3.2492    4.0615    4.8738    4.8738    4.8738    4.0615    3.2492
      0    1.6246    2.4369    3.2492    4.0615    4.0615    4.0615    3.2492    2.4369
-Inf    0.8123    1.6246    2.4369    3.2492    3.2492    3.2492    2.4369    1.6246
```

Translate the structuring element.

```
V = [2 2];
```

```
SE2 = translate(SE,V)
```

```
SE2 =
```

offsetstrel is a ball shaped offset structuring element with properties:

```
        Offset: [15×15 double]
Dimensionality: 2
```

```
SE2.Offset
```

```
ans =
```

-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	-Inf
-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	-Inf
-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	-Inf
-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	-Inf
-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	0	0.8123	1.6246
-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	0.8123	1.6246	2.4369
-Inf	-Inf	-Inf	-Inf	-Inf	0	1.6246	2.4369	3.2492
-Inf	-Inf	-Inf	-Inf	-Inf	0.8123	2.4369	3.2492	4.0615
-Inf	-Inf	-Inf	-Inf	-Inf	1.6246	3.2492	4.0615	4.8738
-Inf	-Inf	-Inf	-Inf	-Inf	2.4369	3.2492	4.0615	4.8738

See Also

Introduced in R2016a

openrset

Open R-Set file

Syntax

```
openrset(filename)
```

Description

`openrset(filename)` opens the reduced resolution dataset (R-Set) specified by *filename* for viewing.

See Also

`imtool` | `rsetwrite`

Introduced in R2010a

ordfilt2

2-D order-statistic filtering

Syntax

```
B = ordfilt2(A,order,domain)
B = ordfilt2(A,order,domain,S)
B = ordfilt2( ____,padopt)
```

Description

`B = ordfilt2(A,order,domain)` replaces each element in `A` by the `order`th element in the sorted set of neighbors specified by the nonzero elements in `domain`.

`B = ordfilt2(A,order,domain,S)` filters `A`, where `ordfilt2` uses the values of `S` corresponding to the nonzero values of `domain` as additive offsets. You can use this syntax to implement grayscale morphological operations, including grayscale dilation and erosion.

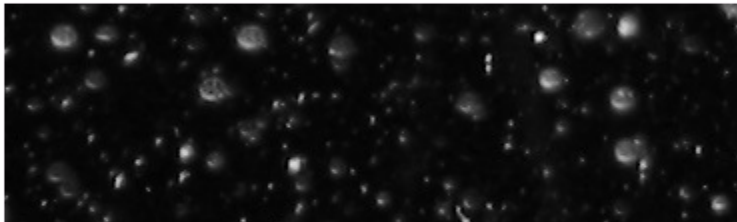
`B = ordfilt2(____,padopt)` filters `A`, where `padopt` specifies how `ordfilt2` pads the matrix boundaries.

Examples

Filter Image with Maximum Filter

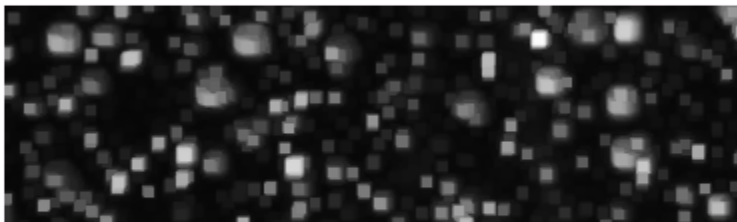
Read image into workspace and display it.

```
A = imread('snowflakes.png');
figure
imshow(A)
```



Filter the image and display the result.

```
B = ordfilt2(A,25,true(5));  
figure  
imshow(B)
```



Input Arguments

A — Input matrix

2-D, real, nonsparse, numeric or logical matrix

Input matrix, specified as a 2-D, real, nonsparse, numeric or logical array.

Example: `A = imread('snowflakes.png');`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

order — Element to replace the target pixel

real scalar integer

Element to replace the target pixel, specified as a real scalar integer of class `double`.

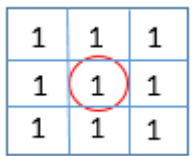
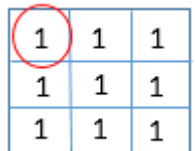
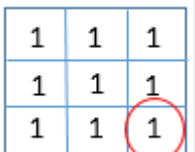
Example: `B = ordfilt2(A,25,true(5));`

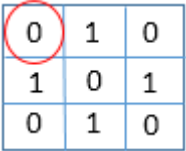
Data Types: `double`

domain — Neighborhood

numeric or logical matrix

Neighborhood, specified as a numeric or logical matrix, containing 1's and 0's. `domain` is equivalent to the structuring element used for binary image operations. The 1-valued elements define the neighborhood for the filtering operation. The following table gives examples of some common filters.

Type of Filtering Operation	MATLAB code	Neighborhood
Median filter	<code>B = ordfilt2(A,5,ones(3,3))</code>	
Minimum filter	<code>B = ordfilt2(A,1,ones(3,3))</code>	
Maximum filter	<code>B = ordfilt2(A,9,ones(3,3))</code>	

Type of Filtering Operation	MATLAB code	Neighborhood
Minimum of north, east, south, and west neighbors	<code>B = ordfilt2(A,1,[0 1 0; 1 0 1; 0 1 0])</code>	 <pre> 0 1 0 1 0 1 0 1 0 </pre>

Example: `B = ordfilt2(A,25,true(5));`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

S — Additive offsets

matrix

Additive offsets, specified as a matrix the same size as `domain`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

padopt — Padding option

'zeros' (default) | 'symmetric'

Padding option, specified as either of the following values:

Option	Description
'zeros'	Pad array boundaries with 0's.
'symmetric'	Pad array with mirror reflections of itself.

Data Types: `char`

Output Arguments

B — Output image

2-D array

Output image, returned as a 2-D array of the same class as the input image `A`.

Tips

- When working with large domain matrices that do not contain any zero-valued elements, `ordfilt2` can achieve higher performance if **A** is in an integer data format (`uint8`, `int8`, `uint16`, `int16`). The gain in speed is larger for `uint8` and `int8` than for the 16-bit data types. For 8-bit data formats, the domain matrix must contain seven or more rows. For 16-bit data formats, the domain matrix must contain three or more rows and 520 or more elements.

References

- [1] Haralick, Robert M., and Linda G. Shapiro, *Computer and Robot Vision*, Volume I, Addison-Wesley, 1992.
- [2] Huang, T.S., G.J.Yang, and G.Y.Tang. "A fast two-dimensional median filtering algorithm.", *IEEE transactions on Acoustics, Speech and Signal Processing*, Vol ASSP 27, No. 1, February 1979

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- When generating code, the `padopt` argument must be a compile-time constant.

See Also

See Also

`medfilt2`

Introduced before R2006a

otf2psf

Convert optical transfer function to point-spread function

Syntax

```
PSF = otf2psf(OTF)
PSF = otf2psf(OTF, OUTSIZE)
```

Description

`PSF = otf2psf(OTF)` computes the inverse Fast Fourier Transform (IFFT) of the optical transfer function (OTF) array and creates a point-spread function (PSF), centered at the origin. By default, the PSF is the same size as the OTF.

`PSF = otf2psf(OTF, OUTSIZE)` converts the OTF array into a PSF array, where `OUTSIZE` specifies the size of the output point-spread function. The size of the output array must not exceed the size of the OTF array in any dimension.

To center the PSF at the origin, `otf2psf` circularly shifts the values of the output array down (or to the right) until the (1,1) element reaches the central position, then it crops the result to match dimensions specified by `OUTSIZE`.

Note that this function is used in image convolution/deconvolution when the operations involve the FFT.

Class Support

OTF can be any nonsparse, numeric array. PSF is of class `double`.

Examples

Convert OTF to PSF

Create a point-spread function (PSF).

```
PSF = fspecial('gaussian',13,1);
```

Convert the PSF to an Optical Transfer Function (OTF).

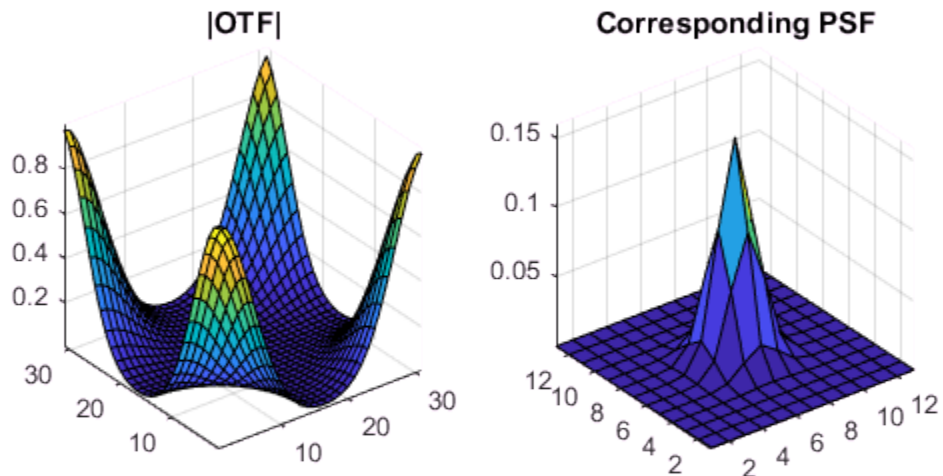
```
OTF = psf2otf(PSF,[31 31]);
```

Convert the OTF back to a PSF.

```
PSF2 = otf2psf(OTF,size(PSF));
```

Plot the PSF and the OTF.

```
subplot(1,2,1)
surf(abs(OTF))
title('|OTF|');
axis square
axis tight
subplot(1,2,2)
surf(PSF2)
title('Corresponding PSF');
axis square
axis tight
```

See Also

[psf2otf](#) | [circshift](#) | [padarray](#)

Introduced before R2006a

otsuthresh

Global histogram threshold using Otsu's method

Syntax

```
T = otsuthresh(counts)
[T,EM] = otsuthresh(counts)
```

Description

`T = otsuthresh(counts)` computes a global threshold `T` from histogram counts, `counts`, using Otsu's method. `T` is a normalized intensity value that lies in the range `[0, 1]` that can be used with `imbinarize` to convert an intensity image to a binary image. Otsu's method chooses a threshold that minimizes the intraclass variance of the thresholded black and white pixels.

`[T,EM] = otsuthresh(counts)` returns the effectiveness metric, `EM`, which indicates the effectiveness of the thresholding. `EM` is in the range `[0, 1]`.

Examples

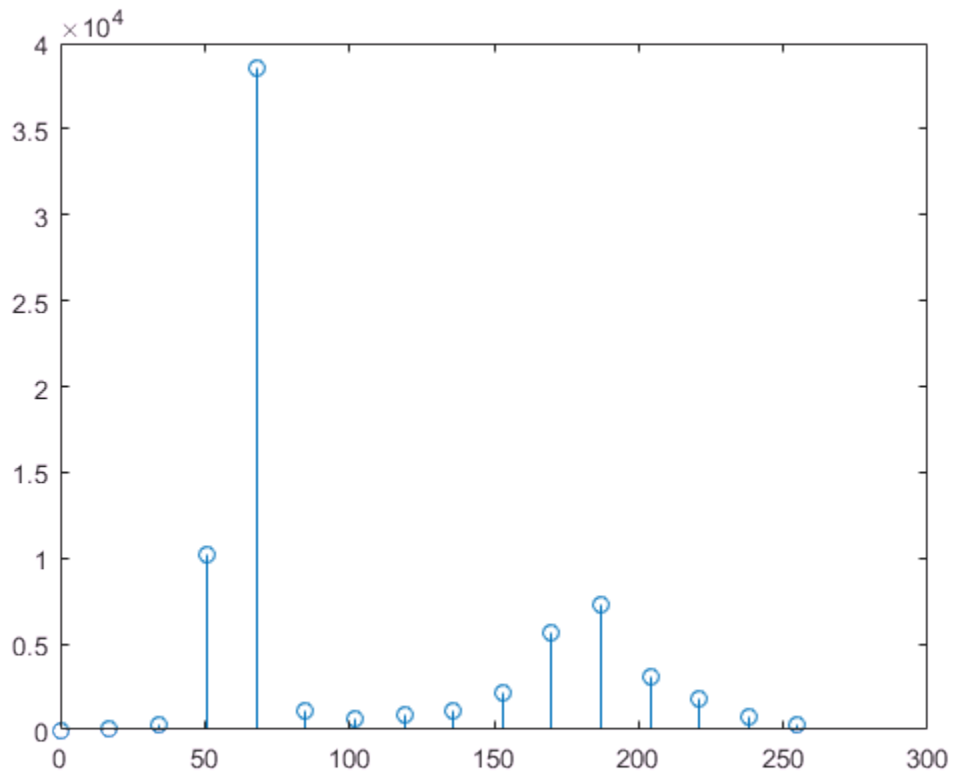
Compute Threshold from Image Histogram and Binarize Image

Read image into the workspace.

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

Calculate a 16-bin histogram for the image.

```
[counts,x] = imhist(I,16);
stem(x,counts)
```

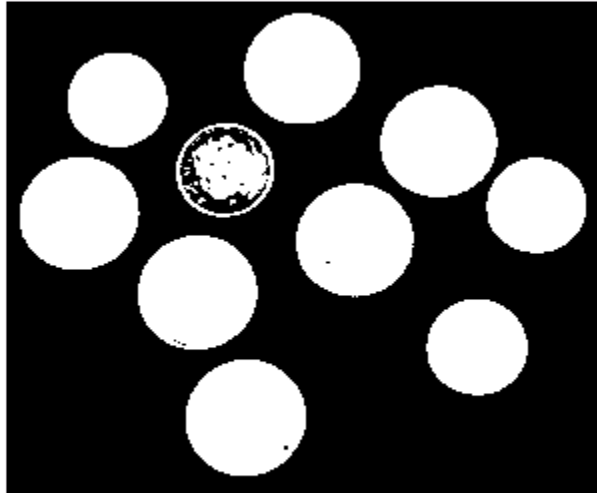


Compute a global threshold using the histogram counts.

```
T = otsuthresh(counts);
```

Create a binary image using the computed threshold and display the image.

```
BW = imbinarize(I,T);  
figure  
imshow(BW)
```



Input Arguments

counts — Histogram counts

real, nonsparse numeric vector of nonnegative values

Histogram counts, specified as a real, nonsparse numeric vector of nonnegative values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

T — Global threshold value

numeric scalar

Global threshold value, returned as a numeric scalar of class `double` in the range [0, 1].

EM — Effectiveness metric

numeric scalar

Effectiveness metric, returned as a numeric scalar of class `double` in the range [0, 1]. The lower bound is attainable only by histogram counts with all data in a single non-zero bin. The upper bound is attainable only by histogram counts with two non-zero bins.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.

See Also

See Also

`adaptthresh` | `graythresh` | `imbinarize`

Introduced in R2016a

padarray

Pad array

Syntax

```
B = padarray(A,padsizel)
B = padarray(A,padsizel,padval)
B = padarray(A,padsizel,padval,direction)
gpuarrayB = padarray(gpuarrayA, ___)
```

Description

`B = padarray(A,padsizel)` pads array `A` with 0's (zeros). `padsizel` is a vector of nonnegative integers that specifies both the amount of padding to add and the dimension along which to add it. The value of an element in the vector specifies the amount of padding to add. The order of the element in the vector specifies the dimension along which to add the padding.

For example, a `padsizel` value of `[2 3]` means add 2 elements of padding along the first dimension and 3 elements of padding along the second dimension. By default, `padarray` adds padding before the first element and after the last element along the specified dimension.

`B = padarray(A,padsizel,padval)` pads array `A` where `padval` specifies the value to use as the pad value. `padarray` uses the value 0 (zero) as the default. `padval` can be a scalar that specifies the pad value directly or one of the following values that specifies the method `padarray` uses to determine the values of the elements added as padding.

Value	Meaning
'circular'	Pad with circular repetition of elements within the dimension.
'replicate'	Pad by repeating border elements of array.
'symmetric'	Pad array with mirror reflections of itself.

`B = padarray(A,padsizel,padval,direction)` pads `A` in the direction specified by the `direction`. `direction` can be one of the following values.

Value	Meaning
'both'	Pads before the first element and after the last array element along each dimension. This is the default.
'post'	Pad after the last array element along each dimension.
'pre'	Pad before the first array element along each dimension.

`gpuarrayB = padarray(gpuarrayA, ___)` performs the padding operation on a GPU, where `gpuarrayA` is a `gpuArray` object that contains the image `A`. The return value `gpuarrayB` is also a `gpuArray`. This syntax requires the Parallel Computing Toolbox.

Class Support

When padding with a constant value, `A` can be numeric or logical. When padding using the `'circular'`, `'replicate'`, or `'symmetric'` methods, `A` can be of any class. `B` is of the same class as `A`.

Examples

Pad Arrays in Various Ways

Add three elements of padding to the beginning of a vector. The padding elements contain mirror copies of the array elements.

```
a = [ 1 2 3 4 ]
```

```
a =
```

```
    1    2    3    4
```

```
b = padarray(a,[0 3], 'symmetric', 'pre')
```

```
b =
```

```
    3    2    1    1    2    3    4
```

Add three elements of padding to the end of the first dimension of the array and two elements of padding to the end of the second dimension. The example uses the value of the last array element as the padding value.

```
A = [1 2; 3 4]
```

```
A =
```

```
    1    2
    3    4
```

```
B = padarray(A,[3 2], 'replicate', 'post')
```

```
B =
```

```
    1    2    2    2
    3    4    4    4
    3    4    4    4
    3    4    4    4
    3    4    4    4
```

Add three elements of padding to the vertical and horizontal dimensions of a three-dimensional array. Use default values for the pad value and direction.

```
A = [ 1 2; 3 4]
```

```
A =
```

```
    1    2
    3    4
```

```
B = [ 5 6; 7 8]
```

```
B =
```

```
    5    6
    7    8
```

```
C = cat(3,A,B)
```

```
C =
```

```
(:,: ,1) =
```

```
    1    2
```



```

    3    4

(:, :, 2) =
    5    6
    7    8

D = padarray(C, [3 3])
D =
(:, :, 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    1    2    0    0    0
    0    0    0    3    4    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0

(:, :, 2) =
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    5    6    0    0    0
    0    0    0    7    8    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0

```

Perform Padding on a GPU

Add padding on all sides of an image.

```

gcam = gpuArray(imread('cameraman.tif'));
padcam = padarray(gcam, [50 50], 'both');
imshow(padcam)

```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, `padarray` supports only up to 3-D inputs, and the input arguments, `padval` and `direction` must be compile-time constants.

See Also

`circshift` | `gpuArray` | `imfilter`

Introduced before R2006a

para2fan

Convert parallel-beam projections to fan-beam

Syntax

```
F = para2fan(P, D)
I = para2fan(..., param1, val1, param2, val2, ...)
[F, fan_positions, fan_rotation_angles] = fan2para(...)
```

Description

`F = para2fan(P, D)` converts the parallel-beam data `P` to the fan-beam data `F`. Each column of `P` contains the parallel-beam sensor samples at one rotation angle. `D` is the distance in pixels from the fan-beam vertex to the center of rotation that was used to obtain the projections.

The sensors are assumed to have a one-pixel spacing. The parallel-beam rotation angles are assumed to be spaced equally to cover $[0,180]$ degrees. The calculated fan-beam rotation angles cover $[0,360)$ with the same spacing as the parallel-beam rotation angles. The calculated fan-beam angles are equally spaced with the spacing set to the smallest angle implied by the sensor spacing.

`I = para2fan(..., param1, val1, param2, val2, ...)` specifies parameters that control various aspects of the `para2fan` conversion. Parameter names can be abbreviated, and case does not matter. Default values are enclosed in braces like this: `{default}`. Parameters include

Parameter	Description
'FanCoverage'	Range of rotation angles used to calculate the projection data. Possible values: {'cycle'} or 'minimal' See <code>ifanbeam</code> for details.
'FanRotationIncrement'	Positive real scalar specifying the rotation angle increment of the fan-beam projections in degrees.

Parameter	Description
	<p>If 'FanCoverage' is 'cycle', 'FanRotationIncrement' must be a factor of 360.</p> <p>If 'FanRotationIncrement' is not specified, then it is set to the same spacing as the parallel-beam rotation angles.</p>
'FanSensorGeometry'	<p>Positioning of sensors, specified as either of the following values:</p> <p>'arc' or 'line'</p> <p>See fanbeam for details.</p>
'FanSensorSpacing'	<p>Positive real scalar specifying the spacing of the fan beams. Interpretation of the value depends on the setting of 'FanSensorGeometry':</p> <p>If 'FanSensorGeometry' is 'arc', the value defines the angular spacing in degrees. Default value is 1.</p> <p>If 'FanSensorGeometry' is 'line', the value defines the linear spacing in pixels.</p> <p>If 'FanSensorSpacing' is not specified, the default is the smallest value implied by 'ParallelSensorSpacing' such that</p> <p>If 'FanSensorGeometry' is 'arc', 'FanSensorSpacing' is</p> $180/\text{PI} * \text{ASIN}(\text{PSPACE}/D)$ <p>where PSPACE is the value of 'ParallelSensorSpacing'.</p> <p>If 'FanSensorGeometry' is 'line', 'FanSensorSpacing' is</p> $D * \text{ASIN}(\text{PSPACE}/D)$

Parameter	Description
'Interpolation'	Type of interpolation used between the parallel-beam and fan-beam data, specified as one of the following values: 'nearest' — Nearest-neighbor 'linear' — Linear (the default) 'spline' — Piecewise cubic spline 'pchip' — Piecewise cubic Hermite (PCHIP) 'v5cubic' — The cubic interpolation from MATLAB 5
'ParallelCoverage'	Range of rotation, specified as one of the following values: 'cycle' -- Parallel data covers 360 degrees 'halfcycle' — Parallel data covers 180 degrees (the default)
'ParallelSensorSpacing'	Positive real scalar specifying the spacing of the parallel-beam sensors in pixels. The range of sensor locations is implied by the range of fan angles and is given by $[D*\sin(\min(\text{FAN_ANGLES})), D*\sin(\max(\text{FAN_ANGLES}))]$ If 'ParallelSensorSpacing' is not specified, the spacing is assumed to be uniform and is set to the minimum spacing implied by the fan angles and sampled over the range implied by the fan angles.

`[F, fan_positions, fan_rotation_angles] = fan2para(...)` returns the fan-beam sensor measurement *angles* in `fan_positions`, if 'FanSensorGeometry' is 'arc'. If 'FanSensorGeometry' is 'line', `fan_positions` contains the fan-beam sensor *positions* along the line of sensors. `fan_rotation_angles` contains rotation angles.

Class Support

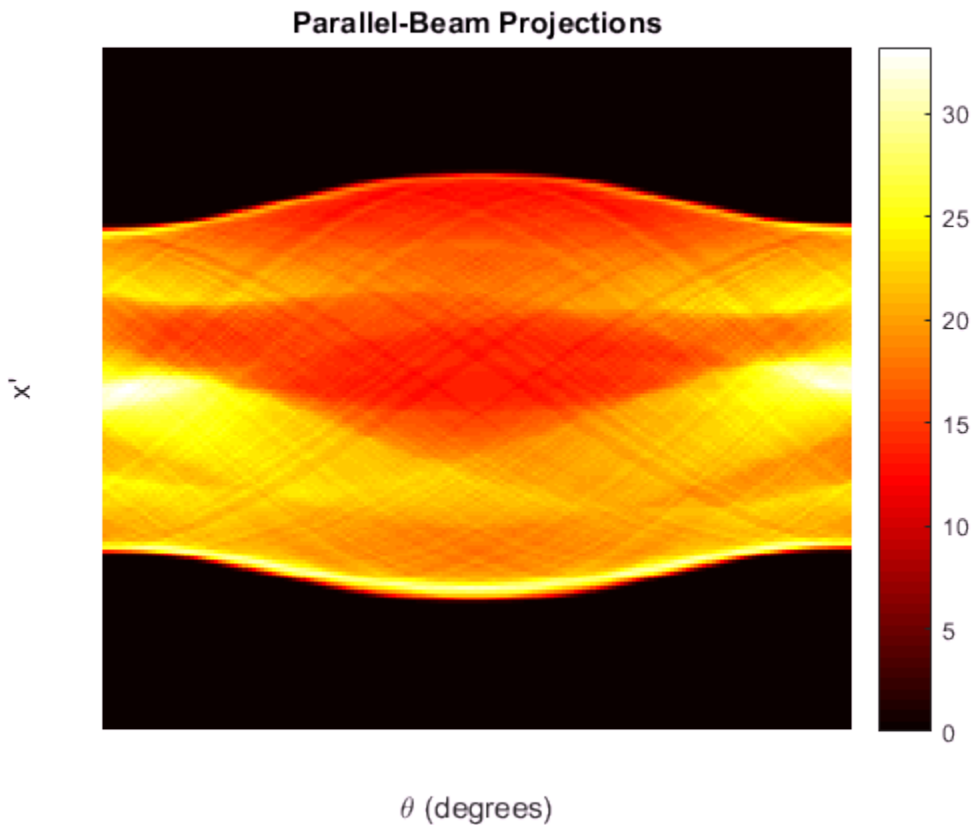
P and D can be `double` or `single`, and must be nonsparse. The other numeric input arguments must be `double`. The output arguments are `double`.

Examples

Convert Parallel-beam Projections to Fan-beam Projections

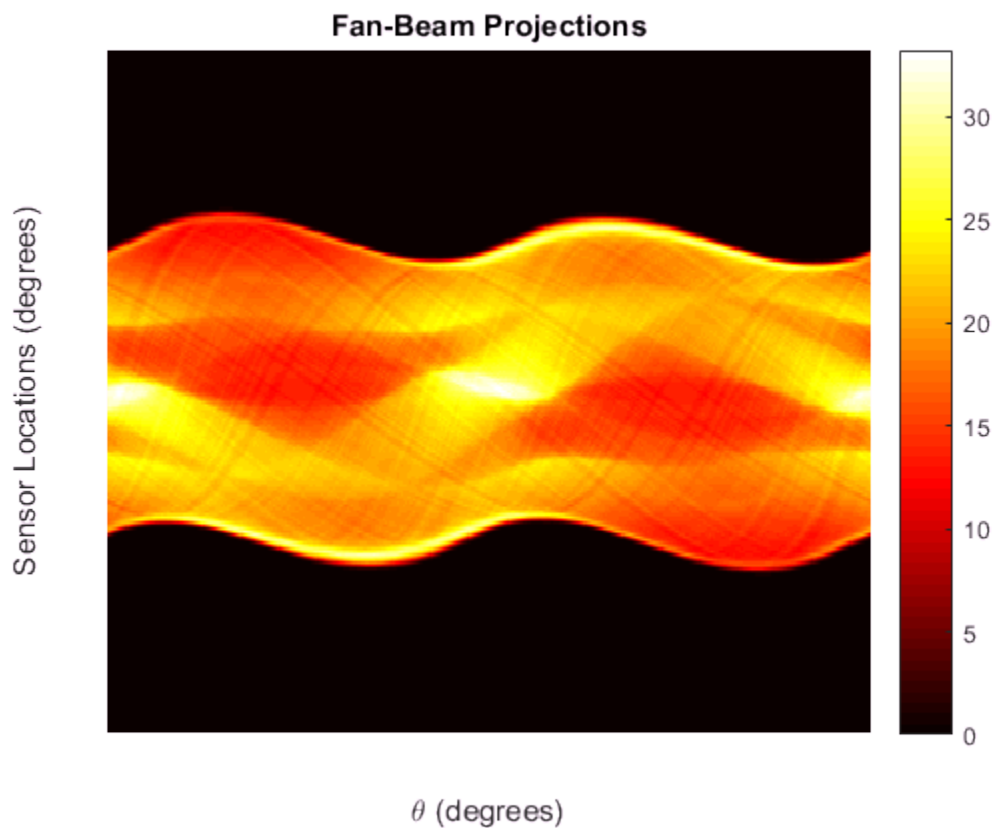
Generate parallel-beam projections

```
ph = phantom(128);
theta = 0:180;
[P, xp] = radon(ph, theta);
imshow(P, [], 'XData', theta, 'YData', xp, 'InitialMagnification', 'fit')
axis normal
title('Parallel-Beam Projections')
xlabel('\theta (degrees)')
ylabel('x')
colormap(gca, hot), colorbar
```



Convert to fan-beam projections

```
[F,Fpos,Fangles] = para2fan(P,100);
figure
imshow(F,[],'XData',Fangles,'YData',Fpos,'InitialMagnification','fit')
axis normal
title('Fan-Beam Projections')
xlabel('\theta (degrees)')
ylabel('Sensor Locations (degrees)')
colormap(gca,hot), colorbar
```

See Also

fan2para | iradon | fanbeam | ifanbeam | phantom | radon

Introduced before R2006a

phantom

Create head phantom image

Syntax

```
P = phantom(def, n)
P = phantom(E, n)
[P, E] = phantom(...)
```

Description

`P = phantom(def, n)` generates an image of a head phantom that can be used to test the numerical accuracy of `radon` and `iradon` or other two-dimensional reconstruction algorithms. `P` is a grayscale intensity image that consists of one large ellipse (representing the brain) containing several smaller ellipses (representing features in the brain).

`def` specifies the type of head phantom to generate. Valid values are

- 'Shepp-Logan' — Test image used widely by researchers in tomography
- 'Modified Shepp-Logan' (default) — Variant of the Shepp-Logan phantom in which the contrast is improved for better visual perception

`n` is a scalar that specifies the number of rows and columns in `P`. If you omit the argument, `n` defaults to 256.

`P = phantom(E, n)` generates a user-defined phantom, where each row of the matrix `E` specifies an ellipse in the image. `E` has six columns, with each column containing a different parameter for the ellipses. This table describes the columns of the matrix.

Column	Parameter	Meaning
Column 1	A	Additive intensity value of the ellipse
Column 2	a	Length of the horizontal semiaxis of the ellipse

Column	Parameter	Meaning
Column 3	b	Length of the vertical semiaxis of the ellipse
Column 4	x0	<i>x</i> -coordinate of the center of the ellipse
Column 5	y0	<i>y</i> -coordinate of the center of the ellipse
Column 6	phi	Angle (in degrees) between the horizontal semiaxis of the ellipse and the <i>x</i> -axis of the image

For purposes of generating the phantom, the domains for the *x*- and *y*-axes span [-1,1]. Columns 2 through 5 must be specified in terms of this range.

[P, E] = phantom(...) returns the matrix E used to generate the phantom.

Class Support

All inputs and all outputs must be of class `double`.

Examples

Create Modified Shepp-Logan Head Phantom Image

Create the modified Shepp-Logan head phantom image and display it.

```
P = phantom('Modified Shepp-Logan',200);  
imshow(P)
```



Tips

For any given pixel in the output image, the pixel's value is equal to the sum of the additive intensity values of all ellipses that the pixel is a part of. If a pixel is not part of any ellipse, its value is 0.

The additive intensity value A for an ellipse can be positive or negative; if it is negative, the ellipse will be darker than the surrounding pixels. Note that, depending on the values of A , some pixels can have values outside the range $[0,1]$.

References

- [1] Jain, Anil K., *Fundamentals of Digital Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1989, p. 439.

See Also

radon | iradon

Introduced before R2006a

images.geotrans.PiecewiseLinearTransformation2D class

Package: images.geotrans

2-D piecewise linear geometric transformation

Description

A `PiecewiseLinearTransformation2D` object encapsulates a 2-D piecewise linear geometric transformation.

You can create a `PiecewiseLinearTransformation2D` object using the following methods:

- `fitgeotrans` — Returns a `PiecewiseLinearTransformation2D` object that maps control point pairs using a piecewise linear transformation
- The `PiecewiseLinearTransformation2D` class constructor

Construction

`tform = images.geotrans.PiecewiseLinearTransformation2D(movingPoints, fixedPoints)` creates a `PiecewiseLinearTransformation2D` object given m -by-2 matrices `movingPoints` and `fixedPoints`, which define matched control points in the moving and fixed images, respectively. The piecewise linear transformation creates a mapping by breaking up the plane into regions and using a different affine transformation to map control points in each region.

Input Arguments

movingPoints — **x - and y -coordinates of control points in the moving image**
 m -by-2 matrix

x - and y -coordinates of control points in the moving image, specified as an m -by-2 matrix.

Data Types: `double` | `single`

fixedPoints — *x- and y-coordinates of control points in the fixed image**m*-by-2 matrix

x- and *y*-coordinates of control points in the fixed image, specified as an *m*-by-2 matrix.

Data Types: double | single

Properties

Dimensionality — *Describes the dimensionality of the geometric transformation*

2

Describes the dimensionality of the geometric transformation for both input and output points.

Methods

transformPointsInverse
outputLimits

Apply inverse geometric transformation
Find output limits of geometric transformation

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB documentation.

Examples

Fit set of control points related by affine transformation

Fit a piecewise linear transformation to a set of fixed and moving control points that are actually related by a single global affine2d transformation across the domain.

Create a 2D affine transformation.

```
theta = 10;
```

```
tformAffine = affine2d([cosd(theta) -sind(theta) 0; sind(theta) cosd(theta) 0; 0 0 1])
```

```
tformAffine =
```

```
    affine2d with properties:
```

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

Arbitrarily choose 6 pairs of control points.

```
fixedPoints = [10 20; 10 5; 2 3; 0 5; -5 3; -10 -20];
```

Apply forward geometric transformation to map fixed points to obtain effect of fixed and moving points that are related by some geometric transformation.

```
movingPoints = transformPointsForward(tformAffine, fixedPoints)
```

```
movingPoints =
```

```
    13.3210    17.9597
    10.7163     3.1876
     2.4906     2.6071
     0.8682     4.9240
    -4.4031     3.8227
   -13.3210   -17.9597
```

Estimate piecewise linear transformation that maps movingPoints to fixedPoints.

```
tformPiecewiseLinear = images.geotrans.PiecewiseLinearTransformation2D(movingPoints, fixedPoints)
```

```
tformPiecewiseLinear =
```

```
    PiecewiseLinearTransformation2D with properties:
```

```
    Dimensionality: 2
```

Verify the fit of our PiecewiseLinearTransformation2D object at the control points.

```
movingPointsComputed = transformPointsInverse(tformPiecewiseLinear, fixedPoints);
```

```
errorInFit = hypot(movingPointsComputed(:,1)-movingPoints(:,1), ...
                  movingPointsComputed(:,2)-movingPoints(:,2))
```

```
errorInFit =
```



```
1.0e-15 *  
  0  
  0  
0.4441  
  0  
  0  
  0
```

See Also

See Also

Functions

[fitgeotrans](#) | [imwarp](#)

Classes

[affine2d](#) | [projective2d](#) | [images.geotrans.PolynomialTransformation2d](#) | [images.geotrans.LocalWeightedMeanTransformation2d](#)

images.geotrans.PiecewiseLinearTransformation2D.transform

Class: images.geotrans.PiecewiseLinearTransformation2D

Package: images.geotrans

Apply inverse geometric transformation

Syntax

```
[u,v] = transformPointsInverse(tform,x,y)
```

```
U = transformPointsInverse(tform,X)
```

Description

`[u,v] = transformPointsInverse(tform,x,y)` applies the inverse transformation of `tform` to the input 2-D point arrays `x` and `y` and outputs the point arrays `u` and `v`. The input point arrays `x` and `y` must be of the same size.

`U = transformPointsInverse(tform,X)` applies the inverse transformation of `tform` to the input n -by-2 point matrix `X` and outputs the n -by-2 point matrix `U`. `transformPointsForward` maps the point `X(k, :)` to the point `U(k, :)`.

Input Arguments

tform

Geometric transformation, specified as an `images.geotrans.PiecewiseLinearTransformation2D` object.

x

Coordinates in X dimension of points to be transformed, specified as a array.

y

Coordinates in Y dimension of points to be transformed, specified as a array.

X

X and Y coordinates of points to be transformed, specified as an n -by-2 matrix

Output Arguments

u

Transformed coordinates in X dimension, returned as an array.

v

Transformed coordinates in Y dimension, returned as an array.

u

Transformed X and Y coordinates, returned as an n -by-2 matrix

See Also

images.geotrans.PiecewiseLinearTransformation2D.outputLimits

Class: images.geotrans.PiecewiseLinearTransformation2D

Package: images.geotrans

Find output limits of geometric transformation

Syntax

```
[xLimitsOut,yLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn)
[xLimitsOut,yLimitsOut,zLimitsOut] =
outputLimits(tform,xLimitsIn,yLimitsIn,zLimitsIn)
```

Description

[xLimitsOut,yLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn) estimates the output spatial limits corresponding to a given 2D geometric transformation and a set of input spatial limits.

[xLimitsOut,yLimitsOut,zLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn,zLimitsIn) estimates the output spatial limits corresponding to a given 3D geometric transformation and a set of input spatial limits.

Input Arguments

tform

Geometric transformation, specified as an `images.geotrans.PiecewiseLinearTransformation2D` object.

xLimitsIn

Limits along the *X* axes, specified as a two-element vector, such as [].

yLimitsIn

Limits along the *Y* axes, specified as a two-element vector, such as [].

zLimitsIn

Limits along the Z axes, specified as a two-element vector, such as [].

Output Arguments

xLimitsOut

Actual limits along the X dimension, returned as an array.

yLimitsOut

Actual limits along the Y dimension, returned as an array.

zLimitsOut

Actual limits along the Z dimension, returned as an array.

See Also

poly2mask

Convert region of interest (ROI) polygon to region mask

Syntax

```
BW = poly2mask(x, y, m, n)
```

Description

`BW = poly2mask(x, y, m, n)` computes a binary region of interest (ROI) mask, `BW`, from an ROI polygon, represented by the vectors `x` and `y`. The size of `BW` is `m`-by-`n`. `poly2mask` sets pixels in `BW` that are inside the polygon (`X, Y`) to 1 and sets pixels outside the polygon to 0.

`poly2mask` closes the polygon automatically if it isn't already closed.

Note on Rectangular Polygons

When the input polygon goes through the middle of a pixel, sometimes the pixel is determined to be inside the polygon and sometimes it is determined to be outside (see Algorithm for details). To specify a polygon that includes a given rectangular set of pixels, make the edges of the polygon lie along the outside edges of the bounding pixels, instead of the center of the pixels.

For example, to include pixels in columns 4 through 10 and rows 4 through 10, you might specify the polygon vertices like this:

```
x = [4 10 10 4 4];  
y = [4 4 10 10 4];  
mask = poly2mask(x,y,12,12)
```

mask =

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

```

0  0  0  0  1  1  1  1  1  1  0  0
0  0  0  0  1  1  1  1  1  1  0  0
0  0  0  0  1  1  1  1  1  1  0  0
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  0  0  0  0

```

In this example, the polygon goes through the center of the bounding pixels, with the result that only some of the desired bounding pixels are determined to be inside the polygon (the pixels in row 4 and column 4 and not in the polygon). To include these elements in the polygon, use fractional values to specify the outside edge of the 4th row (3.5) and the 10th row (10.5), and the outside edge of the 4th column (3.5) and the outside edge of the 10th column (10.5) as vertices, as in the following example:

```

x = [3.5 10.5 10.5 3.5 3.5];
y = [3.5 3.5 10.5 10.5 3.5];
mask = poly2mask(x,y,12,12)

```

mask =

```

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  1  1  1  1  1  1  1  0  0
0  0  0  1  1  1  1  1  1  1  0  0
0  0  0  1  1  1  1  1  1  1  0  0
0  0  0  1  1  1  1  1  1  1  0  0
0  0  0  1  1  1  1  1  1  1  0  0
0  0  0  1  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  0  0  0  0

```

Class Support

The class of BW is `logical`

Examples

Define Polygon and Create Mask

Specify the x- and y-coordinates of the polygon.

```

x = [63 186 54 190 63];
y = [60 60 209 204 60];

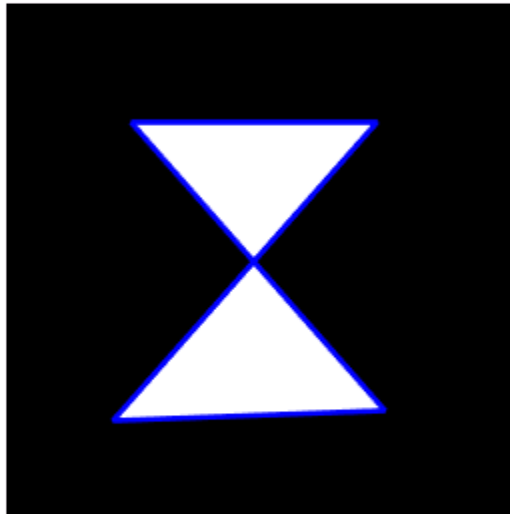
```

Create the mask specifying the size of the image.

```
bw = poly2mask(x,y,256,256);
```

Display the mask, drawing a line around the polygon.

```
imshow(bw)
hold on
plot(x,y,'b','LineWidth',2)
hold off
```



Create Mask Using Random Points to Define Polygon

Define two sets of random points for the x- and y-coordinates.

```
x = 256*rand(1,4);
y = 256*rand(1,4);
x(end+1) = x(1);
```



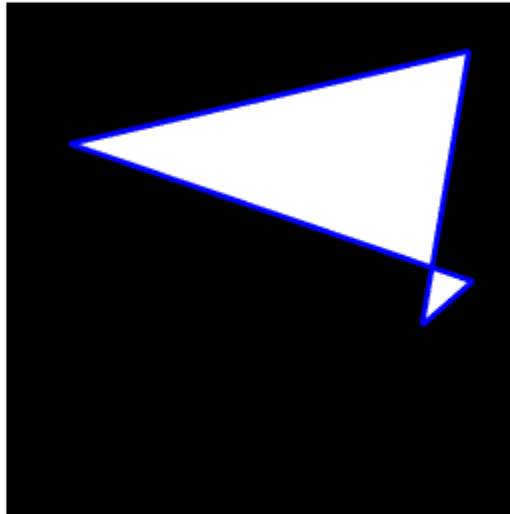
```
y(end+1) = y(1);
```

Create the mask.

```
bw = poly2mask(x,y,256,256);
```

Display the mask and draw a line around the polygon.

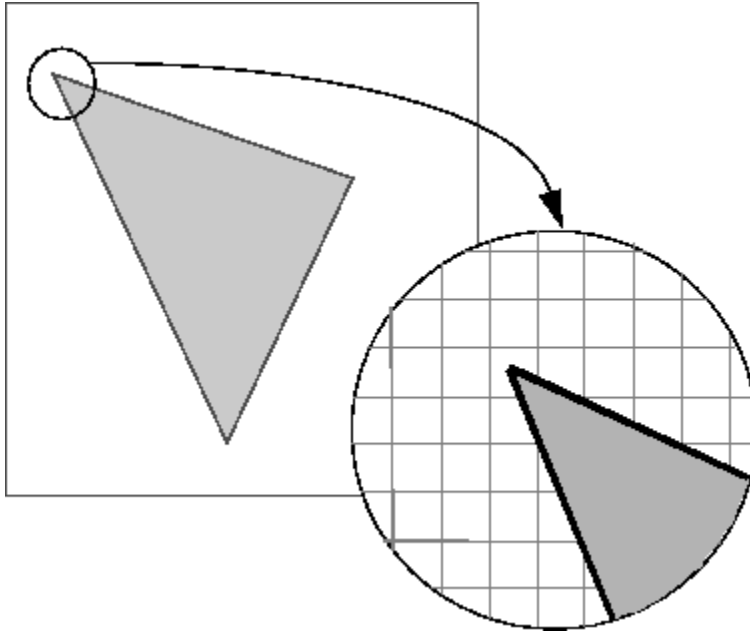
```
imshow(bw)
hold on
plot(x,y,'b','LineWidth',2)
hold off
```



Algorithms

When creating a region of interest (ROI) mask, `poly2mask` must determine which pixels are included in the region. This determination can be difficult when pixels on the edge

of a region are only partially covered by the border line. The following figure illustrates a triangular region of interest, examining in close-up one of the vertices of the ROI. The figure shows how pixels can be partially covered by the border of a region-of-interest.



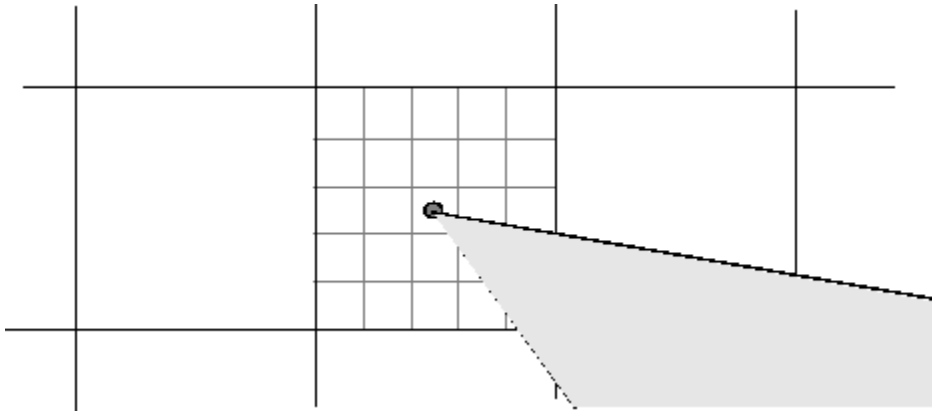
Pixels on the Edge of an ROI Are Only Partially Covered by Border

To determine which pixels are in the region, `poly2mask` uses the following algorithm:

- 1** Divide each pixel (unit square) into a 5-by-5 grid. See “Dividing Pixels into a 5-by-5 Subpixel Grid” on page 1-1597 for an illustration.
- 2** Adjust the position of the vertices to be on the intersections of the subpixel grid. See “Adjusting the Vertices to the Subpixel Grid” on page 1-1597 for an illustration.
- 3** Draw a path from each adjusted vertex to the next, following the edges of the subpixel grid. See “Drawing a Path Between the Adjusted Vertices” on page 1-1598 for an illustration.
- 4** Determine which border pixels are inside the polygon using this rule: if a pixel's central subpixel is inside the boundaries defined by the path between adjusted vertices, the pixel is considered inside the polygon. See “Determining Which Pixels Are in the Region” on page 1-1598 for an illustration.

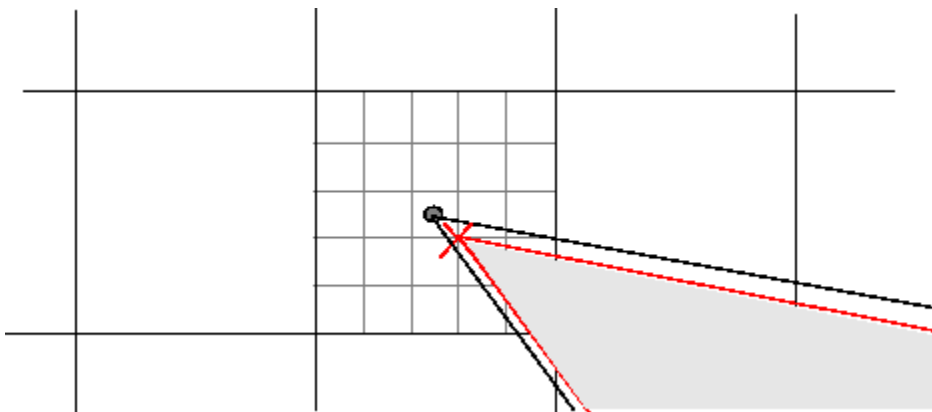
Dividing Pixels into a 5-by-5 Subpixel Grid

The following figure shows the pixel that contains the vertex of the ROI shown previously with this 5-by-5 subpixel grid.



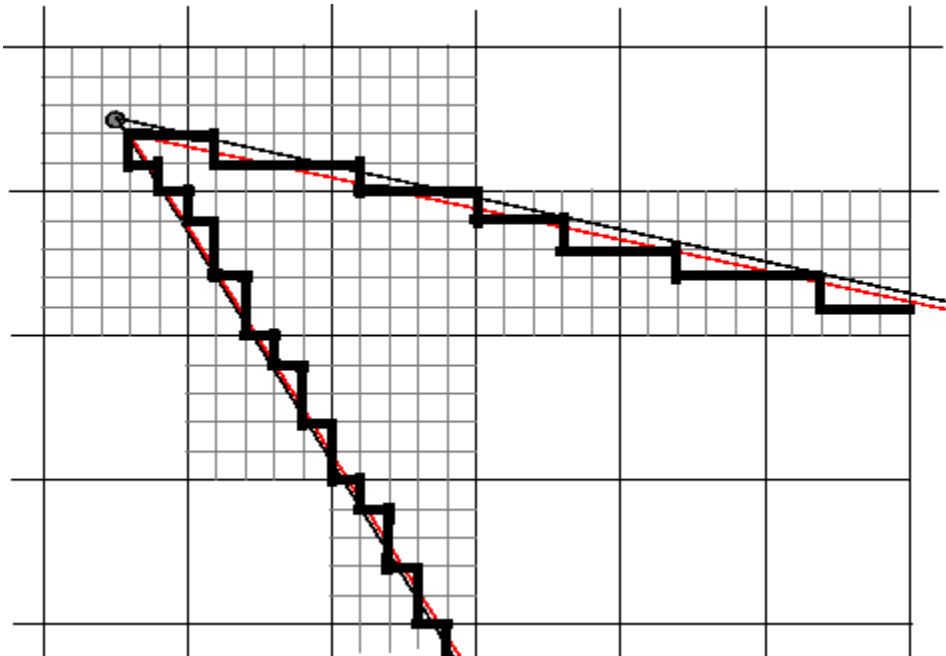
Adjusting the Vertices to the Subpixel Grid

poly2mask adjusts each vertex of the polygon so that the vertex lies on the subpixel grid. Note how poly2mask rounds up x and y coordinates to find the nearest grid corner. This creates a second, modified polygon, slightly smaller, in this case, than the original ROI. A portion is shown in the following figure.



Drawing a Path Between the Adjusted Vertices

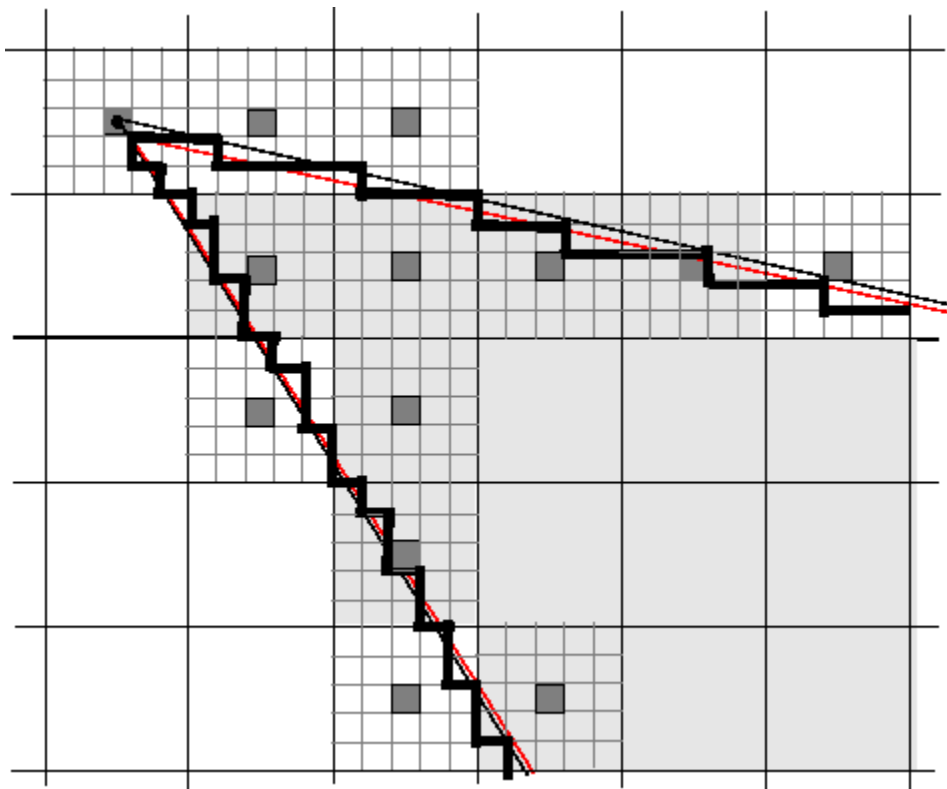
`poly2mask` forms a path from each adjusted vertex to the next, following the edges of the subpixel grid. In the following figure, a portion of this modified polygon is shown by the thick dark lines.



Determining Which Pixels Are in the Region

`poly2mask` uses the following rule to determine which border pixels are inside the polygon: if the pixel's central subpixel is inside the modified polygon, the pixel is inside the region.

In the following figure, the central subpixels of pixels on the ROI border are shaded a dark gray color. Pixels inside the polygon are shaded a lighter gray. Note that the pixel containing the vertex is not part of the ROI because its center pixel is not inside the modified polygon.



See Also

`roipoly`

Introduced before R2006a

images.geotrans.PolynomialTransformation2D class

Package: images.geotrans

2-D polynomial geometric transformation

Description

A `PolynomialTransformation2D` object encapsulates a 2-D polynomial geometric transformation.

You can create a `PolynomialTransformation2D` object using the following methods:

- `fitgeotrans` — Returns a `PolynomialTransformation2D` object that maps control point pairs using a polynomial transformation of degree 2, 3, or 4
- The `PolynomialTransformation2D` class constructors

Construction

`tform = images.geotrans.PolynomialTransformation2D(movingPoints, fixedPoints, degree)` creates a `PolynomialTransformation2D` object given m -by-2 matrices `movingPoints`, and `fixedPoints` which define matched control points in the moving and fixed images, respectively. `degree` is a scalar with value 2, 3, or 4 that specifies the degree of the polynomial that is fit to the control points.

`tform = images.geotrans.PolynomialTransformation2D(a,b)` creates a `PolynomialTransformation2D` object given polynomial coefficient vectors `a` and `b`. `a` is a vector of polynomial coefficients of length n that is used to determine U in the inverse transformation. `b` is a vector of polynomial coefficients of length n that is used to determine V in the inverse transformation. For polynomials of degree 2, 3, and 4, n is 6, 10, and 15, respectively.

Input Arguments

movingPoints — *x- and y-coordinates of control points in the moving image*
m-by-2 matrix

x - and y -coordinates of control points in the moving image, specified as an m -by-2 matrix.

Data Types: double | single

fixedPoints — x - and y -coordinates of control points in the fixed image

m -by-2 matrix

x - and y -coordinates of control points in the fixed image, specified as an m -by-2 matrix.

Data Types: double | single

degree — Degree of the polynomial transformation

2 | 3 | 4

Degree of the polynomial transformation, specified as the scalar values 2, 3, or 4.

a — Polynomial coefficients used to determine U in the inverse transformation

vector of length n

Polynomial coefficients used to determine U in inverse transformation, specified as a vector of length n . For polynomials of degree 2, 3, and 4, n is 6, 10, and 15, respectively. The polynomial coefficient vector a is ordered as follows:

$$U = a(1) + a(2) \cdot X + a(3) \cdot Y + a(4) \cdot X \cdot Y + a(5) \cdot X.^2 + a(6) \cdot Y.^2 + \dots$$

Data Types: double | single

b — Polynomial coefficients used to determine V in inverse transformation

vector of length n

Polynomial coefficients used to determine V in the inverse transformation, specified as a vector of length n . For polynomials of degree 2, 3, and 4, n is 6, 10, and 15, respectively. The polynomial coefficient vector b is ordered as follows:

$$V = b(1) + b(2) \cdot X + b(3) \cdot Y + b(4) \cdot X \cdot Y + b(5) \cdot X.^2 + b(6) \cdot Y.^2 + \dots$$

Data Types: double | single

Properties

A — Polynomial coefficients used to determine U in the inverse transformation

vector of length n

Polynomial coefficients used to determine U in the inverse transformation, specified as a vector of length n . For polynomials of degree 2, 3, and 4, n is 6, 10, and 15, respectively. The polynomial coefficient vector A is ordered as follows:

$$U = A(1) + A(2) \cdot X + A(3) \cdot Y + A(4) \cdot X \cdot Y + A(5) \cdot X.^2 + A(6) \cdot Y.^2 + \dots$$

Data Types: `double` | `single`

B — Polynomial coefficients used to determine V in the inverse transformation

vector of length n

Polynomial coefficients used to determine V in the inverse transformation, specified as a vector of length n . For polynomials of degree 2, 3, and 4, n is 6, 10, and 15, respectively. The polynomial coefficient vector B is ordered as follows:

$$V = B(1) + B(2) \cdot X + B(3) \cdot Y + B(4) \cdot X \cdot Y + B(5) \cdot X.^2 + B(6) \cdot Y.^2 + \dots$$

Data Types: `double` | `single`

Degree — Degree of the polynomial transformation

2 | 3 | 4

Degree of the polynomial transformation, specified as the scalar values 2, 3, or 4.

Dimensionality — Describes the dimensionality of the geometric transformation

2

Describes the dimensionality of the geometric transformation for both input and output points.

Methods

`transformPointsInverse`

Apply inverse 2-D geometric transformation to points

`outputLimits`

Apply inverse 2-D geometric transformation to points

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#) in the MATLAB documentation.

Examples

Fit a second degree polynomial transformation to a set of fixed and moving control points

Fit a second degree polynomial transformation to a set of fixed and moving control points that are actually related by an 2-D affine transformation.

Create 2-D affine transformation.

```
theta = 10;
tformAffine = affine2d([cosd(theta) -sind(theta) 0; sind(theta) cosd(theta) 0; 0 0 1]);
```

Arbitrarily choose six pairs of control points. A second degree polynomial requires six pairs of control points.

```
fixedPoints = [10 20; 10 5; 2 3; 0 5; -5 3; -10 -20];
```

Apply forward geometric transformation to map fixed points to obtain effect of fixed and moving points that are related by some geometric transformation.

```
movingPoints = transformPointsForward(tformAffine, fixedPoints);
```

Estimate second degree `PolynomialTransformation2D` transformation that fits `fixedPoints` and `movingPoints`.

```
tformPolynomial = images.geotrans.PolynomialTransformation2D(movingPoints, fixedPoints, 2);
```

Verify the fit of our `PolynomialTransformation2D` transformation at the control points.

```
movingPointsEstimated = transformPointsInverse(tformPolynomial, fixedPoints);
errorInFit = hypot(movingPointsEstimated(:,1)-movingPoints(:,1), ...
                  movingPointsEstimated(:,2)-movingPoints(:,2))
```

See Also

See Also

Functions

fitgeotrans | imwarp

Classes

affine2d | projective2d | images.geotrans.PiecewiseLinearTransformation2d |
images.geotrans.LocalWeightedMeanTransformation2d

images.geotrans.PolynomialTransformation2d.transformPointsInverse

Apply inverse 2-D geometric transformation to points

Syntax

```
[u,v] = transformPointsInverse(tform,x,y)
U = transformPointsInverse(tform,X)
```

Description

`[u,v] = transformPointsInverse(tform,x,y)` applies the inverse transformation of `tform` to the input 2-D point arrays `x` and `y` and outputs the point arrays `u` and `v`. The input point arrays `x` and `y` must be of the same size.

`U = transformPointsInverse(tform,X)` applies the inverse transformation of `tform` to the input n -by-2 point matrix `X` and outputs the n -by-2 point matrix `U`. `transformPointsInverse` maps the point `X(k, :)` to the point `U(k, :)`.

Input Arguments

tform

Geometric transformation, specified as an `projective2d` geometric transformation object.

x

Coordinates in X dimension of points to be transformed, specified as a array.

y

Coordinates in Y dimension of points to be transformed, specified as a array.

X

X and Y coordinates of points to be transformed, specified as an n -by-2 matrix

Output Arguments

u

Transformed coordinates in X dimension, returned as an array.

v

Transformed coordinates in Y dimension, returned as an array.

U

Transformed X and Y coordinates, returned as an n -by-2 matrix

See Also

images.geotrans.PolynomialTransformation2d.outputLimits

Apply inverse 2-D geometric transformation to points

Syntax

```
[xLimitsOut,yLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn)
[xLimitsOut,yLimitsOut,zLimitsOut] =
outputLimits(tform,xLimitsIn,yLimitsIn,zLimitsIn)
```

Description

`[xLimitsOut,yLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn)` estimates the output spatial limits corresponding to a given 2D geometric transformation and a set of input spatial limits.

`[xLimitsOut,yLimitsOut,zLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn,zLimitsIn)` estimates the output spatial limits corresponding to a given 3D geometric transformation and a set of input spatial limits.

Input Arguments

tform

Geometric transformation, specified as an `images.geotrans.PolynomialTransformation2D` object.

xLimitsIn

Limits along the *X* axes, specified as a two-element vector, such as `[]`.

yLimitsIn

Limits along the *Y* axes, specified as a two-element vector, such as `[]`.

zLimitsIn

Limits along the *Z* axes, specified as a two-element vector, such as [].

Output Arguments

xLimitsOut

Actual limits along the *X* dimension, returned as an array.

yLimitsOut

Actual limits along the *Y* dimension, returned as an array.

zLimitsOut

Actual limits along the *Z* dimension, returned as an array.

See Also

projective2d class

2-D projective geometric transformation

Description

A `projective2d` object encapsulates a 2-D projective geometric transformation.

You can create a `projective2d` object using the following methods:

- `fitgeotrans` — Returns a `projective2d` object that maps control point pairs
- The `projective2d` class constructors

Construction

`tform = projective2d()` creates a `projective2d` object with default property settings that correspond to the identity transformation.

`tform = projective2d(A)` creates a `projective2d` object given an input 3-by-3 matrix `A` that specifies a valid projective transformation.

Input Arguments

A — Forward 2-D projective transformation

3-by-3 numeric matrix

Forward 2-D projective transformation, specified as a 3-by-3 numeric matrix. `A` has the form:

$$A = \begin{bmatrix} a & b & c; \\ d & e & f; \\ g & h & i \end{bmatrix}$$

Data Types: `double` | `single`

Properties

T — Forward 2-D projective transformation

3-by-3 numeric matrix

Forward 2-D projective transformation, specified as a 3-by-3 numeric matrix.

The matrix T uses the convention:

$$[x \ y \ 1] = [u \ v \ 1] * T$$

where T has the form:

```
[a b c;...  
 d e f;...  
 g h i];
```

The default of T is the identity transformation.

Data Types: `double` | `single`

Dimensionality — Describes the dimensionality of the geometric transformation

2

Describes the dimensionality of the geometric transformation for both input and output points.

Methods

<code>invert</code>	Invert geometric transformation
<code>outputLimits</code>	Find output spatial limits given input spatial limits
<code>transformPointsForward</code>	Apply forward geometric transformation
<code>transformPointsInverse</code>	Apply inverse 2-D geometric transformation to points

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#) in the MATLAB documentation.

Examples

Apply Projective Transformation to Image

This example shows how to apply rotation and tilt to an image, using a `projective2d` geometric transformation object created directly from a transformation matrix.

Read a grayscale image into the workspace.

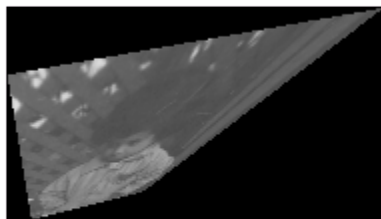
```
I = imread('pout.tif');
```

Create a geometric transformation object. This example combines rotation and tilt into a transformation matrix, `tm`. Use this transformation matrix to construct a `projective2d` class geometric transformation object, `tform`.

```
theta = 10;  
tm = [cosd(theta) -sind(theta) 0.001; ...  
      sind(theta) cosd(theta) 0.01; ...  
      0 0 1];  
tform = projective2d(tm);
```

Apply the transformation using `imwarp`. View the transformed image.

```
outputImage = imwarp(I,tform);  
figure  
imshow(outputImage);
```



Create a Projective2d Object

Create a `projective2d` object that defines the transformation.

```
theta = 10;
tform = projective2d([cosd(theta) -sind(theta) 0.001; sind(theta) cosd(theta) 0.01; 0 0 1]);
tform =
    projective2d with properties:
        T: [3x3 double]
        Dimensionality: 2
```

Apply forward geometric transformation to an input point.

```
[X,Y] = transformPointsForward(tform,5,10)
```

```
X =
```

```
6.0276
```

```
Y =
```

```
8.1265
```

Apply inverse geometric transformation to output point from previous step to recover the point we started with.

```
[U,V] = transformPointsInverse(tform,X,Y)
```

```
U =
```

```
5.0000
```

```
V =
```

```
10
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, you can only specify singular objects—arrays of objects are not supported.

See Also

See Also

Functions

`fitgeotrans` | `imwarp`

Classes

`affine2d` | `images.geotrans.PiecewiseLinearTransformation2d`
| `images.geotrans.PolynomialTransformation2d` |
`images.geotrans.LocalWeightedMeanTransformation2d`

invert

Class: projective2d

Invert geometric transformation

Syntax

```
invtfom = invert(tform)
```

Description

`invtfom = invert(tform)` inverts the geometric transformation `tform` and returns the inverse geometric transform.

Input Arguments

tform

Geometric transformation, specified as an `projective2d` geometric transformation object.

Output Arguments

invtfom

Inverse of the geometric transformation, returned as an `projective2d` geometric transformation object

Examples

Invert Geometric Transformation Object

Create a `projective2d` object that defines a transformation.

```
theta = 10;
tform = projective2d([cosd(theta) -sind(theta) 0.001; sind(theta) cosd(theta) 0.01; 0 0 1]);
invtform =
```

```
projective2d with properties:
```

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

Invert the geometric transformation.

```
invtform = invert(tform)
```

```
invtform =
```

```
projective2d with properties:
```

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

See Also

outputLimits

Class: projective2d

Find output spatial limits given input spatial limits

Syntax

```
[xLimitsOut,yLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn)
```

Description

`[xLimitsOut,yLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn)` estimates the output spatial limits corresponding to a given geometric transformation, `tform`, and a set of input spatial limits.

Input Arguments

tform

Geometric transformation, specified as an `projective2d` geometric transformation object.

xLimitsIn

Input spatial limits in X dimension, specified as a two-element vector of doubles.

yLimitsIn

Input spatial limits in Y dimension, specified as a two-element vector of doubles.

Output Arguments

xLimitsOut

Output spatial limits in X dimension, returned as a two-element vector of doubles.

yLimitsOut

Output spatial limits in Y dimension, returned as a two-element vector of **doubles**.

Examples

Estimate the Output Limits for a Geometric Transformation

Create an `projective2d` object that defines a transformation.

```
theta = 10;  
tform = affine2d([cosd(theta) -sind(theta) 0; sind(theta) cosd(theta) 0; 0 0 1]);  
tform =
```

```
    affine2d with properties:
```

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

Estimate the output spatial limits, given the geometric transformation.

```
[xlim, ylim] = outputLimits(tform,[1 240],[1 291])
```

```
xlim =
```

```
    1.1459    189.2220
```

```
ylim =
```

```
   -32.5526    73.2307
```

See Also

transformPointsForward

Class: projective2d

Apply forward geometric transformation

Syntax

```
[x,y] = transformPointsForward(tform,u,v)  
X = transformPointsForward(tform,U)
```

Description

`[x,y] = transformPointsForward(tform,u,v)` applies the forward transformation of `tform` to the input 2-D point arrays `u` and `v` and returns the point arrays `x` and `y`. The input point arrays `u` and `v` must be of the same size.

`X = transformPointsForward(tform,U)` applies the forward transformation of `tform` to the input n -by-2 point matrix `U` and outputs the n -by-2 point matrix `X`. `transformPointsForward` maps the point `U(k,:)` to the point `X(k,:)`.

Input Arguments

tform

Geometric transformation, specified as an `projective2d` geometric transformation object.

u

Coordinates in X dimension of points to be transformed, specified as an array.

v

Coordinates in Y dimension of points to be transformed, specified as an array.

U

X and Y coordinates of points to be transformed, specified as an n -by-2 matrix

Output Arguments**x**

Transformed coordinates in X dimension, returned as a array.

y

Transformed coordinates in Y dimension, returned as a array.

X

Transformed points in X and Y dimensions, returned as an n -by-2 point matrix

Examples**Apply Forward Geometric Transformation**

Create an `projective2d` object that defines the transformation.

```
theta = 10;
tform = projective2d([cosd(theta) -sind(theta) 0.001; sind(theta) cosd(theta) 0.01; 0 0 1]);
tform =
```

```
projective2d with properties:
```

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

Apply forward geometric transformation to an input point.

```
[X,Y] = transformPointsForward(tform,5,10)
```

```
X =
```

```
    6.0276
```

Y =

8.1265

See Also

transformPointsInverse

Class: projective2d

Apply inverse 2-D geometric transformation to points

Syntax

```
[u,v] = transformPointsInverse(tform,x,y)  
U = transformPointsInverse(tform,X)
```

Description

`[u,v] = transformPointsInverse(tform,x,y)` applies the inverse transformation of `tform` to the input 2-D point arrays `x` and `y` and outputs the point arrays `u` and `v`. The input point arrays `x` and `y` must be of the same size.

`U = transformPointsInverse(tform,X)` applies the inverse transformation of `tform` to the input n -by-2 point matrix `X` and outputs the n -by-2 point matrix `U`. `transformPointsForward` maps the point `X(k, :)` to the point `U(k, :)`.

Input Arguments

tform

Geometric transformation, specified as an `projective2d` geometric transformation object.

x

Coordinates in X dimension of points to be transformed, specified as a array.

y

Coordinates in Y dimension of points to be transformed, specified as a array.

X

X and Y coordinates of points to be transformed, specified as an n -by-2 matrix

Output Arguments

u

Transformed coordinates in X dimension, returned as an array.

v

Transformed coordinates in Y dimension, returned as an array.

U

Transformed X and Y coordinates, returned as an n -by-2 matrix

Examples

Apply Inverse Geometric Transformation

Create an `projective2d` object that defines the transformation.

```
theta = 10;  
tform = projective2d([cosd(theta) -sind(theta) 0.001; sind(theta) cosd(theta) 0.01; 0 0 1]);  
tform =
```

```
projective2d with properties:
```

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

Apply forward geometric transformation to an input point.

```
[X,Y] = transformPointsForward(tform,5,10)
```

```
X =
```

```
    6.0276
```

Y =

8.1265

Apply inverse geometric transformation to output point from the previous step to recover the original coordinates.

```
[U,V] = transformPointsInverse(tform,X,Y)
```

U =

5.0000

V =

10

See Also

psf2otf

Convert point-spread function to optical transfer function

Syntax

```
OTF = psf2otf(PSF)
OTF = psf2otf(PSF,OUTSIZE)
```

Description

`OTF = psf2otf(PSF)` computes the fast Fourier transform (FFT) of the point-spread function (PSF) array and creates the optical transfer function array, `OTF`, that is not influenced by the PSF off-centering. By default, the `OTF` array is the same size as the `PSF` array.

`OTF = psf2otf(PSF,OUTSIZE)` converts the `PSF` array into an `OTF` array, where `OUTSIZE` specifies the size of the `OTF` array. `OUTSIZE` cannot be smaller than the `PSF` array size in any dimension.

To ensure that the `OTF` is not altered because of `PSF` off-centering, `psf2otf` postpads the `PSF` array (down or to the right) with 0's to match dimensions specified in `OUTSIZE`, then circularly shifts the values of the `PSF` array up (or to the left) until the central pixel reaches (1,1) position.

Note that this function is used in image convolution/deconvolution when the operations involve the FFT.

Class Support

`PSF` can be any nonsparse, numeric array. `OTF` is of class `double`.

Examples

Convert PSF to OTF

Create a point-spread function (`PSF`).

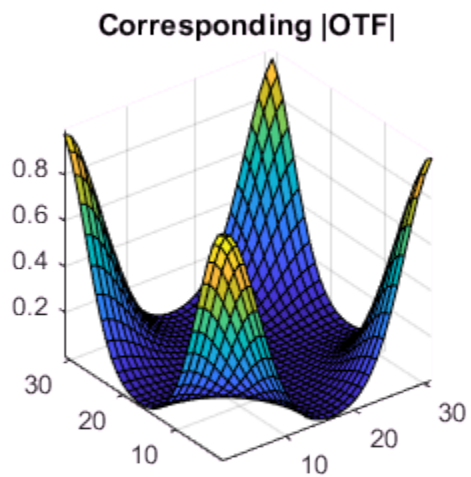
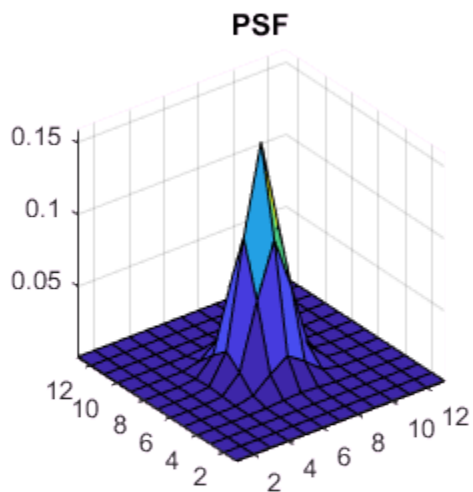
```
PSF = fspecial('gaussian',13,1);
```

Convert the PSF to an Optical Transfer Function (OTF).

```
OTF = psf2otf(PSF,[31 31]);
```

Plot the PSF and the OTF.

```
subplot(1,2,1);  
surf(PSF);  
title('PSF');  
axis square;  
axis tight  
subplot(1,2,2);  
surf(abs(OTF));  
title('Corresponding |OTF|');  
axis square;  
axis tight
```



See Also

`otf2psf` | `circshift` | `padarray`

Introduced before R2006a

psnr

Peak Signal-to-Noise Ratio (PSNR)

Syntax

```
peaksnr = psnr(A,ref)
peaksnr = psnr(A,ref,peakval)
[peaksnr,snr] = psnr( ___ )
```

Description

`peaksnr = psnr(A,ref)` calculates the peak signal-to-noise ratio for the image `A`, with the image `ref` as the reference. `A` and `ref` must be of the same size and class.

`peaksnr = psnr(A,ref,peakval)` uses `peakval` as the peak signal value for calculating the peak signal-to-noise ratio for image `A`.

`[peaksnr,snr] = psnr(___)` returns the simple signal-to-noise ratio, `snr`, in addition to the peak signal-to-noise ratio.

Examples

Calculate PSNR for Noisy Image Given Original Image as Reference

Read image and create a copy with added noise. The original image is the reference image.

```
ref = imread('pout.tif');
A = imnoise(ref,'salt & pepper', 0.02);
```

Calculate the PSNR.

```
[peaksnr, snr] = psnr(A, ref);
fprintf('\n The Peak-SNR value is %0.4f', peaksnr);
```

```
The Peak-SNR value is 22.6437
```

```
fprintf('\n The SNR value is %0.4f \n', snr);
```

```
The SNR value is 15.5524
```

Input Arguments

A — Image to be analyzed

N-D numeric matrix

Image to be analyzed, specified as an N-D numeric matrix.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

ref — Reference image

N-D numeric matrix

Reference image, specified as an N-D numeric matrix.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

peakval — Peak signal level

scalar of any numeric class

Peak signal level, specified as a scalar of any numeric class. If not specified, the default value for `peakval` depends on the class of `A` and `ref`. If the images are of floating point types, `peakval` is 1, assuming that the data is in the range `[0 1]`. If the images are of integer data types, `peakval` is the largest value allowed by the range of the class. For `uint8`, the default value is 255. For `uint16` or `int16`, the default is 65535.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

Output Arguments

peaksnr — Peak signal-to-noise ratio

scalar

Peak signal-to-noise ratio in decibels, returned as a scalar of type `double`, except if `A` and `ref` are of class `single`, in which case `peaksnr` is of class `single`.

Data Types: `single` | `double`

snr — Signal-to-noise ratio

scalar

Signal-to-noise ratio in decibels, returned as a scalar of type `double`, except if `A` and `ref` are of class `single`, in which case `peaksnr` is of class `single`.

Data Types: `single` | `double`

Algorithm

The `psnr` function implements the following equation to calculate the Peak Signal-to-Noise Ratio (PSNR):

$$PSNR = 10 \log_{10} \left(\frac{peakval^2}{MSE} \right)$$

where *peakval* is either specified by the user or taken from the range of the image datatype (e.g. for `uint8` image it is 255). *MSE* is the mean square error, i.e. *MSE* between `A` and `ref`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.

See Also

See Also

`immse` | `mean` | `median` | `ssim` | `sum` | `var`

Topics

“Image Quality Metrics”

Introduced in R2014a

qtdecomp

Quadtree decomposition

Syntax

```
S = qtdecomp(I)
S = qtdecomp(I, threshold)
S = qtdecomp(I, threshold, mindim)
S = qtdecomp(I, threshold, [mindim maxdim])
S = qtdecomp(I, fun)
```

Description

`qtdecomp` divides a square image into four equal-sized square blocks, and then tests each block to see if it meets some criterion of homogeneity. If a block meets the criterion, it is not divided any further. If it does not meet the criterion, it is subdivided again into four blocks, and the test criterion is applied to those blocks. This process is repeated iteratively until each block meets the criterion. The result can have blocks of several different sizes.

`S = qtdecomp(I)` performs a quadtree decomposition on the intensity image `I` and returns the quadtree structure in the sparse matrix `S`. If `S(k,m)` is nonzero, then `(k,m)` is the upper left corner of a block in the decomposition, and the size of the block is given by `S(k,m)`. By default, `qtdecomp` splits a block unless all elements in the block are equal.

`S = qtdecomp(I, threshold)` splits a block if the maximum value of the block elements minus the minimum value of the block elements is greater than `threshold`. `threshold` is specified as a value between 0 and 1, even if `I` is of class `uint8` or `uint16`. If `I` is `uint8`, the threshold value you supply is multiplied by 255 to determine the actual threshold to use; if `I` is `uint16`, the threshold value you supply is multiplied by 65535.

`S = qtdecomp(I, threshold, mindim)` will not produce blocks smaller than `mindim`, even if the resulting blocks do not meet the threshold condition.

`S = qtdecomp(I, threshold, [mindim maxdim])` will not produce blocks smaller than `mindim` or larger than `maxdim`. Blocks larger than `maxdim` are split even if they meet the threshold condition. `maxdim/mindim` must be a power of 2.

`S = qtdecomp(I, fun)` uses the function `fun` to determine whether to split a block. `qtdecomp` calls `fun` with all the current blocks of size `m`-by-`m` stacked into an `m`-by-`m`-by-`k` array, where `k` is the number of `m`-by-`m` blocks. `fun` returns a logical `k`-element vector, whose values are 1 if the corresponding block should be split, and 0 otherwise. (For example, if `k(3)` is 0, the third `m`-by-`m` block should not be split.) `fun` must be a function handle.

Class Support

For the syntaxes that do not include a function, the input image can be of class `logical`, `uint8`, `uint16`, `int16`, `single`, or `double`. For the syntaxes that include a function, the input image can be of any class supported by the function. The output matrix is always of class `sparse`.

Examples

Perform Quadtree Decomposition of Sample Matrix

Create a small sample matrix.

```
I = uint8([1 1 1 2 3 6 6;...
          1 1 2 1 4 5 6 8;...
          1 1 1 1 7 7 7 7;...
          1 1 1 1 6 6 5 5;...
          20 22 20 22 1 2 3 4;...
          20 22 22 20 5 4 7 8;...
          20 22 20 20 9 12 40 12;...
          20 22 20 20 13 14 15 16]);
```

Perform the quadtree decomposition and display the results.

```
S = qtdecomp(I, .05);
disp(full(S));
```

```
    4    0    0    0    4    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    4    0    0    0    2    0    2    0
```

```
0 0 0 0 0 0 0 0
0 0 0 0 2 0 1 1
0 0 0 0 0 0 1 1
```

View Block Representation of Quadtree Decomposition

Read image into the workspace.

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

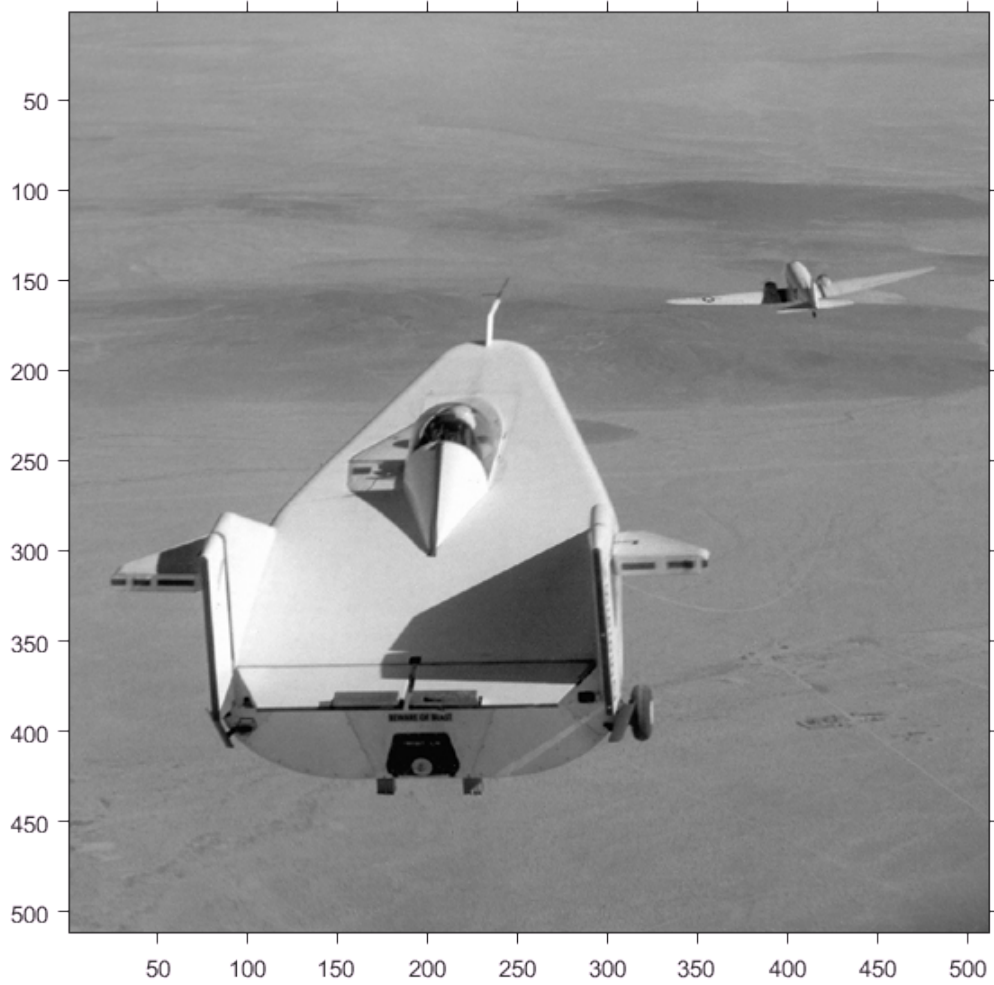
Perform the quadtree decomposition and display the block representation in a figure.

```
S = qtdecomp(I,.27);
blocks = repmat(uint8(0),size(S));

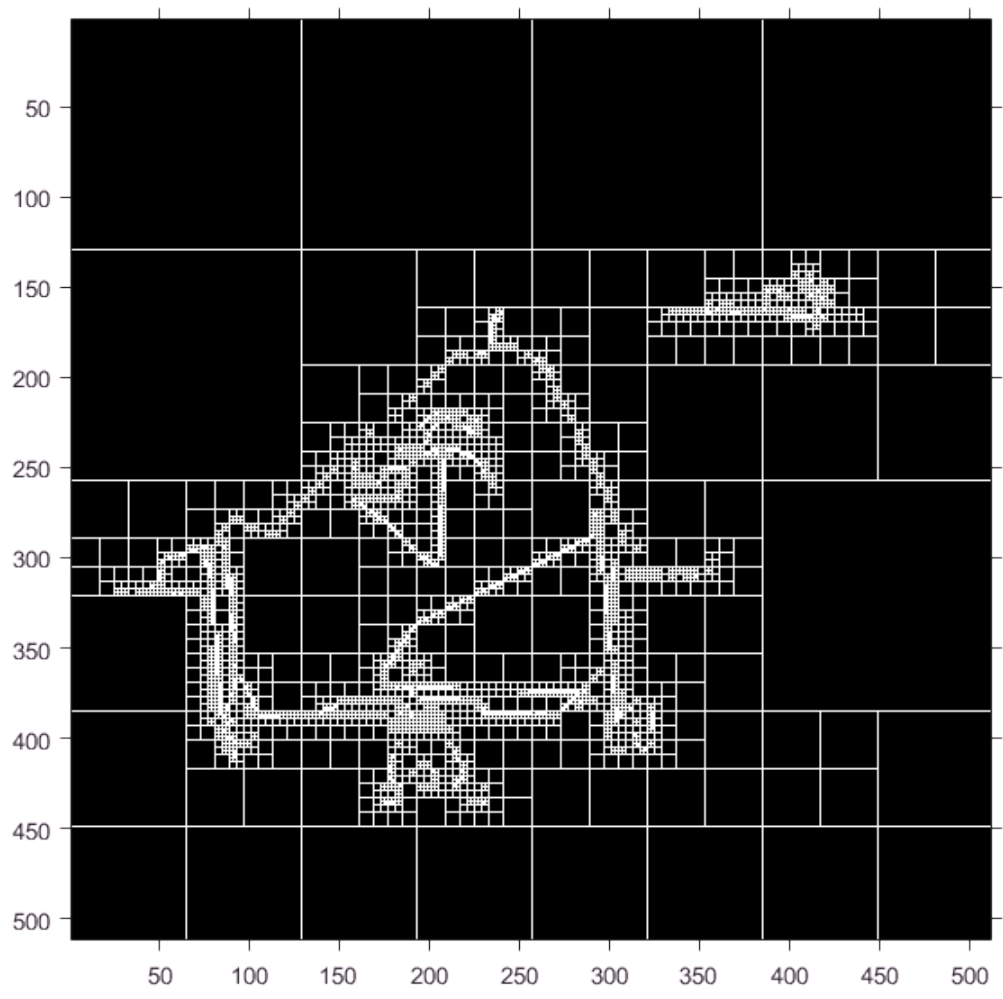
for dim = [512 256 128 64 32 16 8 4 2 1];
    numblocks = length(find(S==dim));
    if (numblocks > 0)
        values = repmat(uint8(1),[dim dim numblocks]);
        values(2:dim,2:dim,:) = 0;
        blocks = qtsetblk(blocks,S,dim,values);
    end
end

blocks(end,1:end) = 1;
blocks(1:end,end) = 1;

imshow(I)
```



```
figure  
imshow(blocks,[])
```

Tips

`qtdecomp` is appropriate primarily for square images whose dimensions are a power of 2, such as 128-by-128 or 512-by-512. These images can be divided until the blocks are as small as 1-by-1. If you use `qtdecomp` with an image whose dimensions are not a power of 2, at some point the blocks cannot be divided further. For example, if an image is 96-by-96, it can be divided into blocks of size 48-by-48, then 24-by-24, 12-by-12, 6-by-6, and finally 3-by-3. No further division beyond 3-by-3 is possible. To process this image, you must set `mindim` to 3 (or to 3 times a power of 2); if you are using the syntax that includes a function, the function must return 0 at the point when the block cannot be divided further.

See Also

`qtgetblk` | `qtsetblk`

Topics

“Anonymous Functions” (MATLAB)

“Parameterizing Functions” (MATLAB)

“Create Function Handle” (MATLAB)

Introduced before R2006a

qtgetblk

Block values in quadtree decomposition

Syntax

```
[vals, r, c] = qtgetblk(I, S, dim)
[vals, idx] = qtgetblk(I, S, dim)
```

Description

`[vals, r, c] = qtgetblk(I, S, dim)` returns in `vals` an array containing the `dim`-by-`dim` blocks in the quadtree decomposition of `I`. `S` is the sparse matrix returned by `qtdecomp`; it contains the quadtree structure. `vals` is a `dim`-by-`dim`-by-`k` array, where `k` is the number of `dim`-by-`dim` blocks in the quadtree decomposition; if there are no blocks of the specified size, all outputs are returned as empty matrices. `r` and `c` are vectors containing the row and column coordinates of the upper left corners of the blocks.

`[vals, idx] = qtgetblk(I, S, dim)` returns in `idx` a vector containing the linear indices of the upper left corners of the blocks.

Class Support

`I` can be of class `logical`, `uint8`, `uint16`, `int16`, `single`, or `double`. `S` is of class `sparse`.

Examples

```
I = [1  1  1  1  2  3  6  6
      1  1  2  1  4  5  6  8
      1  1  1  1 10 15  7  7
      1  1  1  1 20 25  7  7
     20 22 20 22  1  2  3  4
     20 22 22 20  5  6  7  8
     20 22 20 20  9 10 11 12]
```

```
    22  22  20  20  13  14  15  16];  
S = qtdecomp(I,5);  
[vals,r,c] = qtgetblk(I,S,4)
```

Tips

The ordering of the blocks in `vals` matches the columnwise order of the blocks in `I`. For example, if `vals` is 4-by-4-by-2, `vals(:, :, 1)` contains the values from the first 4-by-4 block in `I`, and `vals(:, :, 2)` contains the values from the second 4-by-4 block.

See Also

`qtdecomp` | `qtsetblk`

Introduced before R2006a

qtsetblk

Set block values in quadtree decomposition

Syntax

```
J = qtsetblk(I, S, dim, vals)
```

Description

`J = qtsetblk(I, S, dim, vals)` replaces each `dim-by-dim` block in the quadtree decomposition of `I` with the corresponding `dim-by-dim` block in `vals`. `S` is the sparse matrix returned by `qtdecomp`; it contains the quadtree structure. `vals` is a `dim-by-dim-by-k` array, where `k` is the number of `dim-by-dim` blocks in the quadtree decomposition.

Class Support

`I` can be of class `logical`, `uint8`, `uint16`, `int16`, `single`, or `double`. `S` is of class `sparse`.

Examples

```
I = [1   1   1   1   2   3   6   6
      1   1   2   1   4   5   6   8
      1   1   1   1  10  15   7   7
      1   1   1   1  20  25   7   7
     20  22  20  22   1   2   3   4
     20  22  22  20   5   6   7   8
     20  22  20  20   9  10  11  12
     22  22  20  20  13  14  15  16];
```

```
S = qtdecomp(I,5);
```

```
newvals = cat(3,zeros(4),ones(4));
J = qtsetblk(I,S,4,newvals)
```

Tips

The ordering of the blocks in `vals` must match the columnwise order of the blocks in `I`. For example, if `vals` is 4-by-4-by-2, `vals(:, :, 1)` contains the values used to replace the first 4-by-4 block in `I`, and `vals(:, :, 2)` contains the values for the second 4-by-4 block.

See Also

`qtdecomp` | `qtgetblk`

Introduced before R2006a

radon

Radon transform

Syntax

```
R = radon(I, theta)
[R, xp] = radon(...)
[ ___ ]= radon(gpuarrayI, theta)
```

Description

`R = radon(I, theta)` returns the Radon transform **R** of the intensity image **I** for the angle **theta** degrees.

The Radon transform is the projection of the image intensity along a radial line oriented at a specific angle. If **theta** is a scalar, **R** is a column vector containing the Radon transform for **theta** degrees. If **theta** is a vector, **R** is a matrix in which each column is the Radon transform for one of the angles in **theta**. If you omit **theta**, it defaults to 0:179.

`[R, xp] = radon(...)` returns a vector **xp** containing the radial coordinates corresponding to each row of **R**.

The radial coordinates returned in **xp** are the values along the x' -axis, which is oriented at **theta** degrees counterclockwise from the x -axis. The origin of both axes is the center pixel of the image, which is defined as

```
floor((size(I)+1)/2)
```

For example, in a 20-by-30 image, the center pixel is (10,15).

`[___]= radon(gpuarrayI, theta)` performs the Radon transform on a GPU. The input image and the return values are 2-D `gpuArrays`. **theta** can be a `double` or `gpuArray` of underlying class `double`. This syntax requires the Parallel Computing Toolbox.

Class Support

I can be of class `double`, `logical`, or any integer class. All other inputs and outputs are of class `double`. Neither of the inputs can be sparse.

`gpuarrayI` is a `gpuArray` with underlying class `uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`, `logical`, `single` or `double` and must be two-dimensional. `theta` is a `double` vector or `gpuArray` vector of underlying class `double`.

Examples

Calculate Radon Transform and Display Plot

Make the axes scale visible for this image.

```
iptsetpref('ImshowAxesVisible','on')
```

Create a sample image.

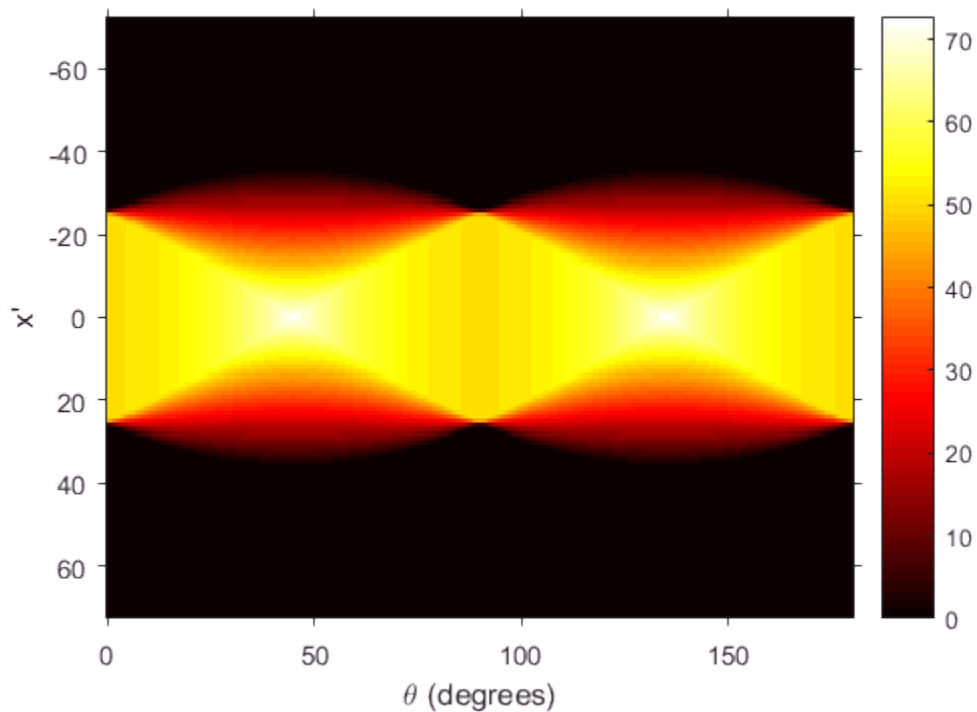
```
I = zeros(100,100);  
I(25:75, 25:75) = 1;
```

Calculate the Radon transform.

```
theta = 0:180;  
[R,xp] = radon(I,theta);
```

Display the transform.

```
imshow(R,[],'Xdata',theta,'Ydata',xp,'InitialMagnification','fit')  
xlabel('\theta (degrees)')  
ylabel('x')  
colormap(gca,hot), colorbar
```

Make the axes scale invisible.

```
iptsetpref('ImshowAxesVisible','off')
```

Calculate Radon transform on a GPU

Calculate Radon transform on a GPU and visualize it.

```
iptsetpref('ImshowAxesVisible','on')  
I = zeros(100,100);  
I(25:75, 25:75) = 1;  
theta = 0:180;  
[R,xp] = radon(gpuArray(I),theta);
```

```

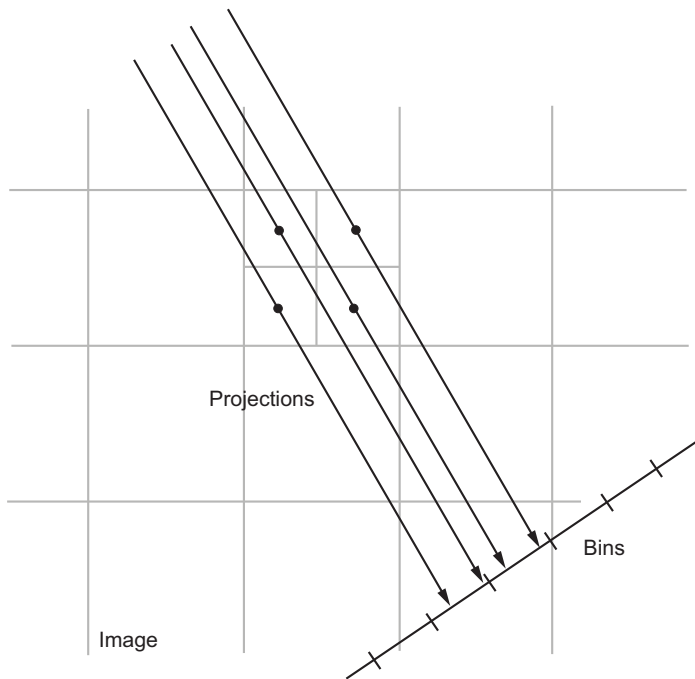
imshow(R,[], 'Xdata',theta, 'Ydata',xp,...
        'InitialMagnification','fit')
xlabel('\theta (degrees)')
ylabel('x''')
colormap(gca,hot), colorbar
iptsetpref('ImshowAxesVisible','off')

```

Algorithms

The Radon transform of an image is the sum of the Radon transforms of each individual pixel.

The algorithm first divides pixels in the image into four subpixels and projects each subpixel separately, as shown in the following figure.



Each subpixel's contribution is proportionally split into the two nearest bins, according to the distance between the projected location and the bin centers. If the subpixel projection

hits the center point of a bin, the bin on the axes gets the full value of the subpixel, or one-fourth the value of the pixel. If the subpixel projection hits the border between two bins, the subpixel value is split evenly between the bins.

References

Bracewell, Ronald N., *Two-Dimensional Imaging*, Englewood Cliffs, NJ, Prentice Hall, 1995, pp. 505-537.

Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, pp. 42-45.

See Also

See Also

fan2para | fanbeam | ifanbeam | iradon | para2fan | phantom

Introduced before R2006a

rangefilt

Local range of image

Syntax

```
J = rangefilt(I)
J = rangefilt(I, NHOOD)
```

Description

`J = rangefilt(I)` returns the array `J`, where each output pixel contains the range value (maximum value – minimum value) of the 3-by-3 neighborhood around the corresponding pixel in the input image `I`. The image `I` can have any dimension. The output image `J` is the same size as the input image `I`.

`J = rangefilt(I, NHOOD)` performs range filtering of the input image `I` where you specify the neighborhood in `NHOOD`. `NHOOD` is a multidimensional array of zeros and ones where the nonzero elements specify the neighborhood for the range filtering operation. `NHOOD`'s size must be odd in each dimension.

By default, `rangefilt` uses the neighborhood `true(3)`. `rangefilt` determines the center element of the neighborhood by `floor((size(NHOOD) + 1)/2)`. For information about specifying neighborhoods, see [Notes](#).

Class Support

`I` can be logical or numeric and must be real and nonsparse. `NHOOD` can be logical or numeric and must contain zeros or ones.

The output image `J` is the same class as `I`, except for signed integer data types. The output class for signed data types is the corresponding unsigned integer data type. For example, if the class of `I` is `int8`, then the class of `J` is `uint8`.

Notes

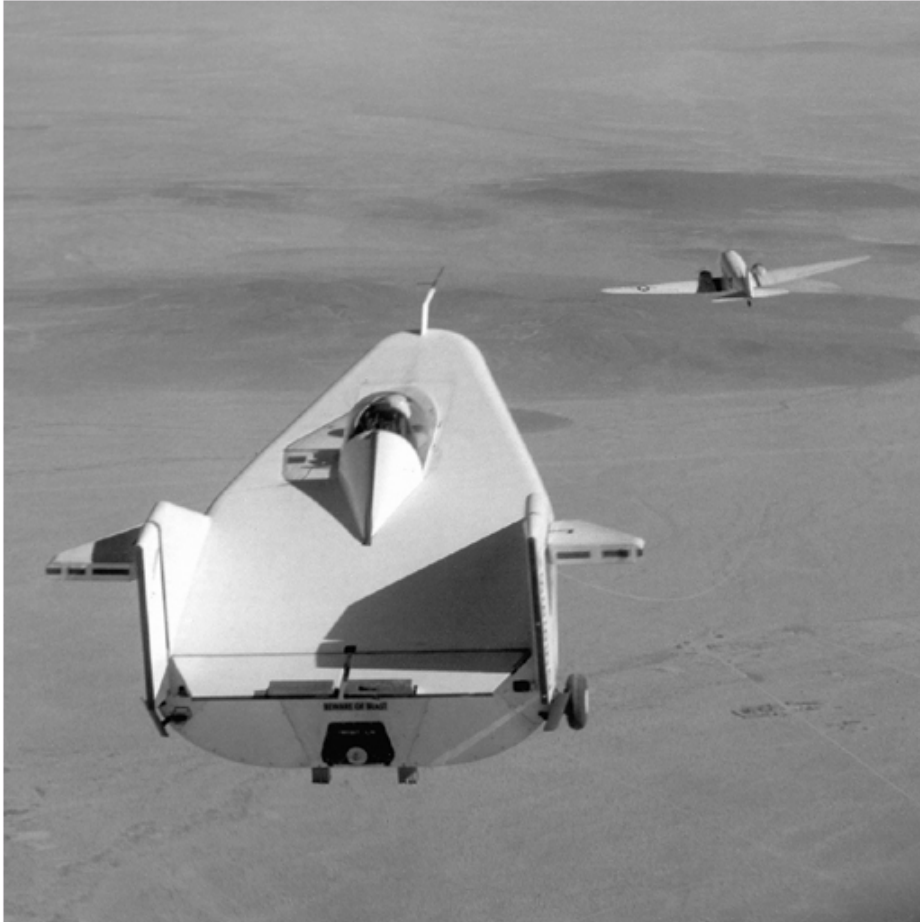
`rangefilt` uses the morphological functions `imdilate` and `imerode` to determine the maximum and minimum values in the specified neighborhood. Consequently, `rangefilt` uses the padding behavior of these morphological functions.

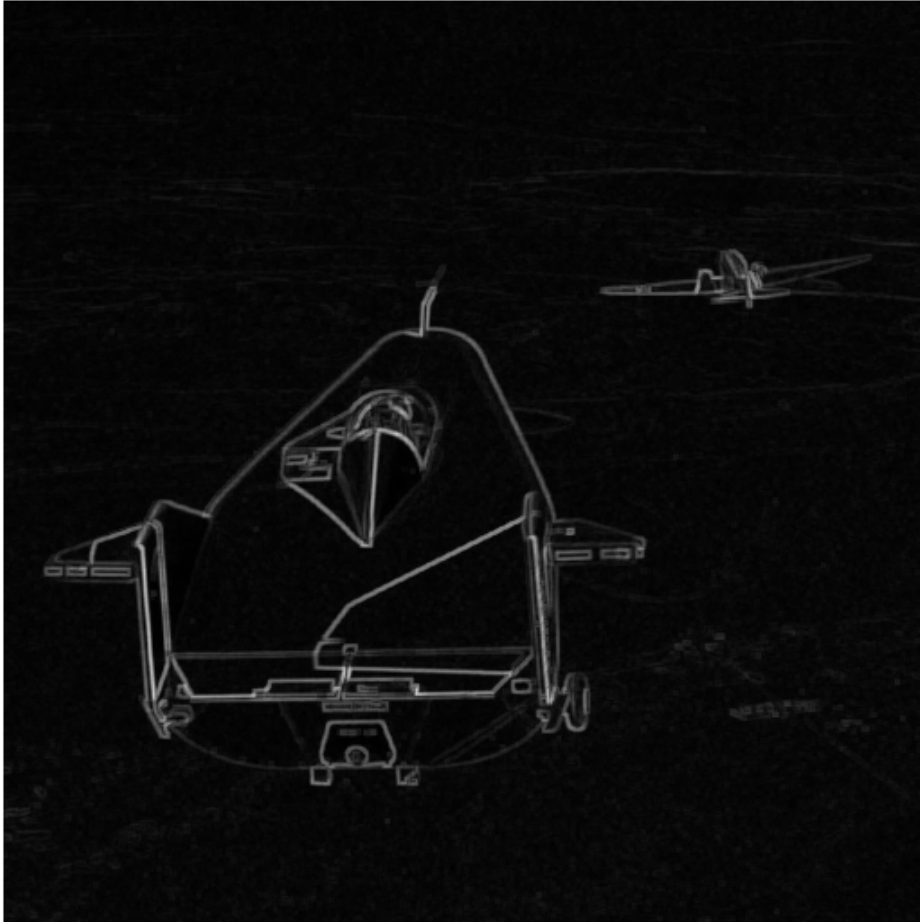
In addition, to specify neighborhoods of various shapes, such as a disk, use the `strel` function to create a structuring element object and then extract the neighborhood from the structuring element object's `neighborhood` property.

Examples

Identify Objects in 2-D Image

```
I = imread('liftingbody.png');  
J = rangefilt(I);  
imshow(I), figure, imshow(J);
```





Quantify Land Cover Changes in an RGB (3-D) Image

Read an image and convert color space from RGB to LAB.

```
I = imread('autumn.tif');  
cform = makecform('srgb2lab');  
LAB = applycform(I, cform);
```

Perform the range filtering on the LAB image.

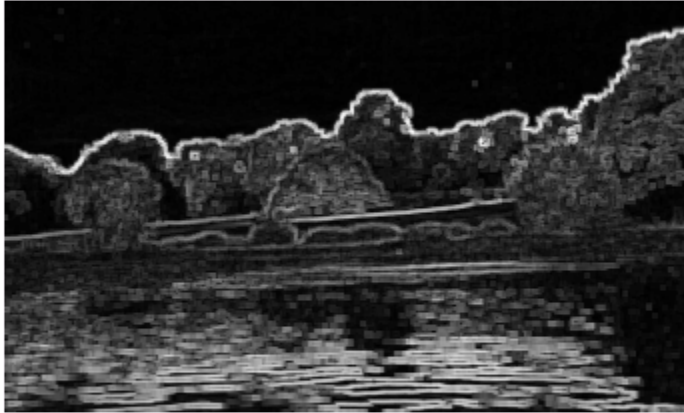
```
rLAB = rangefilt(LAB);
```

Display the images.

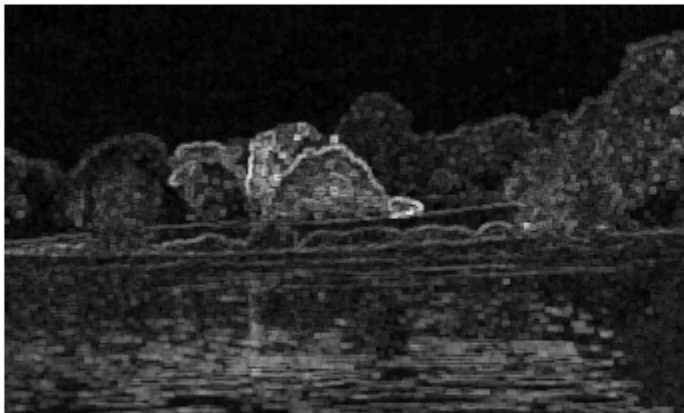
```
imshow(I);
```



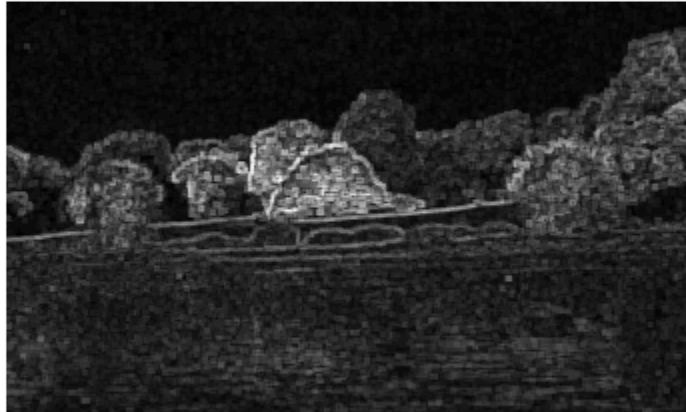
```
figure, imshow(rLAB(:,:,1),[]);
```

```
figure, imshow(rLAB(:,:,2),[]);
```



```
figure, imshow(rLAB(:,:,3),[]);
```



See Also

`entropyfilt` | `getnhood` | `imdilate` | `imerode` | `stdfilt` | `strel` | `offsetstrel`

Introduced before R2006a

regionfill

Fill in specified regions in image using inward interpolation

Syntax

```
J = regionfill(I,mask)
J = regionfill(I,x,y)
```

Description

`J = regionfill(I,mask)` fills the regions in image `I` specified by `mask`. Non-zero pixels in `mask` designate the pixels of image `I` to fill. You can use `regionfill` to remove objects in an image or to replace invalid pixel values using their neighbors.

`J = regionfill(I,x,y)` fills the region in image `I` corresponding to the polygon with vertices specified by `x` and `y`.

Examples

Fill Region in Grayscale Image

Read grayscale image into the workspace.

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

Specify a polygon that completely surrounds one of the coins in the image. This example uses the x-coordinates and y-coordinates (columns and rows) of the polygon vertices to specify the region.

```
x = [222 272 300 270 221 194];
y = [21 21 75 121 121 75];
```

Fill the polygon, using the `regionfill` function.

```
J = regionfill(I,x,y);
```

Display the original image and the filled image side-by-side.

```
figure
subplot(1,2,1)
imshow(I)
title('Original image')
subplot(1,2,2)
imshow(J)
title('Image with one less coin')
```

Original image



Image with one less coin



Fill Regions Using Mask Image

Read grayscale image into the workspace.

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

Create a mask image that covers all the coins.

```
mask = I < 200;
```

Fill holes in the mask image.

```
mask = imfill(mask,'holes');
```

Remove noise in the mask image.

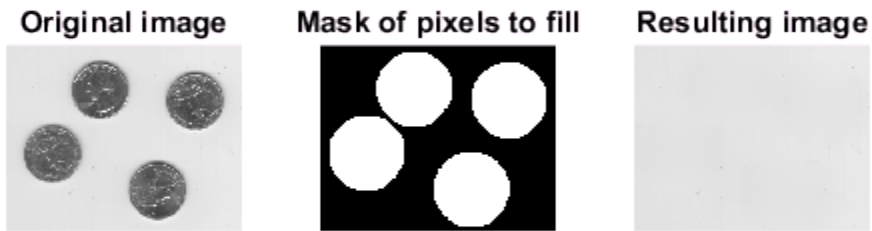
```
mask = imerode(mask,strel('disk',10));  
mask = imdilate(mask,strel('disk',20));
```

Fill the regions in the input image using the mask image.

```
J = regionfill(I,mask);
```

Display the original image next to the mask image and the filled image.

```
figure  
subplot(1,3,1)  
imshow(I)  
title('Original image')  
subplot(1,3,2)  
imshow(mask)  
title('Mask of pixels to fill')  
subplot(1,3,3)  
imshow(J)  
title('Resulting image')
```



Input Arguments

I — Input grayscale image

2-D numeric array, nonsparse and real

Input grayscale image, specified as a 2-D numeric array, nonsparse and real. I must be greater than or equal to a 3-by-3 array.

Example: `I = imread('eight.tif');`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

mask — Mask binary image

nonsparse logical array

Mask binary image, specified as a nonsparse logical array the same size as `I`.

Data Types: `logical`

x — X-coordinates of polygon vertices

numeric vector

X-coordinates of polygon vertices, specified as a numeric vector of class `double`. Must be the same length as `y`.

Example: `x = [222 272 300 270 221 194];`

Data Types: `double`

y — Y-coordinates of polygon vertices

numeric vector

Y-coordinates of polygon vertices, specified as a numeric vector of class `double`. Must be the same length as `x`.

Example: `y = [21 21 75 121 121 75];`

Data Types: `double`

Output Arguments

J — Filled grayscale image

2-D numeric array, nonsparse and real

Filled grayscale image, returned as a 2-D numeric array, nonsparse, and real. `J` has the same size and class as `I`.

Tips

- `regionfill` does not support the interactive syntax that `roifill` supports to specify a region of interest (ROI). To define an ROI interactively, use `roipoly` with `regionfill`.

Algorithms

`regionfill` smoothly interpolates inward from the pixel values on the outer boundary of the regions. `regionfill` computes the discrete Laplacian over the regions and solves the Dirichlet boundary value problem.

See Also

See Also

`imfill` | `impoly` | `poly2mask` | `roifilt2` | `roipoly`

Introduced in R2015a

regionprops

Measure properties of image regions

Syntax

```
stats = regionprops(BW,properties)
stats = regionprops(CC,properties)
stats = regionprops(L,properties)
stats = regionprops( ____,I,properties)
stats = regionprops(output, ____)
stats = regionprops(gpuarrayImg, ____)
```

Description

`stats = regionprops(BW,properties)` returns measurements for the set of properties specified by `properties` for each 8-connected component (object) in the binary image, `BW`. `stats` is struct array containing a struct for each object in the image. You can use `regionprops` on contiguous regions and discontinuous regions (see “Algorithms” on page 1-1676).

`stats = regionprops(CC,properties)` returns measurements for the set of properties specified by `properties` for each connected component (object) in `CC`. `CC` is a structure returned by `bwconncomp`.

`stats = regionprops(L,properties)` returns measurements for the set of properties specified by `properties` for each labeled region in the label matrix `L`.

`stats = regionprops(____,I,properties)` returns measurements for the set of properties specified by `properties` for each labeled region in the image `I`. The first input to `regionprops` (`BW`, `CC`, or `L`) identifies the regions in `I`. The size of the first input must match the size of the image, that is, `size(I)` must equal `size(BW)`, `CC.ImageSize`, or `size(L)`.

`stats = regionprops(output, ____)` returns measurements for a set of properties, where `output` specifies the type of return value. `regionprops` can return these values in a `struct` or a `table`.

`stats = regionprops(gpuarrayImg, ___)` performs the measurements on a GPU. `gpuarrayImg` can be a 2-D binary image (logical `gpuArray`) or a `gpuArray` label matrix. The connected component structure (CC) returned by `bwconncomp` is not supported on the GPU.

When run on a GPU, `regionprops` does not support the following properties: 'ConvexArea', 'ConvexHull', 'ConvexImage', 'EulerNumber', 'FilledArea', 'FilledImage', and 'Solidity'.

Examples

Calculate Centroids and Superimpose Locations on Image

Read binary image into workspace.

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

Calculate centroids for connected components in the image using `regionprops`.

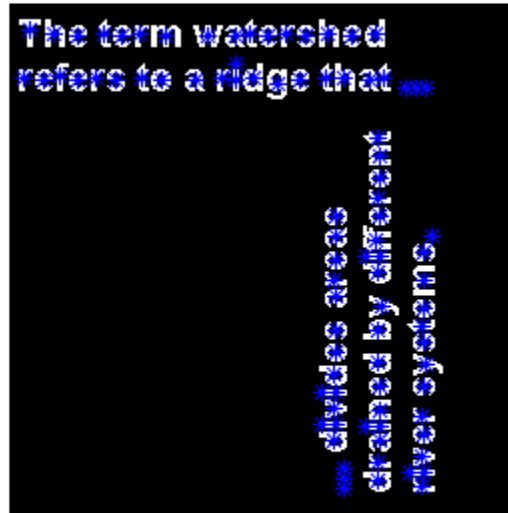
```
s = regionprops(BW, 'centroid');
```

Concatenate structure array containing centroids into a single matrix.

```
centroids = cat(1, s.Centroid);
```

Display binary image with centroid locations superimposed.

```
imshow(BW)
hold on
plot(centroids(:,1),centroids(:,2), 'b*')
hold off
```



Calculate Centroids and Superimpose Locations on Image on a GPU

Read binary image into a gpuArray.

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

Calculate the centroids of objects in the image.

```
s = regionprops(BW, 'centroid');
```

Plot the centroids on the image.

```
centroids = cat(1, s.Centroid);  
imshow(BW)  
hold on  
plot(centroids(:,1), centroids(:,2), 'b*')
```

```
hold off
```

Estimate Center and Radii of Circular Objects and Plot Circles

Estimate the center and radii of circular objects in an image and use this information to plot circles on the image. In this example, `regionprops` returns the information it calculates in a table.

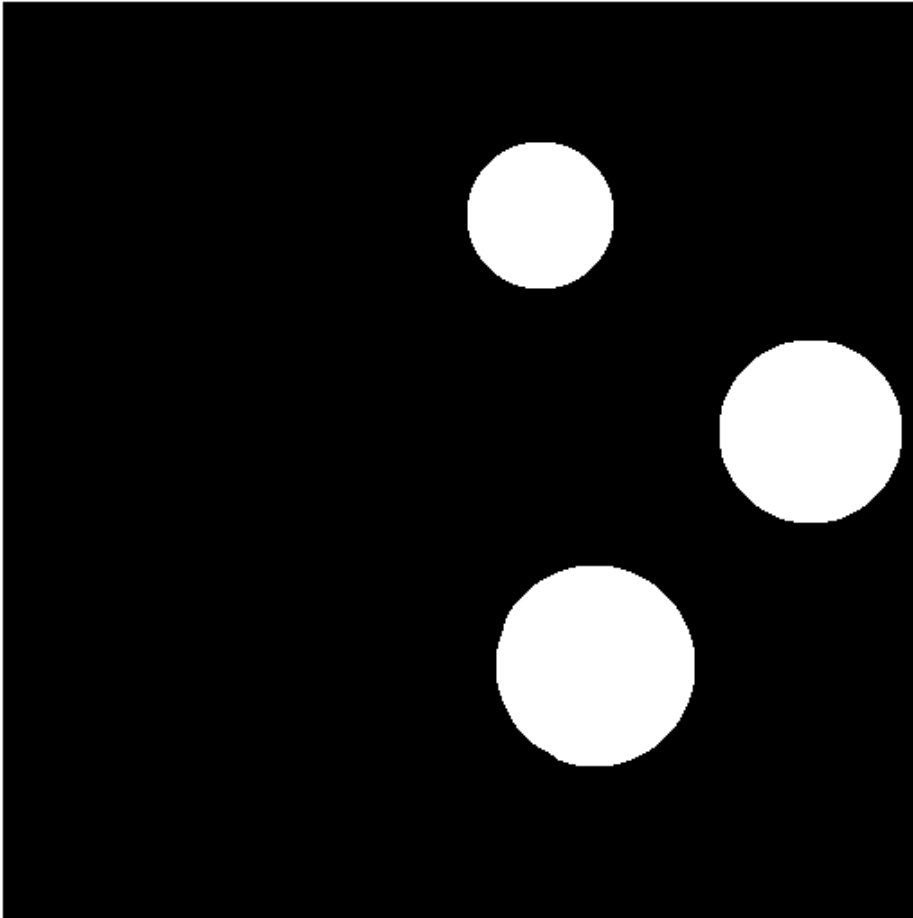
Read an image into workspace.

```
a = imread('circlesBrightDark.png');
```

Turn the input image into a binary image.

```
bw = a < 100;  
imshow(bw)  
title('Image with Circles')
```

Image with Circles



Calculate properties of regions in the image and return the data in a table.

```
stats = regionprops('table',bw,'Centroid',...  
    'MajorAxisLength','MinorAxisLength')  
stats = 4×3 table
```

Centroid		MajorAxisLength	MinorAxisLength
256.5	256.5	834.46	834.46
300	120	81.759	81.759
330.47	369.83	111.78	110.36
450	240	101.72	101.72

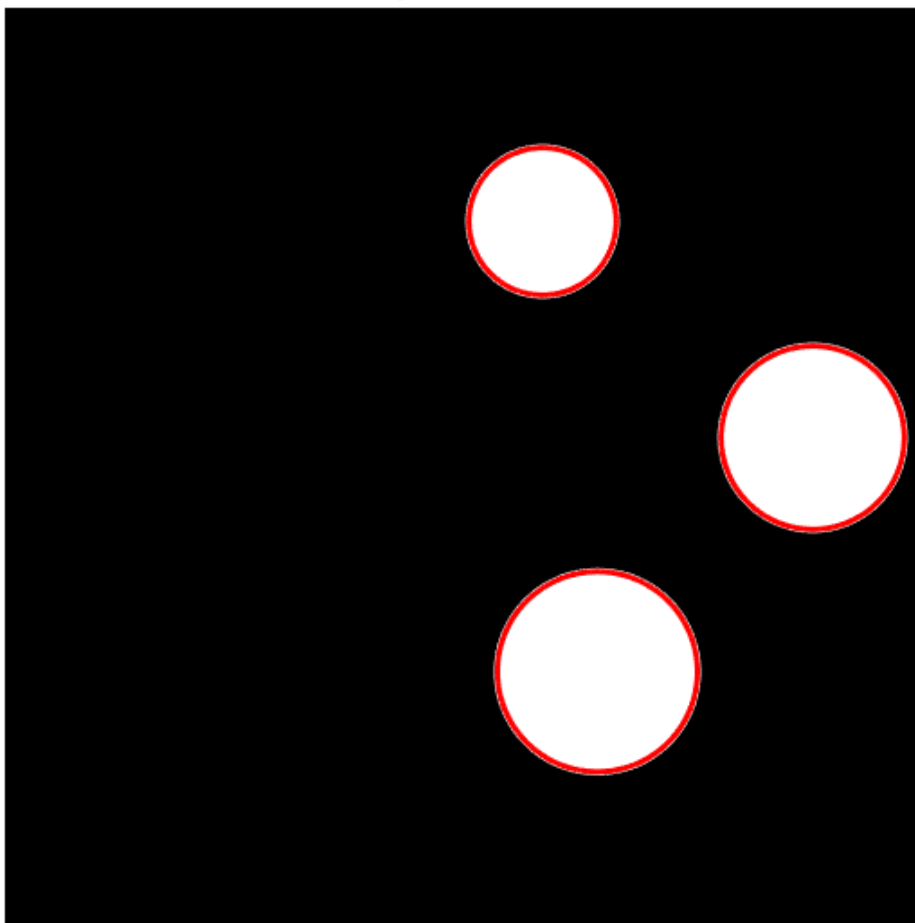
Get centers and radii of the circles.

```
centers = stats.Centroid;  
diameters = mean([stats.MajorAxisLength stats.MinorAxisLength],2);  
radii = diameters/2;
```

Plot the circles.

```
hold on  
viscircles(centers,radii);  
hold off
```

Image with Circles



Input Arguments

BW — Input binary image

logical array of any dimension

Input binary image, specified as a logical array of any dimension.

Data Types: `logical`

properties — Type of measurement

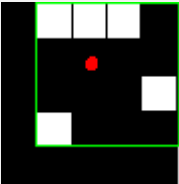
character vector | comma-separated list of character vectors | cell array of character vectors | 'all' | 'basic'

Type of measurement, specified as a character vector, comma-separated list of character vectors, cell array of character vectors, or 'all' or 'basic'. Property names are case-insensitive and can be abbreviated. When used with code generation, `regionprops` does not support cell arrays of character vectors.

The following tables list all the properties that provide shape measurements. The properties listed in the Pixel Value Measurements table are only valid when you specify a grayscale image. If you specify 'all', `regionprops` computes all the shape measurements and, if you specified a grayscale image, the pixel value measurements. If you specify 'basic', or do not specify the `properties` argument, `regionprops` computes only the 'Area', 'Centroid', and 'BoundingBox' measurements. You can calculate the following properties on N-D inputs: 'Area', 'BoundingBox', 'Centroid', 'FilledArea', 'FilledImage', 'Image', 'PixelIdxList', 'PixelList', and 'SubarrayIdx'.

Shape Measurements

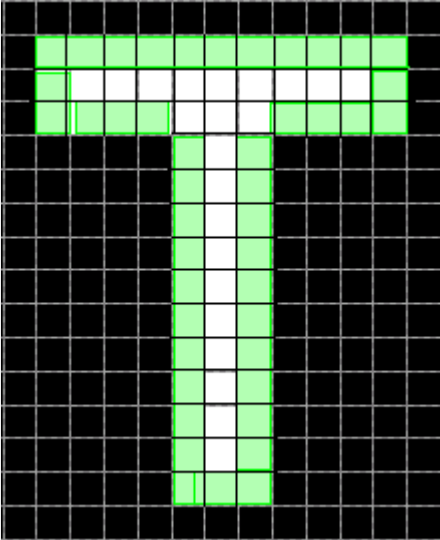
Property Name	Description	N-D Support	GPU Support	Code Generation
'Area'	Returns a scalar that specifies the actual number of pixels in the region. (This value might differ slightly from the value returned by <code>bwarea</code> , which weights different patterns of pixels differently.)	Yes	Yes	Yes
'BoundingBox'	Returns the smallest rectangle containing the region, specified as a 1-by-Q*2 vector, where Q is the number of image dimensions, for example, [<code>ul_corner width</code>]. <code>ul_corner</code> specifies the upper-left corner of the bounding box in the form [<code>x y z ...</code>]. <code>width</code>	Yes	Yes	Yes

Property Name	Description	N-D Support	GPU Support	Code Generation
	specifies the width of the bounding box along each dimension in the form [x_width y_width ...]. <code>regionprops</code> uses <code>ndims</code> to get the dimensions of label matrix or binary image, <code>ndims(L)</code> , and <code>numel</code> to get the dimensions of connected components, <code>numel(CC.ImageSize)</code> .			
'Centroid'	Returns a 1-by- <i>Q</i> vector that specifies the center of mass of the region. The first element of <code>Centroid</code> is the horizontal coordinate (or <i>x</i> -coordinate) of the center of mass, and the second element is the vertical coordinate (or <i>y</i> -coordinate). All other elements of <code>Centroid</code> are in order of dimension. This figure illustrates the centroid and bounding box for a discontinuous region. The region consists of the white pixels; the green box is the bounding box, and the red dot is the centroid. 	Yes	Yes	Yes
'ConvexArea'	Returns a scalar that specifies the number of pixels in 'ConvexImage'.	2-D only	No	No
'ConvexHull'	Returns a <i>p</i> -by-2 matrix that specifies the smallest convex polygon that can contain the region. Each row of the matrix contains the <i>x</i> - and <i>y</i> -coordinates of one vertex of the polygon.	2-D only	No	No
'ConvexImage'	Returns a binary image (logical) that specifies the convex hull, with all pixels within the hull filled in (set to on). The image is the size of the bounding box of the region. (For pixels that the boundary of the hull passes through, <code>regionprops</code> uses the same logic as <code>roipoly</code> to determine whether the pixel is inside or outside the hull.)	2-D only	No	No

Property Name	Description	N-D Support	GPU Support	Code Generation
'Eccentricity'	Returns a scalar that specifies the eccentricity of the ellipse that has the same second-moments as the region. The eccentricity is the ratio of the distance between the foci of the ellipse and its major axis length. The value is between 0 and 1. (0 and 1 are degenerate cases. An ellipse whose eccentricity is 0 is actually a circle, while an ellipse whose eccentricity is 1 is a line segment.)	2-D only	Yes	Yes
'EquivalentDiameter'	Returns a scalar that specifies the diameter of a circle with the same area as the region. Computed as $\sqrt{4*Area/\pi}$.	2-D only	Yes	Yes
'EulerNumber'	Returns a scalar that specifies the number of objects in the region minus the number of holes in those objects. This property is supported only for 2-D label matrices. <code>regionprops</code> uses 8-connectivity to compute the Euler number measurement. To learn more about connectivity, see “Pixel Connectivity”.	Yes	No	Yes
'Extent'	Returns a scalar that specifies the ratio of pixels in the region to pixels in the total bounding box. Computed as the <code>Area</code> divided by the area of the bounding box.	2-D only	Yes	Yes
'Extrema'	Returns an 8-by-2 matrix that specifies the extrema points in the region. Each row of the matrix contains the <i>x</i> - and <i>y</i> -coordinates of one of the points. The format of the vector is [top-left top-right right-top right-bottom bottom-right bottom-left left-bottom left-top]. This figure illustrates the extrema of two different regions. In the region on the left, each extrema point is distinct. In the region on the right, certain extrema points (e.g., top-left and left-top) are identical.	2-D only	Yes	Yes

Property Name	Description	N-D Support	GPU Support	Code Generation
'FilledArea'	Returns a scalar that specifies the number of on pixels in FilledImage.	Yes	No	Yes
'FilledImage'	Returns a binary image (logical) of the same size as the bounding box of the region. The on pixels correspond to the region, with all holes filled in, as shown in this figure. Original Image, Containing a Single Region	Yes	No	Yes
'Image'	Returns a binary image (logical) of the same size as the bounding box of the region. The on pixels correspond to the region, and all other pixels are off.	Yes	Yes	Yes

Property Name	Description	N-D Support	GPU Support	Code Generation
'MajorAxis	Returns a scalar that specifies the length (in pixels) of the major axis of the ellipse that has the same normalized second central moments as the region.	2-D only	Yes	Yes
'MinorAxis	Returns a scalar that specifies the length (in pixels) of the minor axis of the ellipse that has the same normalized second central moments as the region.	2-D only	Yes	Yes
'Orientation	Returns a scalar that specifies the angle between the x -axis and the major axis of the ellipse that has the same second-moments as the region. The value is in degrees, ranging from -90 to 90 degrees. This figure illustrates the axes and orientation of the ellipse. The left side of the figure shows an image region and its corresponding ellipse. The right side shows the same ellipse with the solid blue lines representing the axes, the red dots are the foci, and the orientation is the angle between the horizontal dotted line and the major axis. <div data-bbox="382 904 917 1090" data-label="Image"> </div>	2-D only	Yes	Yes
'Perimeter	Returns a scalar that specifies the distance around the boundary of the region. <code>regionprops</code> computes the perimeter by calculating the distance between each adjoining pair of pixels around the border of the region. If the image contains discontinuous regions, <code>regionprops</code> returns unexpected results. This figure illustrates the pixels included in the perimeter calculation for this object.	2-D only	Yes	Yes

Property Name	Description	N-D Support	GPU Support	Code Generation
				
'PixelIdx'	Returns a p -element vector that contains the linear indices of the pixels in the region.	Yes	Yes	Yes
'PixelList'	Returns a p -by- Q matrix that specifies the locations of pixels in the region. Each row of the matrix has the form $[x \ y \ z \ \dots]$ and specifies the coordinates of one pixel in the region.	Yes	Yes	Yes
'Solidity'	Returns a scalar specifying the proportion of the pixels in the convex hull that are also in the region. Computed as $\text{Area}/\text{ConvexArea}$.	2-D only	No	No
'Subarray'	Returns a cell array that contains indices such that $L(\text{idx}\{\ : \})$ extracts the elements of L inside the object bounding box.	Yes	Yes	No

The pixel value measurement properties in the following table are valid only when you specify a grayscale image, I .

Pixel Value Measurements

Property Name	Description	N-D Support	GPU Support	Code Generation
'MaxInten	Returns a scalar that specifies the value of the pixel with the greatest intensity in the region.	2-D only	Yes	Yes
'MeanInte	Returns a scalar that specifies the mean of all the intensity values in the region.	2-D only	Yes	Yes
'MinInten	Returns a scalar that specifies the value of the pixel with the lowest intensity in the region.	2-D only	Yes	Yes
'PixelVal	Returns a p -by-1 vector, where p is the number of pixels in the region. Each element in the vector contains the value of a pixel in the region.	2-D only	Yes	Yes
'Weighted	Returns a p -by- Q vector of coordinates specifying the center of the region based on location and intensity value. The first element of <code>WeightedCentroid</code> is the horizontal coordinate (or x -coordinate) of the weighted centroid. The second element is the vertical coordinate (or y -coordinate). All other elements of <code>WeightedCentroid</code> are in order of dimension.	2-D only	Yes	Yes

Data Types: char

CC — Connected components

structure

Connected components, specified as a structure returned by `bwconncomp`.

Data Types: struct

L — Labeled regions

label matrix

Labeled regions, specified as a label matrix. `L` can have any numeric class and any dimension. `regionprops` treats negative-valued pixels as background and rounds down input pixels that are not integers. Positive integer elements of `L` correspond to different regions. For example, the set of elements of `L` equal to 1 corresponds to region 1; the set of elements of `L` equal to 2 corresponds to region 2; and so on.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

I — Image to be measured

grayscale image

Image to be measured, specified as a grayscale image.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

output — Return type

'struct' (default) | 'table'

Return type, specified as either of the following values:

Value	Description
'struct'	Returns an array of structures with length equal to the number of objects in <code>BW</code> , <code>CC.NumObjects</code> , or <code>max(L(:))</code> . The fields of the structure array denote different properties for each region, as specified by <code>properties</code> . If you do not specify this argument, <code>regionprops</code> returns a struct by default.
'table'	Returns a MATLAB table with height (number of rows) equal to the number of objects in <code>BW</code> , <code>CC.NumObjects</code> , or <code>max(L(:))</code> . The variables (columns) denote different properties for each region, as specified by <code>properties</code> . To learn more about MATLAB tables, see <code>table</code> . Not supported on a GPU.

Data Types: `char`

gpuarrayImg — Input image

2D logical `gpuArray` | label matrix `gpuArray`

Input image, specified as a 2-D logical `gpuArray` or label matrix `gpuArray`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output Arguments

stats — Measurement values

struct array (default) | table

Measurement values, returned as an array of structs or a table. The number of structs in the array, or the number of rows in the table, corresponds to the number of objects in `BW`,

`CC.NumObjects`, or `max(L(:))`. The fields of each struct, or the variables in each row, denote the properties calculated for each region, as specified by `properties`.

When run on a GPU, `regionprops` can only return struct arrays.

Tips

- The function `ismember` is useful in conjunction with `regionprops`, `bwconncomp`, and `labelmatrix` for creating a binary image containing only objects or regions that meet certain criteria. For example, these commands create a binary image containing only the regions whose area is greater than 80 and whose eccentricity is less than 0.8.

```
cc = bwconncomp(BW);
stats = regionprops(cc, 'Area', 'Eccentricity');
idx = find([stats.Area] > 80 & [stats.Eccentricity] < 0.8);
BW2 = ismember(labelmatrix(cc), idx);
```

- The comma-separated list syntax for structure arrays is very useful when you work with the output of `regionprops`. For example, for a field that contains a scalar, you can use this syntax to create a vector containing the value of this field for each region in the image. For instance, if `stats` is a structure array with field `Area`, then the following expression:

```
stats(1).Area, stats(2).Area, ..., stats(end).Area
```

is equivalent to:

```
stats.Area
```

Therefore, you can use these calls to create a vector containing the area of each region in the image. `allArea` is a vector of the same length as the structure array `stats`.

```
stats = regionprops(L, 'Area');
allArea = [stats.Area];
```

- The functions `bwlabel`, `bwlabeln`, and `bwconncomp` all compute connected components for binary images. `bwconncomp` replaces the use of `bwlabel` and `bwlabeln`. It uses significantly less memory and is sometimes faster than the other functions.

Function	Input Dimension	Output Form	Memory Use	Connectivity
<code>bwlabel</code>	2-D	Label matrix with double-precision	High	4 or 8
<code>bwlabeln</code>	N-D	Double-precision label matrix	High	Any
<code>bwconncomp</code>	N-D	CC struct	Low	Any

The output of `bwlabel` and `bwlabeln` is a double-precision label matrix. To compute a label matrix using a more memory-efficient data type, use the `labelmatrix` function on the output of `bwconncomp`:

```
CC = bwconncomp(BW);
L = labelmatrix(CC);
```

If you are measuring components in a binary image with default connectivity, it is no longer necessary to call `bwlabel` or `bwlabeln` first. You can pass the binary image directly to `regionprops`, which then uses the memory-efficient `bwconncomp` function to compute the connected components automatically. If you need to specify nondefault connectivity, call `bwconncomp` and then pass the result to `regionprops`.

```
CC = bwconncomp(BW, CONN);
S = regionprops(CC);
```

- Most of the measurements take very little time to compute. However, these measurements can take significantly longer, depending on the number of regions in `L`:
 - 'ConvexHull'
 - 'ConvexImage'
 - 'ConvexArea'
 - 'FilledImage'
- Computing certain groups of measurements takes about the same amount of time as computing just one of them because `regionprops` takes advantage of intermediate computations used in both computations. Therefore, it is fastest to compute all the desired measurements in a single call to `regionprops`.

Algorithms

Contiguous regions are also called "objects," "connected components," or "blobs." A label matrix containing contiguous regions might look like this:

```
1 1 0 2 2 0 3 3
1 1 0 2 2 0 3 3
```

Elements of L equal to 1 belong to the first contiguous region or connected component; elements of L equal to 2 belong to the second connected component; and so on.

Discontiguous regions are regions that might contain multiple connected components. A label matrix containing discontiguous regions might look like this:

```
1 1 0 1 1 0 2 2
1 1 0 1 1 0 2 2
```

Elements of L equal to 1 belong to the first region, which is discontiguous and contains two connected components. Elements of L equal to 2 belong to the second region, which is a single connected component.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.
- Supports only 2-D input images or label matrices.
- Specifying the output type `'table'` is not supported.
- Passing a cell array of properties is not supported. Use a comma-separated list instead.

- All properties are supported except 'ConvexArea', 'ConvexHull', 'ConvexImage', 'Solidity', and 'SubarrayIdx'.

See Also

See Also

`bwconncomp` | `bwlabel` | `bwlabeln` | `ismember` | `labelmatrix` | `watershed`

Introduced before R2006a

registration.metric.MattesMutualInformation class

Package: registration.metric

Mattes mutual information metric configuration object

Description

A `MattesMutualInformation` object describes a mutual information metric configuration that you pass to the function `imregister` to solve image registration problems.

You can create a `MattesMutualInformation` object using the following methods:

- `imregconfig` — Returns a `MattesMutualInformation` object for registering multimodal images
- The `MattesMutualInformation` class constructor

Construction

`metric = registration.metric.MattesMutualInformation()` constructs a `MattesMutualInformation` object.

Properties

NumberOfSpatialSamples — Number of spatial samples used to compute the mutual information metric

500 (default) | positive integer scalar

Number of spatial samples used to compute the mutual information metric, specified as a positive integer scalar. `NumberOfSpatialSamples` defines the number of random pixels `imregister` uses to compute the metric. Your registration results are more reproducible (at the cost of performance) as you increase this value. `imregister` only uses `NumberOfSpatialSamples` when `UseAllPixels = 0` (false).

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

NumberOfHistogramBins — Number of histogram bins used to compute the mutual information metric

50 (default) | positive integer scalar

Number of histogram bins used to compute the mutual information metric, specified as a positive integer scalar. `NumberOfHistogramBins` defines the number of bins `imregister` uses to compute the joint distribution histogram. The minimum value is 5.

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

UseAllPixels — Option to include all pixels in the overlap region when computing the mutual information metric

1 (true) (default) | logical scalar

Option to compute the metric using all pixels in the overlap region of the images when computing the mutual information metric, specified as a logical scalar.

You can achieve significantly better performance if you set this property to 0 (false). When `UseAllPixels = 0`, the `NumberOfSpatialSamples` property controls the number of random pixel locations that `imregister` uses to compute the metric. The results of your registration might not be reproducible when `UseAllPixels = 0`. This is because `imregister` selects a random subset of pixels from the images to compute the metric.

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Register Images with Mattes Mutual Information Metric

Create a `MattesMutualInformation` object and use it to register two MRI images of a knee that were obtained using different protocols.

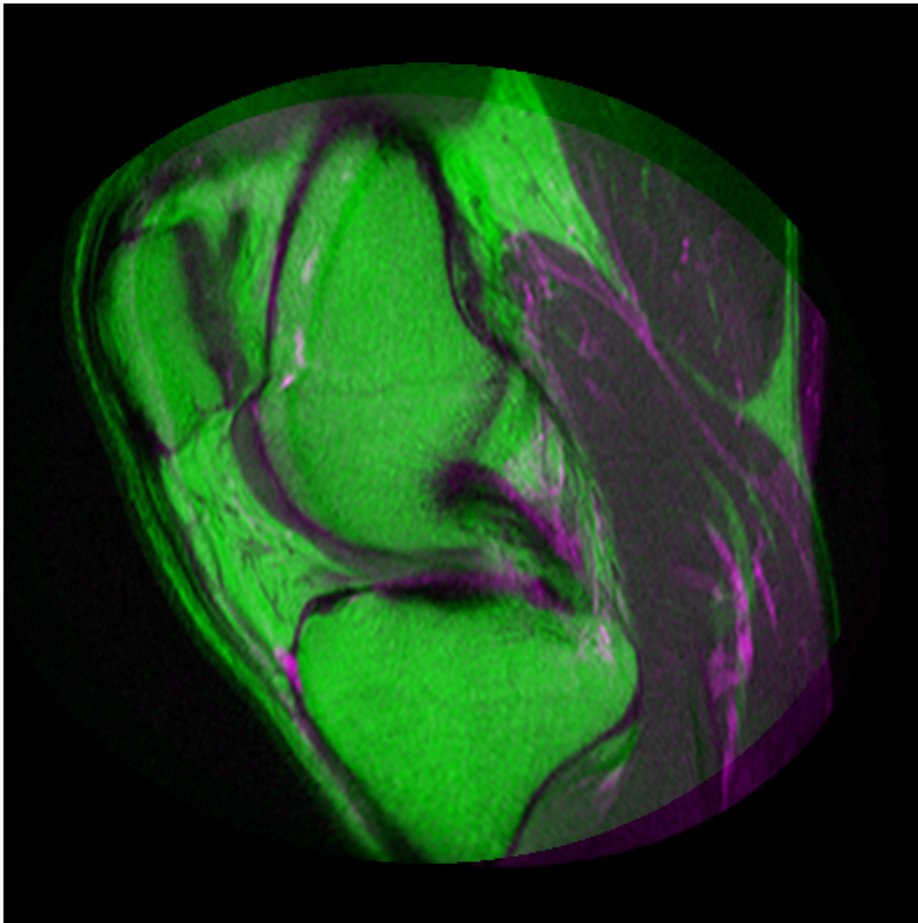
Read the images into the workspace. The images are multimodal because they have different brightness and contrast.

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

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

View the misaligned images.

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



Create the optimizer configuration object suitable for registering multimodal images.

```
optimizer = registration.optimizer.OnePlusOneEvolutionary;
```

Create the metric configuration object suitable for registering multimodal images.

```
metric = registration.metric.MattesMutualInformation
```

```
metric =  
    registration.metric.MattesMutualInformation
```

```
    Properties:
```

```
        NumberOfSpatialSamples: 500
```

```
        NumberOfHistogramBins: 50
```

```
        UseAllPixels: 1
```

Tune the properties of the optimizer so that the problem will converge on a global maxima. Increase the number of iterations the optimizer will use to solve the problem.

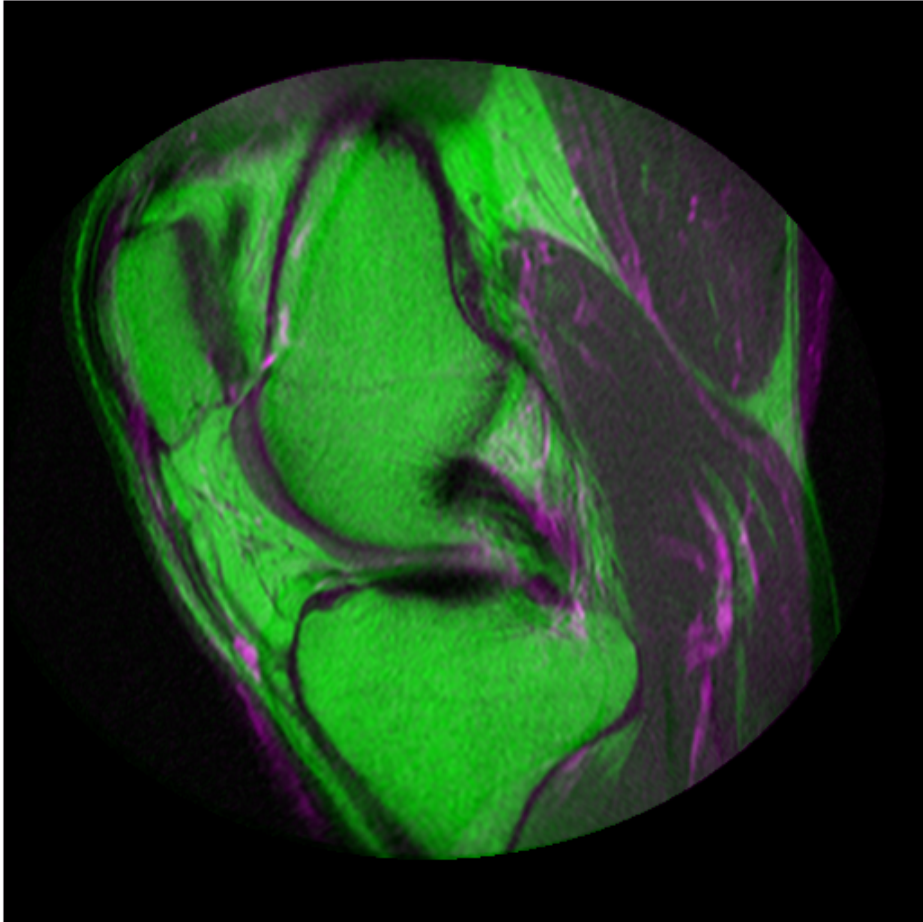
```
optimizer.InitialRadius = 0.009;  
optimizer.Epsilon = 1.5e-4;  
optimizer.GrowthFactor = 1.01;  
optimizer.MaximumIterations = 300;
```

Perform the registration.

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

View the registered images.

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



Definitions

Mutual Information Metric

Metric used to maximize the number of coincident pixels with the same relative brightness value. This metric is best suited for images with different brightness ranges.

Tips

- Larger values of mutual information correspond to better registration results. You can examine the computed values of Mattes mutual information if you enable 'DisplayOptimization' when you call `imregister`, for example:

```
movingRegistered = imregister(moving, fixed, 'rigid', optimizer, metric, 'DisplayOptimization', true);
```

Algorithms

Mutual information metrics are information theoretic techniques for measuring how related two variables are. These algorithms use the joint probability distribution of a sampling of pixels from two images to measure the certainty that the values of one set of pixels map to similar values in the other image. This information is a quantitative measure of how similar the images are. High mutual information implies a large reduction in the uncertainty (entropy) between the two distributions, signaling that the images are likely better aligned.

The Mattes mutual information algorithm uses a single set of pixel locations for the duration of the optimization, instead of drawing a new set at each iteration. The number of samples used to compute the probability density estimates and the number of bins used to compute the entropy are both user selectable. The marginal and joint probability density function is evaluated at the uniformly spaced bins using the samples. Entropy values are computed by summing over the bins. Zero-order and third-order B-spline kernels are used to compute the probability density functions of the fixed and moving images, respectively.[1]

References

- [1] Rahunathan, Smriti, D. Stredney, P. Schmalbrock, and B.D. Clymer. Image Registration Using Rigid Registration and Maximization of Mutual Information.

Poster presented at: MMVR13. The 13th Annual Medicine Meets Virtual Reality Conference; 2005 January 26–29; Long Beach, CA.

[2] D. Mattes, D.R. Haynor, H. Vesselle, T. Lewellen, and W. Eubank. "Non-rigid multimodality image registration." (Proceedings paper). *Medical Imaging 2001: Image Processing*. SPIE Publications, 3 July 2001. pp. 1609–1620.

See Also

See Also

Functions

`imregconfig` | `imregister`

Classes

`registration.metric.MeanSquares` | `registration.optimizer.OnePlusOneEvolutionary` | `registration.optimizer.RegularStepGradientDescent`

Topics

“Create an Optimizer and Metric for Intensity-Based Image Registration”

Class Attributes (MATLAB)

Property Attributes (MATLAB)

registration.metric.MeanSquares class

Package: registration.metric

Mean square error metric configuration object

Description

A `MeanSquares` object describes a mean square error metric configuration that you pass to the function `imregister` to solve image registration problems.

You can create a `MeanSquares` object using the following methods:

- `imregconfig` — Returns a `MeanSquares` object for registering monomodal images
- The `MeanSquares` class constructor

Construction

`metric = registration.metric.MeanSquares()` constructs a `MeanSquares` object.

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Register Images with Mean Squares Metric

Create a `MeanSquares` object and use it to register two images with similar brightness and contrast.

Read the reference image and create an unregistered copy.

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

```
moving = imrotate(fixed, 5, 'bilinear', 'crop');
```

View the misaligned images.

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



Create the metric configuration object suitable for registering monomodal images.

```
metric = registration.metric.MeanSquares
```

```
metric =  
    registration.metric.MeanSquares
```

This class has no properties.

Create the optimizer configuration object.

```
optimizer = registration.optimizer.RegularStepGradientDescent;
```

Modify the metric configuration to get more precision.

```
optimizer.MaximumIterations = 300;  
optimizer.MinimumStepLength = 5e-4;
```

Perform the registration.

```
movingRegistered = imregister(moving, fixed, 'rigid', optimizer, metric);
```

View the registered images.

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



Tips

- The mean squares metric is an element-wise difference between two input images. The ideal value is zero. You can examine the computed values of mean square error if you enable 'DisplayOptimization' when you call `imregister`. For example, `movingRegistered = imregister(moving, fixed, 'rigid', optimizer, metric, 'Display`

Algorithms

The mean squares image similarity metric is computed by squaring the difference of corresponding pixels in each image and taking the mean of the those squared differences.

See Also

See Also

Functions

`imregconfig` | `imregister`

Classes

`registration.metric.MattesMutualInformation` |
`registration.optimizer.OnePlusOneEvolutionary` |
`registration.optimizer.RegularStepGradientDescent`

Topics

“Create an Optimizer and Metric for Intensity-Based Image Registration”

Class Attributes (MATLAB)

Property Attributes (MATLAB)

registration.optimizer.OnePlusOneEvolutionary class

Package: registration.optimizer

One-plus-one evolutionary optimizer configuration object

Description

A `OnePlusOneEvolutionary` object describes a one-plus-one evolutionary optimization configuration that you pass to the function `imregister` to solve image registration problems.

You can create a `OnePlusOneEvolutionary` object using the following methods:

- `imregconfig` — Returns a `OnePlusOneEvolutionary` object for registering multimodal images
- The `OnePlusOneEvolutionary` class constructor

Construction

`optimizer = registration.optimizer.OnePlusOneEvolutionary()` Constructs a `OnePlusOneEvolutionary` object.

Properties

GrowthFactor — Growth factor of the search radius

1.05 (default) | positive scalar

Growth factor of the search radius, specified as a positive scalar. The optimizer uses `GrowthFactor` to control the rate at which the search radius grows in parameter space. If you set `GrowthFactor` to a large value, the optimization is fast, but it might result in finding only the metric's local extrema. If you set `GrowthFactor` to a small value, the optimization is slower, but it is likely to converge on a better solution.

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

Epsilon — Minimum size of the search radius

1.5e-6 (default) | positive scalar

Minimum size of the search radius, specified as a positive scalar. `Epsilon` controls the accuracy of convergence by adjusting the minimum size of the search radius. If you set `Epsilon` to a small value, the optimization of the metric is more accurate, but the computation takes longer. If you set `Epsilon` to a large value, the computation time decreases at the expense of accuracy.

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

InitialRadius — Initial size of search radius

6.25e-3 | positive scalar

Initial size of search radius, specified as a positive scalar. If you set `InitialRadius` to a large value, the computation time decreases. However, overly large values of `InitialRadius` might result in an optimization that fails to converge.

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

MaximumIterations — Maximum number of optimizer iterations

100 (default) | positive integer scalar

Maximum number of optimizer iterations, specified as a positive integer scalar. `MaximumIterations` determines the maximum number of iterations the optimizer performs at any given pyramid level. The registration could converge before the optimizer reaches the maximum number of iterations.

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Register Images with One Plus One Evolutionary Optimizer

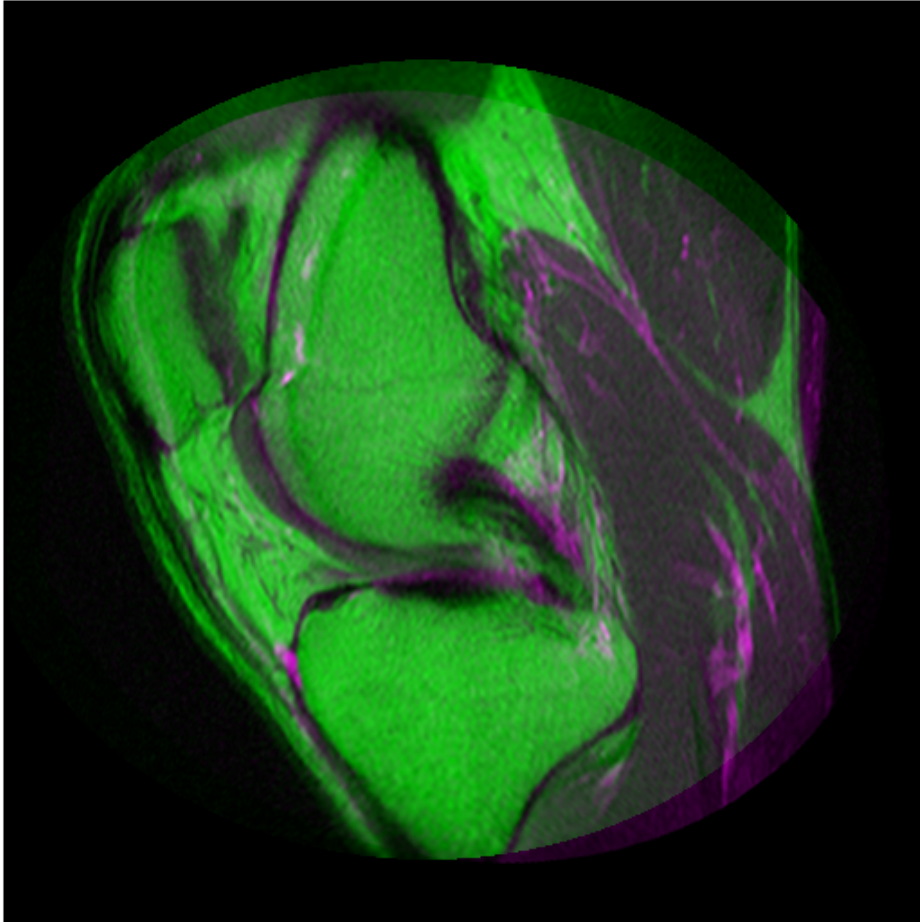
Create a `OnePlusOneEvolutionary` object and use it to register two MRI images of a knee that were obtained using different protocols.

Read the images into the workspace. The images are multimodal because they have different brightness and contrast.

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

View the misaligned images.

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



Create the optimizer configuration object suitable for registering multimodal images.

```
optimizer = registration.optimizer.OnePlusOneEvolutionary
```

```
optimizer =  
    registration.optimizer.OnePlusOneEvolutionary
```

```
Properties:
  GrowthFactor: 1.050000e+00
  Epsilon: 1.500000e-06
  InitialRadius: 6.250000e-03
  MaximumIterations: 100
```

Create the metric configuration object suitable for registering multimodal images.

```
metric = registration.metric.MattesMutualInformation;
```

Tune the properties of the optimizer so that the problem will converge on a global maxima. Increase the number of iterations the optimizer will use to solve the problem.

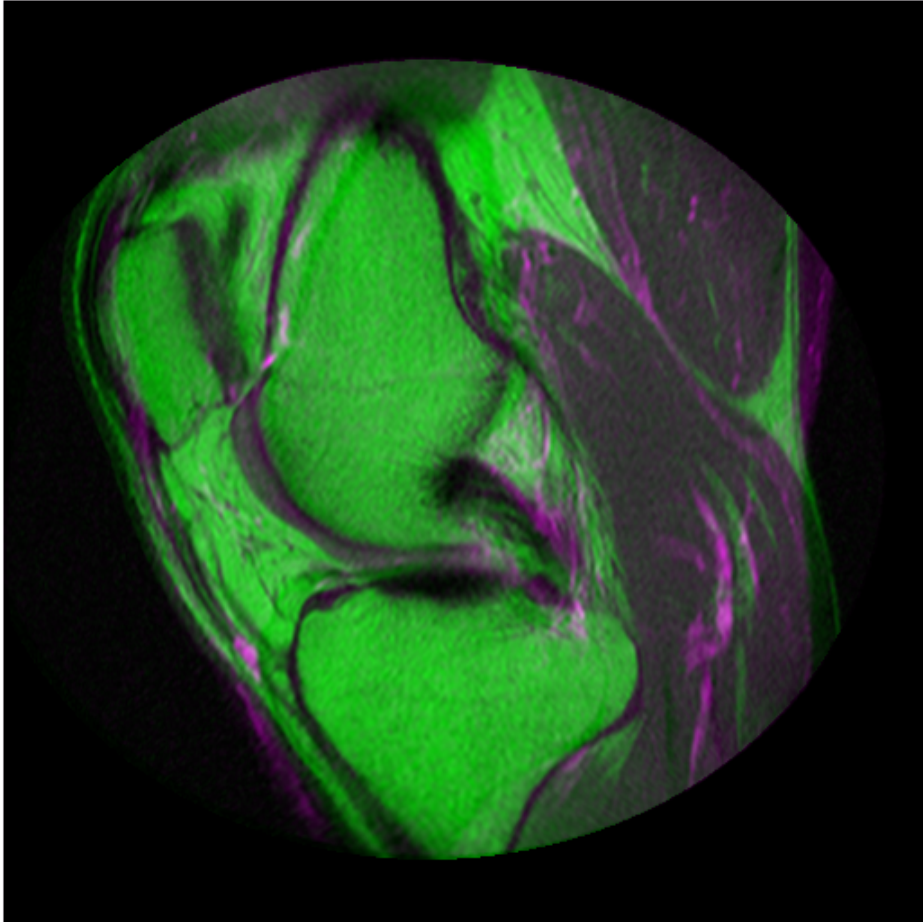
```
optimizer.InitialRadius = 0.009;
optimizer.Epsilon = 1.5e-4;
optimizer.GrowthFactor = 1.01;
optimizer.MaximumIterations = 300;
```

Perform the registration.

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

View the registered images.

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



Algorithms

An evolutionary algorithm iterates to find a set of parameters that produce the best possible registration result. It does this by perturbing, or mutating, the parameters from the last iteration (the parent). If the new (child) parameters yield a better result, then the child becomes the new parent whose parameters are perturbed, perhaps more aggressively. If the parent yields a better result, it remains the parent and the next perturbation is less aggressive.

References

- [1] Styner, M., C. Brechbuehler, G. Székely, and G. Gerig. "Parametric estimate of intensity inhomogeneities applied to MRI." *IEEE Transactions on Medical Imaging*. Vol. 19, Number 3, 2000, pp. 153-165.

See Also

See Also

Functions

`imregconfig` | `imregister`

Classes

`registration.metric.MattesMutualInformation` | `registration.metric.MeanSquares` | `registration.optimizer.RegularStepGradientDescent`

Topics

“Create an Optimizer and Metric for Intensity-Based Image Registration”

Class Attributes (MATLAB)

Property Attributes (MATLAB)

registration.optimizer.RegularStepGradientDescent class

Package: registration.optimizer

Regular step gradient descent optimizer configuration object

Description

A `RegularStepGradientDescent` object describes a regular step gradient descent optimization configuration that you pass to the function `imregister` to solve image registration problems.

You can create a `RegularStepGradientDescent` object using the following methods:

- `imregconfig` — Returns a `RegularStepGradientDescent` object for registering monomodal images
- The `RegularStepGradientDescent` class constructor

Construction

`optimizer = registration.optimizer.RegularStepGradientDescent()`
constructs a `RegularStepGradientDescent` object.

Properties

GradientMagnitudeTolerance — Gradient magnitude tolerance

1e-4 (default) | positive scalar

Gradient magnitude tolerance, specified as a positive scalar.

`GradientMagnitudeTolerance` controls the optimization process. When the value of the gradient is smaller than `GradientMagnitudeTolerance`, it is an indication that the optimizer might have reached a plateau.

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

MinimumStepLength — Tolerance for convergence

1e-5 (default) | positive scalar

Tolerance for convergence, specified as a positive scalar. `MinimumStepLength` controls the accuracy of convergence. If you set `MinimumStepLength` to a small value, the optimization takes longer to compute, but it is likely to converge on a more accurate metric value.

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

MaximumStepLength — Initial step length

0.0625 (default) | positive scalar

Initial step length, specified as a positive scalar. The initial step length is the maximum step length because the optimizer reduces the step size during convergence. If you set `MaximumStepLength` to a large value, the computation time decreases. However, the optimizer might fail to converge if you set `MaximumStepLength` to an overly large value.

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

MaximumIterations — Maximum number of iterations

100 (default) | positive integer scalar

Maximum number of iterations, specified as a positive integer scalar. `MaximumIterations` is a positive scalar integer value that determines the maximum number of iterations the optimizer performs at any given pyramid level. The registration could converge before the optimizer reaches the maximum number of iterations.

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

RelaxationFactor — Step length reduction factor

0.5 (default) | positive scalar between 0 and 1

Step length reduction factor, specified as a positive scalar between 0 and 1. `RelaxationFactor` defines the rate at which the optimizer reduces step size during convergence. Whenever the optimizer determines that the direction of the gradient changed, it reduces the size of the step length. If your metric is noisy, you can set `RelaxationFactor` to a larger value. This leads to a more stable convergence at the expense of computation time.

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

Examples

Register Images with Regular Step Gradient Descent Optimizer

Create a `RegularStepGradientDescent` object and use it to register two images with similar brightness and contrast.

Read the reference image and create an unregistered copy.

```
fixed = imread('pout.tif');  
moving = imrotate(fixed, 5, 'bilinear', 'crop');
```

View the misaligned images.

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




Create the optimizer configuration object suitable for registering monomodal images.

```
optimizer = registration.optimizer.RegularStepGradientDescent
```

```
optimizer =  
    registration.optimizer.RegularStepGradientDescent
```

Properties:

```
    GradientMagnitudeTolerance: 1.000000e-04  
    MinimumStepLength: 1.000000e-05  
    MaximumStepLength: 6.250000e-02  
    MaximumIterations: 100  
    RelaxationFactor: 5.000000e-01
```

Create the metric configuration object.

```
metric = registration.metric.MeanSquares;
```

Modify the optimizer configuration to get more precision.

```
optimizer.MaximumIterations = 300;  
optimizer.MinimumStepLength = 5e-4;
```

Perform the registration.

```
movingRegistered = imregister(moving, fixed, 'rigid', optimizer, metric);
```

View the registered images.

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



Algorithms

The regular step gradient descent optimization adjusts the transformation parameters so that the optimization follows the gradient of the image similarity metric in the direction of the extrema. It uses constant length steps along the gradient between computations until the gradient changes direction. At this point, the step length is reduced based on the `RelaxationFactor`, which halves the step length by default.

See Also

See Also

Functions

`imregconfig` | `imregister`

Classes

`registration.metric.MattesMutualInformation` | `registration.metric.MeanSquares` | `registration.optimizer.OnePlusOneEvolutionary`

Topics

“Create an Optimizer and Metric for Intensity-Based Image Registration”

Class Attributes (MATLAB)

Property Attributes (MATLAB)

rgb2lab

Convert RGB to CIE 1976 L*a*b*

Syntax

```
lab = rgb2lab(rgb)
lab = rgb2lab(rgb,Name,Value)
```

Description

`lab = rgb2lab(rgb)` converts RGB values to CIE 1976 L*a*b* values.

`lab = rgb2lab(rgb,Name,Value)` specifies additional options with one or more Name,Value pair arguments.

Examples

Convert RGB White to L*a*b*

Use `rgb2lab` to convert the RGB white value to L*a*b*.

```
rgb2lab([1 1 1])
```

```
ans =
```

```
    100     0     0
```

Convert Color Value to L*a*b* Specifying Color Space

Convert an Adobe RGB (1998) color value to L*a*b* using the `ColorSpace` parameter.

```
rgb2lab([.2 .3 .4], 'ColorSpace', 'adobe-rgb-1998')
```

```
ans =
```

```
30.1783   -5.6902  -20.8223
```

Convert RGB color to L*a*b* Specifying Reference White

Use `rgb2lab` to convert an RGB color to L*a*b* using the D50 reference white.

```
rgb2lab([.2 .3 .4], 'WhitePoint', 'd50')
```

```
ans =
```

```
31.3294   -4.0732  -18.1750
```

Convert RGB Image to L*a*b* and Display L* Component

Read RGB image into the workspace.

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

Convert the RGB image to the L*a*b* color space.

```
lab = rgb2lab(rgb);
```

Display the L* component of the L*a*b* image.

```
imshow(lab(:,:,1), [0 100])
```



Input Arguments

rgb — Color values to convert

p-by-3 matrix | m-by-n-by-3 image array | m-by-n-by-3-by-f image stack

Color values to convert, specified as a p-by-3 matrix of color values (one color per row), an m-by-n-by-3 image array, or an m-by-n-by-3-by-f image stack.

Data Types: single | double | uint8 | uint16

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `rgb2lab([0.25 0.40 0.10], 'WhitePoint', 'd50')`

'ColorSpace' — Color space of the input RGB values

'srgb' (default) | 'adobe-rgb-1998' | 'linear-rgb'

Color space of the input RGB values, specified as 'srgb', 'adobe-rgb-1998', or 'linear-rgb'.

Data Types: char

'WhitePoint' — Reference white point

'd65' (default) | 'a' | 'c' | 'e' | 'd50' | 'd55' | 'icc' | 1-by-3 vector

Reference white point, specified as a 1-by-3 vector or one of the CIE standard illuminants, listed in the following table.

Value	White Point
'a'	CIE standard illuminant A, [1.0985, 1.0000, 0.3558]. Simulates typical, domestic, tungsten-filament lighting with correlated color temperature of 2856 K.
'c'	CIE standard illuminant C, [0.9807, 1.0000, 1.1822]. Simulates average or north sky daylight with correlated color temperature of 6774 K. Deprecated by CIE.
'e'	Equal-energy radiator. Useful as a theoretical reference. [1.000, 1.000, 1.000].
'd50'	CIE standard illuminant D50, [0.9642, 1.0000, 0.8251]. Simulates warm daylight at sunrise or sunset with correlated color temperature of 5003 K. Also known as <i>horizon light</i> .
'd55'	CIE standard illuminant D55, [0.9568, 1.0000, 0.9214]. Simulates mid-morning or mid-afternoon daylight with correlated color temperature of 5500 K.
'd65'	CIE standard illuminant D65, [0.9504, 1.0000, 1.0888]. Simulates noon daylight with correlated color temperature of 6504 K.

Value	White Point
'icc'	Profile Connection Space (PCS) illuminant used in ICC profiles. Approximation of [0.962, 1.000, 0.8249] using fixed-point, signed, 32-bit numbers with 16 fractional bits. Actual value: [31595, 32768, 27030]/32768.

Data Types: `single` | `double` | `char`

Output Arguments

lab — Converted color values

numeric array

Converted color values, returned as an array the same shape as the input. The output type is `double` unless the input type is `single`, in which case the output type is also `single`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- When generating code, all character vector input arguments must be compile-time constants.

See Also

See Also

`lab2rgb` | `lab2xyz` | `rgb2xyz` | `xyz2lab` | `xyz2rgb`

Introduced in R2014b

rgb2ntsc

Convert RGB color values to NTSC color space

Syntax

```
YIQmap = rgb2ntsc(rgbmap)  
YIQ = rgb2ntsc(RGB)
```

Description

`YIQmap = rgb2ntsc(rgbmap)` converts the m -by-3 RGB values in `rgbmap` to NTSC color space. `YIQmap` is an m -by-3 matrix that contains the NTSC luminance (Y) and chrominance (I and Q) color components as columns that are equivalent to the colors in the RGB colormap.

`YIQ = rgb2ntsc(RGB)` converts the truecolor image `RGB` to the equivalent NTSC image `YIQ`.

Class Support

RGB can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. `RGBMAP` can be `double`. The output is `double`.

Examples

Convert Image from RGB to YIQ

This example shows how to convert an image from RGB to NTSC color space.

Read an RGB image into the workspace.

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

Convert the image to YIQ color space.

```
YIQ = rgb2ntsc(RGB);
```

Display the NTSC luminance value, represented by the first color channel in the YIQ image.

```
imshow(YIQ(:,:,1));  
title('Luminance in YIQ Color Space');
```


Tips

In the NTSC color space, the luminance is the grayscale signal used to display pictures on monochrome (black and white) televisions. The other components carry the hue and saturation information.

`rgb2ntsc` defines the NTSC components using

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

See Also

`ntsc2rgb` | `rgb2ind` | `ind2rgb` | `ind2gray`

Introduced before R2006a

rgb2xyz

Convert RGB to CIE 1931 XYZ

Syntax

```
xyz = rgb2xyz(rgb)  
xyz = rgb2xyz(rgb,Name,Value)
```

Description

`xyz = rgb2xyz(rgb)` converts RGB values to CIE 1931 XYZ values.

`xyz = rgb2xyz(rgb,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments.

Examples

Convert RGB to XYZ

Convert images and color values from RGB to CIE 1931 XYZ color space.

Convert RGB Image to XYZ.

Read an RGB image into the workspace

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

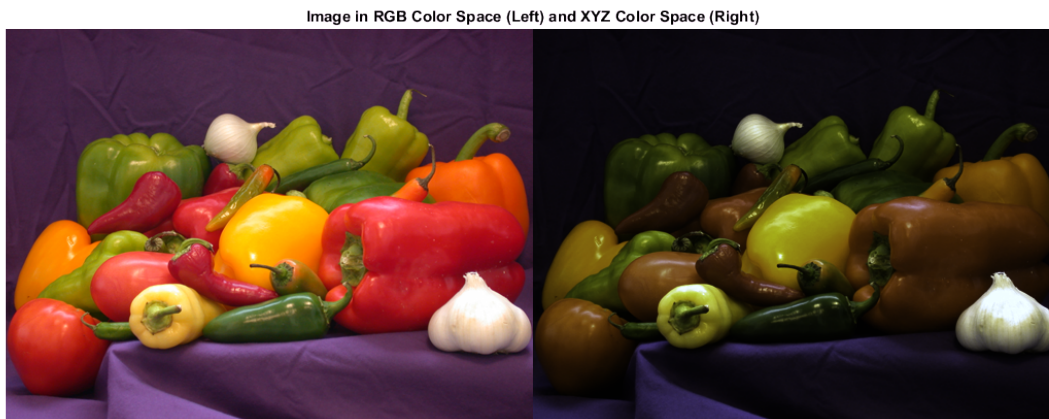
Convert the image to XYZ color space.

```
XYZ = rgb2xyz(RGB);
```

Display the original image alongside the new image.

```
figure  
imshowpair(RGB,XYZ,'montage');
```

```
title('Image in RGB Color Space (Left) and XYZ Color Space (Right)');
```



Convert RGB Color Value to XYZ

Convert the value of white from RGB to XYZ color space. In RGB, white is represented by the vector $[1 \ 1 \ 1]$.

```
rgb2xyz([1 1 1])
```

```
ans =
```

```
0.9505    1.0000    1.0888
```

Convert RGB Color to XYZ using D50 as Reference White

```
XYZ_D50 = rgb2xyz(RGB, 'WhitePoint', 'd50');
```

Display the first output XYZ image alongside the XYZ image with D50 as reference white.

```
figure
```

```
imshowpair(XYZ,XYZ_D50,'montage');
```

```
title('XYZ Image, Without (Left) and With (Right) Reference White');
```

XYZ Image, Without (Left) and With (Right) Reference White



Convert Adobe RGB (1998) Color to XYZ

```
XYZ_Adobe = rgb2xyz(RGB, 'ColorSpace', 'adobe-rgb-1998');
```

Display the XYZ images generated from the default RGB and the Adobe RGB (1998) color spaces.

```
figure
imshowpair(XYZ,XYZ_Adobe,'montage');
title(['XYZ Image, Starting From Default RGB (Left) and Adobe RGB ',...
      '(Right) Color Space']);
```


XYZ Image, Starting From Default RGB (Left) and Adobe RGB (Right) Color Space



Input Arguments

rgb — Color values to convert

p-by-3 matrix | m-by-n-by-3 image array | m-by-n-by-3-by-f image stack

Color values to convert, specified as a p-by-3 matrix of color values (one color per row), an m-by-n-by-3 image array, or an m-by-n-by-3-by-f image stack.

Example: `rgb2xyz([1 1 1])`

Data Types: `single` | `double` | `uint8` | `uint16`

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example: `rgb2xyz([.2 .3 .4], 'WhitePoint', 'd50')`

'ColorSpace' — Color space of the input RGB values

'srgb' (default) | 'adobe-rgb-1998' | 'linear-rgb'

Color space of the input RGB values, specified as 'srgb', 'adobe-rgb-1998', or 'linear-rgb'.

Data Types: char

'WhitePoint' — Reference white point

'd65' (default) | 'a' | 'c' | 'e' | 'd50' | 'd55' | 'icc' | 1-by-3 vector

Reference white point, specified as a 1-by-3 vector or one of the CIE standard illuminants, listed in the following table.

Value	White Point
'a'	CIE standard illuminant A, [1.0985, 1.0000, 0.3558]. Simulates typical, domestic, tungsten-filament lighting with correlated color temperature of 2856 K.
'c'	CIE standard illuminant C, [0.9807, 1.0000, 1.1822]. Simulates average or north sky daylight with correlated color temperature of 6774 K. Deprecated by CIE.
'e'	Equal-energy radiator. Useful as a theoretical reference. [1.000, 1.000, 1.000].
'd50'	CIE standard illuminant D50, [0.9642, 1.0000, 0.8251]. Simulates warm daylight at sunrise or sunset with correlated color temperature of 5003 K. Also known as <i>horizon light</i> .
'd55'	CIE standard illuminant D55, [0.9568, 1.0000, 0.9214]. Simulates mid-morning or mid-afternoon daylight with correlated color temperature of 5500 K.
'd65'	CIE standard illuminant D65, [0.9504, 1.0000, 1.0888]. Simulates noon daylight with correlated color temperature of 6504 K.
'icc'	Profile Connection Space (PCS) illuminant used in ICC profiles. Approximation of [0.962, 1.000, 0.8249] using fixed-point, signed, 32-bit numbers with 16 fractional bits. Actual value: [31595, 32768, 27030] / 32768.

Data Types: single | double | char

Output Arguments

xyz — Converted color values

array the same shape as the input

Converted color values, returned as an array the same shape as the input. The output type is class `double` unless the input type is `single`, in which case the output type is also `single`.

See Also

See Also

`lab2rgb` | `lab2xyz` | `rgb2lab` | `xyz2lab` | `xyz2rgb`

Introduced in R2014b

rgb2ycbcr

Convert RGB color values to YCbCr color space

Syntax

```
ycbcrmap = rgb2ycbcr(rgbmap)
YCBCR = rgb2ycbcr(RGB)
gpuarrayB = rgb2ycbcr(gpuarrayA)
```

Description

`ycbcrmap = rgb2ycbcr(rgbmap)` converts the RGB color space values in `rgbmap` to the YCbCr color space. `ycbcrmap` is an `m`-by-3 matrix that contains the YCbCr luminance (*Y*) and chrominance (*Cb* and *Cr*) color values as columns. Each row in `ycbcrmap` represents the equivalent color to the corresponding row in `rgbmap`.

`YCBCR = rgb2ycbcr(RGB)` converts the truecolor image `RGB` to the equivalent image in YCbCr color space.

`gpuarrayB = rgb2ycbcr(gpuarrayA)` performs the conversion on a GPU. The input image, `gpuarrayA`, is a `gpuArray` containing RGB color space values or an RGB image. The output is a `gpuArray` containing YCbCr color space values or a YCbCr image, depending on the input type. This syntax requires the Parallel Computing Toolbox.

Examples

Convert RGB to YCbCr

Convert Image from RGB to YCbCr.

Read an RGB image into the workspace

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

Convert the image to YCbCr.

```
YCBCR = rgb2ycbcr(RGB);
```

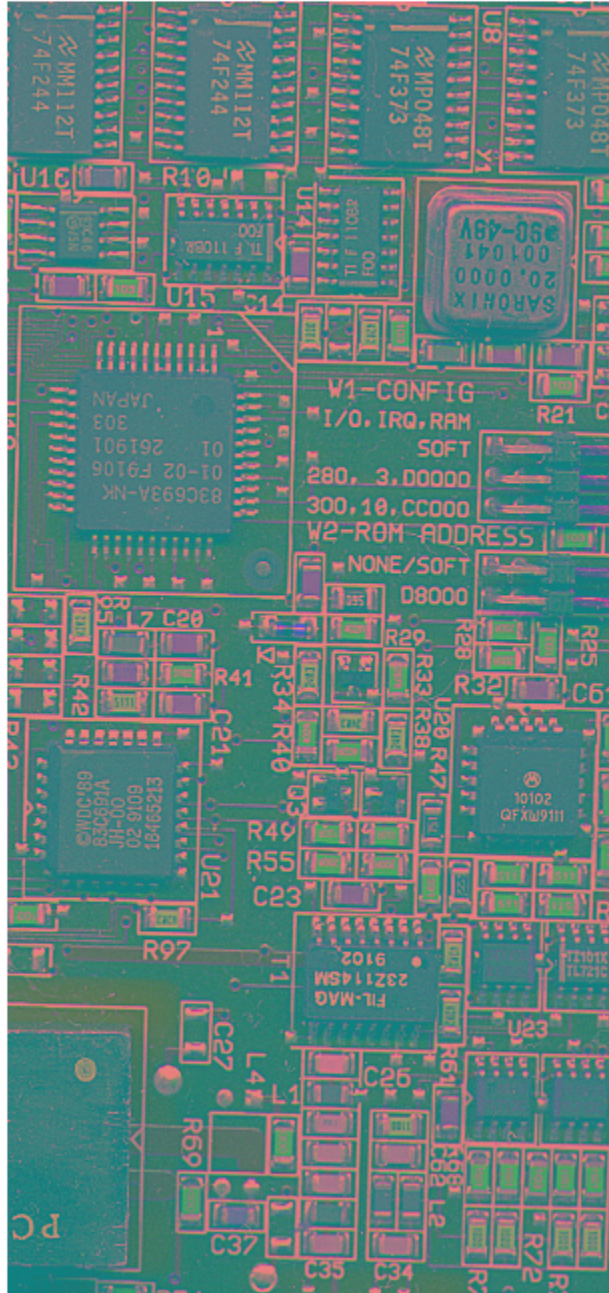
Display the original image and the new image

```
figure  
imshow(RGB);  
title('Image in RGB Color Space');
```



```
figure
imshow(YCBCR);
title('Image in YCbCr Color Space');
```

Image in YCbCr Color Space



Convert Colormap from RGB to YCbCr.

Load an indexed image into the workspace. The colormap is in RGB colorspace.

```
[I,map] = imread('forest.tif');
```

Convert the colormap to YCbCr.

```
newmap = rgb2ycbcr(map);
```

Display the grayscale image with the original map and with the new map.

```
figure  
imshow(I,map)  
title('Indexed Image with RGB Colormap');
```

Indexed Image with RGB Colormap



```
figure  
imshow(I,newmap)
```

```
title('Indexed Image with YCbCr Colormap');
```

Indexed Image with YCbCr Colormap



Input Arguments

rgbmap — RGB color space values

m-by-3 array

RGB color space values, specified as an m-by-3 array.

Data Types: `single` | `double`

RGB — RGB image

m-by-n-by-3 array

RGB image, specified as an m-by-n-by-3 array.

Data Types: `single` | `double` | `uint8` | `uint16`

gpuarrayA — RGB color space values or RGB image to be processed on a graphics processing unit (GPU)

gpuArray object

RGB color space values or RGB image to be processed on a graphics processing unit (GPU), specified as a `gpuArray` object.

Output Arguments

ycbcrmap — YCbCr color space values

m-by-3 array

YCbCr color space values, returned as an m-by-3 array. The first column corresponds to Y. The second and third columns correspond to Cb and Cr. Y is in the range [16/255, 235/255], and Cb and Cr are in the range [16/255, 240/255].

YCBCR — Image in YCbCr color space

m-by-n-by-3 array

Image in YCbCr color space, returned as an m-by-n-by-3 array.

- If the input is `double` or `single`, Y is in the range [16/255, 235/255], and Cb and Cr are in the range [16/255, 240/255].
- If the input is `uint8`, Y, is in the range [16, 235], and Cb and Cr, are in the range [16, 240].
- If the input is `uint16`, Y is in the range [4112, 60395] and Cb and Cr are in the range [4112, 61680].

gpuarrayB — Output in YCbCr color space when run on a graphics processing unit (GPU)

gpuArray object

Output in YCbCr color space when run on a graphics processing unit (GPU), specified as a `gpuArray` object. The output is either an array of YCbCr color space values or a YCbCr image, depending on the input type.

References

- [1] Poynton, C. A. *A Technical Introduction to Digital Video*, John Wiley & Sons, Inc., 1996, p. 175.

[2] Rec. ITU-R BT.601-5, *Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-screen 16:9 Aspect Ratios*, (1982-1986-1990-1992-1994-1995), Section 3.5.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

See Also

See Also

`gpuArray` | `ntsc2rgb` | `rgb2ntsc` | `ycbcr2rgb`

Introduced before R2006a

roicolor

Select region of interest (ROI) based on color

Syntax

```
BW = roicolor(A,low,high)
```

```
BW = roicolor(A,v)
```

Description

`roicolor` selects a region of interest (ROI) within an indexed or intensity image and returns a binary image. (You can use the returned image as a mask for masked filtering using `roifilt2`.)

`BW = roicolor(A,low,high)` returns an ROI selected as those pixels that lie within the colormap range `[low high]`.

```
BW = (A >= low) & (A <= high)
```

`BW` is a binary image with 0's outside the region of interest and 1's inside.

`BW = roicolor(A,v)` returns an ROI selected as those pixels in `A` that match the values in vector `v`. `BW` is a binary image with 1's where the values of `A` match the values of `v`.

Class Support

The input image `A` must be `numeric`. The output image `BW` is of class `logical`.

Examples

Select Region-of-Interest Based on Color

Load an indexed image.

```
load clown
```

Create binary mask image based on color.

```
BW = roicolor(X,10,20);
```

Display the original image and the binary mask.

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



```
figure  
imshow(BW)
```



See Also

roifilt2 | roipoly

Introduced before R2006a

roifill

Fill in specified region of interest (ROI) polygon in grayscale image

Compatibility

`roifill` is not recommended. Use `regionfill` instead.

Syntax

```
J = roifill
J = roifill(I)
J = roifill(I, c, r)
J = roifill(I, BW)
[J,BW] = roifill(...)
J = roifill(x, y, I, xi, yi)
[x, y, J, BW, xi, yi] = roifill(...)
```

Description

Use `roifill` to fill in a specified region of interest (ROI) polygon in a grayscale image. `roifill` smoothly interpolates inward from the pixel values on the boundary of the polygon by solving Laplace's equation. The boundary pixels are not modified. `roifill` can be used, for example, to erase objects in an image.

`J = roifill` creates an interactive polygon tool, associated with the image displayed in the current figure, called the target image. You use the mouse to define the ROI – see “Interactive Behavior” on page 1-1731. When you are finished defining the ROI, fill in the area specified by the ROI by double-clicking inside the region or by right-clicking anywhere inside the region and selecting **Fill Area** from the context menu. `roifill` returns the image, `J`, which is the same size as `I` with the region filled in (see “Examples” on page 1-1733).

Note: If you do not specify an output argument, `roifill` displays the filled image in a new figure.

`J = roifill(I)` displays the image `I` and creates an interactive polygon tool associated with the image.

`J = roifill(I, c, r)` fills in the polygon specified by `c` and `r`, which are equal-length vectors containing the row-column coordinates of the pixels on vertices of the polygon. The k th vertex is the pixel $(r(k), c(k))$.


`J = roifill(I, BW)` uses `BW` (a binary image the same size as `I`) as a mask. `roifill` fills in the regions in `I` corresponding to the nonzero pixels in `BW`. If there are multiple regions, `roifill` performs the interpolation on each region independently.

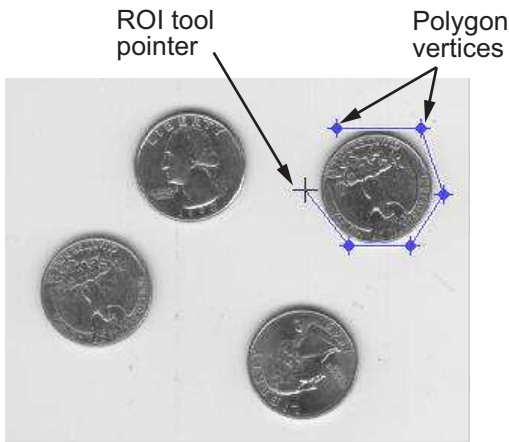
`[J,BW] = roifill(...)` returns the binary mask used to determine which pixels in `I` get filled. `BW` is a binary image the same size as `I` with 1's for pixels corresponding to the interpolated region of `I` and 0's elsewhere.

`J = roifill(x, y, I, xi, yi)` uses the vectors `x` and `y` to establish a nondefault spatial coordinate system. `xi` and `yi` are equal-length vectors that specify polygon vertices as locations in this coordinate system.




`[x, y, J, BW, xi, yi] = roifill(...)` returns the `XData` and `YData` in `x` and `y`, the output image in `J`, the mask image in `BW`, and the polygon coordinates in `xi` and `yi`. `xi` and `yi` are empty if the `roifill(I,BW)` form is used.

Interactive Behavior

When you call `roifill` with an interactive syntax, the pointer changes to a cross hairs shape  when you move it over the target image. Using the mouse, you specify a region-of-interest by selecting vertices of a polygon. You can change the size or shape of the polygon using the mouse. The following figure illustrates a polygon defined by multiple vertices. For more information about all the interactive capabilities of `roifill`, see the table that follows.



Interactive Behavior	Description
<p>Closing the polygon. (Completing the region-of-interest.)</p>	<p>Use any of the following mechanisms:</p> <ul style="list-style-type: none"> • Move the pointer over the initial vertex of the polygon that you selected. The shape changes to a circle ○. Click either mouse button. • Double-click the left mouse button. This action creates a vertex at the point under the mouse and draws a straight line connecting this vertex with the initial vertex. • Click the right mouse button. This action draws a line connecting the last vertex selected with the initial vertex; it does not create a new vertex.
<p>Deleting the polygon</p>	<p>Press Backspace, Escape or Delete, or right-click inside the region and select Cancel from the context menu.</p> <p>Note: If you delete the ROI, the function returns empty values.</p>
<p>Moving the polygon</p>	<p>Move the pointer inside the region. The pointer changes to a fleur ✚. Click and drag the mouse to move the polygon.</p>

Interactive Behavior	Description
Changing the color of the polygon	Move the pointer inside the region. Right-click and select Set color from the context menu.
Adding a new vertex.	Move the pointer over an edge of the polygon and press the A key. The shape of the pointer changes  . Click the left mouse button to create a new vertex at that position on the line.
Moving a vertex. (Reshaping the region-of-interest.)	Move the pointer over a vertex. The pointer changes to a circle  . Click and drag the vertex to its new position.
Deleting a vertex.	Move the pointer over a vertex. The pointer changes to a circle  . Right-click and select Delete Vertex from the context menu. This action deletes the vertex and adjusts the shape of the polygon, drawing a new straight line between the two vertices that were neighbors of the deleted vertex.
Retrieving the coordinates of the vertices	Move the pointer inside the region. Right-click and select Copy position from the context menu to copy the current position to the Clipboard. Position is an n -by-2 array containing the x - and y -coordinates of each vertex, where n is the number of vertices you selected.

Class Support

The input image **I** can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. The input binary mask **BW** can be any numeric class or `logical`. The output binary mask **BW** is always `logical`. The output image **J** is of the same class as **I**. All other inputs and outputs are of class `double`.

Examples

This example uses `roifill` to fill a region in the input image, **I**. For more examples, especially of the interactive syntaxes, see “Fill Region of Interest in an Image”.

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

```
c = [222 272 300 270 221 194];  
r = [21 21 75 121 121 75];  
J = roifill(I,c,r);  
imshow(I)  
figure, imshow(J)
```



See Also

[impoly](#) | [roifilt2](#) | [roipoly](#)

Introduced before R2006a

roifilt2

Filter region of interest (ROI) in image

Syntax

```
J = roifilt2(h, I, BW)
J = roifilt2(I, BW, fun)
```

Description

`J = roifilt2(h, I, BW)` filters the data in `I` with the two-dimensional linear filter `h`. `BW` is a binary image the same size as `I` that defines an ROI used as a mask for filtering. `roifilt2` returns an image that consists of filtered values for pixels in locations where `BW` contains 1's, and unfiltered values for pixels in locations where `BW` contains 0's. For this syntax, `roifilt2` calls `filter2` to implement the filter.

`J = roifilt2(I, BW, fun)` processes the data in `I` using the function `fun`. The result `J` contains computed values for pixels in locations where `BW` contains 1's, and the actual values in `I` for pixels in locations where `BW` contains 0's. `fun` must be a function handle. Parameterizing Functions, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`.

Class Support

For the syntax that includes a filter `h`, the input image can be logical or numeric, and the output array `J` has the same class as the input image. For the syntax that includes a function, `I` can be of any class supported by `fun`, and the class of `J` depends on the class of the output from `fun`.

Examples

This example continues the `roipoly` example, filtering the region of the image `I` specified by the mask `BW`. The `roifilt2` function returns the filtered image `J`, shown in the following figure.

Filter Image Using Polygonal Mask

Read an image into the workspace.

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

Define the vertices of the mask polygon.

```
c = [222 272 300 270 221 194];  
r = [21 21 75 121 121 75];
```

Create the binary mask image.

```
BW = roipoly(I,c,r);
```

Filter the region of the image **I** specified by the mask **BW**.

```
H = fspecial('unsharp');  
J = roifilt2(H,I,BW);
```

Display the original image and the filtered image.

```
imshow(I)
```



figure
imshow(J)



See Also

`filter2` | `imfilter` | `roipoly`

Topics

“Anonymous Functions” (MATLAB)
“Parameterizing Functions” (MATLAB)
“Create Function Handle” (MATLAB)

Introduced before R2006a

roipoly

Specify polygonal region of interest (ROI)


Syntax

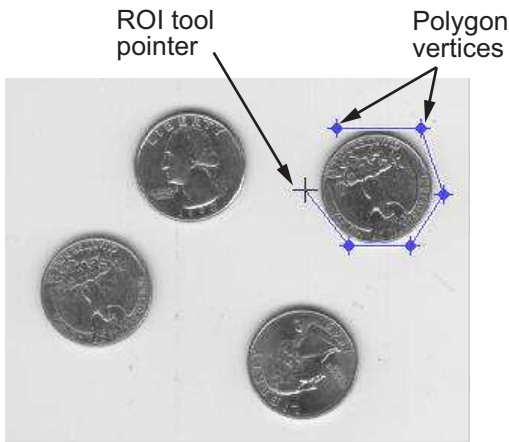
```
BW = roipoly
BW = roipoly(I)
BW = roipoly(I, c, r)
BW = roipoly(x, y, I, xi, yi)
[BW, xi, yi] = roipoly(...)
[x, y, BW, xi, yi] = roipoly(...)
```

Description



Use `roipoly` to specify a polygonal region of interest (ROI) within an image. `roipoly` returns a binary image that you can use as a mask for masked filtering.




`BW = roipoly` creates an interactive polygon tool, associated with the image displayed in the current figure, called the target image. With the polygon tool active, the pointer

changes to cross hairs  when you move the pointer over the image in the figure. Using the mouse, you specify the region by selecting vertices of the polygon. You can move or resize the polygon using the mouse. The following figure illustrates a polygon defined by multiple vertices. The following table describes all the interactive behavior of the polygon tool.



When you are finished positioning and sizing the polygon, create the mask by double-clicking, or by right-clicking inside the region and selecting **Create mask** from the context menu. `roipoly` returns the mask as a binary image, `BW`, the same size as `I`. In the mask image, `roipoly` sets pixels inside the region to 1 and pixels outside the region to 0.

Interactive Behavior	Description
Closing the polygon. (Completing the region-of-interest.)	Use any of the following mechanisms: <ul style="list-style-type: none"> • Move the pointer over the initial vertex of the polygon that you selected. The pointer changes to a circle . Click either mouse button. • Double-click the left mouse button. This action creates a vertex at the point under the mouse pointer and draws a straight line connecting this vertex with the initial vertex. • Right-click the mouse. This draws a line connecting the last vertex selected with the initial vertex; it does not create a new vertex at the point under the mouse.
Moving the entire polygon	Move the pointer inside the region. The pointer changes to a fleur shape  . Click and drag the polygon over the image.

Interactive Behavior	Description
Deleting the polygon	Press Backspace , Escape or Delete , or right-click inside the region and select Cancel from the context menu. Note: If you delete the ROI, the function returns empty values.
Moving a vertex. (Reshaping the region-of-interest.)	Move the pointer over a vertex. The pointer changes to a circle  . Click and drag the vertex to its new position.
Adding a new vertex.	Move the pointer over an edge of the polygon and press the A key. The pointer changes shape to  . Click the left mouse button to create a new vertex at that point on the edge.
Deleting a vertex. (Reshaping the region-of-interest.)	Move the pointer over the vertex. The pointer changes to a circle  . Right-click and select Delete vertex from the context menu. <code>roipoly</code> draws a new straight line between the two vertices that were neighbors of the deleted vertex.
Changing the color of the polygon	Move the pointer anywhere inside the boundary of the region and click the right mouse button. Select Set color from the context menu.
Retrieving the coordinates of the vertices	Move the pointer inside the region. Right-click and select Copy position from the context menu to copy the current position to the Clipboard. The position is an n -by-2 array containing the x - and y -coordinates of each vertex, where n is the number of vertices.

Note: If you call `roipoly` without specifying any output arguments, `roipoly` displays the resulting mask image in a new figure window.

`BW = roipoly(I)` displays the image `I` and creates an interactive polygon tool associated with that image.

`BW = roipoly(I, c, r)` returns the ROI specified by the polygon described by vectors `c` and `r`, which specify the column and row indices of each vertex, respectively. `c` and `r` must be the same size.

`BW = roipoly(x, y, I, xi, yi)` uses the vectors `x` and `y` to establish a nondefault spatial coordinate system. `xi` and `yi` are equal-length vectors that specify polygon vertices as locations in this coordinate system.

`[BW, xi, yi] = roipoly(...)` returns the *x*- and *y*-coordinates of the polygon vertices in `xi` and `yi`.

Note: `roipoly` always produces a closed polygon. If the points specified describe a closed polygon (i.e., if the last pair of coordinates is identical to the first pair), the length of `xi` and `yi` is equal to the number of points specified. If the points specified do not describe a closed polygon, `roipoly` adds a final point having the same coordinates as the first point. (In this case the length of `xi` and `yi` is one greater than the number of points specified.)

`[x, y, BW, xi, yi] = roipoly(...)` returns the XData and YData in `x` and `y`, the mask image in `BW`, and the polygon coordinates in `xi` and `yi`.

Class Support

The input image `I` can be of class `uint8`, `uint16`, `int16`, `single`, or `double`. The output image `BW` is of class `logical`. All other inputs and outputs are of class `double`.

Examples

Use `roipoly` to create a mask image, `BW`, the same size as the input image, `I`. The example in `roifilt2` continues this example, filtering the specified region in the image. To see another example of using `roipoly`, especially of the interactive syntaxes, see “Fill Region of Interest in an Image”.

Create Polygonal Mask

Read an image into the workspace.

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

Define the vertices of the mask polygon.

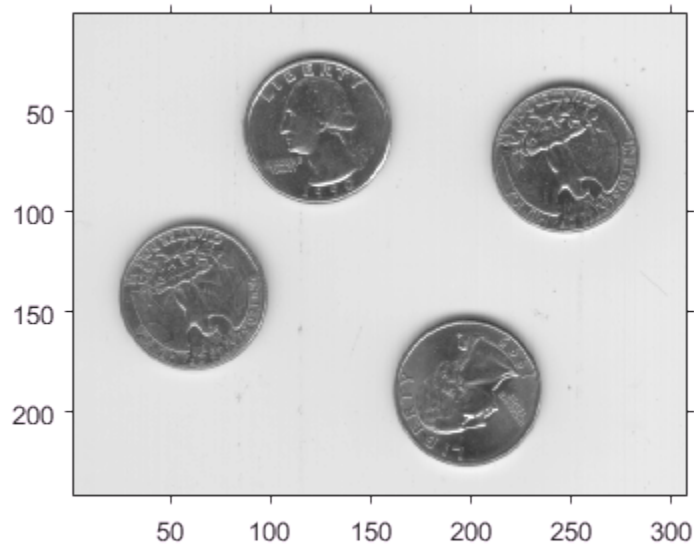
```
c = [222 272 300 270 221 194];  
r = [21 21 75 121 121 75];
```

Create the binary mask image.

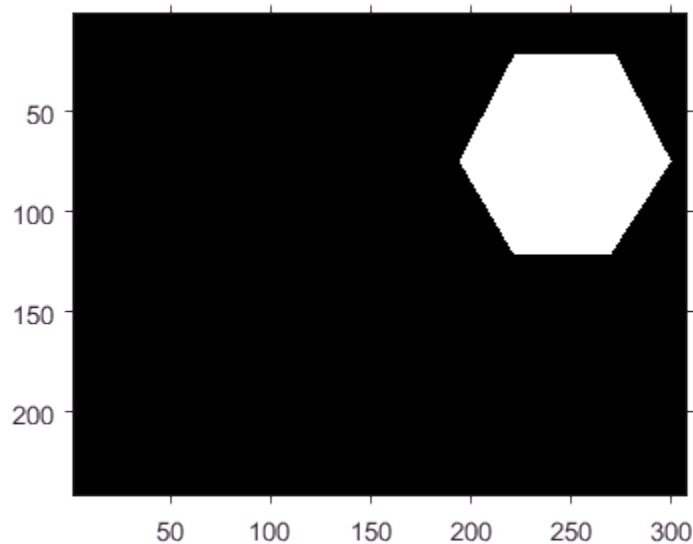
```
BW = roipoly(I,c,r);
```

Display the original image and the polygonal mask.

```
imshow(I)
```



```
figure  
imshow(BW)
```



Tips

For any of the `roipoly` syntaxes, you can replace the input image `I` with two arguments, `m` and `n`, that specify the row and column dimensions of an arbitrary image. For example, these commands create a 100-by-200 binary mask.

```
c = [112 112 79 79];  
r = [37 66 66 37];  
BW = roipoly(100,200,c,r);
```

If you specify `m` and `n` with an interactive form of `roipoly`, an `m`-by-`n` black image is displayed, and you use the mouse to specify a polygon within this image.

See Also

`impoly` | `poly2mask` | `roifilt2` | `roicolor` | `regionfill`

Introduced before R2006a

rsetwrite

Create reduced resolution data set from image file

Syntax

```
rsetfile = rsetwrite(File_Name)
rsetfile = rsetwrite(File_Name, output_filename)
rsetfile = rsetwrite(adapter, output_filename)
```

Description

`rsetfile = rsetwrite(File_Name)`, where `File_Name` is a TIFF or NITF image file, creates a reduced resolution data set (R-Set) from the specified file. The R-Set file is written to the current working directory with a name based on the input file name. For example, if `File_Name` is `'VeryLargeImage.tiff'`, `rsetfile` will be `'VeryLargeImage.rset'`. If an image file contains multiple images, only the first one is used.

`rsetfile = rsetwrite(File_Name, output_filename)` creates an R-Set from the specified image file, using `output_filename` as the name of the new file. In this case, `rsetfile` and `output_filename` are identical.

`rsetfile = rsetwrite(adapter, output_filename)` creates an R-Set from the specified Image Adapter object, `adapter`. Image Adapters are user-defined classes that provide `rsetwrite` a common API for reading a particular image file format. See the documentation for `ImageAdapter` on page 1-632 for more details.

Examples

Example 1: Create an R-Set File

Visualize a very large image by using an R-Set. Replace `'MyReallyBigImage.tif'` in the example below with the name of your file:

```
big_file = 'MyReallyBigImage.tif';
```



```
rset_file = rsetwrite(big_file);  
imtool(rset_file)
```

Example 2: Convert TIFF Files to R-Set Files

Create R-Set files for every TIFF in a directory containing very large images. Put the R-Set files into a temporary directory:

```
d = dir('*.tif*');  
image_dir = pwd;  
cd(tempdir)  
for p = 1:numel(d)  
    big_file = fullfile(image_dir, d(p).name);  
    rsetwrite(big_file);  
end
```

Tips

`rsetwrite` creates an R-Set file by dividing an image into spatial tiles and resampling the image at different resolution levels. When you open the R-Set file in the Image Tool and zoom in, you view tiles at a higher resolution. When you zoom out, you view tiles at a lower resolution. In this way, clarity of the image and memory usage are balanced for optimal performance. The R-Set file contains a compressed copy of the full-resolution data.

Because R-Set creation can be time consuming, a progress bar shows the status of the operation. If you cancel the operation, processing stops, no file is written, and the `rsetfile` variable will be empty.

`rsetwrite` supports NITF image files that are uncompressed and Version 2.0 or higher. It does not support NITF files with more than three bands or with floating point data. Images with more than one data band are OK if they contain unsigned integer data.

While it is possible to create an R-Set from an image where the dimensions are smaller than the size of a single R-Set tile, the resulting R-set file will likely be larger and take longer to load than the original file. The current size of an R-Set tile is 512 x 512 pixels.

See Also

`imread` | `imtool`

Introduced in R2009a

ssim

Structural Similarity Index (SSIM) for measuring image quality

Syntax

```
ssimval = ssim(A,ref)
[ssimval,ssimmap] = ssim(A,ref)
___ = ssim(__,Name,Value,...)
```

Description

`ssimval = ssim(A,ref)` computes the Structural Similarity Index (SSIM) value for image `A` using `ref` as the reference image.

`[ssimval,ssimmap] = ssim(A,ref)` returns the local SSIM value for each pixel in `A`.

`___ = ssim(__,Name,Value,...)` computes the SSIM, using name-value pairs to control aspects of the computation. Parameter names can be abbreviated.

Examples

Calculate Structural Similarity Index (SSIM)

Read an image into the workspace. Create another version of the image, applying a blurring filter. Display both images.

```
ref = imread('pout.tif');
H = fspecial('Gaussian',[11 11],1.5);
A = imfilter(ref,H,'replicate');

subplot(1,2,1); imshow(ref); title('Reference Image');
subplot(1,2,2); imshow(A); title('Blurred Image');
```

Reference Image



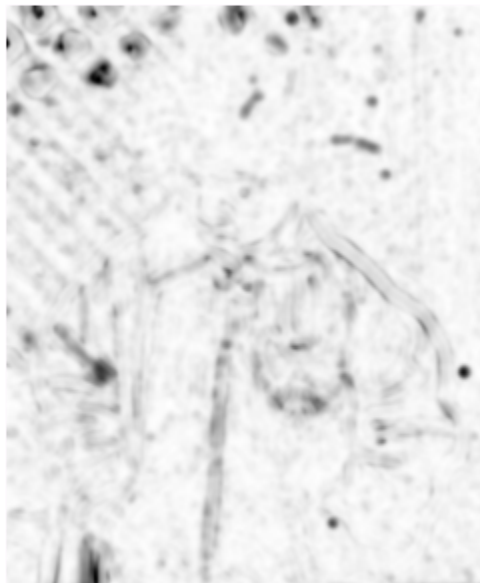
Blurred Image



Calculate the global SSIM value for the image and local SSIM values for each pixel. Return the global SSIM value and display the local SSIM value map.

```
[ssimval, ssimmap] = ssim(A,ref);  
fprintf('The SSIM value is %0.4f.\n',ssimval);  
  
The SSIM value is 0.9407.  
  
figure, imshow(ssimmap,[]);  
title(sprintf('ssim Index Map - Mean ssim Value is %0.4f',ssimval));
```

ssim Index Map - Mean ssim Value is 0.9407



- “Compare Image Quality at Various Compression Levels”

Input Arguments

A — Image whose quality is to be measured

2-D grayscale image | 3-D volume image

Image whose quality is to be measured, specified as a 2-D grayscale image or 3-D volume image. Must be the same size and class as `ref`

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

ref — Reference image against which quality is measured

2-D grayscale image | 3-D volume image

Reference image against which quality is measured, specified as a 2-D grayscale image or 3-D volume image. Must be the same size and class as `A`

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

'DynamicRange' — Dynamic range of the input image

`diff(getrangefromclass(A))` (default) | positive scalar

Dynamic range of the input image, specified as a positive scalar. By default, this value is chosen based on the class of the input image `A`, as `diff(getrangefromclass(A))`. When class of `A` is `single` or `double`, this value is 1, by default.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

'Exponents' — Exponents for the luminance, contrast, and structural terms respectively

`[1 1 1]` (default) | three-element vector of nonnegative real numbers, [`alpha beta gamma`]

Exponents for the luminance, contrast, and structural terms, specified as a three-element vector of nonnegative real numbers, [`alpha beta gamma`].

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

'Radius' — Standard deviation of isotropic Gaussian function

1.5 (default) | positive scalar

Standard deviation of isotropic Gaussian function, specified as a positive scalar. This value is used for weighting the neighborhood pixels around a pixel for estimating local statistics. This weighting is used to avoid blocking artifacts in estimating local statistics.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

'RegularizationConstants' — Regularization constants for the luminance, contrast, and structural terms

three-element vector of nonnegative real numbers, [`C1 C2 C3`]

Regularization constants for the luminance, contrast, and structural terms, specified as a three-element vector of nonnegative real numbers. `ssim` uses these regularization constants to avoid instability for image regions where the local mean or standard deviation is close to zero. Therefore, small non-zero values should be used for these constants.

By default,

- $C1 = (0.01 * L)^2$, where L is the specified `DynamicRange` value.
- $C2 = (0.03 * L)^2$, where L is the specified `DynamicRange` value.
- $C3 = C2 / 2$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output Arguments

ssimval — Structural Similarity (SSIM) Index

scalar

Structural Similarity (SSIM) Index, returned as a scalar `double`, except when `A` and `ref` are of class `single`, in which case `ssimval` is of class `single`.

ssimmap — Local values of Structural Similarity (SSIM) Index

numeric array

Local values of Structural Similarity (SSIM) Index, returned as a numeric array of class `double` except when `A` and `ref` are of class `single`, in which case `ssimmap` is of class `single`. `ssimmap` is an array of the same size as input image `A`.

Definitions

Structural Similarity Index

An image quality metric that assesses the visual impact of three characteristics of an image: luminance, contrast and structure.

Algorithm

The Structural Similarity (SSIM) Index quality assessment index is based on the computation of three terms, namely the luminance term, the contrast term and the structural term. The overall index is a multiplicative combination of the three terms.

$$SSIM(x, y) = [l(x, y)]^\alpha \cdot [c(x, y)]^\beta \cdot [s(x, y)]^\gamma$$

where

$$l(x, y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1},$$

$$c(x, y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2},$$

$$s(x, y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}$$

where μ_x , μ_y , σ_x , σ_y , and σ_{xy} are the local means, standard deviations, and cross-covariance for images x , y . If $\alpha = \beta = \gamma = 1$ (the default for Exponents), and $C_3 = C_2/2$ (default selection of C_3) the index simplifies to:

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

References

- [1] Wang Zhou, Bovik, Alan C., Sheikh, Hamid R., and Simoncelli, Eero P. *Image Quality Assessment: From Error Visibility to Structural Similarity*. IEEE Transactions on Image Processing, Volume 13, Issue 4, pp. 600–612, April 2004

See Also

See Also

immse | mean | median | psnr | sum | var

Topics

“Compare Image Quality at Various Compression Levels”

Introduced in R2014a

std2

Standard deviation of matrix elements

Syntax

```
B = std2(A)
gpuarrayB = std2(gpuarrayA)
```

Description

`B = std2(A)` returns the scalar `B`, the standard deviation of the values in `A`.

`gpuarrayB = std2(gpuarrayA)` performs the operation on a GPU. The input image is a `gpuArray` image. The output is a `gpuArray` scalar. This syntax requires the Parallel Computing Toolbox.

Examples

Compute 2-D Standard Deviation

Read a grayscale image into the workspace, then calculate the standard deviation of the pixel intensity values.

```
I = imread('liftingbody.png');
val = std2(I)
```

```
val = 31.6897
```

Compute 2-D Standard Deviation on a GPU

Read a grayscale image into the workspace as a `gpuArray` object, then calculate the standard deviation of the pixel intensity values using a GPU.

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

```
val = std2(I)
```

Input Arguments

A — Input data

numeric or logical array

Input data, specified as a numeric or logical array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

gpuarrayA — Input image when run on a GPU

`gpuArray`

Input image when run on a GPU, specified as a `gpuArray`.

Output Arguments

B — Standard deviation

numeric scalar

Standard deviation, returned as a numeric scalar. If the datatype of input A is `single`, the standard deviation B is returned as a `single`. Otherwise, B is returned as a `double`.

gpuarrayB — Standard deviation when run on a GPU

`gpuArray`

Standard deviation when run on a GPU, returned as a `gpuArray`.

See Also

See Also

`corr2` | `gpuArray` | `mean` | `mean2` | `std`

Introduced before R2006a

stdfilt

Local standard deviation of image

Syntax

```
J = stdfilt(I)
J = stdfilt(I, NHOOD)
gpuarrayJ = stdfilt(gpuarrayI, ___)
```

Description

`J = stdfilt(I)` returns the array `J`, where each output pixel contains the standard deviation of the 3-by-3 neighborhood around the corresponding pixel in the input image `I`. `I` can have any dimension. The output image `J` is the same size as the input image `I`.

For pixels on the borders of `I`, `stdfilt` uses symmetric padding. In symmetric padding, the values of padding pixels are a mirror reflection of the border pixels in `I`.

`J = stdfilt(I, NHOOD)` calculates the local standard deviation of the input image `I`, where you specify the neighborhood in `NHOOD`. `NHOOD` is a multidimensional array of zeros and ones where the nonzero elements specify the neighbors. `NHOOD`'s size must be odd in each dimension. By default, `stdfilt` uses the neighborhood `ones(3)`. `stdfilt` determines the center element of the neighborhood by `floor((size(NHOOD) + 1)/2)`.

`gpuarrayJ = stdfilt(gpuarrayI, ___)` performs the conversion on a GPU. The input image and the output image are `gpuArrays`. This syntax requires the Parallel Computing Toolbox.

Class Support

`I` can be logical or numeric and must be real and nonsparse. `NHOOD` can be logical or numeric and must contain zeros and/or ones. `J` is of class `double`.

The `gpuArray` `gpuarrayI` can be logical or numeric and must be real and nonsparse. The `gpuArray` returned `gpuarrayJ` is of class `double`.

Notes

To specify neighborhoods of various shapes, such as a disk, use the `strel` function to create a structuring element object and then extract the neighborhood from the structuring element object's `neighborhood` property.

Examples

Perform Standard Deviation Filtering

This example shows how to perform standard deviation filtering using `stdfilt`. Brighter pixels in the filtered image correspond to neighborhoods in the original image with larger standard deviations.

Read an image into the workspace.

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

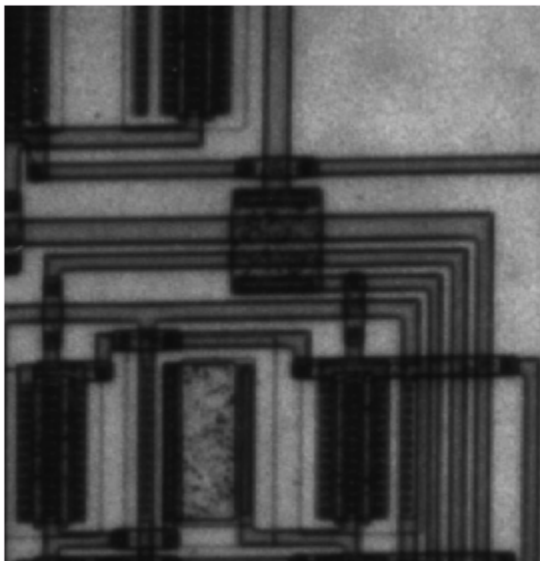
Perform standard deviation filtering using `stdfilt`.

```
J = stdfilt(I);
```

Show the original image and the processed image.

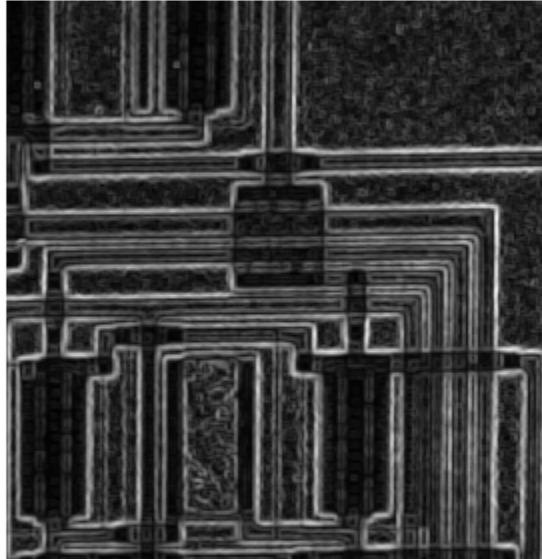
```
imshow(I)  
title('Original Image')
```

Original Image



```
figure  
imshow(J,[])  
title('Result of Standard Deviation Filtering')
```

Result of Standard Deviation Filtering



Perform standard deviation filtering on a GPU.

Read images into `gpuArrays`.

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

Perform standard deviation filtering using `stdfilt`.

```
J = stdfilt(I);
```

Show the original image and the processed image.

```
imshow(I)
title('Original Image')
figure
imshow(J,[])
```

```
title('Result of Standard Deviation Filtering')
```

See Also

entropyfilt | getnhood | rangefilt | std2 | strel | offsetstrel | gpuArray

Introduced before R2006a

strel class

Morphological structuring element

Note: To create the following nonflat structuring element shapes, use `offsetstrel` instead.

- `SE = strel('arbitrary',NHOOD,HEIGHT)`
- `SE = strel('ball',R,H,N)`

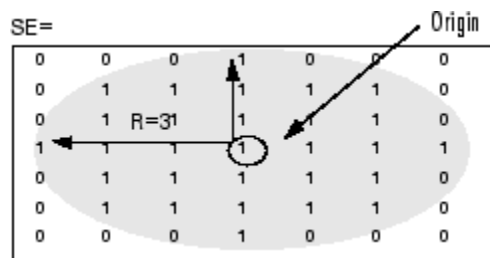
The following syntaxes still work, but are not recommended for use.

- `SE = strel('pair',OFFSET)`
 - `SE = strel('periodicline',P,V)`
-

Description

A `strel` object represents a flat morphological *structuring element*, which is an essential part of morphological dilation and erosion operations. (To create a nonflat structuring element, use `offsetstrel`.)

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.



Construction

`SE = strel('diamond',R)` creates a diamond-shaped structuring element, where `R` specifies the distance from the structuring element origin to the points of the diamond.

`SE = strel('disk',R,N)` creates a disk-shaped structuring element, where `R` specifies the radius. `N` specifies the number of line structuring elements used to approximate the disk shape. Morphological operations using disk approximations run much faster when the structuring element uses approximations.

`SE = strel('line',len,deg)` creates a linear structuring element that is symmetric with respect to the neighborhood center. `deg` specifies the angle (in degrees) of the line as measured in a counterclockwise direction from the horizontal axis. `len` is approximately the distance between the centers of the structuring element members at opposite ends of the line.

`SE = strel('octagon',R)` creates a octagonal structuring element, where `R` specifies the distance from the structuring element origin to the sides of the octagon, as measured along the horizontal and vertical axes. `R` must be a nonnegative multiple of 3.

`SE = strel('rectangle',MN)` creates a rectangular structuring element, where `MN` specifies the size.

`SE = strel('square',W)` creates a square structuring element whose width is `W` pixels.

`SE = strel('cube',W)` creates a cubic structuring element whose width is `W` pixels. `W` must be a nonnegative integer scalar.

`SE = strel('cuboid',XYZ)` creates a cuboidal structuring element of size `XYZ`.

`SE = strel('sphere',R)` creates a spherical structuring element whose radius is `R` pixels.

`SE = strel('arbitrary',nhood)` creates a structuring element, where `NHOOD` is a matrix of 1s and 0s that specifies the neighborhood. You can omit `'arbitrary'` and specify `strel(nhood)`.

Input Arguments

R — Radius of the structuring element in the X-Y plane

nonnegative integer

Radius of the structuring element in the X-Y plane, specified as a nonnegative integer.

For the disk shape, R is the distance from the origin to the edge of the disk.

For the diamond shape, R is the distance from the structuring element origin to the points of the diamond.

Data Types: double

N — Number of periodic line structuring elements used to approximate shape

4 (default) | 0 | 6 | 8

Number of periodic line structuring elements used to approximate shape, specified as the scalar value 0, 4, 6, or 8. When N is greater than 0, the disk-shaped structuring element is approximated by a sequence of N periodic-line structuring elements. When N is 0, `strel` does no approximation, and the structuring element members comprise all pixels whose centers are no greater than R away from the origin. Morphological operations using disk approximations run much faster when the structuring element uses approximations ($N > 0$). Sometimes it is necessary for `strel` to use two extra line structuring elements in the approximation, in which case the number of decomposed structuring elements used is $N + 2$.

Value of N	Behavior
$N > 0$	<code>strel</code> uses a sequence of N (or sometimes $N+2$) periodic line-shaped structuring elements to approximate the shape.
$N = 0$	<code>strel</code> does not use any approximation. The structuring element members comprise all pixels whose centers are no greater than R away from the origin and the corresponding height values are determined from the formula of the ellipsoid specified by R and H.

Data Types: double

MN — Size of rectangle-shaped structuring element

two-element vector of nonnegative integers

Size of rectangle-shaped structuring element, specified as a two-element vector of nonnegative integers. The first element of MN is the number of rows in the structuring element neighborhood and the second element is the number of columns.

Data Types: `double`

W — Width of square-shaped or cube-shaped structuring element

nonnegative integer scalar

Width of square-shaped or cube-shaped structuring element, specified as a nonnegative integer scalar.

Data Types: `double`

XYZ — Dimensions of cuboidal-shaped structuring element

three-element vector

Dimensions of cuboidal-shaped structuring element, specified as a three-element vector of nonnegative integers, of the form `[X Y Z]`. X is the number of rows, Y is the number of columns, and Z is the number of planes in the third dimension.

Data Types: `double`

nhood — Neighborhood

matrix

Neighborhood, specified as a matrix containing 1's and 0's. The location of the 1's defines the neighborhood for the morphological operation. The center (or origin) of `nhood` is its center element, given by `floor((size(nhood) + 1)/2)`.

Data Types: `double`

Properties

Neighborhood — Structuring element neighborhood

logical matrix

Structuring element neighborhood, specified as a matrix of class `logical`.

Data Types: `logical`

Dimensionality — Dimensions of structuring element

nonnegative scalar

Dimensions of structuring element, specified as a nonnegative scalar of class `double`.

Data Types: double

Examples

Create Square Structuring Element

Create an 11-by-11 square structuring element.

```
SE = strel('square', 11)
```

```
SE =
```

```
strel is a square shaped structuring element with properties:
```

```
    Neighborhood: [11×11 logical]  
    Dimensionality: 2
```

Create Line-Shaped Structuring Element

Create a line-shaped structuring element with a length of 10 at an angle of 45 degrees.

```
SE = strel('line', 10, 45)
```

```
SE =
```

```
strel is a line shaped structuring element with properties:
```

```
    Neighborhood: [7×7 logical]  
    Dimensionality: 2
```

View the structuring element.

```
SE.Neighborhood
```

```
ans = 7×7 logical array  
    0    0    0    0    0    0    1  
    0    0    0    0    0    1    0  
    0    0    0    0    1    0    0  
    0    0    0    1    0    0    0  
    0    0    1    0    0    0    0  
    0    1    0    0    0    0    0  
    1    0    0    0    0    0    0
```

Create Disk-Shaped Structuring Element

Create a disk-shaped structuring element with a radius of 15.

```
SE3 = strel('disk', 15)
```

```
SE3 =
```

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

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

Display the disk-shaped structuring element.

```
figure  
imshow(SE3.Neighborhood)
```



Create 3-D Sphere-shaped Structuring Element

Create a 3-D sphere-shaped structuring element with a radius of 15.

```
SE = strel('sphere', 15)
```

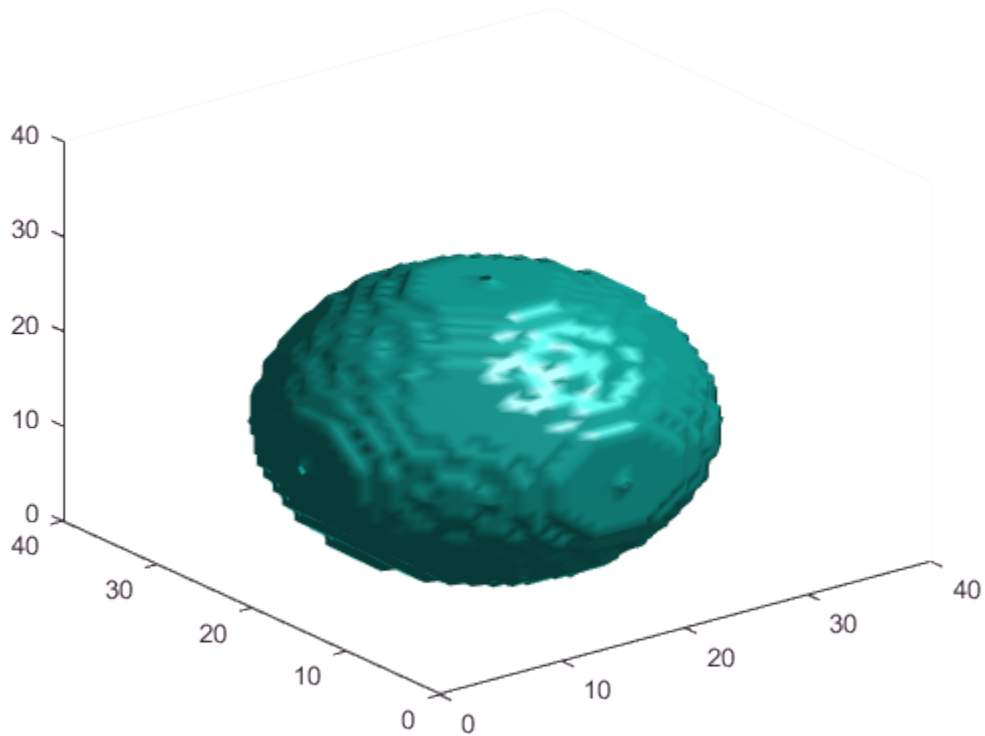
```
SE =
```

```
strel is a sphere shaped structuring element with properties:
```

```
    Neighborhood: [31×31×31 logical]  
    Dimensionality: 3
```

Display the structuring element.

```
figure  
isosurface(SE.Neighborhood)
```



Methods

decompose

Return sequence of decomposed structuring elements

reflect

Reflect structuring element

translate

Translate structuring element

Tips

- Structuring elements that do not use approximations ($N = 0$) are not suitable for computing granulometries.

Algorithms

For all shapes except 'arbitrary', structuring elements are constructed using a family of techniques known collectively as *structuring element decomposition*. The principle is that dilation by some large structuring elements can be computed faster by dilation with a sequence of smaller structuring elements. For example, dilation by an 11-by-11 square structuring element can be accomplished by dilating first with a 1-by-11 structuring element and then with an 11-by-1 structuring element. This results in a theoretical performance improvement of a factor of 5.5, although in practice the actual performance improvement is somewhat less. Structuring element decompositions used for the 'disk' shape is an approximations—all other decompositions are exact.

References

- [1] van den Boomgard, R, and R. van Balen, "Methods for Fast Morphological Image Transforms Using Bitmapped Images," *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, Vol. 54, Number 3, pp. 252–254, May 1992.
- [2] Adams, R., "Radial Decomposition of Discs and Spheres," *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, Vol. 55, Number 5, pp. 325–332, September 1993.
- [3] Jones, R., and P. Soille, "Periodic lines: Definition, cascades, and application to granulometrie," *Pattern Recognition Letters*, Vol. 17, pp. 1057–1063, 1996.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- All input arguments must be compile-time constants.
- The methods associated with `strel` objects are not supported in code generation.
- Arrays of `strel` objects are not supported.

See Also

See Also

`offsetstrel`

Topics

“Structuring Elements”

Property Attributes (MATLAB)

Introduced before R2006a

decompose

Class: `strel`

Return sequence of decomposed structuring elements

Syntax

`SEQ = decompose(SE)`

Description

`SEQ = decompose(SE)` returns an array of structuring elements `SEQ` that are the decomposition of the input structuring element `SE`. `SEQ` is equivalent to `SE`, but the elements of `SEQ` cannot be decomposed further.

`strel` uses decomposition by default for square structuring elements larger than 3-by-3, and disk-shaped structuring elements.

Input Arguments

SE — Structuring element

`strel` object

Structuring element, specified as a `strel` object.

Output Arguments

SEQ — Sequence of structuring elements that approximate the desired shape

array of `strel` objects

Sequence of structuring elements that approximate the desired shape, returned as an array of `strel` objects.

Examples

View Decomposition of Structuring Element

Create a disk-shaped structuring element.

```
se = strel('square',5)
se =
strel is a square shaped structuring element with properties:
    Neighborhood: [5x5 logical]
    Dimensionality: 2
```

Extract the decomposition of the structuring element.

```
seq = decompose(se)
seq =
    2x1 strel array with properties:
    Neighborhood
    Dimensionality
```

To see that dilating sequentially with the decomposed structuring elements really does form a 5-by-5 square, use `imdilate` with the full option.

```
imdilate(1,seq,'full')
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
```

See Also

Introduced before R2006a

reflect

Class: strel

Reflect structuring element

Syntax

```
se2 = reflect(se)
```

Description

`se2 = reflect(se)` reflects the structuring element (or array of structuring elements) specified by `se`. The method reflects the structuring element through its center. The effect is the same as if you rotated the structuring element's domain 180 degrees around its center (for a 2-D structuring element).

Input Arguments

se — Structuring element or structuring elements

strel object or array of strel objects.

Structuring element or structuring elements, specified as a strel object or array of strel objects. If `se` is an array of structuring element objects, then `reflect(se)` reflects each element of `se`, and `se2` has the same size as `se`.

Output Arguments

se2 — Reflected structuring element or structuring elements

strel object or array of strel objects

Reflected structuring element, returned as a strel object or array of structuring elements.

Examples

Reflect a Structuring Element

Create a structuring element.

```
se = strel([0 0 1; 0 0 0; 0 0 0])
```

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

```
    Neighborhood: [3×3 logical]  
    Dimensionality: 2
```

Look at the neighborhood.

```
se.Neighborhood
```

```
ans = 3×3 logical array  
    0    0    1  
    0    0    0  
    0    0    0
```

Reflect it.

```
se2 = reflect(se)
```

```
se2 =  
strel is a arbitrary shaped structuring element with properties:
```

```
    Neighborhood: [3×3 logical]  
    Dimensionality: 2
```

Look at the reflected neighborhood.

```
se2.Neighborhood
```

```
ans = 3×3 logical array  
    0    0    0  
    0    0    0  
    1    0    0
```

See Also

Introduced before R2006a

translate

Class: `strel`

Translate structuring element

Syntax

`SE2 = translate(SE,V)`

Description

`SE2 = translate(SE,V)` translates the structuring element `SE` in N-D space. `V` is an N-element vector containing the offsets of the desired translation in each dimension.

Input Arguments

SE — Structuring element

`strel` object

Structuring element, specified as a `strel` object.

V — Translation offsets

`strel` object

Translation offsets, specified as a numeric vector.

Output Arguments

SE2 — Translated structuring element

`strel` object

Translated structuring element, returned as a `strel` object.

Examples

Translate Structuring Element

Read an image into the workspace.

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

Create a structuring element and translate it down and to the right by 25 pixels.

```
se = translate(strel(1), [25 25]);
```

Dilate the image using the translated structuring element.

```
J = imdilate(I,se);
```

Display the original image and the translated image.

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

Original




```
figure  
imshow(J), title('Translated');
```

Translated



See Also

Introduced before R2006a

stretchlim

Find limits to contrast stretch image

Syntax

```
Low_High = stretchlim(I)
Low_High = stretchlim(I,Tol)
Low_High = stretchlim(RGB,Tol)
Low_High = stretchlim(gpuarrayI, ___ )
```

Description

`Low_High = stretchlim(I)` returns `Low_High`, a two-element vector of pixel values that specify lower and upper limits that can be used for contrast stretching image `I`. By default, values in `Low_High` specify the bottom 1% and the top 1% of all pixel values. The gray values returned can be used by the `imadjust` function to increase the contrast of an image.

`Low_High = stretchlim(I,Tol)` returns `Low_High`, a two-element vector of pixel values that specify lower and upper limits that can be used for contrast stretching image `I`. where `Tol` specifies the fraction of the image to saturate at low and high pixel values.

`Low_High = stretchlim(RGB,Tol)` returns `Low_High`, a two-element vector of pixel values that specify lower and upper limits that can be used for contrast stretching truecolor image `RGB`.

`Low_High = stretchlim(gpuarrayI, ___)` performs the operation on a GPU. . This syntax requires the Parallel Computing Toolbox.

Examples

Find Limits to Stretch Contrast in Grayscale Image

Read grayscale image into the workspace and display it.

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



Adjust the contrast in the image using `stretchlim` to set the limits, and display the result. The example uses the default limits `[0.01 0.99]`, saturating the upper 1% and the lower 1%.

```
J = imadjust(I,stretchlim(I),[]);  
figure  
imshow(J)
```



Find Limits to Stretch Contrast in Grayscale Image on a GPU

Read grayscale image, creating a `gpuArray`.

```
gpuarrayI = gpuArray(imread('pout.tif'));  
figure, imshow(gpuarrayI)
```

Adjust the contrast in the image using `stretchlim` to set the limits. Display the result.

```
gpuarrayJ = imadjust(gpuarrayI,stretchlim(I),[]);
```

```
figure, imshow(gpuarrayJ)
```

Input Arguments

I — Grayscale image

real, nonsparse, numeric array

Grayscale image, specified as a real, nonsparse, numeric array.

Example: `I = imread('pout.tif');` `lohi = stretchlim(I);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

To1 — Fraction of the image to saturate

[0.01 0.99] (default) | numeric scalar or two-element vector in the range [0 1]

Fraction of the image to saturate, specified as a numeric scalar or two-element vector [Low_Fract High_Fract] in the range [0 1].

Value	Description
Scalar	If <code>To1</code> is a scalar, <code>Low_Fract</code> = <code>To1</code> , and <code>High_Fract</code> = <code>1 - Low_Fract</code> , which saturates equal fractions at low and high pixel values.
0	If <code>To1</code> = 0, <code>Low_High</code> = [<code>min(I(:)); max(I(:))</code>].
Default	If you omit the <code>To1</code> argument, [<code>Low_Fract High_Fract</code>] defaults to [0.01 0.99], saturating 2%.
Too big	If <code>To1</code> is too big, such that no pixels would be left after saturating low and high pixel values, <code>stretchlim</code> returns [0 1]

Example: `lohi = stretchlim(I,[.02 .80]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

RGB — Truecolor image

real, nonsparse, numeric array

Truecolor image, specified as a numeric array.

Example: `RGB = imread('peppers.png'); lohi = stretchlim(RGB);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

gpuarrayI — Input image

`gpuArray`

Input image, specified as a `gpuArray`.

Output Arguments

Low_High — Lower and upper limits for contrast stretching

two-element vector of pixel values

Lower and upper limits for contrast stretching, returned as a two-element vector of pixel values

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

See Also

See Also

`brighten` | `decorrstretch` | `gpuArray` | `histeq` | `imadjust`

Introduced before R2006a

subimage

Display multiple images in single figure

Compatibility

`subimage` is not recommended. Use `imshow` instead.

Syntax

```
subimage(X, map)
subimage(I)
subimage(BW)
subimage(RGB)
subimage(x, y...)
h = subimage(...)
```

Description

You can use `subimage` in conjunction with `subplot` to create figures with multiple images, even if the images have different colormaps. `subimage` works by converting images to truecolor for display purposes, thus avoiding colormap conflicts.

`subimage(X, map)` displays the indexed image `X` with colormap `map` in the current axes.

`subimage(I)` displays the intensity image `I` in the current axes.

`subimage(BW)` displays the binary image `BW` in the current axes.

`subimage(RGB)` displays the truecolor image `RGB` in the current axes.

`subimage(x, y...)` displays an image using a nondefault spatial coordinate system.

`h = subimage(...)` returns a handle to an image object.

Class Support

The input image can be of class `logical`, `uint8`, `uint16`, or `double`.

Examples

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

See Also

`imshow` | `subplot`

Introduced before R2006a

superpixels

2-D superpixel oversegmentation of images

Syntax

```
[L, NumLabels] = superpixels(A, N)  
[L, NumLabels] = superpixels( ____, Name, Value, ...)
```

Description

`[L, NumLabels] = superpixels(A, N)` computes superpixels of the 2-D grayscale or RGB image `A`. `N` specifies the number of superpixels you want to create. The function returns `L`, a label matrix of type `double`, and `NumLabels`, the actual number of superpixels that were computed.

The `superpixels` function uses the simple linear iterative clustering (SLIC) algorithm [1]. This algorithm groups pixels into regions with similar values. Using these regions in image processing operations, such as segmentation, can reduce the complexity of these operations.

`[L, NumLabels] = superpixels(____, Name, Value, ...)` computes superpixels of image `A` using with `Name-Value` pairs used to control aspects of the segmentation.

Examples

Compute Superpixels of Input RGB Image

Read image into the workspace.

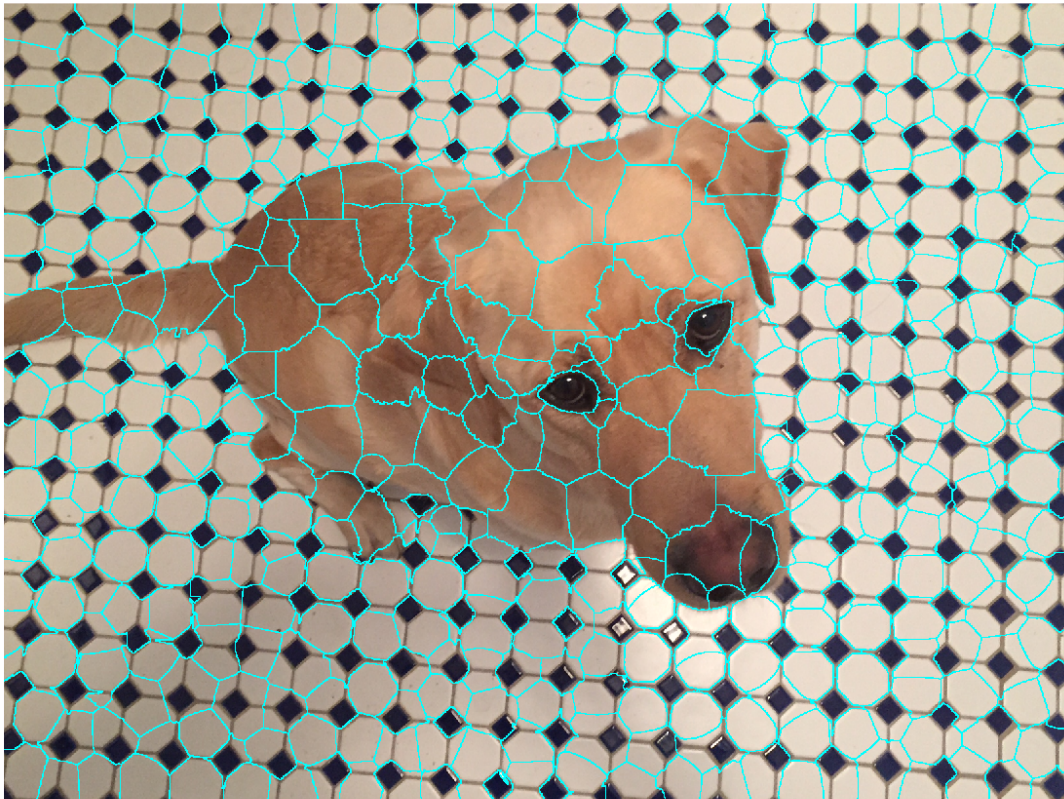
```
A = imread('kobi.png');
```

Calculate superpixels of the image.

```
[L, N] = superpixels(A, 500);
```

Display the superpixel boundaries overlaid on the original image.

```
figure
BW = boundarymask(L);
imshow(imoverlay(A,BW,'cyan'),'InitialMagnification',67)
```



Set the color of each pixel in the output image to the mean RGB color of the superpixel region.

```
outputImage = zeros(size(A),'like',A);
idx = label2idx(L);
numRows = size(A,1);
numCols = size(A,2);
for labelVal = 1:N
```

```
redIdx = idx{labelVal};  
greenIdx = idx{labelVal}+numRows*numCols;  
blueIdx = idx{labelVal}+2*numRows*numCols;  
outputImage(redIdx) = mean(A(redIdx));  
outputImage(greenIdx) = mean(A(greenIdx));  
outputImage(blueIdx) = mean(A(blueIdx));  
end  
  
figure  
imshow(outputImage, 'InitialMagnification',67)
```



- “Plot Land Classification with Color Features and Superpixels”

Input Arguments

A — Input image

real, nonsparse matrix

Input image, specified as a real, nonsparse matrix. For `int16` data, `A` must be a 2-D grayscale image. For all other data types, `A` can be a 2-D grayscale or 2-D RGB image. When the parameter `isInputLab` is `true`, the input image must be `single` or `double`.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

N — Desired number of superpixels

numeric scalar

Desired number of superpixels, specified as a numeric scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `B = superpixels(A,100,'NumIterations', 20);`

'Compactness' — Shape of superpixels

10 (default) | numeric scalar

Shape of superpixels, specified as a numeric scalar. The compactness parameter of the SLIC algorithm controls the shape of superpixels. A higher value makes superpixels more regularly shaped, that is, a square. A lower value makes superpixels adhere to boundaries better, making them irregularly shaped. The allowed range is `[0 Inf)`. Typical values for compactness are in the range `[1,20]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'IsInputLab' — Input image data is in the $L^*a^*b^*$ colorspace

false (default) | true

Input image data is in the $L*a*b^*$ colorspace, specified as the logical scalar `true` or `false`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

'Method' — Algorithm used to compute superpixels

`'slic0'` (default) | `'slic'`

Algorithm used to compute superpixels, specified as one of the following values. The `superpixels` function uses two variations of the simple linear iterative clustering (SLIC) algorithm.

Value	Meaning
<code>'slic0'</code>	<code>superpixels</code> uses the SLIC0 algorithm to refine <code>'Compactness'</code> adaptively after the first iteration. This is the default.
<code>'slic'</code>	<code>'Compactness'</code> is constant during clustering.

Data Types: `char`

'NumIterations' — Number of iterations used in the clustering phase of the algorithm

10 (default) | numeric scalar

Number of iterations used in the clustering phase of the algorithm, specified as a numeric scalar. For most problems, it is not necessary to adjust this parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

L — Label matrix

numeric array

Label matrix, returned as a numeric array of type `double`. The values are positive integers, where 1 indicates the first region, 2 the second region, and so on for each superpixel region in the image.

NumLabels — Number of superpixels computed

numeric scalar

Number of superpixels computed, returned as a numeric scalar of type double.

References

- [1] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Susstrunk, *SLIC Superpixels Compared to State-of-the-art Superpixel Methods*. IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume 34, Issue 11, pp. 2274-2282, May 2012

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. For more information, see “Code Generation for Image Processing”.
- All character vector inputs must be compile-time constants.
- The value of 'IsInputLabel' (true or false) must be a compile-time constant.

See Also

See Also

[boundarymask](#) | [imoverlay](#) | [label2idx](#) | [label2rgb](#) | [superpixels3](#)

Topics

“Plot Land Classification with Color Features and Superpixels”

Introduced in R2016a

superpixels3

3-D superpixel oversegmentation of 3-D image

Syntax

```
[L, NumLabels] = superpixels3(A, N)
[L, NumLabels] = superpixels3( ____, Name, Value, ...)
```

Description

`[L, NumLabels] = superpixels3(A, N)` computes 3-D superpixels of the 3-D image `A`. `N` specifies the number of superpixels you want to create. The function returns `L`, a 3-D label matrix, and `NumLabels`, the actual number of superpixels returned.

`[L, NumLabels] = superpixels3(____, Name, Value, ...)` computes superpixels of image `A` using `Name-Value` pairs to control aspects of the segmentation.

Examples

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);
```

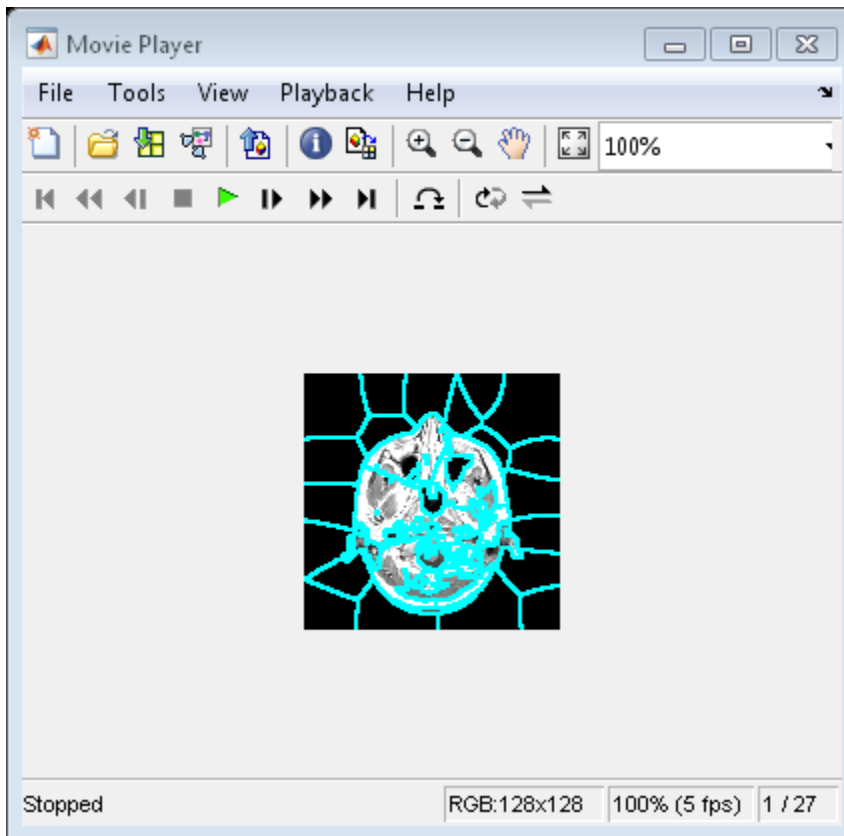
Create a stack of RGB images to display the boundaries in color.


```

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)

```



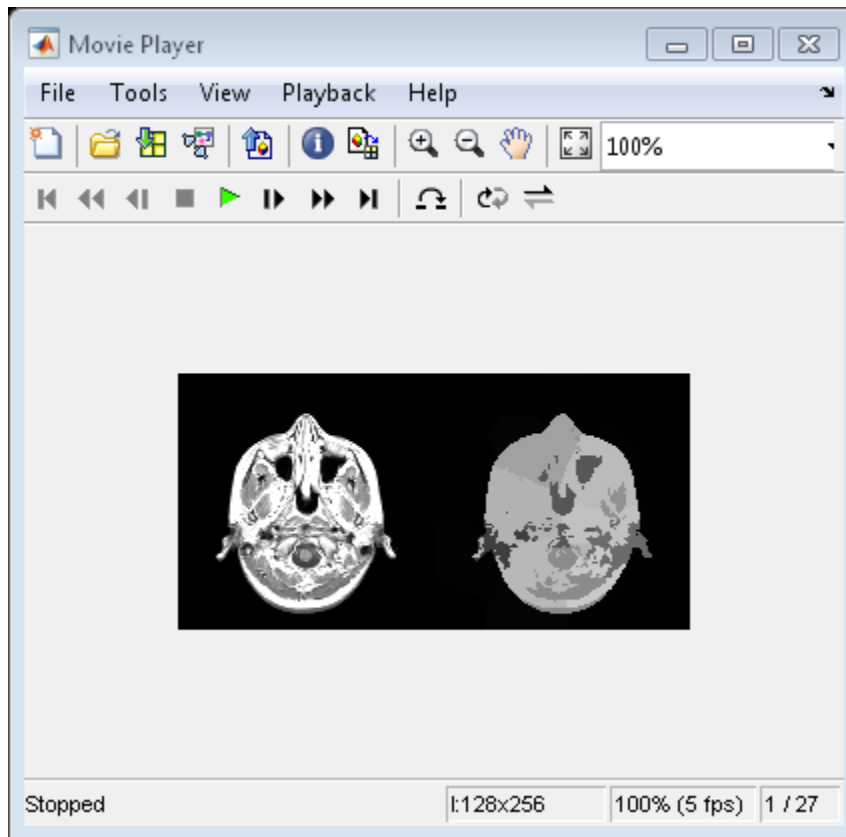
Set the color of each pixel in output image to the mean intensity of the superpixel region. Show the mean image next to the original. If you run this code, you can use `implay` to view each slice of the MRI data.

```

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);
```



Input Arguments

A — Input image

real, nonsparse 3-D array

Input image, specified as a real, nonsparse 3-D array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

N — Desired number of superpixels

numeric scalar

Desired number of superpixels, specified as a numeric scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `B = superpixels3(A,100,'NumIterations', 20);`

'Compactness' — Shape of superpixels

0.001 if method is `slic0` and 0.05 if method is `slic` (default) | numeric scalar

Shape of superpixels, specified as a numeric scalar. The compactness parameter of the SLIC algorithm controls the shape of the superpixels. A higher value makes the superpixels more regularly shaped, that is, a square. A lower value makes the superpixels adhere to boundaries better, making them irregularly shaped. You can specify any value in the range $[0 \text{ Inf})$ but typical values are in the range $[0.01, 0.1]$.

Note: If you specify the '`slic0`' method, you typically do not need to adjust the '`Compactness`' parameter. With the '`slic0`' method, `superpixel3` adaptively refines the '`Compactness`' parameter automatically, thus eliminating the need to determine a good value.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'Method' — Algorithm used to compute superpixels

'`slic0`' (default) | '`slic`'

Algorithm used to compute the superpixels, specified as one of the following values. For more information, see “Algorithms” on page 1-1798.

Value	Meaning
'slic0'	superpixels3 uses the SLIC0 algorithm to refine 'Compactness' adaptively after the first iteration. This is the default.
'slic'	'Compactness' is constant during clustering.

Data Types: char

'NumIterations' — Number of iterations used in the clustering phase of the algorithm

10 (default) | numeric scalar

Number of iterations used in the clustering phase of the algorithm, specified as a numeric scalar. For most problems it is not necessary to adjust this parameter.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Output Arguments

L — Label matrix

3-D array of type double

Label matrix, returned as a 3-D array of type double. The values are positive integers, where 1 indicates the first region, 2 the second region, and so on for each superpixel region in the image.

NumLabels — Number of superpixels computed

numeric scalar

Number of superpixels computed, returned as a numeric scalar of type double.

Algorithms

The algorithm used in `superpixels3` is a modified version of the Simple Linear Iterative Clustering (SLIC) algorithm used by `superpixels`. At a high level, it creates cluster centers and then iteratively alternates between assigning pixels to the closest cluster center and updating the locations of the cluster centers. `superpixels3` uses a distance metric to determine the closest cluster center for each pixel. This distance metric combines intensity distance and spatial distance.

The function's **Compactness** argument comes from the mathematical form of the distance metric. The compactness parameter of the algorithm is a scalar value that controls the shape of the superpixels. The distance between two pixels i and j , where m is the compactness value, is:

$$d_{intensity} = \sqrt{(l_i - l_j)^2}$$
$$d_{spatial} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$
$$D = \sqrt{\left(\frac{d_{intensity}}{m}\right)^2 + \left(\frac{d_{spatial}}{S}\right)^2}$$

Compactness has the same meaning as in the 2-D **superpixels** function: It determines the relative importance of the intensity distance and the spatial distance in the overall distance metric. A lower value makes the superpixels adhere to boundaries better, making them irregularly shaped. A higher value makes the superpixels more regularly shaped. The allowable range for compactness is $(0 \text{ } \text{Inf})$, as in the 2-D function. The typical range has been found through experimentation to be $[0.01 \text{ } 0.1]$. The dynamic range of input images is normalized within the algorithm to be from 0 to 1. This enables a consistent meaning of compactness values across images.

See Also

See Also

[boundarymask](#) | [imoverlay](#) | [label2idx](#) | [label2rgb](#) | [superpixels](#)

Introduced in R2016b

tformarray

Apply spatial transformation to N-D array

Syntax

```
B = tformarray(A,T,R,TDIMS_A, TDIMS_B,TSIZE_B,TMAP_B,F)
```

Description

`B = tformarray(A,T,R,TDIMS_A, TDIMS_B,TSIZE_B,TMAP_B,F)` applies a spatial transformation to array `A` to produce array `B`.

Examples

Transform Checkerboard Image

Create a 2-by-2 checkerboard image where each square is 20 pixels wide and display it.

```
I = checkerboard(20,1,1);  
figure  
imshow(I)
```



Transform the checkerboard with a projective transformation. First create the spatial transformation structure (TFORM).

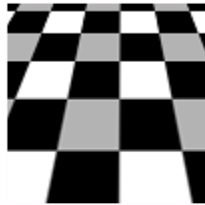
```
T = maketform('projective',[1 1; 41 1; 41 41; 1 41],...
             [5 5; 40 5; 35 30; -10 30]);
```

Create a resampler. Use the pad method 'circular' when creating a resampler, so that the output appears to be a perspective view of an infinite checkerboard.

```
R = makesampler('cubic','circular');
```

Perform the transformation, specifying the TFORM and the resampler. Swap the output dimensions. Specify a 100-by-100 output image. Leave TMAP_B empty, since you specify TSIZE_B. Leave the fill value empty since it won't be needed.

```
J = tformarray(I,T,R,[1 2],[2 1],[100 100],[],[]);
figure
imshow(J)
```



Input Arguments

A — Input image

nonsparse numeric array, that can be real or complex

Input image, specified as a nonsparse numeric array, that can be real or complex.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | logical

T — Spatial transformation

spatial transformation structure (TFORM)

Spatial transformation, specified as a spatial transformation structure, called a TFORM. You typically use the `make_tform` function to create a TFORM structure. `tformarray` uses `T` and the function `tforminv` to compute the corresponding location in the input transform subscript space for each location in the output transform subscript space. `tformarray` defines the input transform space by `TDIMS_B` and `TSIZE_B` and the output transform subscript space by `TDIMS_A` and `size(A)`.

If `T` is empty, `tformarray` operates as a direct resampling function. It applies the resampler defined in `R` to compute values at each transform space location defined in `TMAP_B`. If `TMAP_B` is empty, `tformarray` applies the resampler at each location in the output transform subscript grid.

Data Types: struct

R — Resampler

structure

Resampler, specified as a structure. A resampler structure defines how to interpolate values of the input array at specified locations. `R` is created with `makeresampler`, which allows fine control over how to interpolate along each dimension. `makeresampler` also controls what input array values to use when interpolating close to the edge of the array.

Data Types: struct

TDIMS_A — Input transform dimensions

row vector of finite, positive integers

Input transform dimensions, specified as a row vector of finite, positive integers.

`TDIMS_A` and `TDIMS_B` indicate which dimensions of the input and output arrays are involved in the spatial transformation. Each element must be unique, and must be a positive integer. The entries need not be listed in increasing order, but the order matters. It specifies the precise correspondence between dimensions of arrays `A` and `B` and the input and output spaces of the transformer `T`. `length(TDIMS_A)` must equal `T.ndims_in`, and `length(TDIMS_B)` must equal `T.ndims_out`.

For example, if `T` is a 2-D transformation, `TDIMS_A = [2 1]`, and `TDIMS_B = [1 2]`, then the column dimension and row dimension of `A` correspond to the first and second transformation input-space dimensions, respectively. The row and column dimensions of `B` correspond to the first and second output-space dimensions, respectively.

Data Types: double

TDIMS_B — Size of output array in the transform dimensions

row vector of finite, positive integers

Size in the transform dimensions, specified as a row vector of finite, positive integers. For more information, see TDIMS_A.

Data Types: double

TSIZE_B — Size of output array in the transform dimensions

row vector of finite, positive integers

Size of the output array transform dimensions, specified as a row vector of finite, positive integers. The size of B along nontransform dimensions is taken directly from the size of A along those dimensions. For example, if T is a 2-D transformation, `size(A) = [480 640 3 10]`, TDIMS_B is [2 1], and TSIZE_B is [300 200], then `size(B)` is [200 300 3].

Data Types: double

TMAP_B — Point locations in output space

nonsparse, finite, real-valued array

Point locations in output space, specified as a nonsparse, finite real-valued array. TMAP_B is an optional argument that provides an alternative way of specifying the correspondence between the position of elements of B and the location in output transform space. TMAP_B can be used, for example, to compute the result of an image warp at a set of arbitrary locations in output space. If TMAP_B is not empty, then the size of TMAP_B takes the form

$$[D1 \ D2 \ D3 \ \dots \ DN \ L]$$

where N equals `length(TDIMS_B)`. The vector [D1 D2 ... DN] is used in place of TSIZE_B. If TMAP_B is not empty, then TSIZE_B is [].

The value of L depends on whether T is empty. If T is not empty, then L is `T.ndims_out`, and each L-dimension point in TMAP_B is transformed to an input-space location using T. If T is empty, then L is `length(TDIMS_A)`, and each L-dimensional point in TMAP_B is used directly as a location in input space.

Data Types: double

F — Fill values

array

Fill values, specified as an array. The fill values in **F** can be used in three situations:

- When a separable resampler is created with `makeresampler` and its `padmethod` is set to either `'fill'` or `'bound'`.
- When a custom resampler is used that supports the `'fill'` or `'bound'` pad methods (with behavior that is specific to the customization).
- When the map from the transform dimensions of **B** to the transform dimensions of **A** is deliberately undefined for some points. Such points are encoded in the input transform space by NaNs in either `TMAP_B` or in the output of `TFORMINV`.

In the first two cases, fill values are used to compute values for output locations that map outside or near the edges of the input array. Fill values are copied into **B** when output locations map well outside the input array. See `makeresampler` for more information about `'fill'` and `'bound'`.

F can be a scalar (including NaN), in which case its value is replicated across all the nontransform dimensions. **F** can also be a nonscalar, whose size depends on `size(A)` in the nontransform dimensions. Specifically, if **K** is the **J**th nontransform dimension of **A**, then `size(F,J)` must be either `size(A,K)` or 1. As a convenience to the user, `tformarray` replicates **F** across any dimensions with unit size such that after the replication `size(F,J)` equals `size(A,K)`.

For example, suppose **A** represents 10 RGB images and has size 200-by-200-by-3-by-10, **T** is a 2-D transformation, and `TDIMS_A` and `TDIMS_B` are both `[1 2]`. In other words, `tformarray` applies the same 2-D transform to each color plane of each of the 10 RGB images. In this situation you have several options for **F**:

- **F** can be a scalar, in which case the same fill value is used for each color plane of all 10 images.
- **F** can be a 3-by-1 vector, `[R G B]'`. `tformarray` uses the RGB value as the fill value for the corresponding color planes of each of the 10 images.
- **F** can be a 1-by-10 vector. `tformarray` uses a different fill value for each of 10 images, with that fill value being used for all three color planes.
- **F** can be a 3-by-10 matrix. `tformarray` uses a different RGB fill color for each of the 10 images.

Data Types: double

Output Arguments

B — Transformed image

array

Transformed image, returned as an array.

See Also

See Also

[findbounds](#) | [imtransform](#) | [makesampler](#) | [maketform](#)

Introduced before R2006a

tformfwd

Apply forward spatial transformation

Syntax

```
[X,Y] = tformfwd(T,U,V)
[X1,X2,...,X_ndims_out] = tformfwd(T,U1,U2,...,U_ndims_in)
X = tformfwd(T,U)
[X1,X2,X3,...] = tformfwd(T,U)
X = tformfwd(T,U1,U2,U3,...)
```

Description

`[X,Y] = tformfwd(T,U,V)` applies the 2D-to-2D spatial transformation defined in `T` to coordinate arrays `U` and `V`, mapping the point `[U(k) V(k)]` to the point `[X(k) Y(k)]`.

`[X1,X2,...,X_ndims_out] = tformfwd(T,U1,U2,...,U_ndims_in)` applies the `ndims_in`-to-`ndims_out` spatial transformation defined in `T` to the coordinate arrays `U1,U2,...,U_ndims_in` (where `ndims_in = T.ndims_in`). The number of output arguments must equal `T.ndims_out`. The transformation maps the point `[U1(k) U2(k) ... U_ndims_in(k)]` to the point `[X1(k) X2(k) ... X_ndims_out(k)]`. The arrays `U1,U2,...,U_ndims_in` can have any dimensionality, but must be the same size. `X1,X2,...,X_ndims_out` must have this size also.

`X = tformfwd(T,U)` applies the `ndims_in`-to-`ndims_out` spatial transformation defined in `T` to each row of `U`, where `U` is an `M`-by-`ndims_in` matrix. `tformfwd` maps the point `U(k,:)` to the point `X(k,:)`. `X` is an `M`-by-`ndims_out` matrix.

When `U` is an `(N+1)`-dimensional array, `tformfwd` maps the point `U(k1,k2,...,kN,:)` to the point `X(k1,k2,...,kN,:)`. `size(U,N+1)` must equal `ndims_in`. `X` is an `(N+1)`-dimensional array, with `size(X,I)` equal to `size(U,I)` for `I = 1,...,N` and `size(X,N+1)` equal to `ndims_out`.

The syntax `X = tformfwd(U,T)` is an older form of this syntax that remains supported for backward compatibility.

`[X1,X2,X3,...] = tformfwd(T,U)` maps an `(N+1)`-dimensional array to `ndims_out` equally sized `N`-dimensional arrays.

`X = tformfwd(T,U1,U2,U3,...)` maps `ndims_in` N-dimensional arrays to one (N+1)-dimensional array.

Examples

Create Affine Transformation and Validate

Create an affine transformation that maps the triangle with vertices (0,0), (6,3), (-2,5) to the triangle with vertices (-1,-1), (0,-10), (4,4).

```
u = [ 0  6 -2]';
v = [ 0  3  5]';
x = [-1  0  4]';
y = [-1 -10 4]';
tform = maketform('affine',[u v],[x y]);
```

Validate the mapping by applying `tformfwd`. Results should equal `[x,y]`.

```
[xm, ym] = tformfwd(tform, u, v)
```

```
xm =
```

```
-1
 0
 4
```

```
ym =
```

```
-1
-10
 4
```

Input Arguments

T — Spatial transformation

structure

Spatial transformation, specified as a structure (`tform`), created with `maketform`, `fliptform`, or `cp2tform`. Both `T.ndims_in` and `T.ndims_out` must equal 2.

Data Types: `struct`

U — Input coordinate points

numeric array

Input coordinate points, specified as a numeric array or an M -by-`ndims_in` matrix. For the syntaxes where you also specify V , or specify multiple U 's, U is typically a column vector, but can have any dimensionality. When you specify the argument V , it must have the same size as U .

Data Types: `double`

V — Input coordinate points

column vector

Input coordinate points, specified as a numeric array. V is typically a column vector, but can have any dimensionality. V must have the same size U .

Data Types: `double`

U1, U2, ..., U_ndims_in — Input coordinate points

arrays

Input coordinate points, specified as multiple arrays. `ndims_in` = `T.ndims_in` and `ndims_out` = `T.ndims_out`. The number of output arguments must equal `ndims_out`.

Data Types: `double`

Output Arguments

X — Coordinate array of output points

array

Coordinate array of output points, returned as an array the same size as U .

Y — Coordinate array of output points

array

Coordinate array of output points, returned as an array the same size as V .

X1, X2, ..., X_ndims_out — Coordinates of output points

multiple arrays

Coordinates of output points, returned as multiple arrays.

See Also

See Also

`fliptform` | `tforminv`

Introduced before R2006a

tforminv

Apply inverse spatial transformation

Syntax

```
[U,V] = tforminv(T,X,Y)
[U1,U2,...,U_ndims_in] = tforminv(T,X1,X2,...,X_ndims_in)
U = tforminv(T,X)
[U1,U2,U3,...] = tforminv(T,X)
U = tforminv(T,X1,X2,X3,...)
```

Description

`[U,V] = tforminv(T,X,Y)` applies the 2D-to-2D inverse transformation defined in `T` to coordinate arrays `X` and `Y`, mapping the point `[X(k) Y(k)]` to the point `[U(k) V(k)]`.

`[U1,U2,...,U_ndims_in] = tforminv(T,X1,X2,...,X_ndims_in)` applies the `ndims_out`-to-`ndims_in` inverse transformation defined in `T` to the coordinate arrays `X1,X2,...,X_ndims_out` (where `ndims_in = T.ndims_in` and `ndims_out = T.ndims_out`). The number of output arguments must equal `ndims_in`. The transformation maps the point `[X1(k) X2(k) ... X_ndims_out(k)]` to the point `[U1(k) U2(k) ... U_ndims_in(k)]`.

`U = tforminv(T,X)` applies the `ndims_out`-to-`ndims_in` inverse transformation defined in `T` to each row of `X`, where `X` is an `M`-by-`ndims_out` matrix. `tforminv` maps the point `X(k,:)` to the point `U(k,:)`. `U` is an `M`-by-`ndims_in` matrix.

When `X` is an `(N+1)`-dimensional array, `tforminv` maps the point `X(k1,k2,...,kN,:)` to the point `U(k1,k2,...,kN,:)`. `size(X,N+1)` must equal `ndims_out`. `U` is an `(N+1)`-dimensional array, with `size(U,I)` equal to `size(X,I)` for `I = 1,...,N` and `size(U,N+1)` equal to `ndims_in`.

The syntax `U = tforminv(X,T)` is an older form of the two-argument syntax that remains supported for backward compatibility.

`[U1,U2,U3,...] = tforminv(T,X)` maps an `(N+1)`-dimensional array to `ndims_in` equally sized `N`-dimensional arrays.

`U = tforminv(T,X1,X2,X3,...)` maps `ndims_out` N-dimensional arrays to one (N+1)-dimensional array.

Examples

Create Affine Transformation and Validate By Inverse Mapping

Create an affine transformation that maps the triangle with vertices (0,0), (6,3), (-2,5) to the triangle with vertices (-1,-1), (0,-10), (4,4).

```
u = [ 0  6 -2]';
v = [ 0  3  5]';
x = [-1  0  4]';
y = [-1 -10  4]';
tform = maketform('affine',[u v],[x y]);
```

Validate the mapping by applying `tforminv`. Results should equal `[u, v]`.

```
[um, vm] = tforminv(tform, x, y)
```

```
um =
```

```
    0
  6.0000
 -2.0000
```

```
vm =
```

```
    0
  3.0000
  5.0000
```

Input Arguments

T — Spatial transformation

structure

Spatial transformation, specified as a structure (`tform`), created with `maketform`, `fliptform`, or `cp2tform`. Both `T.ndims_in` and `T.ndims_out` must equal 2.

Data Types: `struct`

X — Input coordinate points

numeric array | M-by-`ndims_out` matrix

Input coordinate points, specified as a numeric array or an M-by-`ndims_out` matrix. X is typically a column vector, but can have any dimensionality. X must have the same size as Y.

When X is an M-by-`ndims_in` matrix, it maps the point X(k, :) to the point U(k, :). U is an M-by-`ndims_in` matrix.

Data Types: `double`

Y — Input coordinate points

numeric array

Input coordinate points, specified as a numeric array or an M-by-`ndims_in` matrix. For the syntaxes where you also specify Y, or specify multiple X's, X is typically a column vector, but can have any dimensionality. When you specify the argument V, it must have the same size U.

Data Types: `double`

X1, X2, ..., X_ndims_in — Input coordinate points

arrays

Input coordinate points, specified as multiple arrays. `ndims_in = T.ndims_in` and `ndims_out = T.ndims_out`). The number of output arguments must equal `ndims_out`.

Data Types: `double`

Output Arguments

U — Coordinate array of output points

array

Coordinate array of output points, returned as an array the same size as X.

V — Coordinate array of output points

array

Coordinate array of output points, returned as an array the same size as Y.

U1,U2,...,U_ndims_in — Coordinates of output points

multiple arrays

Coordinates of output points, returned as multiple arrays.

See Also

See Also

fliptform | tformfwd

Introduced before R2006a

tonemap

Render high dynamic range image for viewing

Syntax

```
RGB = tonemap(HDR)
RGB = tonemap(HDR,Name,Value)
```

Description

`RGB = tonemap(HDR)` converts the high dynamic range image `HDR` to a lower dynamic range image, `RGB`, suitable for display, using a process called tone mapping. Tone mapping is a technique used to approximate the appearance of high dynamic range images on a display with a more limited dynamic range.

`RGB = tonemap(HDR,Name,Value)` allows parameters to control various aspects of the tone mapping.

Examples

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)
```



Input Arguments

HDR — High dynamic range image

m-by-*n*-by-3 array

High dynamic range image, specified by an *m*-by-*n*-by-3 array.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:

'AdjustLightness' — Overall lightness of the rendered image

two-element vector

Overall lightness of the rendered image, specified as a two-element vector. The vector takes the form `[low high]`, where `low` and `high` are luminance values of the low dynamic range image, in the range (0, 1]. These values are passed to `imadjust`.

Data Types: `double`

'AdjustSaturation' — Saturation of colors in the rendered image

1 (default) | positive scalar

Saturation of colors in the rendered image, specified as a positive scalar. When the value is greater than 1, the colors are more saturated. When the value is in the range (0, 1], colors are less saturated.

Data Types: `double`

'NumberOfTiles' — Number of tiles used during adaptive histogram equalization

[4 4] (default) | two-element vector of positive integers

Number of tiles used during the adaptive histogram equalization part of the tone mapping operation, specified as a two-element vector of positive integers. The vector takes the form `[rows cols]`, where `rows` and `cols` specify the number of rows and columns of tiles. Both `rows` and `cols` must be at least 2. The total number of image tiles is equal to `rows*cols`. A larger number of tiles results in an image with greater local contrast.

Data Types: `double`

Output Arguments

RGB — Low dynamic range image

m-by-*n*-by-3 array

Low dynamic range image, specified as an *m*-by-*n*-by-3 array.

Data Types: `uint8`

See Also

See Also

`adaphisteq` | `hdrread` | `stretchlim`

Introduced in R2007b

truesize

Adjust display size of image

Syntax

```
truesize(fig,[mrows ncols])
```

Description

`truesize(fig,[mrows ncols])` adjusts the display size of an image. `fig` is a figure containing a single image or a single image with a colorbar. `[mrows ncols]` is a 1-by-2 vector that specifies the requested screen area (in pixels) that the image should occupy.

`truesize(fig)` uses the image height and width for `[mrows ncols]`. This results in the display having one screen pixel for each image pixel.

If you do not specify a figure, `truesize` uses the current figure.

Examples

Fit image to figure window.

```
imshow(checkerboard,'InitialMagnification','fit')
```

Resize image and figure to show image at its 80-by-80 pixel size.

```
truesize
```

See Also

`imshow` | `iptsetpref` | `iptgetpref`

Introduced before R2006a

uintlut

Compute new values of **A** based on lookup table (LUT)

Syntax

```
B = uintlut(A,LUT)
```

Note: `uintlut` has been removed. Use `intlut` instead.

Class Support

A must be `uint8` or `uint16`. If **A** is `uint8`, then **LUT** must be a `uint8` vector with 256 elements. If **A** is `uint16`, then **LUT** must be a `uint16` vector with 65536 elements. **B** has the same size and class as **A**.

Examples

```
A = uint8([1 2 3 4; 5 6 7 8;9 10 11 12]);  
LUT = repmat(uint8([0 150 200 255]),1,64);  
B = uintlut(A,LUT);  
imshow(A,[]), figure, imshow(B);
```

See Also

`impixel` | `improfile`

Introduced before R2006a

visboundaries

Plot region boundaries

Syntax

```
visboundaries(BW)
visboundaries(B)
visboundaries(AX, ___ )
obj = visboundaries( ___ )
H = visboundaries( ___, Name, Value)
```

Description

`visboundaries(BW)` draws boundaries of regions in the binary image `BW` on the current axes. `BW` is a 2D binary image where pixels that are logical `true` belong to the foreground region and pixels that are logical `false` constitute the background. `visboundaries` uses `bwboundaries` to find the boundary pixel locations in the image.

`visboundaries(B)` draws region boundaries specified by `B`, where `B` is a cell array containing the boundary pixel locations of the regions, similar in structure to the first output from `bwboundaries`. Each cell contains a Q -by-2 matrix, where Q is the number of boundary pixels for the corresponding region. Each row of these Q -by-2 matrices contains the row and column coordinates of a boundary pixel.

`visboundaries(AX, ___)` draws region boundaries on the axes specified by `AX`.

`obj = visboundaries(___)` returns an `hggroup` object for the boundaries. The `hggroup` object, `obj`, is the child of the axes object, `AX`.

`H = visboundaries(___, Name, Value)` passes the name-value pair arguments to specify additional properties of the boundaries. Parameter names can be abbreviated.

Examples

Compute Boundaries and Plot on Image

Read image.

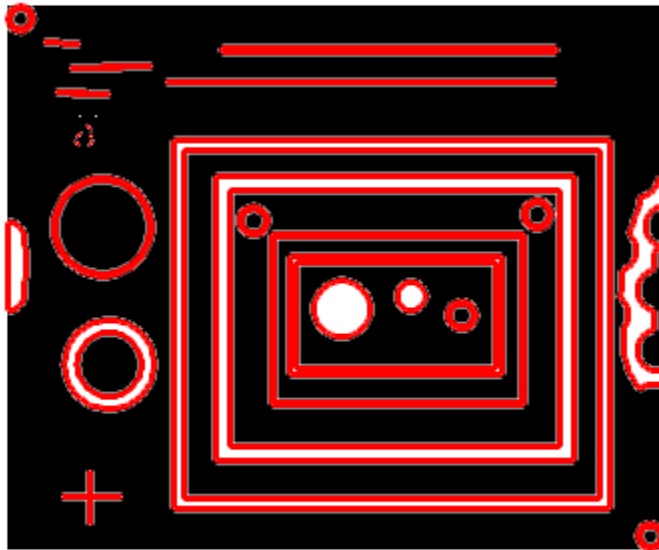
```
BW = imread('blobs.png');
```

Compute boundaries.

```
B = bwboundaries(BW);
```

Display image and plot boundaries on image.

```
imshow(BW)  
hold on  
visboundaries(B)
```



Visualize Segmentation Result

Read image and display it.

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

```
imshow(I)  
hold on
```

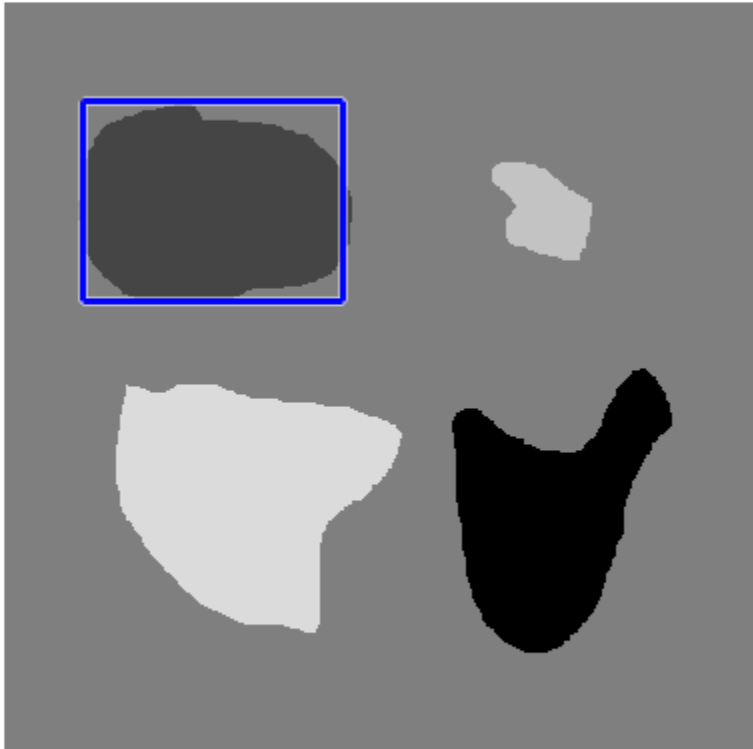


Segment the image using active contour. First, specify the initial contour location close to the object that is to be segmented.

```
mask = false(size(I));  
mask(50:150,40:170) = true;
```

Display the initial contour on the original image in blue.

```
visboundaries(mask, 'Color', 'b');
```



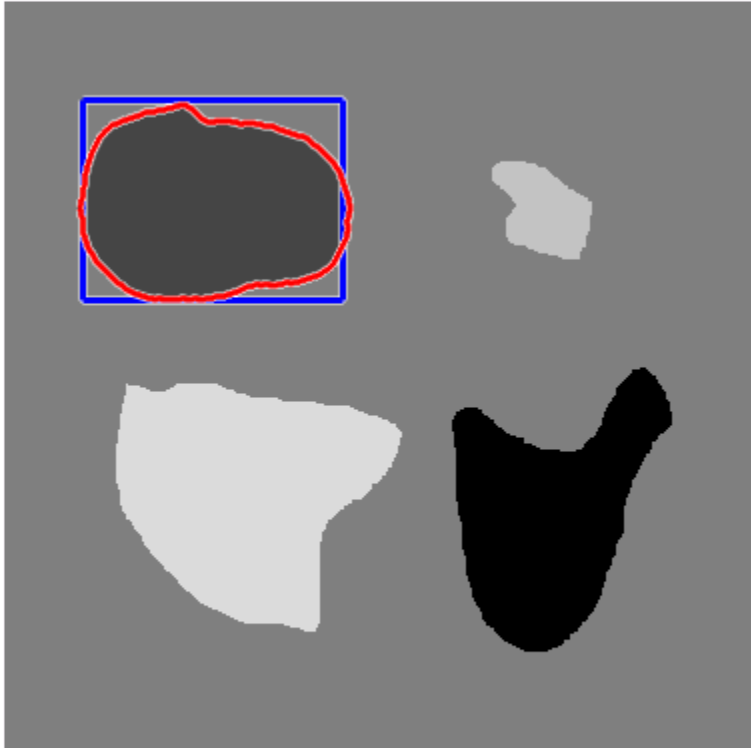
Segment the image using the 'edge' method using 200 iterations.

```
bw = activecontour(I, mask, 200, 'edge');
```

Display the final contour on the original image in red.

```
visboundaries(bw, 'Color', 'r');  
title('Blue - Initial Contour, Red - Final Contour');
```

Blue - Initial Contour, Red - Final Contour



Input Arguments

BW – Binary image

logical array

Binary image, specified as a logical array.

Data Types: `logical`

B — Boundary pixel locations

cell array of Q -by-2 matrices containing row and column coordinates

Boundary pixel locations, specified as a cell array of Q -by-2 matrices containing row and column coordinates, where Q is the number of boundary pixels for the corresponding region.

Data Types: `cell`

AX — Image on which to draw boundaries

current axes (default) | axes object

Image on which to draw boundaries, specified as an axes object.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `visboundaries(bw, 'Color', 'r');`

'Color' — Color of the boundary

MATLAB color spec

Color of the boundary, specified as a MATLAB colorspec.

Example: `visboundaries(bw, 'Color', 'r');`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Complex Number Support: Yes

'LineStyle' — Style of boundary line

solid line('-') (default) | MATLAB line spec

Style of boundary line, specified as a MATLAB line spec.

Example: `visboundaries(bw, 'LineStyle', '-.');`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char`

'LineWidth' — Width of the line used for the boundary

2 points (default) | numeric value

Width of the line used for the boundary, specified as a numeric value. Specify this value in points, where one point = 1/72 inch.

Example: `visboundaries(bw, 'LineWidth', 4);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'EnhanceVisibility' — Augment the drawn boundary with contrasting features`true` (default) | `false`

Augment the drawn boundary with contrasting features to improve visibility on a varying background, specified as the logical flag `true` or `false`.

Example: `visboundaries(bw, 'EnhanceVisibility', true);`

Data Types: `logical`

Output Arguments

obj — Boundary lines`hgroup` object

Boundary line, specified as an `hgroup` object. The `hgroup` object is the child of the axes object, `AX`.

See Also

See Also`bwboundaries` | `bwperim` | `bwtraceboundary` | `viscircles`

Introduced in R2015a

viscircles

Create circle

Syntax

```
viscircles(centers, radii)  
viscircles(ax, centers, radii)  
h = viscircles(ax, centers, radii)  
h = viscircles( ____, Name, Value)
```

Description

`viscircles(centers, radii)` draws circles with specified `centers` and `radii` onto the current axes.

`viscircles(ax, centers, radii)` draws circles onto the axes specified by `ax`.

`h = viscircles(ax, centers, radii)` draws circles and returns a handle to the circles created. This handle is an `hggroup` object that is a child of the `axes` object, `ax`.

`h = viscircles(____, Name, Value)` specifies additional options with one or more `Name, Value` pair arguments, using any of the previous syntaxes. Parameter names can be abbreviated.

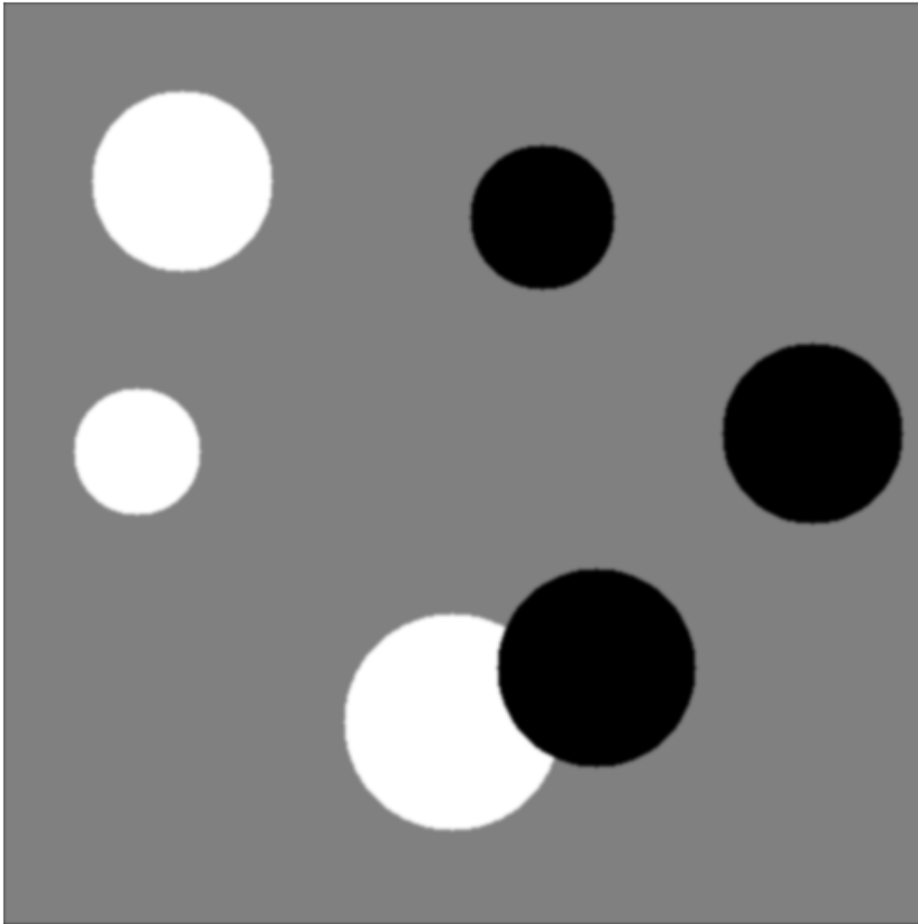
Examples

Draw Lines Around Bright and Dark Circles in Image

This example shows how to draw lines around both bright and dark circles in an image.

Read the image into the workspace and display it.

```
A = imread('circlesBrightDark.png');  
imshow(A)
```



Define the radius range.

```
Rmin = 30;  
Rmax = 65;
```

Find all the bright circles in the image within the radius range.

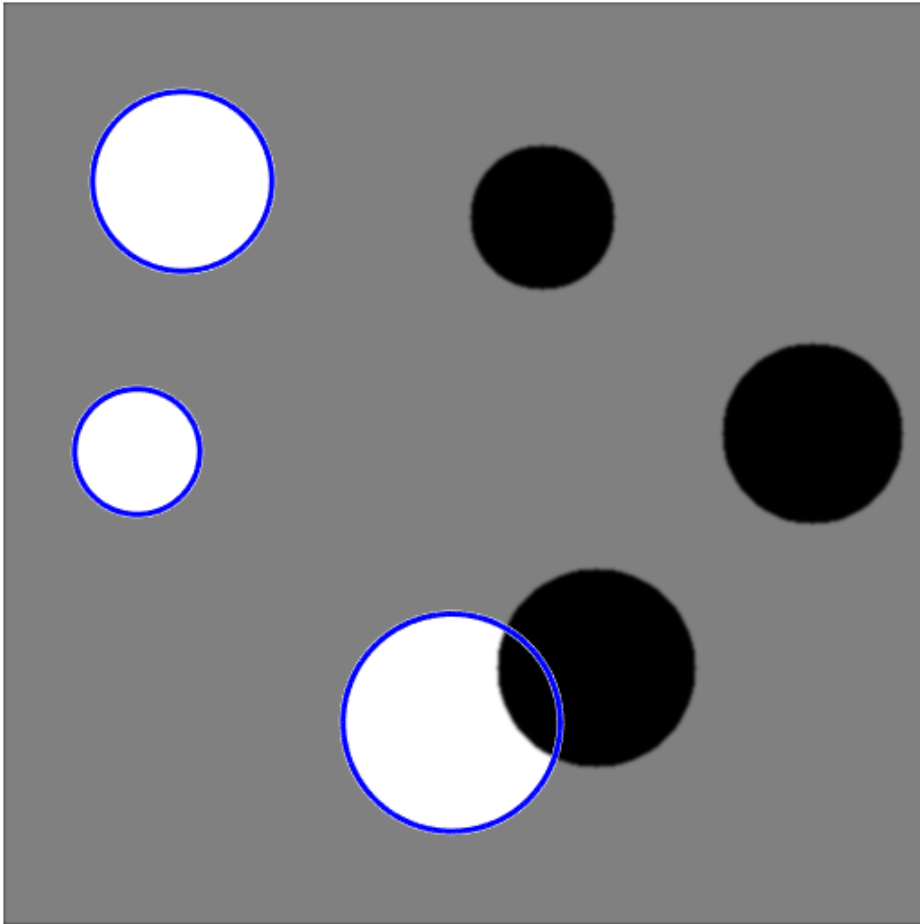
```
[centersBright, radiiBright] = imfindcircles(A,[Rmin Rmax], 'ObjectPolarity', 'bright');
```

Find all the dark circles in the image within the radius range.

```
[centersDark, radiiDark] = imfindcircles(A,[Rmin Rmax], 'ObjectPolarity', 'dark');
```

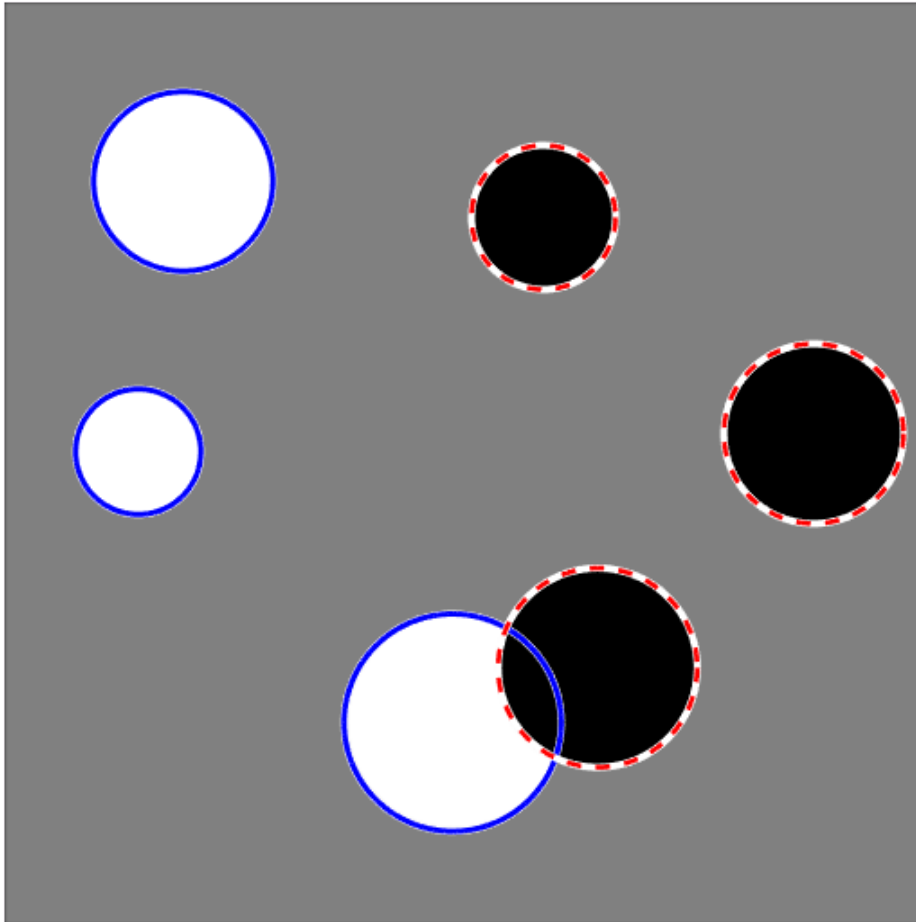
Draw blue lines around the edges of the bright circles.

```
viscircles(centersBright, radiiBright, 'Color', 'b');
```



Draw red dashed lines around the edges of the dark circles.

```
viscircles(centersDark, radiiDark, 'LineStyle', '--');
```



Clear Axes Before Plotting Circles

The `viscircles` function does not clear the target axes before plotting circles. To remove circles that have been previously plotted in an axes, use the `cla` function. To

illustrate, this example creates a new figure and then loops, drawing a set of circles with each iteration, clearing the axes each time.

```
figure
colors = {'b', 'r', 'g', 'y', 'k'};

for k = 1:5
    % Create 5 random circles to display,
    X = rand(5,1);
    Y = rand(5,1);
    centers = [X Y];
    radii = 0.1*rand(5,1);

    % Clear the axes.
    cla

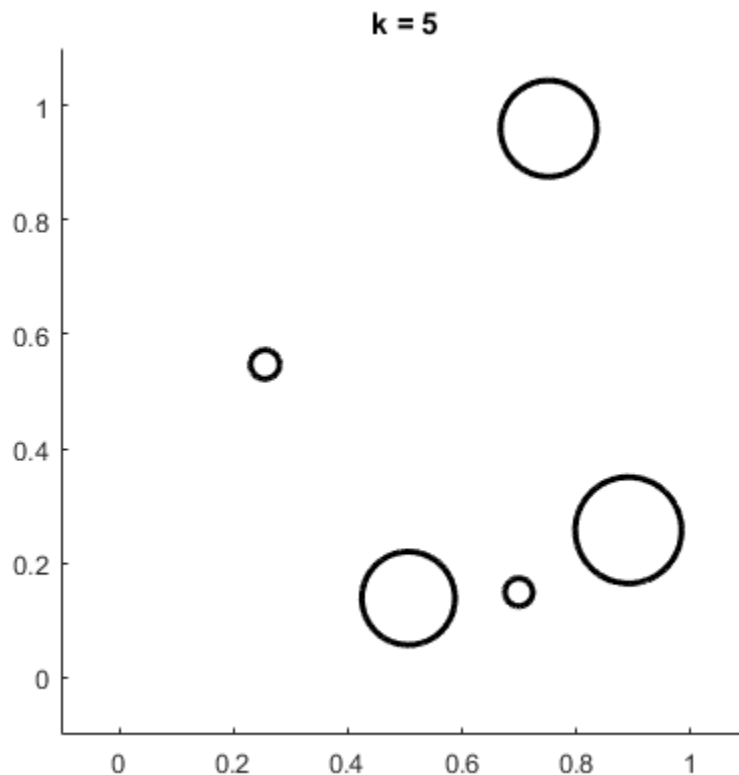
    % Fix the axis limits.
    xlim([-0.1 1.1])
    ylim([-0.1 1.1])

    % Set the axis aspect ratio to 1:1.
    axis square

    % Set a title.
    title(['k = ' num2str(k)])

    % Display the circles.
    viscircles(centers,radii,'Color',colors{k});

    % Pause for 1 second.
    pause(1)
end
```



Input Arguments

centers — Coordinates of circle centers

two-column matrix

Coordinates of circle centers, specified as a P-by-2 matrix, such as that obtained from `imfindcircles`. The *x*-coordinates of the circle centers are in the first column and the *y*-coordinates are in the second column. The coordinates can be integers (of any numeric type) or floating-point values (of type `double` or `single`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

radii — Circle radii

column vector

Circle radii, specified as a column vector such as that returned by `imfindcircles`. The radius value at `radii(j)` corresponds to the circle with center coordinates `centers(j,:)`. The values of `radii` can be nonnegative integers (of any numeric type) or floating-point values (of type `double` or `single`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

ax — Axes in which to draw circles

handle

Axes in which to draw circles, specified as a handle object returned by `gca` or `axes`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `viscircles(centers,radii,'Color','b')` specifies blue circle edges, using the short name for blue.

'EnhanceVisibility' — Augment drawn circles with contrasting features to improve visibility`'true'` (default) | `'false'`

Augment drawn circles with contrasting features to improve visibility, specified as a logical value `true` or `false`. If you set the value to `true`, `viscircles` draws a contrasting circle below the colored circle.

Data Types: `logical`

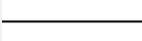

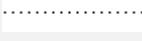

'Color' — Color of circle edge`'red'` (default) | [R G B] | short name | long name

Color of circle edges, specified as a MATLAB `ColorSpec` value.

'LineStyle' — Line style of circle edge

'-' (default) | '--' | ':'

Line style of circle edge, specified as the comma-separated pair consisting of 'LineStyle' and any line specifier in the table below.

Line Style	Description	Resulting Line
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	
'none'	No line	No line

'LineWidth' — Width of circle edge

2 (default) | double

Width of circle edge, specified a positive floating-point double value. Line width is expressed in points, where each point equals 1/72 of an inch.

Output Arguments

h — Circles drawn

handle to an hggroup object

Circles drawn, returned as a handle to an hggroup object. This object is a child of the axes object, ax.

See Also

See Also

imdistline | imfindcircles | imtool

Introduced in R2012a

warp

Display image as texture-mapped surface

Syntax

```
warp(X, map)
warp(I, n)
warp(BW)
warp(RGB)
warp(Z, ___ )
warp(X, Y, Z, ___ )
h = warp( ___ )
```

Description

`warp(X, map)` displays the indexed image `X` with colormap `map` as a texture map on a simple rectangular surface.

`warp(I, n)` displays the intensity image `I` with `n` levels as a texture map on a simple rectangular surface.

`warp(BW)` displays the binary image `BW` as a texture map on a simple rectangular surface.

`warp(RGB)` displays the truecolor image `RGB` as a texture map on a simple rectangular surface.

`warp(Z, ___)` displays the image on the surface `Z`.

`warp(X, Y, Z, ___)` displays the image on the surface `(X, Y, Z)`.

`h = warp(___)` returns a handle to the texture-mapped surface.

Examples

Warp Indexed Image over Curved Surface

This example shows how to warp an indexed image over a nonuniform surface. This example uses a curved surface centered at the origin.

Read an indexed image into the workspace.

```
[I,map] = imread('forest.tif');
```

Create the surface. First, define the x - and y -coordinates of the surface. This example uses arbitrary coordinates that are unrelated to the indexed image. Note that the size of the coordinate matrices X and Y do not need to match the size of the image.

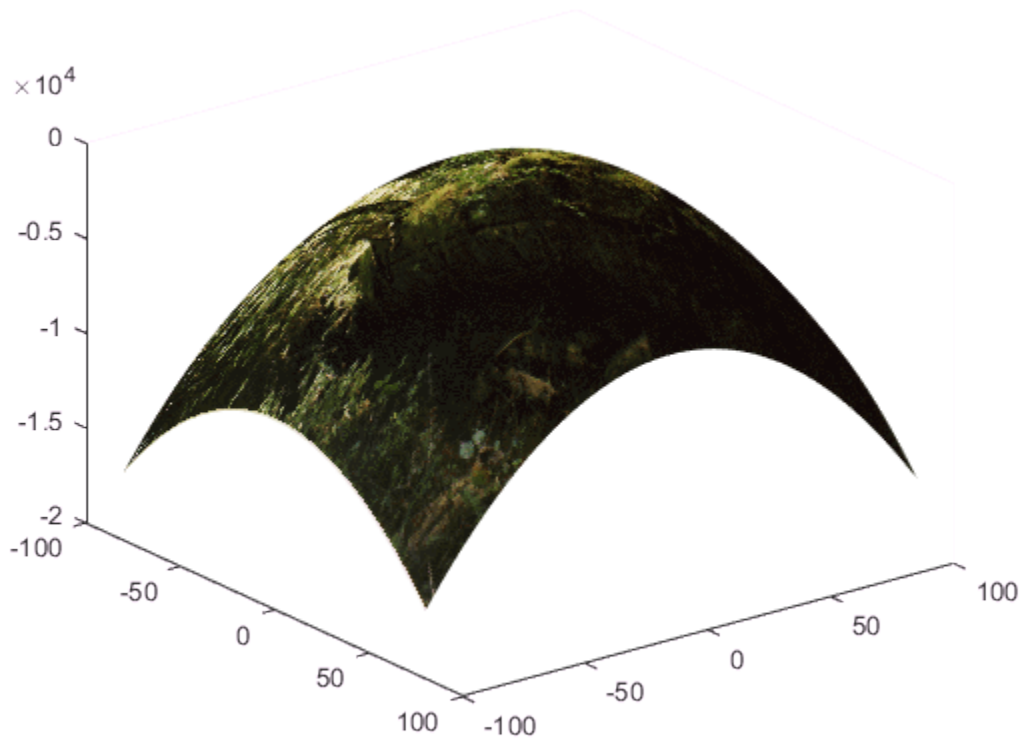
```
[X,Y] = meshgrid(-100:100,-80:80);
```

Define the height Z of the surface at the coordinates given by (X,Y) .

```
Z = -(X.^2 + Y.^2);
```

Warp the image over the surface defined by the coordinates (X,Y,Z) .

```
figure  
warp(X,Y,Z,I,map);
```



Explore the warped image interactively using the rotate and data cursor tools.

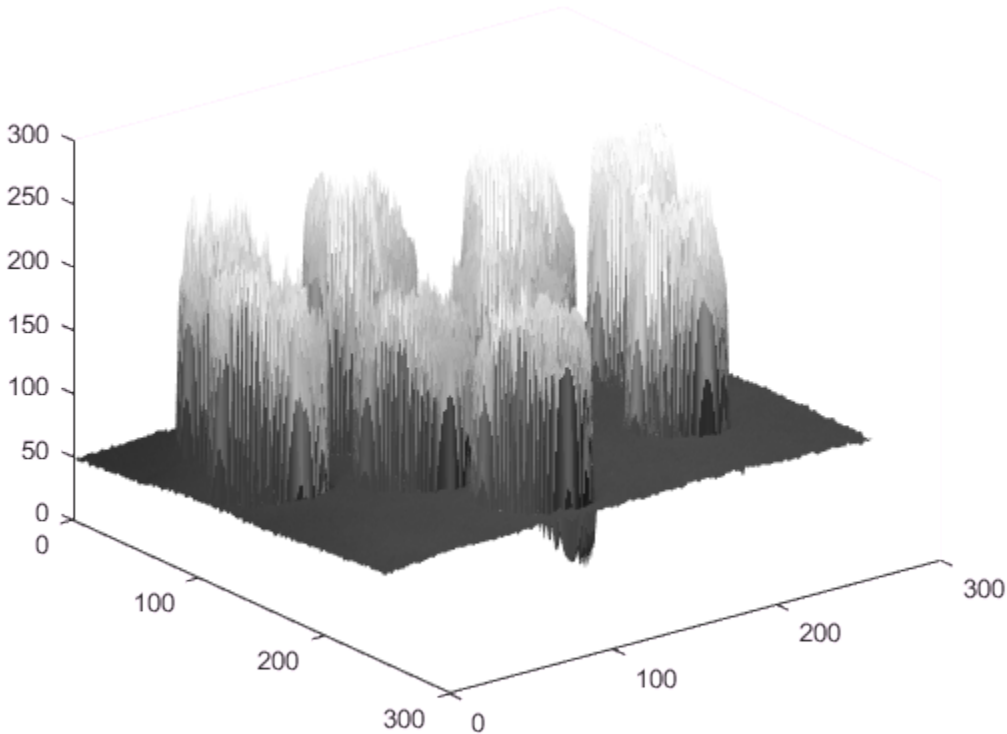
Warp Grayscale Image Based on Intensity

Read a grayscale image into the workspace.

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

Warp the image over the surface whose height is equal to the intensity of the image I. Specify the number of graylevels.

```
figure  
warp(I,I,128);
```



Note that the x - and y -coordinates of the surface were not specified in the call to `warp` and thus default to the image pixel indices. Explore the warped image interactively using the rotate and data cursor tools.

Input Arguments

X — Indexed image

2-D array of real numeric values

Indexed image, specified as a 2-D array of real numeric values. The values in X are an index into `map`, an n -by-3 array of RGB values.

Data Types: `single` | `double` | `uint8` | `uint16` | `int16` | `logical`

map — Colormap

n-by-3 array of real numeric values

Colormap, specified as an *n*-by-3 array of real numeric values. Each row specifies an RGB color value. When `map` is type `single` or `double`, values must be in the range [0, 1].

Data Types: `single` | `double` | `uint8`

I — Intensity image

2-D array of real numeric values

Intensity image, specified as a 2-D array of real numeric values.

Data Types: `single` | `double` | `uint8` | `uint16` | `int16` | `logical`

n — Number of grayscale levels

positive integer

Number of grayscale levels, specified as a positive integer.

Data Types: `double` | `uint8` | `uint16` | `logical`

BW — Binary image

2-D array of logical values

Binary image, specified as a 2-D array of logical values.

Data Types: `single` | `double` | `uint8` | `uint16` | `int16` | `logical`

RGB — Truecolor image

m-by-*n*-by-3 array of real numeric values

Truecolor image, specified as an *m*-by-*n*-by-3 array of real numeric values.

Data Types: `single` | `double` | `uint8` | `uint16` | `int16` | `logical`

Z — Height of surface

2-D array of real numeric values

Height of surface, specified as a 2-D array of logical values. When `Z` is not specified, the surface is flat with a uniform height of 0.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64` | `logical`

X — x-coordinates of surface

2-D array of real numeric values

x-coordinates of surface, specified as a 2-D array of real numeric values.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64` | `logical`

Y — y-coordinates of surface

2-D array of real numeric values

y-coordinates of surface, specified as a 2-D array of real numeric values.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64` | `logical`

Output Arguments

h — Texture-mapped surface object created by warp

texture-mapped surface object

Texture-mapped surface object created by `warp`, specified as a texture-mapped surface object.

Tips

- Texture-mapped surfaces are generally rendered more slowly than images.

See Also

See Also

`image` | `imagesc` | `imshow` | `surf`

Introduced before R2006a

watershed

Watershed transform

Syntax

```
L = watershed(A)
L = watershed(A,conn)
```

Description

`L = watershed(A)` returns a label matrix `L` that identifies the watershed regions of the input matrix `A`, which can have any dimension. The watershed transform finds "catchment basins" or "watershed ridge lines" in an image by treating it as a surface where light pixels represent high elevations and dark pixels represent low elevations. The elements of `L` are integer values greater than or equal to 0. The elements labeled 0 do not belong to a unique watershed region. The elements labeled 1 belong to the first watershed region, the elements labeled 2 belong to the second watershed region, and so on. By default, `watershed` uses 8-connected neighborhoods for 2-D inputs and 26-connected neighborhoods for 3-D inputs. For higher dimensions, `watershed` uses the connectivity given by `conndef(ndims(A), 'maximal')`.

`L = watershed(A,conn)` specifies the connectivity to be used in the watershed computation.

Examples

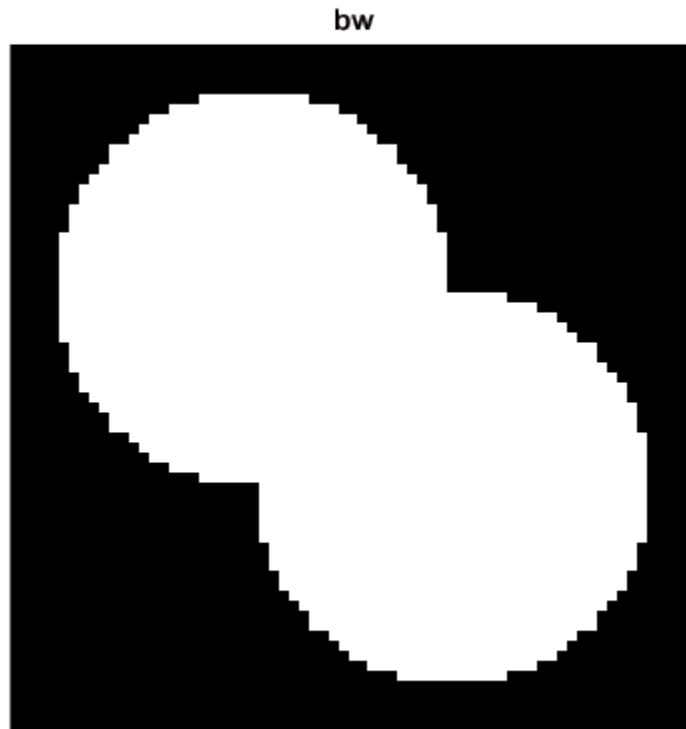
Compute Watershed Transform and Display Resulting Label Matrix

Compute the watershed transform and display the resulting label matrix as an RGB image. This example works with 2-D images.

Create a binary image containing two overlapping circular objects and display it.

```
center1 = -10;
center2 = -center1;
dist = sqrt(2*(2*center1)^2);
radius = dist/2 * 1.4;
```

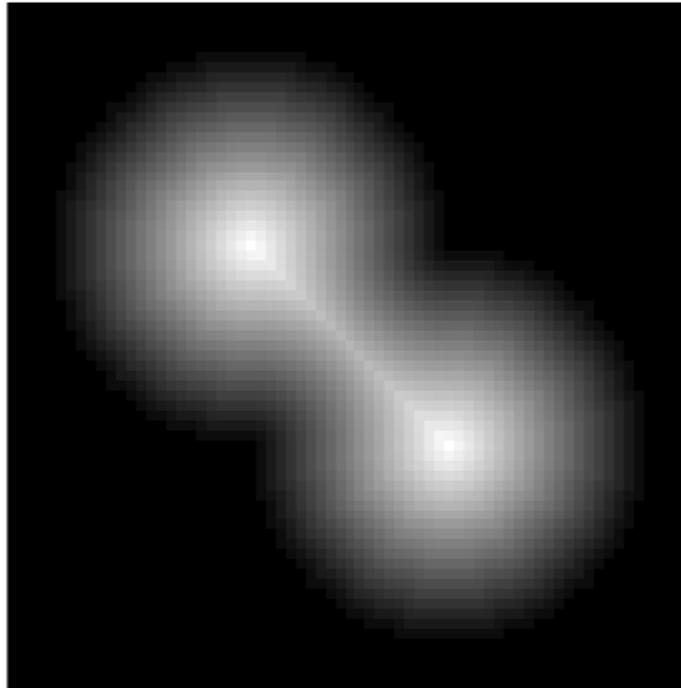
```
lims = [floor(center1-1.2*radius) ceil(center2+1.2*radius)];  
[x,y] = meshgrid(lims(1):lims(2));  
bw1 = sqrt((x-center1).^2 + (y-center1).^2) <= radius;  
bw2 = sqrt((x-center2).^2 + (y-center2).^2) <= radius;  
bw = bw1 | bw2;  
figure  
imshow(bw, 'InitialMagnification','fit'), title('bw')
```



Compute the distance transform of the complement of the binary image.

```
D = bwdist(~bw);  
figure  
imshow(D,[], 'InitialMagnification','fit')  
title('Distance transform of ~bw')
```

Distance transform of ~bw



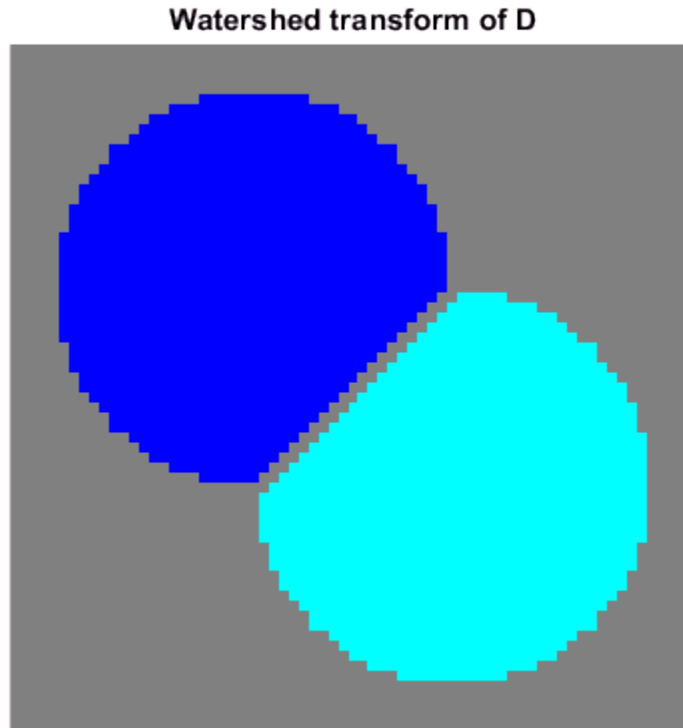
Complement the distance transform, and force pixels that don't belong to the objects to be at Inf .

```
D = -D;  
D(~bw) = Inf;
```

Compute the watershed transform and display the resulting label matrix as an RGB image.

```
L = watershed(D);  
L(~bw) = 0;  
rgb = label2rgb(L, 'jet', [.5 .5 .5]);  
figure  
imshow(rgb, 'InitialMagnification', 'fit')
```

```
title('Watershed transform of D')
```



Compute Watershed Transform of 3-D Binary Image

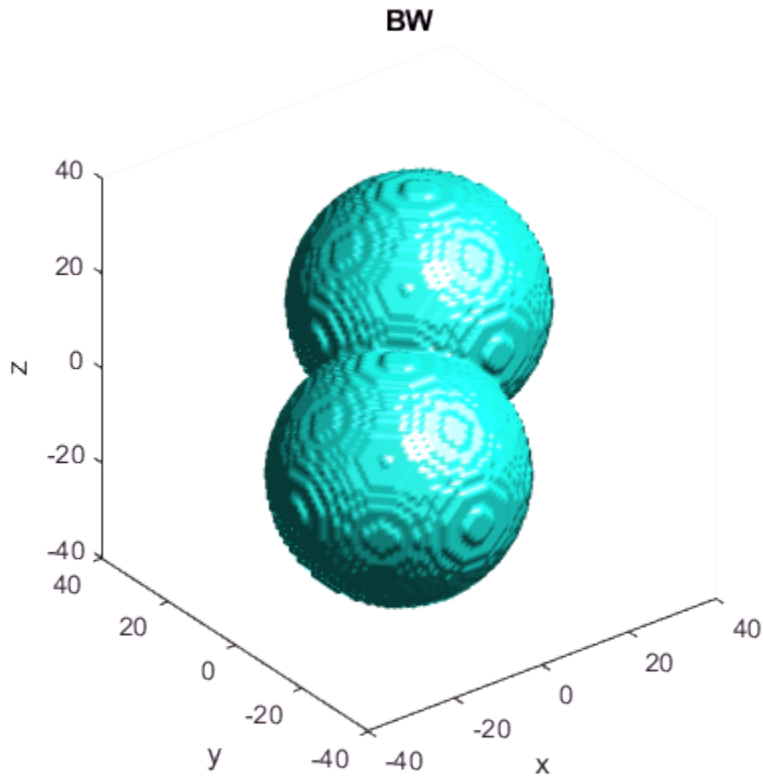
Make a 3-D binary image containing two overlapping spheres.

```
center1 = -10;  
center2 = -center1;  
dist = sqrt(3*(2*center1)^2);  
radius = dist/2 * 1.4;  
lims = [floor(center1-1.2*radius) ceil(center2+1.2*radius)];  
[x,y,z] = meshgrid(lims(1):lims(2));  
bw1 = sqrt((x-center1).^2 + (y-center1).^2 + ...  
          (z-center1).^2) <= radius;
```

```

bw2 = sqrt((x-center2).^2 + (y-center2).^2 + ...
           (z-center2).^2) <= radius;
bw = bw1 | bw2;
figure, isosurface(x,y,z,bw,0.5), axis equal, title('BW')
xlabel x, ylabel y, zlabel z
xlim(lims), ylim(lims), zlim(lims)
view(3), camlight, lighting gouraud

```



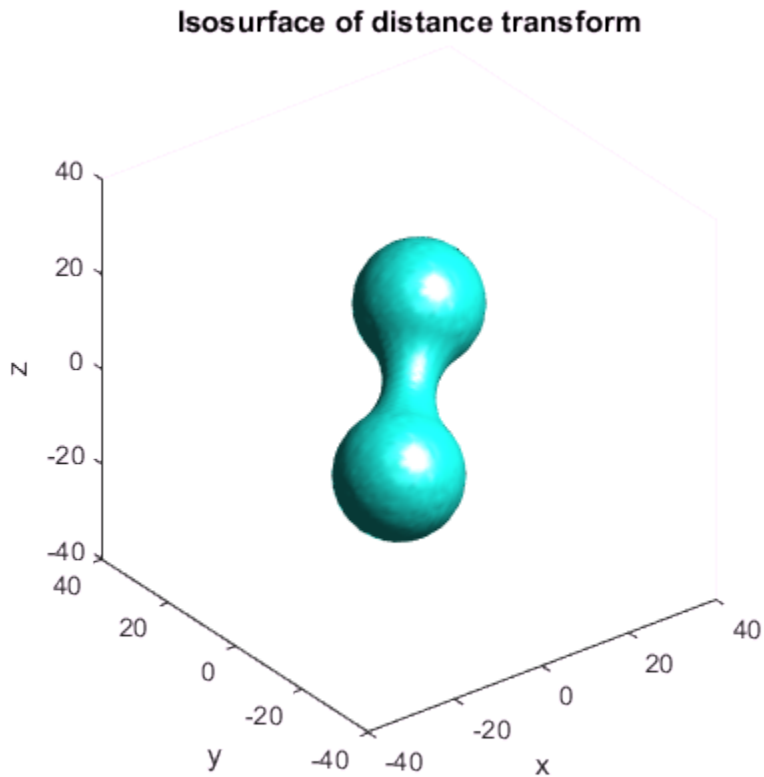
Compute the distance transform.

```

D = bwdist(~bw);
figure, isosurface(x,y,z,D,radius/2), axis equal
title('Isosurface of distance transform')
xlabel x, ylabel y, zlabel z
xlim(lims), ylim(lims), zlim(lims)

```

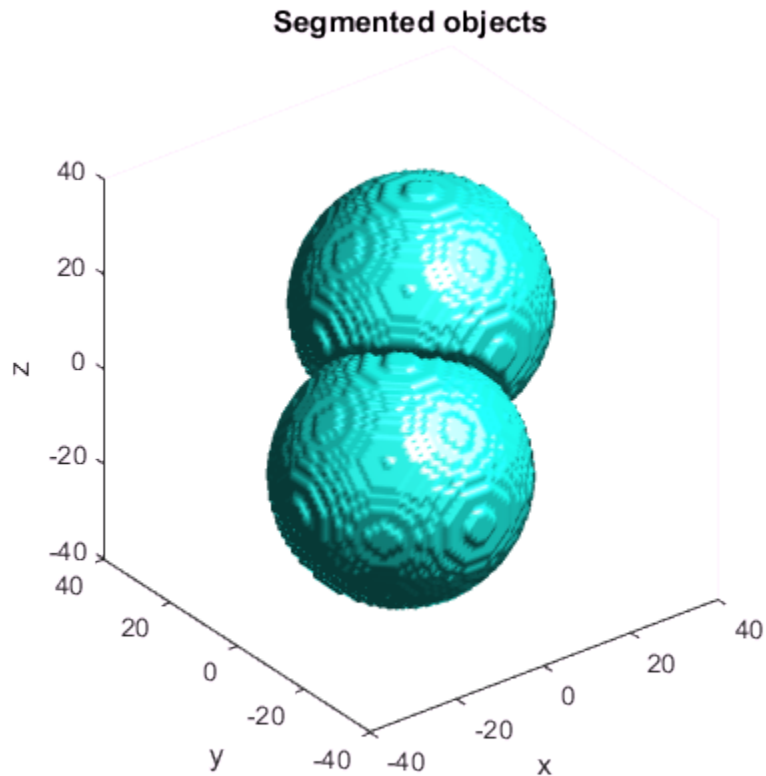
```
view(3), camlight, lighting gouraud
```



Complement the distance transform, force nonobject pixels to be Inf, and then compute the watershed transform.

```
D = ~D;
D(~bw) = Inf;
L = watershed(D);
L(~bw) = 0;
figure
isosurface(x,y,z,L==1,0.5)
isosurface(x,y,z,L==2,0.5)
axis equal
title('Segmented objects')
```

```
xlabel x, ylabel y, zlabel z  
xlim(lims), ylim(lims), zlim(lims)  
view(3), camlight, lighting gouraud
```



Input Arguments

A — Input image

real, nonsparse, numeric or logical array of any dimension

Input image, specified as a real, nonsparse, numeric, or logical array of any dimension.

Example: `RGB = imread('pears.png');` `I = rgb2gray(RGB);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

conn — Connectivity

8 (for 2-D images) (default) | 4 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix

Connectivity, specified as one of the values in this table.

Value	Meaning
Two-dimensional connectivities	
4	4-connected neighborhood
8	8-connected neighborhood
Three-dimensional connectivities	
6	6-connected neighborhood
18	18-connected neighborhood
26	26-connected neighborhood

Connectivity can also be defined in a more general way for any dimension by using for `conn` a 3-by-3-by- ... -by-3 matrix of 0s and 1s. The 1-valued elements define neighborhood locations relative to the center element of `conn`, which must be symmetric around its center element.

Note If you specify a nondefault connectivity, pixels on the edge of the image might not be considered to be border pixels. For example, if `conn = [0 0 0; 1 1 1; 0 0 0]`, elements on the first and last row are not considered to be border pixels because, according to that connectivity definition, they are not connected to the region outside the image.

Example: `L = watershed(I,4);`

Data Types: `double` | `logical`

Output Arguments

L — Label matrix

numeric array of unsigned integers

Label matrix, specified as a numeric array of unsigned integers.

Tips

- The watershed transform algorithm used by this function changed in version 5.4 (R2007a) of the Image Processing Toolbox software. The previous algorithm occasionally produced labeled watershed basins that were not contiguous. If you need to obtain the same results as the previous algorithm, use the function `watershed_old`.

Algorithms

`watershed` uses the Fernand Meyer algorithm [1].

References

- [1] Meyer, Fernand, "Topographic distance and watershed lines," *Signal Processing*, Vol. 38, July 1994, pp. 113-125.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic **MATLAB Host Computer** target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see "Understanding Code Generation with Image Processing Toolbox".
- Supports only 2-D images

- Supports only 4 or 8 connectivity
- Supports images containing up to 65,535 regions
- Output type is always `uint16`

See Also

See Also

`bwdist` | `bwlabel` | `bwlabeln` | `regionprops`

Introduced before R2006a

whitepoint

XYZ color values of standard illuminants

Syntax

```
xyz = whitepoint(illuminant)
xyz = whitepoint
```

Description

`xyz = whitepoint(illuminant)` returns `xyz`, a three-element row vector of *XYZ* values scaled so that $Y = 1$. `illuminant` specifies the white reference illuminant. The following table lists all the possible values.

Value	White Point
'a'	CIE standard illuminant A, [1.0985, 1.0000, 0.3558]. Simulates typical, domestic, tungsten-filament lighting with correlated color temperature of 2856 K.
'c'	CIE standard illuminant C, [0.9807, 1.0000, 1.1822]. Simulates average or north sky daylight with correlated color temperature of 6774 K. Deprecated by CIE.
'e'	Equal-energy radiator. Useful as a theoretical reference. [1.000, 1.000, 1.000].
'd50'	CIE standard illuminant D50, [0.9642, 1.0000, 0.8251]. Simulates warm daylight at sunrise or sunset with correlated color temperature of 5003 K. Also known as <i>horizon light</i> .
'd55'	CIE standard illuminant D55, [0.9568, 1.0000, 0.9214]. Simulates mid-morning or mid-afternoon daylight with correlated color temperature of 5500 K.
'd65'	CIE standard illuminant D65, [0.9504, 1.0000, 1.0888]. Simulates noon daylight with correlated color temperature of 6504 K.
'icc'	Profile Connection Space (PCS) illuminant used in ICC profiles. Approximation of [0.962, 1.000, 0.8249] using fixed-point, signed,

Value	White Point
	32-bit numbers with 16 fractional bits. Actual value: [31595,32768,27030]/32768.

`xyz = whitepoint` is the same as `xyz = whitepoint('icc')`.

Class Support

`illuminant` is a character array. `xyz` is of class `double`.

Examples

Return the *XYZ* color space representation of the default white reference illuminant 'icc'.

```
wp_icc = whitepoint
```

```
wp_icc =
```

```
    0.9642    1.0000    0.8249
```

See Also

[applycform](#) | [lab2double](#) | [lab2uint8](#) | [lab2uint16](#) | [makecform](#) | [xyz2double](#) | [xyz2uint16](#)

Introduced before R2006a

wiener2

2-D adaptive noise-removal filtering

Compatibility

The syntax `wiener2(I,[m n],[mblock nblock],noise)` has been removed. Use the `wiener2(I,[m n],noise)` syntax instead.

Syntax

```
J = wiener2(I,[m n],noise)
[J,noise] = wiener2(I,[m n])
```

Description

`wiener2` lowpass-filters a grayscale image that has been degraded by constant power additive noise. `wiener2` uses a pixelwise adaptive Wiener method based on statistics estimated from a local neighborhood of each pixel.

`J = wiener2(I,[m n],noise)` filters the image `I` using pixelwise adaptive Wiener filtering, using neighborhoods of size `m`-by-`n` to estimate the local image mean and standard deviation. If you omit the `[m n]` argument, `m` and `n` default to 3. The additive noise (Gaussian white noise) power is assumed to be `noise`.

`[J,noise] = wiener2(I,[m n])` also estimates the additive noise power before doing the filtering. `wiener2` returns this estimate in `noise`.

Class Support

The input image `I` is a two-dimensional image of class `uint8`, `uint16`, `int16`, `single`, or `double`. The output image `J` is of the same size and class as `I`.

Examples

Remove Noise By Adaptive Filtering

This example shows how to use the `wiener2` function to apply a Wiener filter (a type of linear filter) to an image adaptively. The Wiener filter tailors itself to the local image variance. Where the variance is large, `wiener2` performs little smoothing. Where the variance is small, `wiener2` performs more smoothing.

This approach often produces better results than linear filtering. The adaptive filter is more selective than a comparable linear filter, preserving edges and other high-frequency parts of an image. In addition, there are no design tasks; the `wiener2` function handles all preliminary computations and implements the filter for an input image. `wiener2`, however, does require more computation time than linear filtering.

`wiener2` works best when the noise is constant-power ("white") additive noise, such as Gaussian noise. The example below applies `wiener2` to an image of Saturn with added Gaussian noise.

Read the image into the workspace.

```
RGB = imread('saturn.png');
```

Convert the image from truecolor to grayscale.

```
I = rgb2gray(RGB);
```

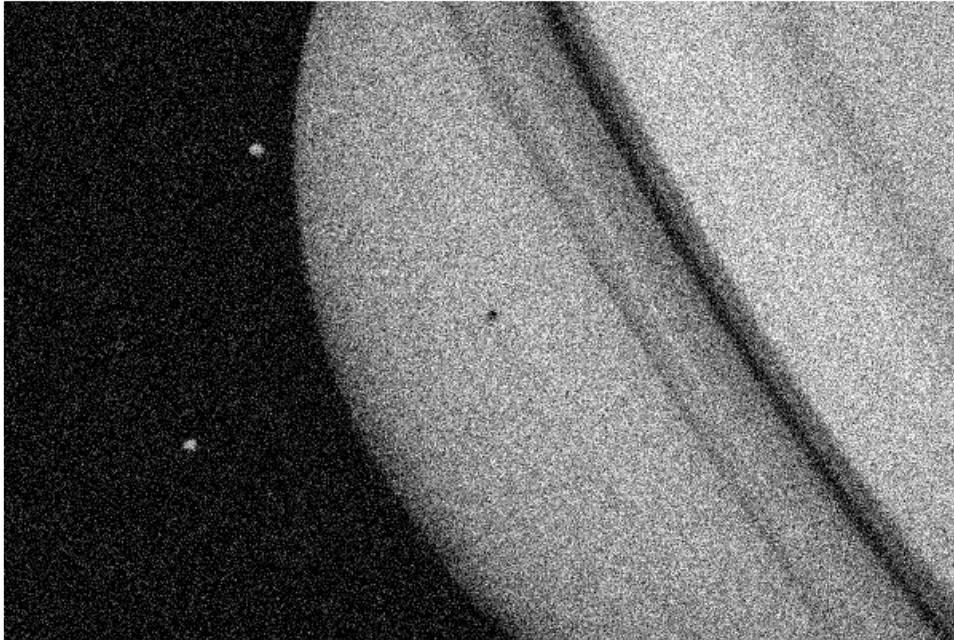
Add Gaussian noise to the image

```
J = imnoise(I,'gaussian',0,0.025);
```

Display the noisy image. Because the image is quite large, display only a portion of the image.

```
imshow(J(600:1000,1:600));  
title('Portion of the Image with Added Gaussian Noise');
```

Portion of the Image with Added Gaussian Noise



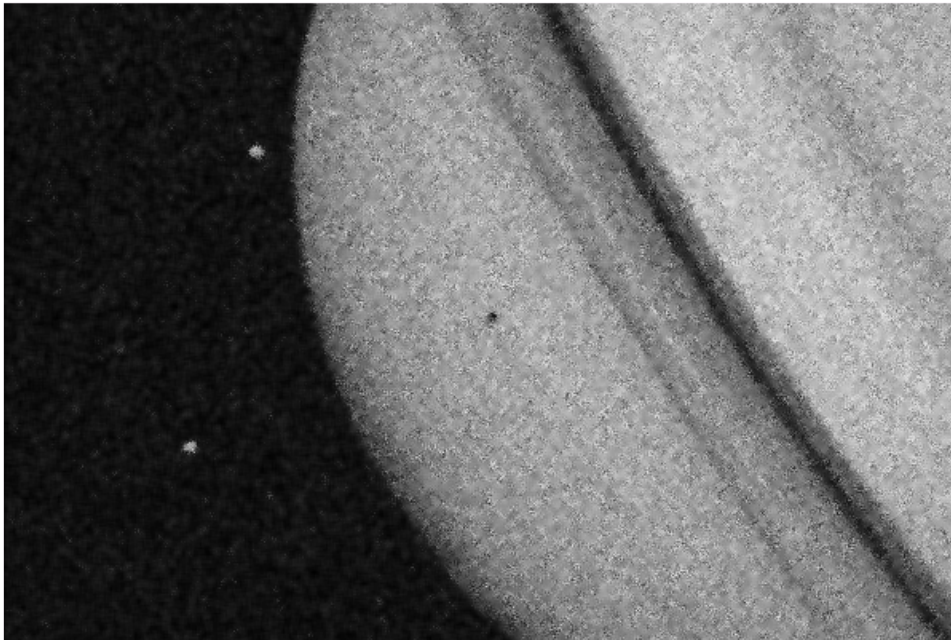
Remove the noise using the `wiener2` function.

```
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



Algorithms

wiener2 estimates the local mean and variance around each pixel.

$$\mu = \frac{1}{NM} \sum_{n_1, n_2 \in \eta} a(n_1, n_2)$$

and

$$\sigma^2 = \frac{1}{NM} \sum_{n_1, n_2 \in \eta} a^2(n_1, n_2) - \mu^2,$$

where η is the N -by- M local neighborhood of each pixel in the image A . `wiener2` then creates a pixelwise Wiener filter using these estimates,

$$b(n_1, n_2) = \mu + \frac{\sigma^2 - v^2}{\sigma^2} (a(n_1, n_2) - \mu),$$

where v^2 is the noise variance. If the noise variance is not given, `wiener2` uses the average of all the local estimated variances.

References

- [1] Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, p. 548, equations 9.26, 9.27, and 9.29.

See Also

`filter2` | `medfilt2`

Introduced before R2006a

xyz2double

Convert *XYZ* color values to `double`

Syntax

```
xyxd = xyz2double(XYZ)
```

Description

`xyxd = xyz2double(XYZ)` converts an *M*-by-3 or *M*-by-*N*-by-3 array of *XYZ* color values to `double`. `xyxd` has the same size as *XYZ*.

The Image Processing Toolbox software follows the convention that double-precision *XYZ* arrays contain 1931 CIE *XYZ* values. *XYZ* arrays that are `uint16` follow the convention in the ICC profile specification (ICC.1:2001-4, www.color.org) for representing *XYZ* values as unsigned 16-bit integers. There is no standard representation of *XYZ* values as unsigned 8-bit integers. The ICC encoding convention is illustrated by this table.

Value (X, Y, or Z)	uint16 Value
0.0	0
1.0	32768
1.0 + (32767/32768)	65535

Class Support

`xyz` is a `uint16` or `double` array that must be real and nonsparse. `xyzd` is of class `double`.

Examples

Convert *XYZ* Color Values to `double`

This example shows how to convert `uint16`-encoded *XYZ* values to `double`.

Create a `uint16` vector specifying a color in XYZ colorspace.

```
c = uint16([100 32768 65535]);
```

Convert the XYZ color value to double.

```
xyz2double(c)
```

```
ans =
```

```
    0.0031    1.0000    2.0000
```

See Also

[applycform](#) | [lab2double](#) | [lab2uint16](#) | [lab2uint8](#) | [makecform](#) | [whitepoint](#)
| [xyz2uint16](#)

Introduced before R2006a

xyz2rgb

Convert CIE 1931 XYZ to RGB

Syntax

```
rgb = xyz2rgb(xyz)
rgb = xyz2rgb(xyz, Name, Value)
```

Description

`rgb = xyz2rgb(xyz)` converts CIE 1931 XYZ values to RGB values.

`rgb = xyz2rgb(xyz, Name, Value)` specifies additional options with one or more `Name, Value` pair arguments.

Examples

Convert XYZ color to sRGB

Convert a color value in the XYZ color space to the sRGB color space.

```
xyz2rgb([0.25 0.40 0.10])
ans =
    0.4174    0.7434    0.2152
```

Convert XYZ Color to Adobe RGB

Convert the color value in XYZ color space to the Adobe RGB (1998) color space.

```
xyz2rgb([0.25 0.40 0.10], 'ColorSpace', 'adobe-rgb-1998')
ans =
    0.5323    0.7377    0.2730
```

Convert XYZ color to sRGB Specifying Whitepoint

Convert an XYZ color value to sRGB specifying the D50 whitepoint.

```
xyz2rgb([0.25 0.40 0.10], 'WhitePoint', 'd50')
```

```
ans =
```

```
    0.3276    0.7517    0.2869
```

Convert XYZ color to 8-bit-encoded RGB Color

Convert an XYZ color value to an 8-bit encoded RGB color value.

```
xyz2rgb([0.25 0.40 0.10], 'OutputType', 'uint8')
```

```
ans = 1×3 uint8 row vector
```

```
    106    190    55
```

Input Arguments

xyz — Color values to convert

p-by-3 matrix | m-by-n-by-3 image array | m-by-n-by-3-by-f image stack

Color values to convert, specified as a p-by-3 matrix of color values (one color per row), an m-by-n-by-3 image array, or an m-by-n-by-3-by-f image stack.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `xyz2rgb([0.25 0.40 0.10], 'ColorSpace', 'adobe-rgb-1998')`

'ColorSpace' — Color space of the input RGB values

'srgb' (default) | 'adobe-rgb-1998' | 'linear-rgb'

Color space of the input RGB values, specified as 'srgb', 'adobe-rgb-1998', or 'linear-rgb'.

Data Types: char

'WhitePoint' — Reference white point

'd65' (default) | 'a' | 'c' | 'e' | 'd50' | 'd55' | 'icc' | 1-by-3 vector

Reference white point, specified as a 1-by-3 vector or one of the CIE standard illuminants, listed in the following table.

Value	White Point
'a'	CIE standard illuminant A, [1.0985, 1.0000, 0.3558]. Simulates typical, domestic, tungsten-filament lighting with correlated color temperature of 2856 K.
'c'	CIE standard illuminant C, [0.9807, 1.0000, 1.1822]. Simulates average or north sky daylight with correlated color temperature of 6774 K. Deprecated by CIE.
'e'	Equal-energy radiator. Useful as a theoretical reference. [1.000, 1.000, 1.000].
'd50'	CIE standard illuminant D50, [0.9642, 1.0000, 0.8251]. Simulates warm daylight at sunrise or sunset with correlated color temperature of 5003 K. Also known as <i>horizon light</i> .
'd55'	CIE standard illuminant D55, [0.9568, 1.0000, 0.9214]. Simulates mid-morning or mid-afternoon daylight with correlated color temperature of 5500 K.
'd65'	CIE standard illuminant D65, [0.9504, 1.0000, 1.0888]. Simulates noon daylight with correlated color temperature of 6504 K.
'icc'	Profile Connection Space (PCS) illuminant used in ICC profiles. Approximation of [0.962, 1.000, 0.8249] using fixed-point, signed, 32-bit numbers with 16 fractional bits. Actual value: [31595, 32768, 27030]/32768.

Data Types: single | double | char

'OutputType' — Data type of returned RGB values`'double' | 'single' | 'uint8' | 'uint16'`

Data type of returned RGB values, specified as one of the following values: `'double'`, `'single'`, `'uint8'`, or `'uint16'`. If you do not specify `OutputType`, the output type is the same type as the input.

Data Types: char

Output Arguments

rgb — Converted color values

Array the same shape as the input

Converted color values, returned as an array the same shape as the input. The output type is the same as the input type unless you specify the `'OutputType'` parameter.

See Also

See Also

`rgb2xyz`

Introduced in R2014b

xyz2lab

Convert CIE 1931 XYZ to CIE 1976 L*a*b*

Syntax

```
lab = xyz2lab(xyz)
lab = xyz2lab(xyz,Name,Value)
```

Description

`lab = xyz2lab(xyz)` converts CIE 1931 XYZ values to CIE 1976 L*a*b* values.

`lab = xyz2lab(xyz,Name,Value)` specifies additional options with one or more Name,Value pair arguments.

Examples

Convert XYZ Color to L*a*b*

Convert an XYZ color value to L*a*b* using the default reference white point, D65.

```
xyz2lab([0.25 0.40 0.10])
ans =
    69.4695   -48.0439    57.1259
```

Convert XYZ Color to L*a*b* Specifying Whitepoint

Convert an XYZ color value to L*a*b* specifying the D50 whitepoint.

```
xyz2lab([0.25 0.40 0.10], 'WhitePoint', 'd50')
ans =
```


69.4695 -49.5717 48.3864

Input Arguments

xyz — Color values to convert

P-by-3 matrix | M-by-N-by-3 image array | M-by-N-by-3-by-F image stack

Color values to convert, specified as a P-by-3 matrix of color values (one color per row), an M-by-N-by-3 image array, or an M-by-N-by-3-by-F image stack.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, . . . , **NameN**, **ValueN**.

Example: `xyz2lab([0.25 0.40 0.10], 'WhitePoint', 'd50')`

'WhitePoint' — Reference white point

'd65' (default) | 'a' | 'c' | 'e' | 'd50' | 'd55' | 'icc' | 1-by-3 vector

Reference white point, specified as a 1-by-3 vector or one of the CIE standard illuminants, listed in the following table.

Value	White Point
'a'	CIE standard illuminant A, [1.0985, 1.0000, 0.3558]. Simulates typical, domestic, tungsten-filament lighting with correlated color temperature of 2856 K.
'c'	CIE standard illuminant C, [0.9807, 1.0000, 1.1822]. Simulates average or north sky daylight with correlated color temperature of 6774 K. Deprecated by CIE.
'e'	Equal-energy radiator. Useful as a theoretical reference. [1.000, 1.000, 1.000].

Value	White Point
'd50'	CIE standard illuminant D50, [0.9642, 1.0000, 0.8251]. Simulates warm daylight at sunrise or sunset with correlated color temperature of 5003 K. Also known as <i>horizon light</i> .
'd55'	CIE standard illuminant D55, [0.9568, 1.0000, 0.9214]. Simulates mid-morning or mid-afternoon daylight with correlated color temperature of 5500 K.
'd65'	CIE standard illuminant D65, [0.9504, 1.0000, 1.0888]. Simulates noon daylight with correlated color temperature of 6504 K.
'icc'	Profile Connection Space (PCS) illuminant used in ICC profiles. Approximation of [0.962, 1.000, 0.8249] using fixed-point, signed, 32-bit numbers with 16 fractional bits. Actual value: [31595, 32768, 27030] / 32768.

Data Types: single | double | char

Output Arguments

1ab — Converted color values

array the same shape and type as the input

Converted color values, returned as an array the same shape and type as the input.

See Also

See Also

rgb2lab | rgb2xyz | rgb2xyz | xyz2rgb

Introduced in R2014b

xyz2uint16

Convert *XYZ* color values to `uint16`

Syntax

```
xyz16 = xyz2uint16(xyz)
```

Description

`xyz16 = xyz2uint16(xyz)` converts an *M*-by-3 or *M*-by-*N*-by-3 array of *XYZ* color values to `uint16`. `xyz16` has the same size as `xyz`.

The Image Processing Toolbox software follows the convention that double-precision *XYZ* arrays contain 1931 CIE *XYZ* values. *XYZ* arrays that are `uint16` follow the convention in the ICC profile specification (ICC.1:2001-4, www.color.org) for representing *XYZ* values as unsigned 16-bit integers. There is no standard representation of *XYZ* values as unsigned 8-bit integers. The ICC encoding convention is illustrated by this table.

Value (X, Y, or Z)	uint16 Value
0.0	0
1.0	32768
1.0 + (32767/32768)	65535

Class Support

`xyz` is a `uint16` or `double` array that must be real and nonsparse. `xyz16` is `uint8`.

Examples

Convert XYZ Color Values to uint16

This example shows how to convert *XYZ* color values from `double` to `uint16`.

Create a double vector specifying a color in XYZ colorspace.

```
c = [0.1 0.5 1.0];
```

Convert the XYZ color value to uint16.

```
xyz2uint16(c)
```

```
ans = 1×3 uint16 row vector
```

```
    3277    16384    32768
```

See Also

[applycform](#) | [lab2double](#) | [lab2uint16](#) | [lab2uint8](#) | [makecform](#) | [whitepoint](#)
| [xyz2double](#)

Introduced before R2006a

ybcrc2rgb

Convert YCbCr color values to RGB color space

Syntax

```
rgbmap = ybcrc2rgb(ybcrcmap)
RGB = ybcrc2rgb(YCBCR)
gpuarrayB = ybcrc2rgb(gpuarrayA)
```

Description

`rgbmap = ybcrc2rgb(ybcrcmap)` converts the YCbCr color space values in `ybcrcmap` to the RGB color space. `ybcrcmap` is an m -by-3 matrix that contains the YCbCr luminance (Y) and chrominance (Cb and Cr) color values as columns. Each row in `rgbmap` represents the equivalent color to the corresponding row in `ybcrcmap`.

`RGB = ybcrc2rgb(YCBCR)` converts the YCbCr image `YCBCR` to the equivalent truecolor image `RGB`.

`gpuarrayB = ybcrc2rgb(gpuarrayA)` performs the conversion on a GPU. The input image, `gpuarrayA`, is a `gpuArray` containing YCbCr color space values or a YCbCr image. The output is a `gpuArray` containing RGB color space values or an RGB image, depending on the input type. This syntax requires the Parallel Computing Toolbox.

Examples

Convert Image from YCbCr to RGB

This example shows how to convert an image from RGB to YCbCr color space and back.

Read an RGB image into the workspace.

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

Convert the image to YCbCr color space.

```
YCBCR = rgb2ybcrc(RGB);
```

Convert the YCbCr image back to RGB color space.

```
RGB2 = ycbcr2rgb(YCBCR);
```

Display the luminance channel of the image in YCbCr color space alongside the image that was converted from YCbCr to RGB color space.

```
figure  
subplot(1,2,1)  
imshow(YCBCR(:,:,1))  
title('Original Luminance (Y)');  
subplot(1,2,2)  
imshow(RGB2);  
title('Image Converted to RGB');
```

Original Luminance (Y)

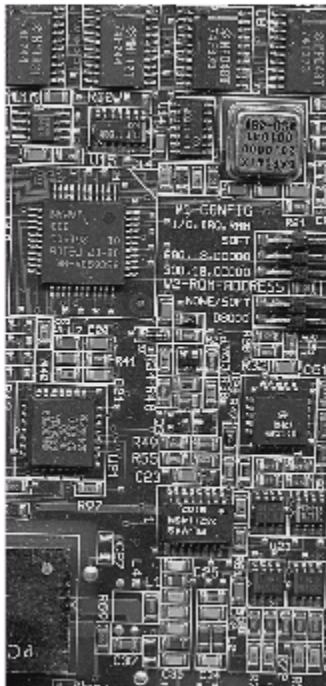
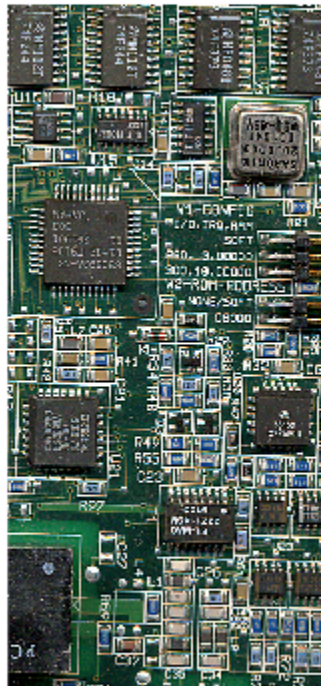


Image Converted to RGB



Input Arguments

ycbcrmap — YCbCr color space values

`m-by-3` array

YCbCr color space values, specified as an `m-by-3` array. The first column corresponds to luminance `Y`. The second and third columns correspond to chrominance `Cb` and `Cr`.

Data Types: `single` | `double`

YCBCR — YCbCr image

`m-by-n-by-3` array

YCbCr image, specified as an `m-by-n-by-3` array.

Data Types: `single` | `double` | `uint8` | `uint16`

gpuarrayA — YCbCr color space values or YCbCr image to be processed on a graphics processing unit (GPU)

`gpuArray` object

YCbCr color space values or YCbCr image to be processed on a graphics processing unit (GPU), specified as a `gpuArray` object.

Output Arguments

rgbmap — RGB color space values

`m-by-3` array

RGB color space values, returned as an `m-by-3` array. The three columns represent the red, green, and blue channels.

RGB — Image in RGB color space

`m-by-n-by-3` array

Image in RGB color space, returned as an `m-by-n-by-3` array.

gpuarrayB — Output in RGB color space when run on a graphics processing unit (GPU)

`gpuArray` object

Output in RGB color space when run on a graphics processing unit (GPU), returned as a `gpuArray` object. The output is either an array of RGB color space values or an RGB image, depending on the input type.

References

- [1] Poynton, C. A. *A Technical Introduction to Digital Video*, John Wiley & Sons, Inc., 1996, p. 175.
- [2] Rec. ITU-R BT.601-5, *Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-screen 16:9 Aspect Ratios*, (1982-1986-1990-1992-1994-1995), Section 3.5.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This function supports the generation of C code using MATLAB Coder. Note that if you choose the generic `MATLAB Host Computer` target platform, the function generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated. For more information, see “Understanding Code Generation with Image Processing Toolbox”.

See Also

See Also

`gpuArray` | `ntsc2rgb` | `rgb2ntsc` | `rgb2ycbcr`

Introduced before R2006a