

# Image Processing Toolbox™

Reference



# MATLAB®

R2018b

 MathWorks®

# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://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 1993–2018 by The MathWorks, Inc.

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

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

### **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://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](http://www.mathworks.com/patents) for more information.

## Revision History

August 1993	First printing	Version 1
May 1997	Second printing	Version 2
April 2001	Third printing	Revised for Version 3.0
June 2001	Online only	Revised for Version 3.1 (Release 12.1)
July 2002	Online only	Revised for Version 3.2 (Release 13)
May 2003	Fourth printing	Revised for Version 4.0 (Release 13.0.1)
September 2003	Online only	Revised for Version 4.1 (Release 13.SP1)
June 2004	Online only	Revised for Version 4.2 (Release 14)
August 2004	Online only	Revised for Version 5.0 (Release 14+)
October 2004	Fifth printing	Revised for Version 5.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 5.0.2 (Release 14SP2)
September 2005	Online only	Revised for Version 5.1 (Release 14SP3)
March 2006	Online only	Revised for Version 5.2 (Release 2006a)
September 2006	Online only	Revised for Version 5.3 (Release 2006b)
March 2007	Online only	Revised for Version 5.4 (Release 2007a)
September 2007	Online only	Revised for Version 6.0 (Release 2007b)
March 2008	Online only	Revised for Version 6.1 (Release 2008a)
October 2008	Online only	Revised for Version 6.2 (Release 2008b)
March 2009	Online only	Revised for Version 6.3 (Release 2009a)
September 2009	Online only	Revised for Version 6.4 (Release 2009b)
March 2010	Online only	Revised for Version 7.0 (Release 2010a)
September 2010	Online only	Revised for Version 7.1 (Release 2010b)
April 2011	Online only	Revised for Version 7.2 (Release 2011a)
September 2011	Online only	Revised for Version 7.3 (Release 2011b)
March 2012	Online only	Revised for Version 8.0 (Release 2012a)
September 2012	Online only	Revised for Version 8.1 (Release 2012b)
March 2013	Online only	Revised for Version 8.2 (Release 2013a)
September 2013	Online only	Revised for Version 8.3 (Release 2013b)
March 2014	Online only	Revised for Version 9.0 (Release 2014a)
October 2014	Online only	Revised for Version 9.1 (Release 2014b)
March 2015	Online only	Revised for Version 9.2 (Release 2015a)
September 2015	Online only	Revised for Version 9.3 (Release 2015b)
March 2016	Online only	Revised for Version 9.4 (Release 2016a)
September 2016	Online only	Revised for Version 9.5 (Release 2016b)
March 2017	Online only	Revised for Version 10.0 (Release 2017a)
September 2017	Online only	Revised for Version 10.1 (Release 2017b)
March 2018	Online only	Revised for Version 10.2 (Release 2018a)
September 2018	Online only	Revised for Version 10.3 (Release 2018b)



<b>1</b>	<b>Functions – Alphabetical List</b>
----------	--------------------------------------



# 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(IMG)` opens the Color Thresholder app, loading the image `IMG` into the app.

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



## See Also

### **Apps**

**Image Segmenter**

### **Functions**

`imcontrast`

### **Topics**

“Image Segmentation Using the Color Thresholder App”

**Introduced in R2014a**

## DICOM Browser

Explore collection of DICOM files

### Description

The **DICOM Browser** app lets you explore the contents of collections of DICOM files. The app sorts images by study and series. You can select a series and save it to the MATLAB workspace. The DICOM Browser stores the data as a volume, with separate variables for a colormap and for spatial details.

### Open the DICOM Browser App

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

### Examples

#### Explore by Folder Name

Open the DICOM Browser, displaying DICOM files from the sample image folder.

```
dicomBrowser(fullfile(matlabroot, 'toolbox/images/imdata'))
```

#### Explore by DICOMDIR File

Open the DICOM Browser and explore a DICOM folder by using the DICOMDIR file.

```
dicomBrowser(fullfile(matlabroot, 'toolbox/images/imdata/DICOMDIR'))
```

#### Programmatic Use

`dicomBrowser` opens the DICOM Browser app for exploring the contents of collections of DICOM files.

`dicomBrowser(DIR)` opens the DICOM Browser app, displaying details about the files in the folder `DIR` and its subfolders. `DIR` can contain a full path name, a relative path name to the file, or the name of a file on the MATLAB search path.

`dicomBrowser(DICOMDIR)` opens the DICOM Browser app and gathers details from the DICOM directory file, named `DICOMDIR`. A DICOM directory file is a special DICOM file that serves as a directory to a collection of DICOM files stored on removable media, such as CD/DVD ROMs. `DICOMDIR` can contain a full path name or a relative path name to the file. The name of this file is `DICOMDIR`, with no file extension.

## See Also

### Apps

### Functions

`dicomanon` | `dicomdict` | `dicomdisp` | `dicominfo` | `dicomlookup` | `dicomuid` | `dicomwrite`

**Introduced in R2017b**

## 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

#### 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.

### 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

#### 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 (also called snakes) 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

### Functions

`activecontour` | `grabcut` | `grayconnected` | `imbinarize` | `imfindcircles` | `lazysnapping`

### 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

### 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 Image Viewer App”

## Programmatic Use

```
imtool
```

## See Also

**Apps**  
**Video Viewer**

**Functions**  
`imshow`

## **Topics**

“Explore Images with 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

### 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

**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 Volume Viewer App”

## Programmatic Use

`volumeViewer` opens a volume visualization app.

`volumeViewer(V)` loads the volume *V* into the app. *V* is a scalar-valued *m*-by-*n*-by-*p* image of class `logical`, `uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`, `single`, or `double`.

`volumeViewer(V,Ref)` loads the volume *V* with spatial referencing information *Ref* into the app. *Ref* is a 3-D scale transformation of size [4 4] of class `double`.

`volumeViewer close` closes all open Volume Viewer apps.

## See Also

### Functions

`isosurface` | `slice`

### Topics

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

**Introduced in R2017a**

## activecontour

Segment image into foreground and background using active contours (snakes)

### 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 contours. Using the active contour algorithm, also called snakes, you specify curves on the image that move to find object boundaries. The `activecontour` function evolves the segmentation using an iterative process and, by default, `activecontour` performs 100 iterations.

The `mask` argument 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.

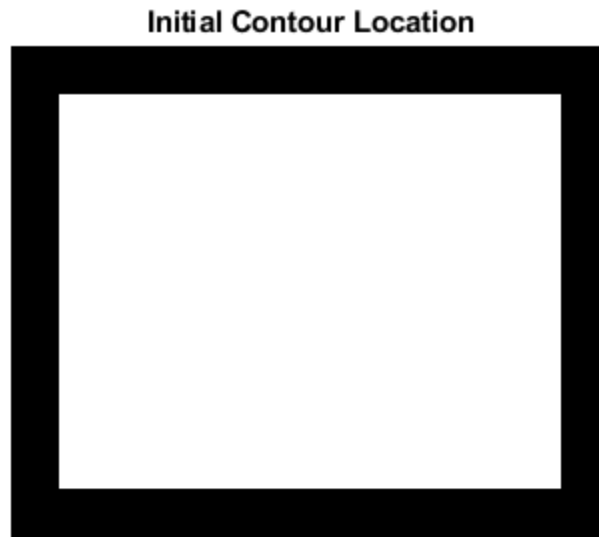
```
I = imread('coins.png');  
imshow(I)  
title('Original Image')
```

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')
```

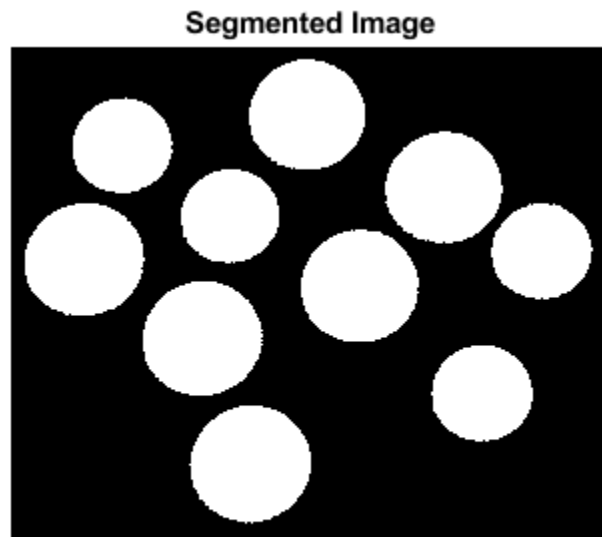


Segment the image using the default method and 300 iterations.

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

Display the result.

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



### 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



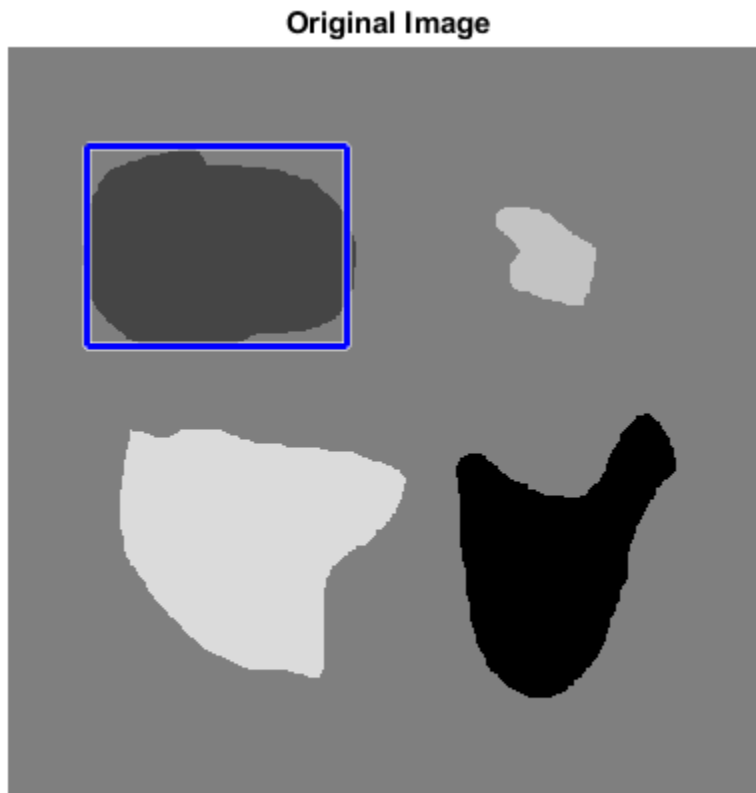
Specify initial contour location close to the object that is to be segmented.

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

Display the initial contour on the original image in blue.

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





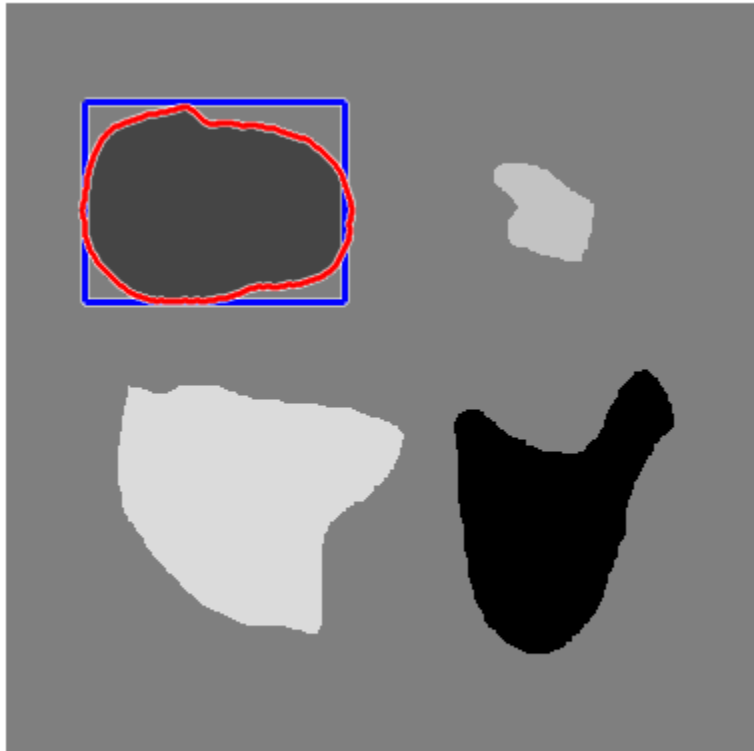
Segment the image using the 'edge' method and 200 iterations.

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

Display the final contour on the original image in red.

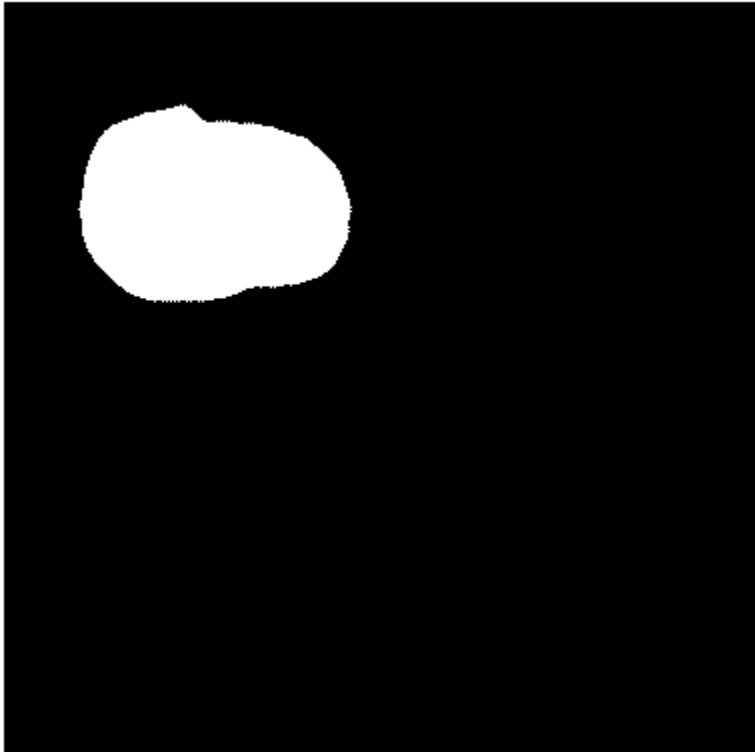
```
visboundaries(bw, 'Color', 'r');  
title('Initial contour (blue) and final contour (red)');
```

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.'))
```

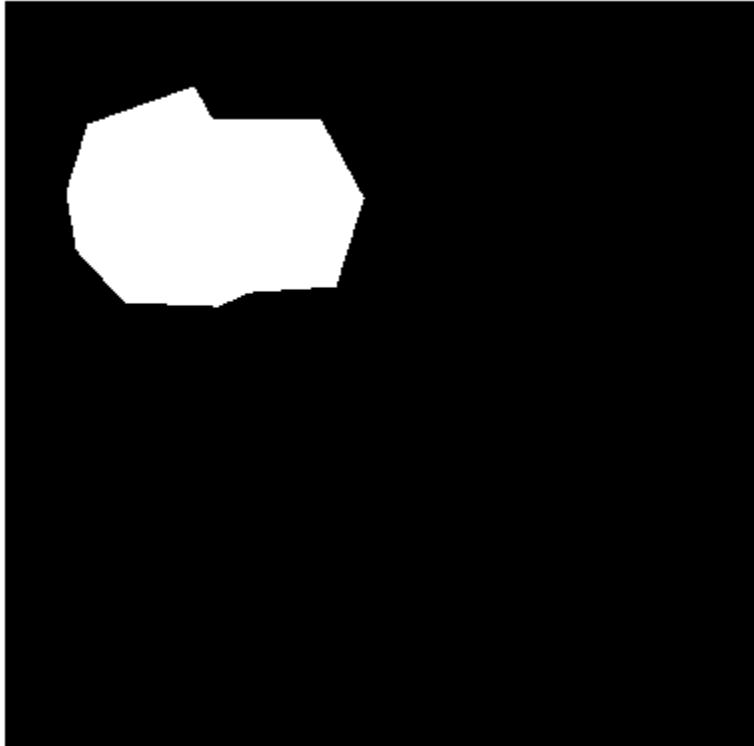
Click to select initial contour location. Double-click to confirm and proceed.



Specify initial contour interactively.

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

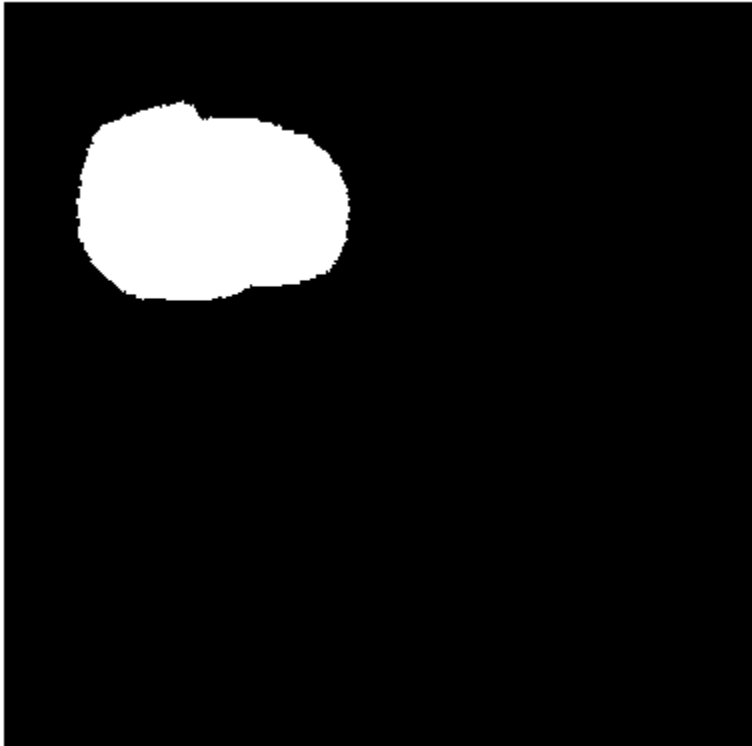
Initial MASK



Segment the image, specifying 200 iterations.

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

Segmented Image



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

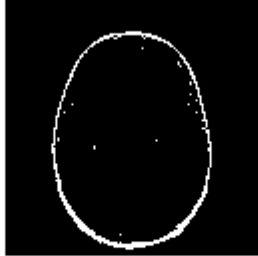
Load 3-D volumetric image data, removing the singleton dimension.

```
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.

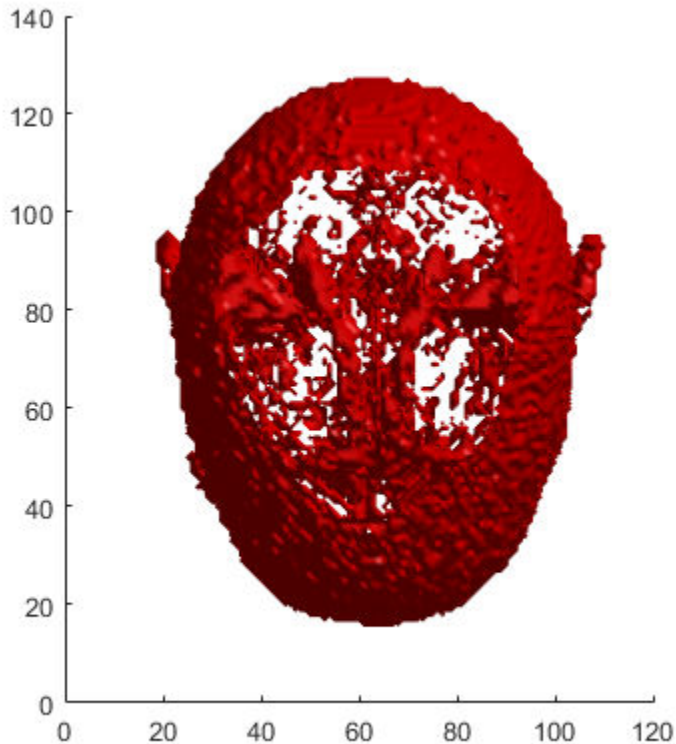
```
mask = zeros(size(A));  
mask(:,:,seedLevel) = seed;
```

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 'Chan-Vese' or 'edge'. The Chan and Vese region-based energy model is described in [1] on page 1-37. The edge-based model, similar to Geodesic Active Contour, is described in [2] on page 1-37.

Data Types: `char` | `string`

## **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 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**

**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].

`J = adapthisteq(I,Name,Value)` specifies additional name-value pairs. Parameter names can be abbreviated, and case does not matter.

## Examples

### Apply Contrast-Limited Adaptive Histogram Equalization (CLAHE)

Apply CLAHE to an image and display the results.

```
I = imread('tire.tif');
J = adapthisteq(I,'clipLimit',0.02,'Distribution','rayleigh');
imshowpair(I,J,'montage');
title('Original Image (left) and Contrast Enhanced Image (right)')
```

**Original Image (left) and Contrast Enhanced Image (right)**



## Apply CLAHE to Indexed Color Image

Read the indexed color image into the workspace.

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

Convert the indexed image into a truecolor (RGB) image, then convert the RGB image into the L\*a\*b\* color space.

```
RGB = ind2rgb(X,MAP);  
LAB = rgb2lab(RGB);
```

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

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

Perform CLAHE on the L channel. Scale the result to get back to the range used by the L\*a\*b\* color space.

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

Convert the resulting image back into the RGB color space.

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

Display the original image and the processed image.

```
figure  
imshowpair(RGB,J,'montage')  
title('Original (left) and Contrast Enhanced (right) Image')
```

**Original (left) and Contrast Enhanced (right) Image**



Shadows in the enhanced image look darker and highlights look brighter. The overall contrast is improved.

## 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 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) | 2-element vector of positive integers

Number of rectangular contextual regions (tiles) into which `adaptHisteq` divides the image, specified as a 2-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`. The optimal number of tiles depends on the type of the input image, and it is best determined through experimentation.

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.

`'ClipLimit'` is a contrast factor that prevents oversaturation 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.

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

Data Types: char | string

### 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.

'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.

Data Types: char | string

### 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.

## Algorithms

CLAHE operates on small regions in the image, called *tiles*, rather than the entire image. `adapthisteq` calculates the contrast transform function for each tile individually. Each tile's contrast is enhanced, so that the histogram of the output region approximately matches the histogram specified by the 'Distribution' value. 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.

## References

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

## 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)
T = adapthresh(V, ____,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.

`T = adapthresh(V, ____,Name,Value)` computes a locally adaptive threshold for the 3-D input volume `V`.

## Examples

### Find Threshold and Segment Bright Rice Grains from Dark Background

Read image into the workspace.

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

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

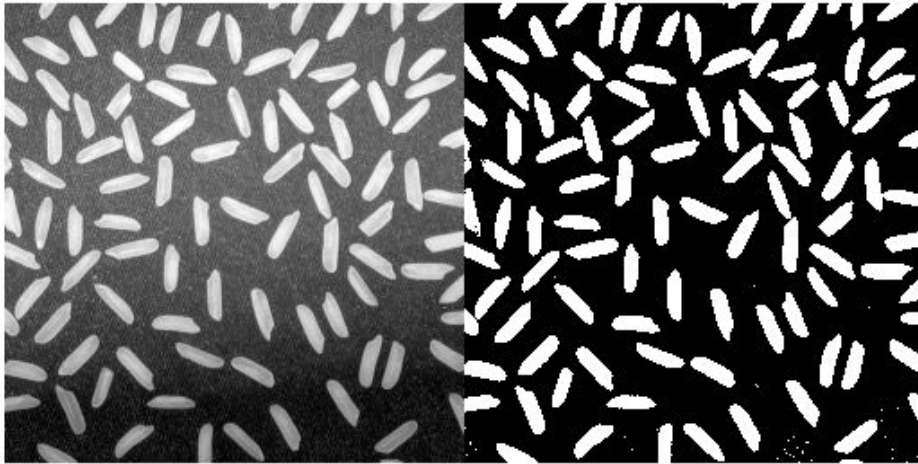
```
T = adaptthresh(I, 0.4);
```

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

```
T = adapthresh(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 22 1 1]
```

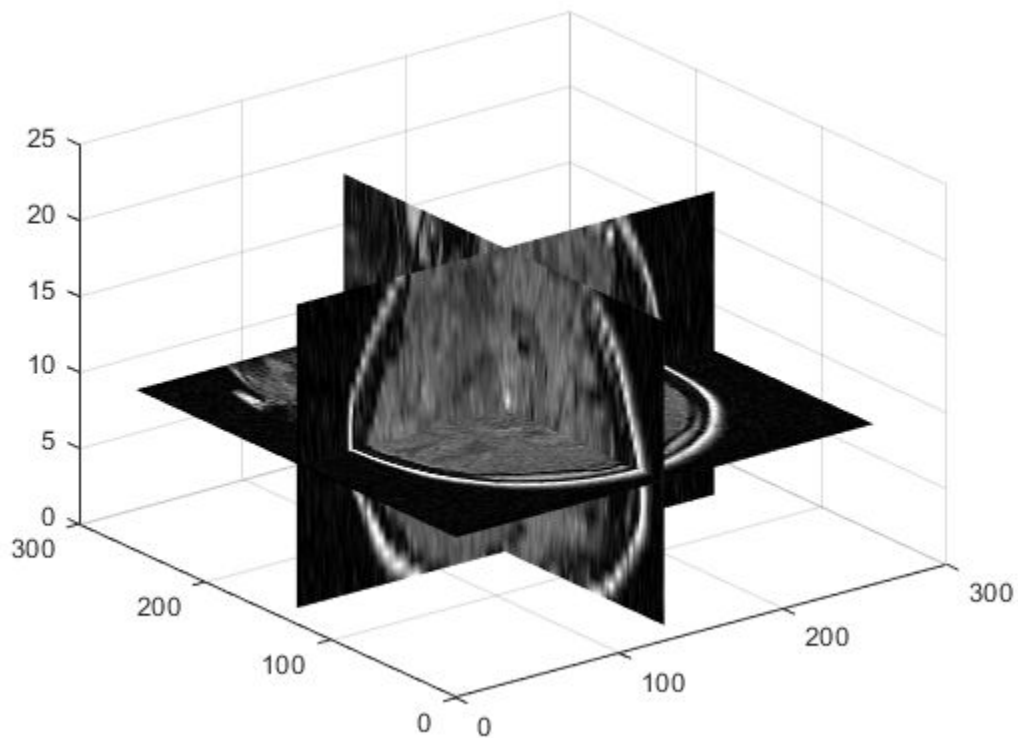
## Calculate Threshold for 3-D Volume

Load 3-D volume into the workspace.

```
load mrystack;
V = mrystack;
```

Display the data.

```
figure
slice(double(V), size(V,2)/2, size(V,1)/2, size(V,3)/2)
colormap gray
shading interp
```

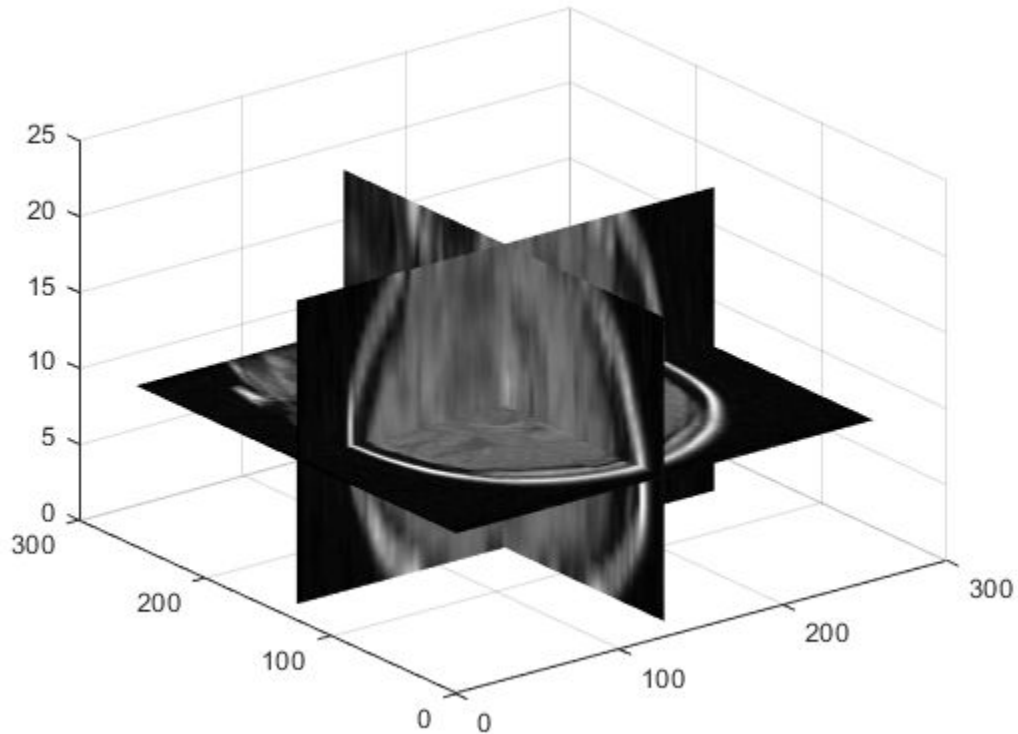


Calculate the threshold.

```
J = adapthresh(V, 'neigh', [3 3 3], 'Fore', 'bright');
```

Display the threshold.

```
figure  
slice(double(J), size(J,2)/2, size(J,1)/2, size(J,3)/2)  
colormap gray  
shading interp
```



## 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 might 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

**V — Input intensity volume**

real, nonsparse, 3-D array

Input intensity volume, specified as a real, nonsparse, 3-D array. If the image contains Infs or NaNs, the behavior of adapthresh is undefined. Propagation of Infs or NaNs might not be localized to the neighborhood around Inf or NaN pixels.

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 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.
'dark'	The foreground is darker than the background

Data Types: `char` | `string`

### **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 technique is also called Bradley's method [1].
'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` | `string`

## **Output Arguments**

### **T — Normalized intensity values**

2-D matrix | 3-D array

Normalized intensity values, returned as a 2-D matrix or 3-D array of class `double`. The return value `T` is the same size as the input image or volume.

## **References**

[1] Bradley, D., G. Roth, "Adapting Thresholding Using the Integral Image," *Journal of Graphics Tools*. Vol. 12, No. 2, 2007, pp.13-21.

## 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 “Understand Code Generation with Image Processing Toolbox”.
- The `ForegroundPolarity` and `Statistic` arguments must be compile-time constants.

### See Also

`graythresh` | `imbinarize` | `otsuthresh`

**Introduced in R2016a**

## affine2d

2-D affine geometric transformation

### Description

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

### Creation

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

- `imregtform` — Estimates a geometric transformation that maps a moving image to a fixed image using similarity optimization
- `imregcorr` — Estimates a geometric transformation that maps a moving image to a fixed image using phase correlation
- `fitgeotrans` — Estimates a geometric transformation that maps pairs of control points between two images
- The `affine2d` function described here

### Syntax

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

### Description

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

`tform = affine2d(A)` sets the property `T` with a valid affine transformation defined by nonsingular matrix `A`.

## Properties

### T — Forward 2-D affine transformation

nonsingular 3-by-3 numeric matrix

Forward 2-D affine transformation, specified as a nonsingular 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 — Dimensionality of the geometric transformation

2

Dimensionality of the geometric transformation for both input and output points, specified as the value 2.

## Object Functions

<code>invert</code>	Invert geometric transformation
<code>isRigid</code>	Determine if transformation is rigid transformation
<code>isSimilarity</code>	Determine if transformation is similarity transformation
<code>isTranslation</code>	Determine if transformation is pure translation
<code>outputLimits</code>	Find output spatial limits given input spatial limits
<code>transformPointsForward</code>	Apply forward geometric transformation
<code>transformPointsInverse</code>	Apply inverse geometric transformation

## Examples

## Define 2-D Affine Transformation Object for Rotation

Create an `affine2d` object that defines a 30 degree rotation in the counterclockwise direction around the origin.

```
theta = 30;
tform = affine2d([cosd(theta) sind(theta) 0;...
    -sind(theta) cosd(theta) 0; 0 0 1])

tform =
    affine2d with properties:

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

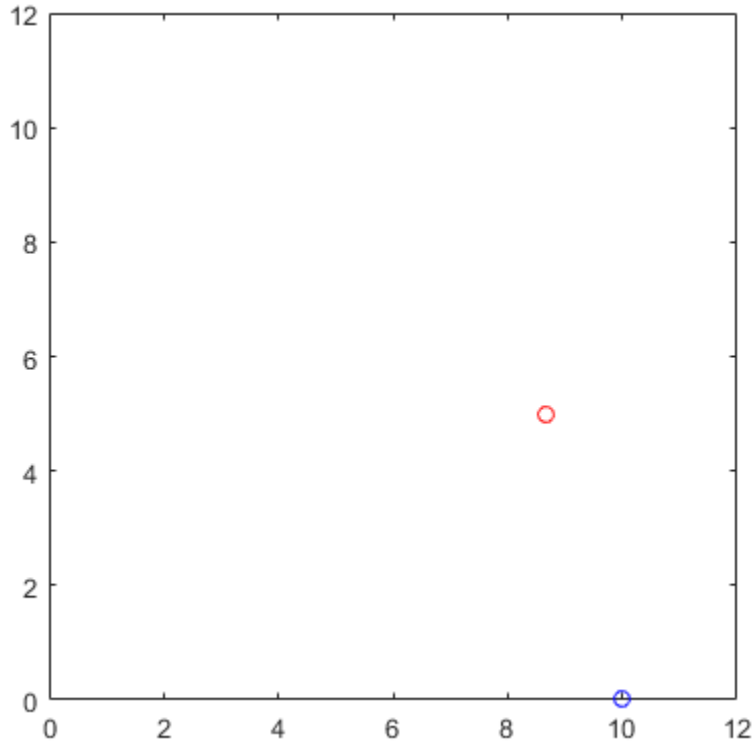
Apply the forward geometric transformation to a point (10,0).

```
[x,y] = transformPointsForward(tform,10,0)

x = 8.6603
y = 5.0000
```

Validate the transformation by plotting the original point (in blue) and the transformed point (in red).

```
figure
plot(10,0, 'bo', x,y, 'ro')
axis([0 12 0 12]); axis square;
```



### Transform Image Using 2-D Affine Transformation Object

Read an image into the workspace.

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

Create an `affine2d` object that defines an affine geometric transformation. This example combines vertical shear and horizontal stretch.

```
tform = affine2d([2 0.33 0; 0 1 0; 0 0 1])
```

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

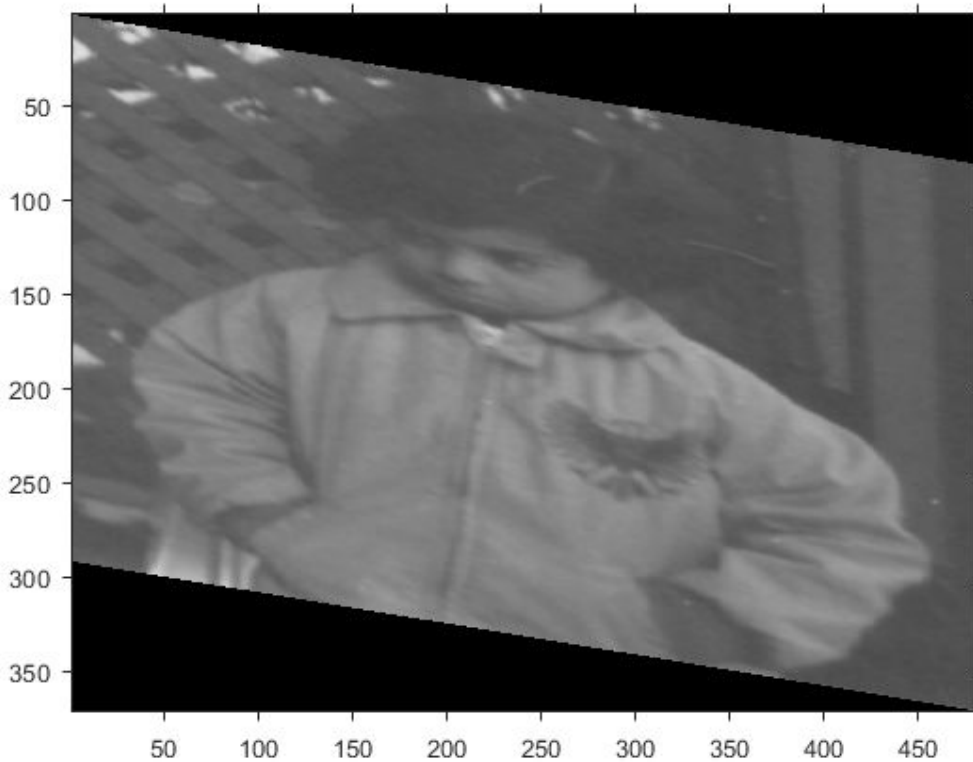
Apply the geometric transformation to the image using `imwarp`.

```
B = imwarp(A,tform);
```

Display the resulting image.

```
figure  
imshow(B);  
axis on equal;
```





## 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

### Functions

`fitgeotrans` | `imregcorr` | `imregister` | `imregtform` | `imwarp`

### Objects

`LocalWeightedMeanTransformation2D` | `PiecewiseLinearTransformation2D` | `PolynomialTransformation2D` | `affine3d` | `geometricTransform2d` | `projective2d`

## Topics

“2-D and 3-D Geometric Transformation Process Overview”  
“Matrix Representation of Geometric Transformations”

### Introduced in R2013a

# affine3d

3-D affine geometric transformation

## Description

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

## Creation

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

- `imregtform` — Estimates a geometric transformation that maps a moving image to a fixed image using similarity optimization
- The `affine3d` function described here

## Syntax

```
tform = affine3d  
tform = affine3d(A)
```

## Description

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

`tform = affine3d(A)` sets the property `T` with a valid affine transformation defined by nonsingular matrix `A`.

## Properties

### **T** — Forward 3-D affine transformation

nonsingular 4-by-4 numeric matrix

Forward 3-D affine transformation, specified as a nonsingular 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, specified as the value 3.

## Object Functions

<code>invert</code>	Invert geometric transformation
<code>isRigid</code>	Determine if transformation is rigid transformation
<code>isSimilarity</code>	Determine if transformation is similarity transformation
<code>isTranslation</code>	Determine if transformation is pure translation
<code>outputLimits</code>	Find output spatial limits given input spatial limits
<code>transformPointsForward</code>	Apply forward geometric transformation
<code>transformPointsInverse</code>	Apply inverse geometric transformation

## Examples

### Define 3-D Affine Transformation Object for Anisotropic Scaling

Create an `affine3d` object that scales a 3-D image by a different 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
```

Load a 3-D volume into the workspace.

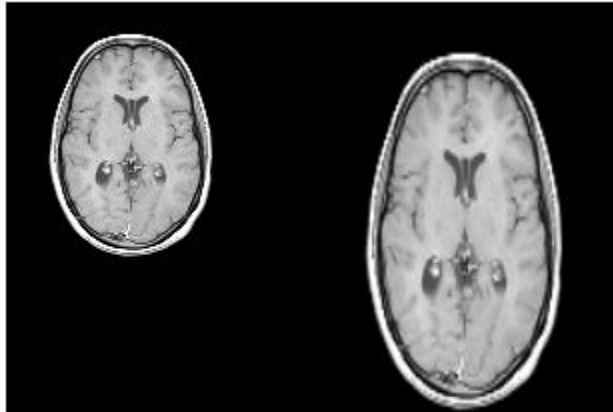
```
load('mri');
D = squeeze(D);
```

Apply the geometric transformation to the image using `imwarp`.

```
B = imwarp(D,tform);
```

Visualize an axial slice through the center of each volume to see the effect of scale translation. Note that the center slice of the transformed volume has a different index than the center slice of the original volume because of the scaling in the z-dimension.

```
figure
imshowpair(D(:,:,14),B(:,:,33),'montage');
```



The original image is on the left, and the transformed image is on the right. The transformed image is scaled more in the vertical direction than in the horizontal direction, as expected since  $S_y$  is larger than  $S_x$ .

## See Also

### Functions

`imregister` | `imregtform` | `imwarp`

### Objects

`affine2d` | `geometricTransform3d`

### Topics

“2-D and 3-D Geometric Transformation Process Overview”

“Matrix Representation of Geometric Transformations”

**Introduced in R2013a**

# 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 `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 | string

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

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

Data Types: `char` | `string`

## 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 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 | string

Endianness of the data, specified as one of the strings or 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` | `string`



## Output Arguments

**info** — Information about Analyze 7.5 data set  
structure

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

## 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 1-68.

---

`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.

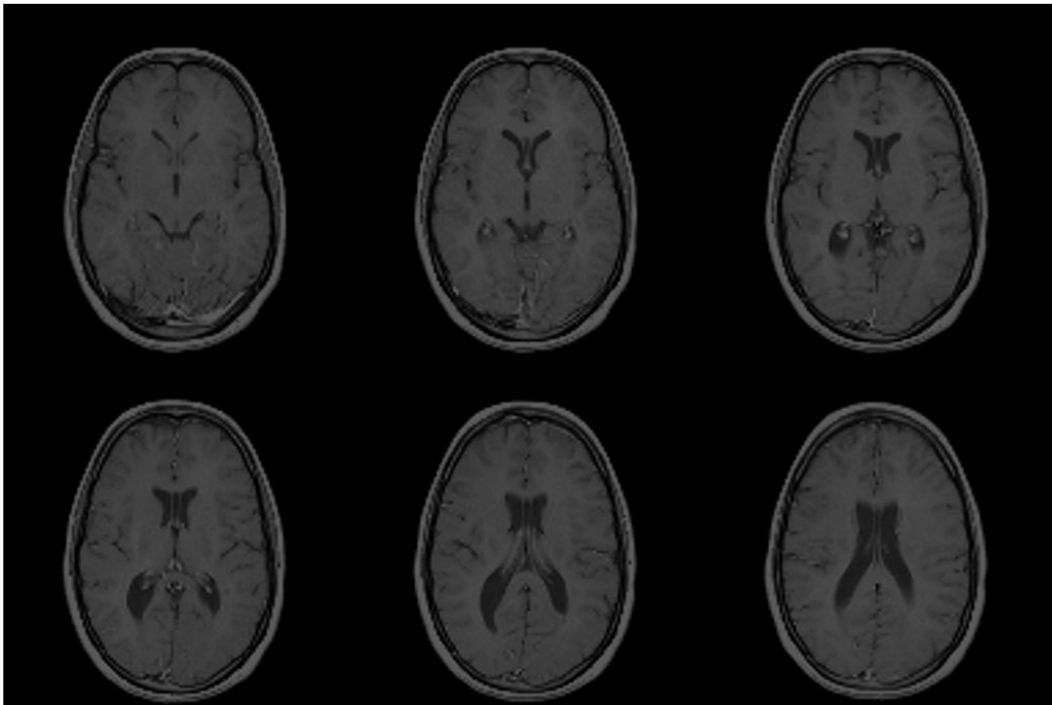
```
X = analyze75read('brainMRI');
```

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**

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`.

### 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);
```

### Input Arguments

#### A — Input color space

2-D numeric matrix | 3-D numeric array | string | character vector

Input color space, specified as one of the following:

- 2-D numeric matrix. `applycform` interprets each row as a color unless the color transformation structure, `C`, contains a grayscale ICC profile. In that case, `applycform` interprets each pixel in `A` as a color.
- 3-D numeric matrix. Each row-column location is interpreted as a color. `size(A,3)` is typically 1 or more, depending on the input color space.
- string or character vector. `A` is only a string or character vector if `C` is created with the following syntax:

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

Data Types: `double` | `uint8` | `uint16` | `char` | `string`

### **C — Color transformation**

structure

Color transformation, specified as a structure. The color transformation structure specifies various parameters of the transformation. You can create a color transformation structure using `makecform`.

## **Output Arguments**

### **B — Output color space**

numeric array

Output color space, returned as a numeric array. The size of `B` depends on the dimensionality and size of the input color space, `A`:

- When `A` is two-dimensional, `B` has the same number of rows and one or more columns, depending on the output color space. (The ICC specification currently supports up to 15-channel device spaces).
- When `A` is three-dimensional, `B` is the same number of rows and columns as `A`, and `size(B,3)` is 1 or more, depending on the output color space.

## **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

---

**Note** `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.

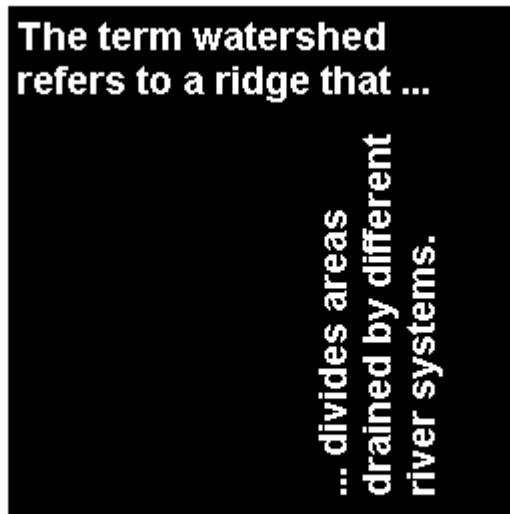
```
lutfun = @(x)(sum(x(:))==4);  
lut     = makelut(lutfun,2);
```

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 .

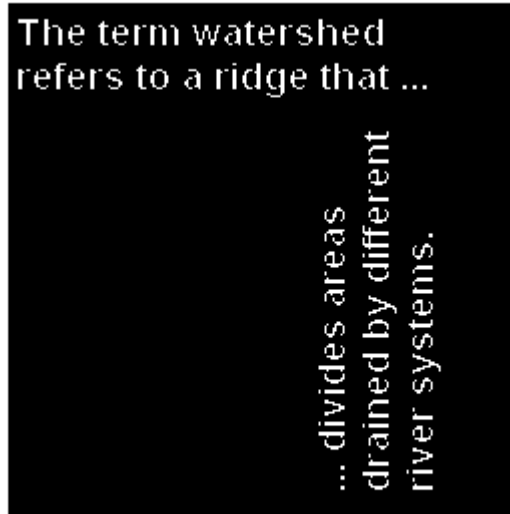
```
BW1 = imread('text.png');  
BW2 = applylut(BW1,lut);
```

Show the original image and the eroded image.

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



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



## 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

```
pixelCoord = axes2pix(n,extent,axesCoord)
```

### Description

`pixelCoord = axes2pix(n,extent,axesCoord)` converts an axes coordinate into an intrinsic ("pixel") coordinate.

---

**Note** The `imref2d` object has several methods that facilitate conversion between intrinsic coordinates, world coordinates and array indices.

---

### 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 = 1×2
```

```
    1    240
```

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

```
ydata = 1×2
```

---

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**

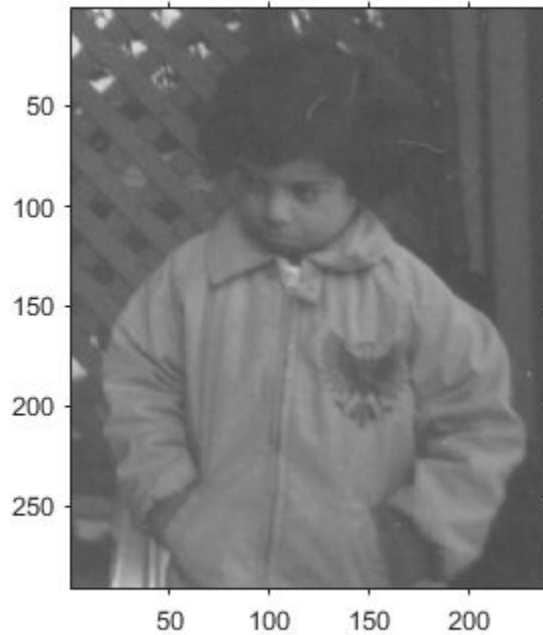
Read an image and display it. Get the size of the image.

```
I = imread('pout.tif');
[nrows,ncols] = size(I)

nrows = 291
ncols = 240
```

Create a spatial referencing object for this image, with default property settings. By default, the upper-left corner of the image has intrinsic coordinate (1,1).

```
RI = imref2d(size(I));
h = imshow(I,RI);
```



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

```
xData = 1×2
```

```
1    240
```

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

```
yData = 1×2
```

```
1    291
```

For illustrative purposes, specify an arbitrary image extent in the  $x$ - and  $y$ -directions. This example shifts the image up by 20 pixels and to the right by 400 pixels. The example also



shifts the image to the right by 100 pixels and compresses the image horizontally by a factor of 2.

```
xWorldLimits = 0.5*xData + 400;  
yWorldLimits = yData - 20;  
RA = imref2d(size(I),xWorldLimits,yWorldLimits);  
imshow(I,RA)
```



Select a pixel, such as a pixel near the nose of the child. This pixel occurs around the axes coordinate  $(x, y) = (450, 90)$  in the modified image.

Convert the axes coordinate to an intrinsic coordinate.

```
px = axes2pix(ncols,xWorldLimits,450)
```

```
px = 100
```

```
py = axes2pix(nrows,yWorldLimits,90)
```

```
py = 110
```

The intrinsic coordinate of the point is at (100, 110). This agrees with the location of the nose in the original image.

## Input Arguments

### **n** — Number of image rows or columns

positive integer

Number of image rows or columns, specified as a positive integer. `n` is the number of image columns for the `x`-coordinate, or the number of image rows for the `y`-coordinate.

### **extent** — Image world extent

2-element numeric vector

Image world extent, specified as a 2-element numeric vector. `extent` is returned by `get(image_handle, 'XData')` or `get(image_handle, 'YData')`.

### **axesCoord** — Axes coordinates to convert

numeric vector

Axes coordinate to convert to intrinsic coordinates, specified as a numeric vector.

## Output Arguments

### **pixelCoord** — Intrinsic coordinates

numeric vector

Intrinsic coordinates, returned as a numeric vector.

Data Types: `double`

## Tips

- `axes2pix` performs minimal checking on the validity of the `n`, `axesCoord`, or `extent` arguments. For example, `axes2pix` can extrapolate from `extent` to return a negative coordinate. The function calling `axes2pix` bears responsibility for error checking.

## See Also

`bwselect` | `imixel` | `imixelinfo` | `improfile` | `imref2d` | `roipoly`

## Topics

“Image Coordinate Systems”

**Introduced before R2006a**

## bestblk

Determine optimal block size for block processing

### Syntax

```
siz = bestblk([M N],k)
[m,n] = bestblk([M N],k)
```

### Description

`siz = bestblk([M N],k)` returns the optimal block size for block processing of an M-by-N image. The optimal block size minimizes the padding required along the outer partial blocks. `k` specifies the maximum row and column dimensions for the block.

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

### Examples

#### Determine Optimal Block Size

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

```
siz = 1×2
```

```
    64    50
```

### Input Arguments

#### [M N] — Image size

2-element vector of positive integers

Image size, specified as a 2-element vector of positive integers.  $M$  is the number of rows and  $N$  is the number of columns in the image.

Data Types: double

**k — Maximum number of block rows or columns**

100 (default) | positive integer

Maximum number of block rows or columns, specified as a positive integer.

Data Types: double

## Output Arguments

**siz — Optimal block size**

2-element numeric row vector

Optimal block size, returned as a 2-element numeric row vector. `siz` is equivalent to `[m n]`.

**m, n — Optimal number of block rows or columns**

numeric scalar

Optimal number of block rows or columns, returned as a numeric scalar.

## Algorithms

The algorithm for determining the optimal value of  $m$  from  $M$  and  $k$  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 used to find the optimal value of  $n$  from  $N$  and  $k$ .

## See Also

`blockproc`

**Introduced before R2006a**

# bfscore

Contour matching score for image segmentation

## Syntax

```
score = bfscore(prediction,groundTruth)
[score,precision,recall] = bfscore(prediction,groundTruth)
[ ___ ] = bfscore(prediction,groundTruth,threshold)
```

## Description

`score = bfscore(prediction,groundTruth)` computes the BF (Boundary F1) contour matching score between the predicted segmentation in `prediction` and the true segmentation in `groundTruth`. `prediction` and `groundTruth` can be a pair of logical arrays for binary segmentation, or a pair of label or categorical arrays for multiclass segmentation.

`[score,precision,recall] = bfscore(prediction,groundTruth)` also returns the precision and recall values for the `prediction` image compared to the `groundTruth` image.

`[ ___ ] = bfscore(prediction,groundTruth,threshold)` computes the BF score using a specified threshold as the distance error tolerance, to decide whether a boundary point has a match or not.

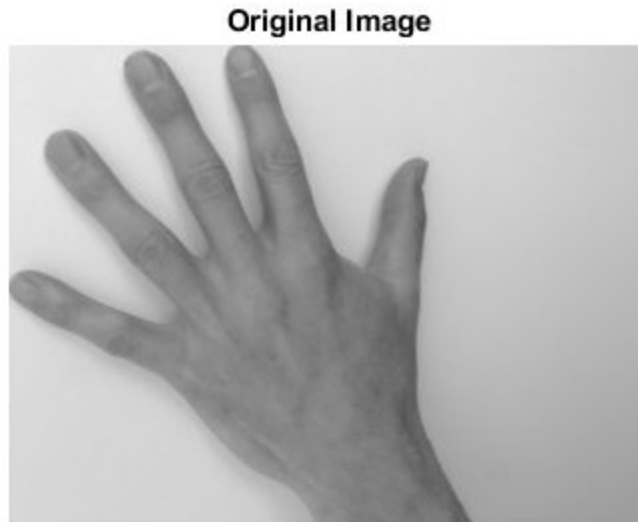
## Examples

### Compute BF Score for Binary Segmentation

Read an image with an object to segment. Convert the image to grayscale, and display the result.

```
A = imread('hands1.jpg');
I = rgb2gray(A);
```

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



Use the active contours (snakes) method to segment the hand.

```
mask = false(size(I));
mask(25:end-25,25:end-25)= true;
BW = activecontour(I, mask, 300);
```

Read the ground truth segmentation.

```
BW_groundTruth = imread('hands1-mask.png');
```

Compute the BF score of the active contours segmentation against the ground truth.

```
similarity = bfscore(BW, BW_groundTruth);
```

Display the masks on top of each other. Colors indicate differences in the masks.



```
figure
imshowpair(BW, BW_groundTruth)
title(['BF Score = ' num2str(similarity)])
```



### Compute BF Score for Multi-Region Segmentation

This example shows how to segment an image into multiple regions. The example then computes the BF score for each region.

Read an image with several regions to segment.

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

Create scribbles for three regions that distinguish their typical color characteristics. The first region classifies the yellow flower. The second region classifies the green stem and leaves. The last region classifies the brown dirt in two separate patches of the image. Regions are specified by a 4-element vector, whose elements indicate the x- and y-

coordinate of the upper left corner of the ROI, the width of the ROI, and the height of the ROI.

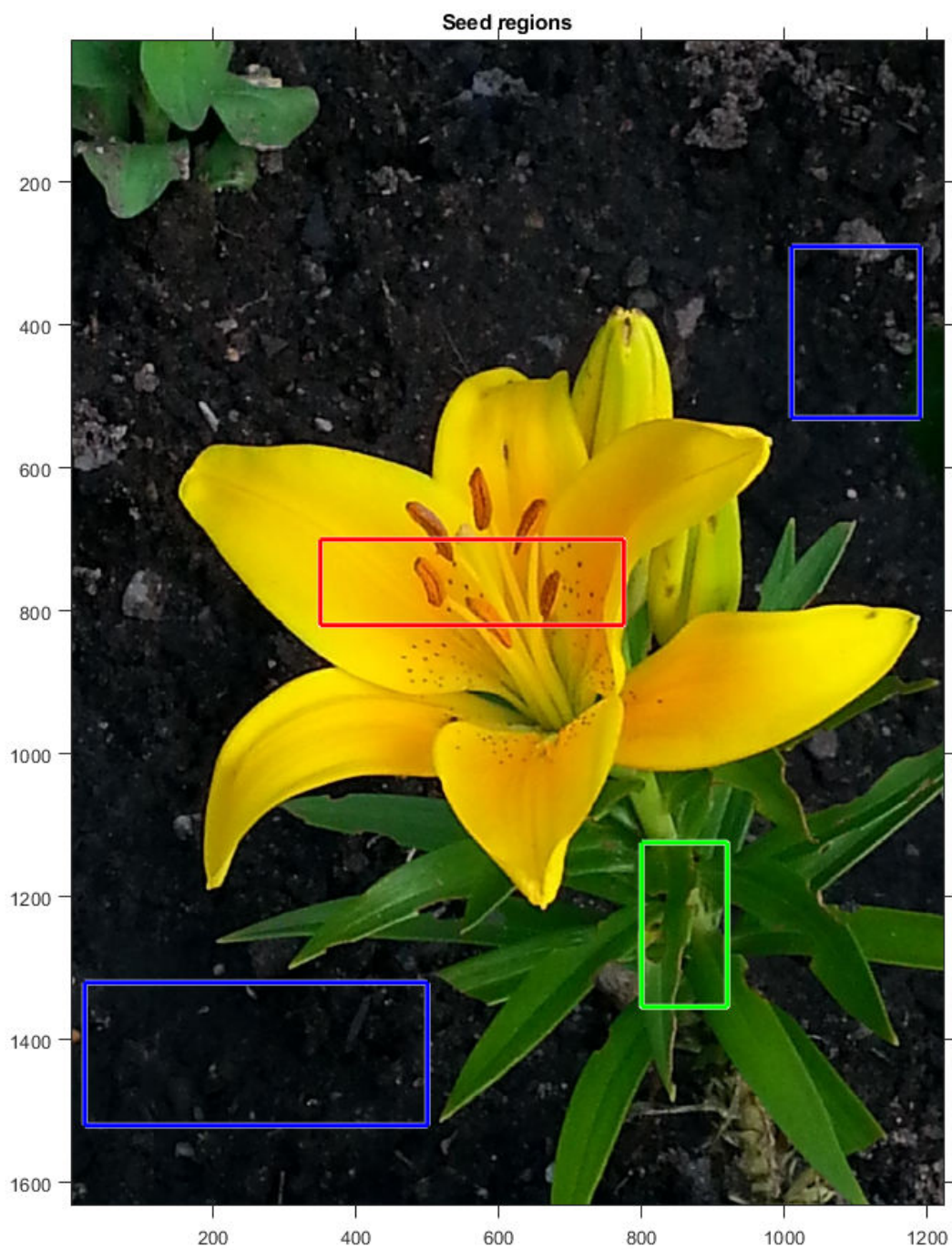
```
region1 = [350 700 425 120]; % [x y w h] format
BW1 = false(size(RGB,1),size(ROI,2));
BW1(region1(2):region1(2)+region1(4),region1(1):region1(1)+region1(3)) = true;

region2 = [800 1124 120 230];
BW2 = false(size(ROI,1),size(ROI,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(ROI,1),size(ROI,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 on top of the image.

```
figure
imshow(ROI)
hold on
visboundaries(BW1,'Color','r');
visboundaries(BW2,'Color','g');
visboundaries(BW3,'Color','b');
title('Seed regions')
```



Segment the image into three regions using geodesic distance-based color segmentation.

```
L = imseggeodesic(RGB,BW1,BW2,BW3,'AdaptiveChannelWeighting',true);
```

Load a ground truth segmentation of the image.

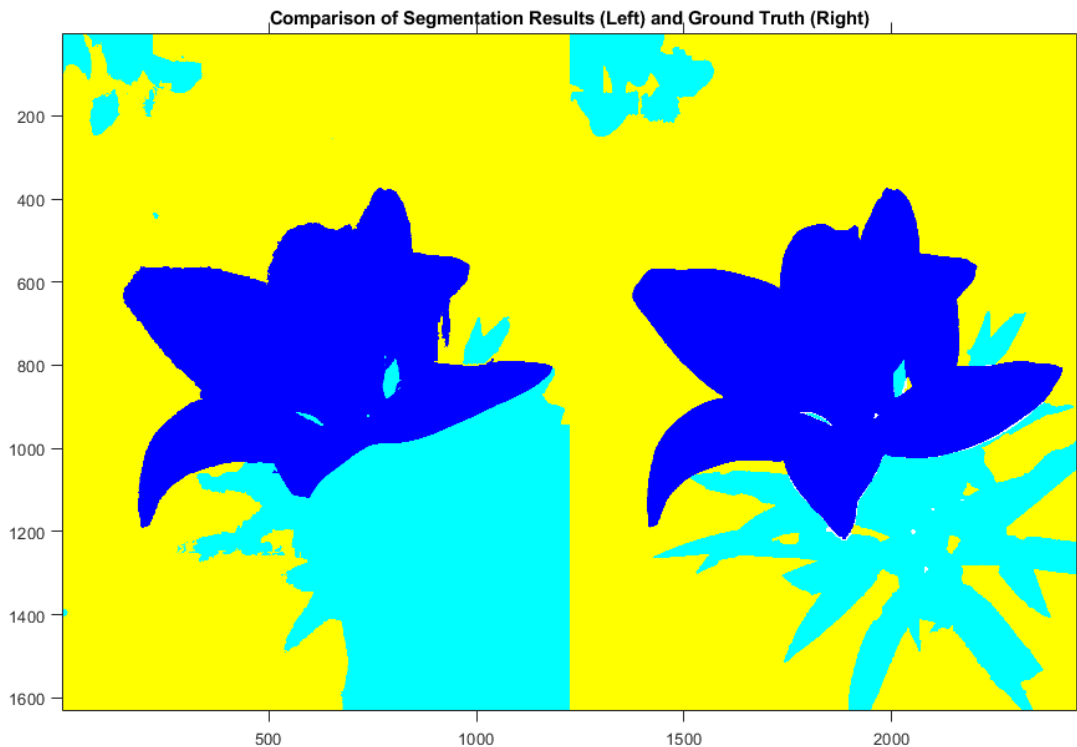
```
L_groundTruth = double(imread('yellowlily-segmented.png'));
```

Visually compare the segmentation results with the ground truth.

```
figure
```

```
imshowpair(label2rgb(L),label2rgb(L_groundTruth),'montage')
```

```
title('Comparison of Segmentation Results (Left) and Ground Truth (Right)')
```



Compute the BF score for each segmented region.

```
similarity = bfscore(L, L_groundTruth)
```

```
similarity = 3x1
    0.7992
    0.5333
    0.7466
```

The BF score is noticeably smaller for the second region. This result is consistent with the visual comparison of the segmentation results, which erroneously classifies the dirt in the lower right corner of the image as leaves.

## Input Arguments

### **prediction** — Predicted segmentation

2-D or 3-D logical, numeric, or categorical array

Predicted segmentation, specified as a 2-D or 3-D logical, numeric, or categorical array. If **prediction** is a numeric array, then it represents a label array and must contain nonnegative integers of data type `double`.

Data Types: `logical` | `double` | `categorical`

### **groundTruth** — Ground truth segmentation

2-D or 3-D logical, numeric, or categorical array

Ground truth segmentation, specified as a 2-D or 3-D logical, numeric, or categorical array of the same size and data type as **prediction**. If **groundTruth** is a numeric array, then it represents a label array and must contain nonnegative integers of data type `double`.

Data Types: `logical` | `double` | `categorical`

### **threshold** — Distance error tolerance threshold

positive scalar

Distance error tolerance threshold in pixels, specified as a positive scalar. The threshold determines whether a boundary point has a match or not. If **threshold** is not specified, then the default value is 0.75% of the length of the image diagonal.

Example: 3

Data Types: `double`

## Output Arguments

### **score — BF score**

numeric scalar | numeric vector

BF score, returned as a numeric scalar or vector with values in the range [0, 1]. A score of 1 means that the contours of objects in the corresponding class in `prediction` and `groundTruth` are a perfect match. If the input arrays are:

- logical arrays, `score` is a scalar and represents the BF score of the foreground.
- label or categorical arrays, `score` is a vector. The first coefficient in `score` is the BF score for the first foreground class, the second coefficient is the score for the second foreground class, and so on.

### **precision — Precision**

numeric scalar | numeric vector

Precision, returned as a numeric scalar or numeric vector with values in the range [0, 1]. Each element indicates the precision of object contours in the corresponding foreground class.

*Precision* is the ratio of the number of points on the boundary of the predicted segmentation that are close enough to the boundary of the ground truth segmentation to the length of the predicted boundary. In other words, precision is the fraction of detections that are true positives rather than false positives.

### **recall — Recall**

numeric scalar | numeric vector

Recall, returned as a numeric scalar or numeric vector with values in the range [0, 1]. Each element indicates the recall of object contours in the corresponding foreground class.

*Recall* is the ratio of the number of points on the boundary of the ground truth segmentation that are close enough to the boundary of the predicted segmentation to the length of the ground truth boundary. In other words, recall is the fraction of true positives that are detected rather than missed.

## Definitions

### BF (Boundary F1) Score

The BF score measures how close the predicted boundary of an object matches the ground truth boundary.

The BF score is defined as the harmonic mean (F1-measure) of the `precision` and `recall` values with a distance error tolerance to decide whether a point on the predicted boundary has a match on the ground truth boundary or not.

$$\text{score} = 2 * \text{precision} * \text{recall} / (\text{recall} + \text{precision})$$

### References

- [1] Csurka, G., D. Larlus, and F. Perronnin. "What is a good evaluation measure for semantic segmentation?" *Proceedings of the British Machine Vision Conference*, 2013, pp. 32.1-32.11.

### See Also

`dice` | `jaccard`

**Introduced in R2017b**

## blendexposure

Create well-exposed image from images with different exposures

### Syntax

```
J = blendexposure(I1,I2,...,In)
J = blendexposure(I1,I2,...,In,Name,Value)
```

### Description

`J = blendexposure(I1,I2,...,In)` blends grayscale or RGB images that have different exposures. `blendexposure` blends the images based on their contrast, saturation, and well-exposedness, and returns the well-exposed image, `J`.

`J = blendexposure(I1,I2,...,In,Name,Value)` blends images that have different exposures, using name-value pairs to adjust how each input image contributes to the blended image.

### Examples

#### Blend Images with Strong Light Sources

Read a series of images with different exposures that were captured from a fixed camera with no moving objects in the scene.

```
I1 = imread('car_1.jpg');
I2 = imread('car_2.jpg');
I3 = imread('car_3.jpg');
I4 = imread('car_4.jpg');
```

Display the images. In the underexposed images, only bright regions like headlights have informative details. Conversely, the headlights are saturated in the overexposed images, and the best contrast comes from darker regions such as the brick floor and the roof.

```
montage({I1,I2,I3,I4})
```





Blend the images using exposure fusion. By default, the `blendexposure` function attempts to suppress highlights from strong light sources. For comparison, also blend the images without suppressing the highlights. Display the two results.

```
E = blendexposure(I1,I2,I3,I4);  
F = blendexposure(I1,I2,I3,I4,'ReduceStrongLight',false);  
montage({E,F})  
title('Exposure Fusion With (Left) and Without (Right) Strong Light Suppression')
```

Exposure Fusion With (Left) and Without (Right) Strong Light Suppression



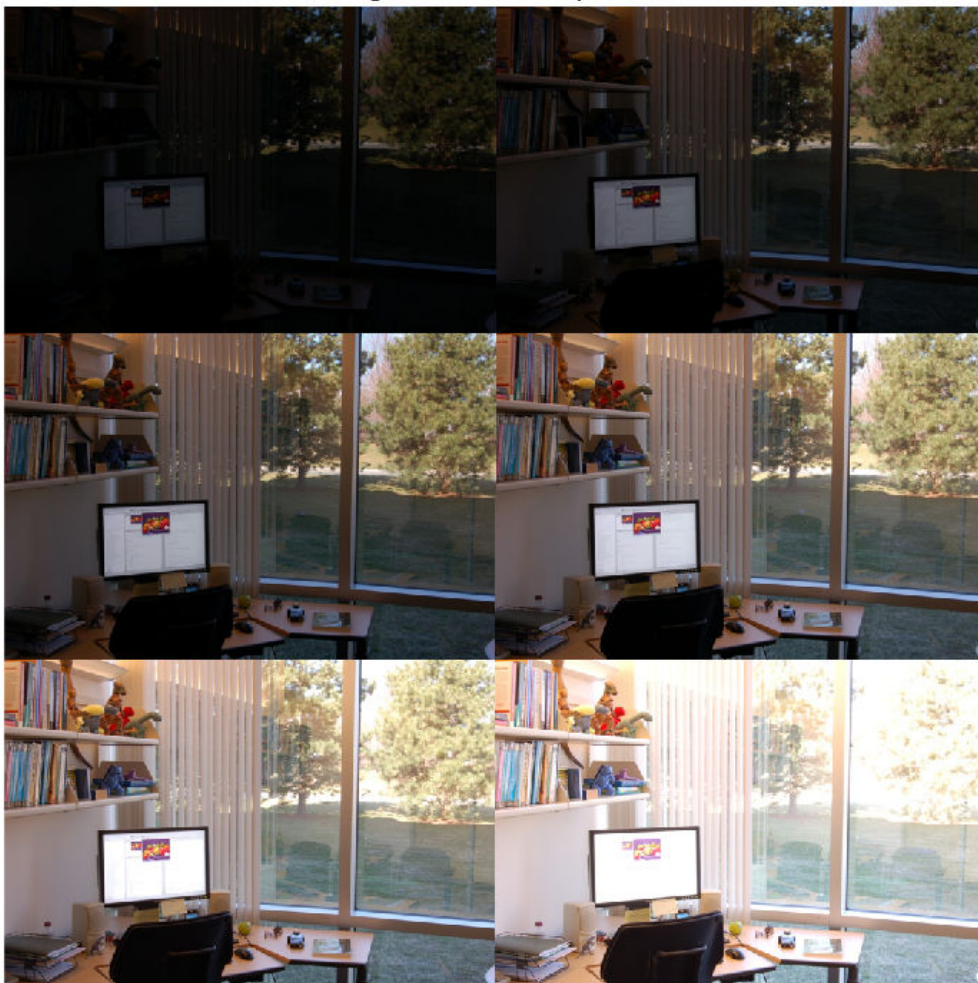
In the fused images, bright regions and dark regions retain informative details. With strong light suppression, the shape of the headlights is identifiable, and saturated pixels do not extend past the boundary of the headlights. Without strong light perception, the shape of the headlights is not identifiable, and there are saturated pixels in the reflection of the headlights on the ground and on some parts of the other cars.

## Blend Images of Stationary Scene Using Exposure Fusion

Read a series of images with different exposures. The images were captured from a fixed camera, and there are no moving objects in the scene.

```
I1 = imread('office_1.jpg');  
I2 = imread('office_2.jpg');  
I3 = imread('office_3.jpg');  
I4 = imread('office_4.jpg');  
I5 = imread('office_5.jpg');  
I6 = imread('office_6.jpg');  
montage({I1,I2,I3,I4,I5,I6})  
title('Images with Different Exposures')
```

Images with Different Exposures



Blend the registered images using exposure fusion, optionally varying the weight of contrast, saturation and well-exposedness in the fusion, and without reducing strong light sources. Display the result.

```
E = blendexposure(I1,I2,I3,I4,I5,I6,'contrast',0.8,...  
    'saturation',0.8,'wellexposedness',0.8,'reduceStrongLight',false);  
imshow(E)  
title('Blended Image Using Exposure Fusion')
```

Blended Image Using Exposure Fusion



## Input Arguments

### **I1, I2, . . . , In — Grayscale or RGB images**

*m*-by-*n* numeric matrices | *m*-by-*n*-by-3 numeric arrays

Grayscale or RGB images, specified as a series of *m*-by-*n* numeric matrices or *m*-by-*n*-by-3 numeric arrays. All images must have the same size and data type.

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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `blendexposure(I1,I2,I3,'Contrast',0.5,'Saturation',0.9)`

### **Contrast** — Relative weight given to contrast

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

Relative weight given to contrast during blending, specified as the comma-separated pair consisting of `'Contrast'` and a numeric scalar in the range [0, 1].

### **Saturation** — Relative weight given to saturation

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

Relative weight given to saturation during blending, specified as the comma-separated pair consisting of `'Saturation'` and a numeric scalar in the range [0, 1].

### **WellExposedness** — Relative weight given to exposure quality

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

Relative weight given to exposure quality during blending, specified as the comma-separated pair consisting of `'WellExposedness'` and a numeric scalar in the range [0, 1]. The exposure quality of each image is based on the divergence of the pixel intensities from a model of pixels with good exposure.

### **ReduceStrongLight** — Reduce strong light

true (default) | false

Reduce strong light, specified as the comma-separated pair consisting of `'ReduceStrongLight'` and true or false. If `'ReduceStrongLight'` is true, then `blendexposure` attempts to suppress highlights from strong light sources in the images.

---

**Note** If the input images do not have strong light sources and you specify `ReduceStrongLight` as true, then the output image `J` has less contrast.

---

## Output Arguments

### **J — Fused image**

numeric matrix or array

Fused image, returned as a numeric matrix or array of the same size and data type as the input images  $I_1, I_2, \dots, I_n$ .

## Tips

- To blend images of moving scenes or with camera jitter, first register the images by using the `imregmtb` function. `imregmtb` considers only translations, not rotations or other types of geometric transformations, when registering the images.

## Algorithms

The `blendexposure` function computes the weight of each quality measure as follows:

- Contrast weights are computed using Laplacian filtering.
- Saturation weights are computed from the standard deviation of each image.
- Well-exposedness is determined by comparing parts of the image to a Gaussian distribution with a mean of 0.5 and a standard deviation of 0.2.
- Strong light reduction weights are computed as a mixture of the other three weights, multiplied by a Gaussian distribution with a fixed mean and variance.

The weights are decomposed using Gaussian pyramids for seamless blending with a Laplacian pyramid of the corresponding image, which helps preserve scene details.

## References

- [1] Mertens, T., J. Kautz, and F. V. Reeth. "Exposure Fusion." *Pacific Graphics 2007: Proceedings of the Pacific Conference on Computer Graphics and Applications*. Maui, HI, 2007, pp. 382–390.

## **See Also**

imregmtb | makehdr | tonemap

**Introduced in R2018a**

## blockproc

Distinct block processing for image

### Syntax

```
B = blockproc(A,[m n],fun)
B = blockproc(src_filename,[m n],fun)
B = blockproc(adapter,[m n],fun)
blockproc( ____,Name,Value)
```

### Description

`B = blockproc(A,[m n],fun)` processes the image `A` by applying the function `fun` to each distinct block of size `[m n]` and concatenating the results into the output matrix, `B`.

`B = blockproc(src_filename,[m n],fun)` processes the image with file name `src_filename`, reading and processing one block at a time. This syntax is useful for processing large images.

`B = blockproc(adapter,[m n],fun)` processes the source image specified by `adapter`, an `ImageAdapter` object.

`blockproc( ____,Name,Value)` processes the input image, specifying arguments and corresponding values that control various aspects of the block behavior. Argument names are case insensitive.

### 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.

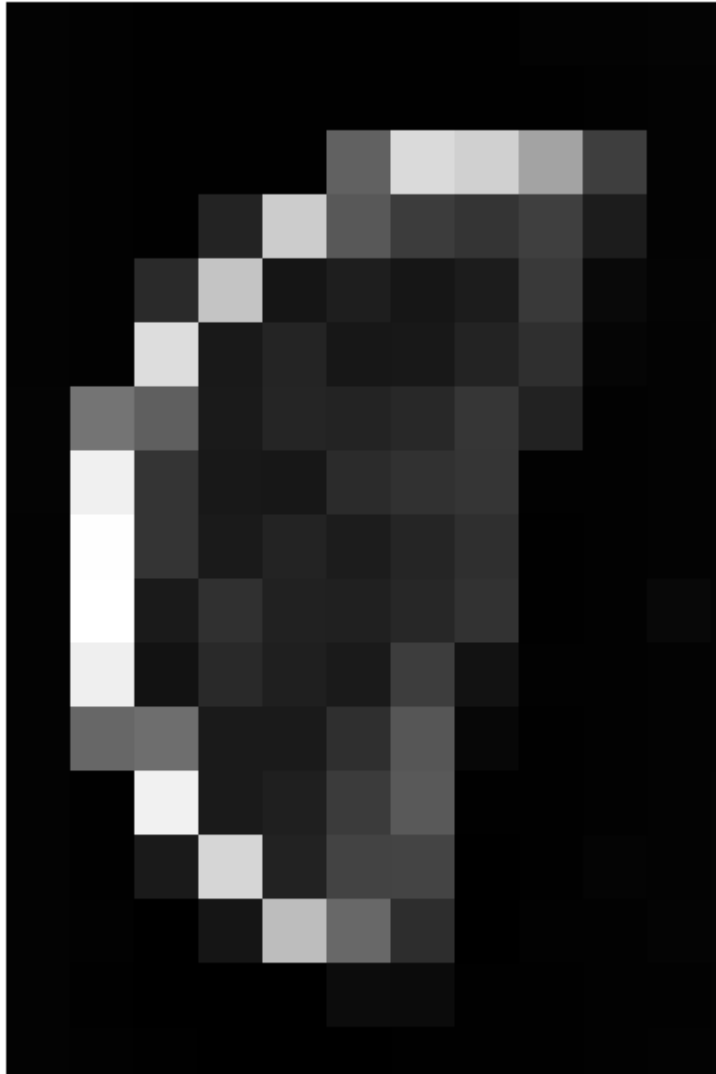
```
I2 = blockproc('moon.tif', [32 32], fun);
```

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');
```

## Input Arguments

### **A — Input image**

numeric matrix

Input image, specified as a numeric matrix.

### **src\_filename — Source file name**

character vector

Source file name, specified as a character vector. Files must have one of these file types and must be named with one of the listed file extensions.

- TIFF (\*.tif, \*.tiff)
- JPEG2000 (\*.jp2, \*.jpf, \*.jpx, \*.j2c, \*.j2k)

Data Types: char

### **adapter — Image adapter**

ImageAdapter object

Image adapter, specified as 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 “Perform Block Processing on Image Files in Unsupported Formats”.

### **[m n] — Block size**

2-element vector

Block size, specified as a 2-element vector. *m* is the number of rows and *n* is the number of columns in the block.

### **fun — Function handle**

handle

Function handle, specified as a handle. The function must accept a *block\_struct* on page 1-118 as input and return a matrix, vector, or scalar. If *fun* returns empty, then blockproc does not generate any output and returns empty after processing all blocks.

For more information about function handles, see “Create Function Handle” (MATLAB).

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'BorderSize', [8 4]`

### Destination — Destination

character vector | `ImageAdapter` object

Destination for the output, specified as the comma-separated pair consisting of `'Destination'` and one of the following.

- A character vector with a destination filename. Files must have one of these file types and must be named with one of the listed file extensions.
  - TIFF (\*.tif, \*.tiff)
  - JPEG2000 (\*.jp2, \*.j2c, \*.j2k)

If a file with this name exists, it is overwritten.

- An `ImageAdapter` object, which provides a common API for reading and writing to a particular image file format. For more information, see “Perform Block Processing on Image Files in Unsupported Formats”.

When you specify the `'Destination'` argument, `blockproc` does not return the processed image as an output argument, but instead writes the output to the `'Destination'`. The `'Destination'` argument 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.

---

**Note** You cannot request an output argument when the `'Destination'` argument is specified.

---

### BorderSize — Border size

[0 0] (default) | 2-element vector of positive integers

Number of border pixels to add to each block, specified as the comma-separated pair consisting of 'BorderSize' and a 2-element vector of positive integers, of the form [ $v$   $h$ ]. 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` argument for more information.

The function pads blocks with borders extending beyond the image edges with zeros.

### **PadPartialBlocks — Pad partial blocks**

false (default) | true

Pad partial blocks to make them full-sized, specified as the comma-separated pair consisting of 'PadPartialBlocks' and `false` or `true`. 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.

When set to `true`, `blockproc` pads partial blocks to make them full-sized  $m$ -by- $n$  blocks. 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.

### **PadMethod — Pad method**

0 (default) | 'replicate' | 'symmetric' | numeric scalar

Method used to pad the image boundary, specified as the comma-separated pair consisting of 'PadMethod' and one of the following.

Value	Description
'replicate'	Repeat border elements.
'symmetric'	Pad image with mirror reflections of itself.
numeric scalar	Pad image with a scalar value. By default, the image boundary is padded with the value 0.

Data Types: char | string

### **TrimBorder — Remove border pixels**

true (default) | false

Remove border pixels from the output of the user function, specified as the comma-separated pair consisting of 'TrimBorder' and `true` or `false`. When set to `true`, the `blockproc` function removes 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` argument defines `v` and `h`.

### **UseParallel — Use parallel processing**

`false` (default) | `true`

Use parallel processing, specified as the comma-separated pair consisting of 'UseParallel' and `false` or `true`. If you have Parallel Computing Toolbox™ installed, when set to `true`, MATLAB automatically opens a parallel pool of workers on your local machine. `blockproc` runs the computation across the available workers. For more information, see “Parallel Block Processing on Large Image Files”.

### **DisplayWaitbar — Display wait bar**

`true` (default) | `false`

Display wait bar, specified as the comma-separated pair consisting of 'DisplayWaitbar' and `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`.

## **Output Arguments**

### **B — Output matrix**

numeric matrix

Output matrix, returned as a numeric matrix.

## **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:

Field	Description
<code>border</code>	2-element vector of the form <code>[v h]</code> . The <code>border</code> field specifies the size of the vertical and horizontal padding around the block of data. See the <code>BorderSize</code> argument for more information.
<code>blockSize</code>	2-element vector of the form <code>[rows cols]</code> . The <code>blockSize</code> field specifies the size of the block data. If a border has been specified, the size does not include the border pixels.
<code>data</code>	m-by-n or m-by-n-by-p matrix of block data.
<code>imageSize</code>	2-element vector of the form <code>[rows cols]</code> . The <code>imageSize</code> field specifies the full size of the input image.
<code>location</code>	2-element vector of the form <code>[row col]</code> . The <code>location</code> field 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.

## Tips

- Choosing an appropriate block size can significantly improve performance. For more information, see “Block Size and Performance”.
- If the output matrix `B` is too large to fit into memory, omit the output argument and instead use the `Destination` name-value pair argument to write the output to a file.

## Extended Capabilities

### Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function supports automatic parallel processing (requires Parallel Computing Toolbox). To run in parallel, specify the `'UseParallel'` argument as `true`. For more information, see “Parallel Block Processing on Large Image Files”.

- Control parallel behavior with the parallel preferences, including scaling up to a cluster. See `parpool` for information on configuring your parallel environment.
- To run in parallel, this function requires a parallel pool with SPMD enabled.
- Parallel processing does not support an `adapter` source image.

### See Also

`ImageAdapter` | `colfilt` | `nlfilter`

### Topics

“Distinct Block Processing”

“Parallel Block Processing on Large Image Files”

**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.

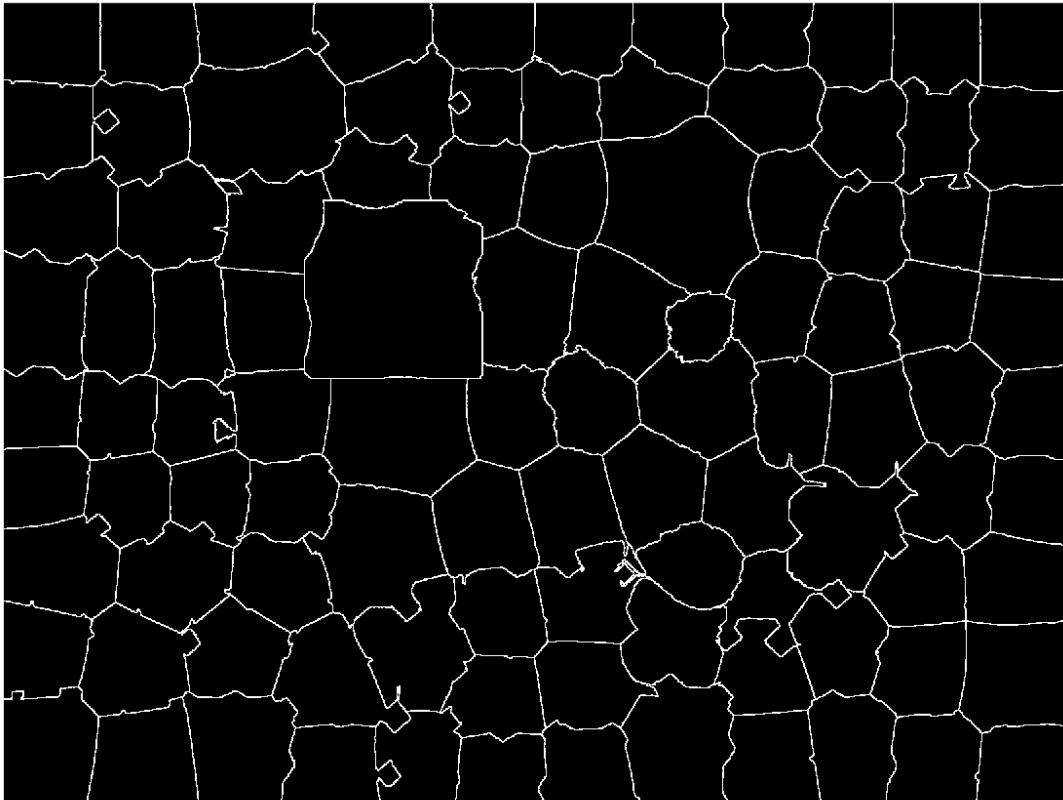
```
L = superpixels(A,100);
```

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 — Label matrix

2-D numeric matrix | 2-D logical matrix

Label matrix, specified as a 2-D numeric matrix of nonnegative numbers or a 2-D logical matrix.

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

### BW — Binary image

numeric matrix | logical matrix

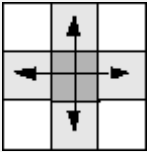
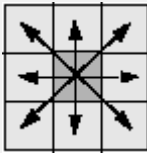
Binary image, specified as a numeric or logical matrix of the same size as L.

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

### conn — Pixel connectivity

8 (default) | 4

Pixel connectivity, specified as 4 or 8.

Value	Meaning
<b>Two-Dimensional Connectivities</b>	
4-connected	<p>Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction.</p> 
8-connected	<p>Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.</p> 

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 of the same size as the input image. A pixel in `mask` is `true` when the corresponding pixel in the input image with value  $P$  has a neighboring pixel with a different value than  $P$ .

Data Types: `logical`

## 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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, the input argument `conn` must be a compile-time constant.

### See Also

`imoverlay` | `label2idx` | `superpixels`

**Introduced in R2016a**

# brisque

Blind/Referenceless Image Spatial Quality Evaluator (BRISQUE) no-reference image quality score

## Syntax

```
score = brisque(A)  
score = brisque(A,model)
```

## Description

`score = brisque(A)` calculates the no-reference image quality score for image `A` using the Blind/Referenceless Image Spatial Quality Evaluator (BRISQUE). `brisque` compare `A` to a default model computed from images of natural scenes with similar distortions. A smaller score indicates better perceptual quality.

`score = brisque(A,model)` calculates the image quality score using a custom feature model.

## Examples

### Calculate BRISQUE Score Using Default Feature Model

Compute the BRISQUE score for a natural image and its distorted versions using the default model.

Read an image into the workspace. Create copies of the image with noise and blurring distortions.

```
I = imread('lighthouse.png');  
Inoise = imnoise(I,'salt & pepper',0.02);  
Iblur = imgaussfilt(I,2);
```

Display the images.

```
montage({I,Inoise,Iblur},'Size',[1 3],'ThumbnailSize',([]))  
title('Original Image | Noisy Image | Blurry Image')
```



Calculate the BRISQUE score for each image using the default model, and display the score.

```
brisqueI = brisque(I);  
fprintf('BRISQUE score for original image is %0.4f.\n',brisqueI)
```

```
BRISQUE score for original image is 20.6586.
```

```
brisqueInoise = brisque(Inoise);  
fprintf('BRISQUE score for noisy image is %0.4f.\n',brisqueInoise)
```

```
BRISQUE score for noisy image is 52.6074.
```

```
brisqueIblur = brisque(Iblur);  
fprintf('BRISQUE score for blurry image is %0.4f.\n',brisqueIblur)
```

```
BRISQUE score for blurry image is 47.7552.
```

The original undistorted image has the best perceptual quality and therefore the lowest BRISQUE score.

## Calculate BRISQUE Score Using Custom Feature Model

Train a custom BRISQUE model from a set of quality-aware features and corresponding human opinion scores. Use the custom model to calculate a BRISQUE score for an image of a natural scene.

Save images from an image datastore. These images all have compression artifacts resulting from JPEG compression.

```
setDir = fullfile(toolboxdir('images'),'imdata');  
imds = imageDatastore(setDir,'FileExtensions',{' .jpg'});
```

Specify the opinion score for each image. The following differential mean opinion score (DMOS) values are for illustrative purposes only. They are not real DMOS values obtained through experimentation.

```
opinionScores = 100*rand(1,size(imds.Files,1));
```

Create the custom model of quality-aware features using the image datastore and the opinion scores. Because the scores are random, the property values will vary.

```
model = fitbrisque(imds,opinionScores')
```

```
Extracting features from 37 images.  
..  
Completed 4 of 37 images. Time: Calculating...  
...  
Completed 13 of 37 images. Time: 00:22 of 00:51  
...  
Completed 23 of 37 images. Time: 00:32 of 00:48  
..  
Completed 31 of 37 images. Time: 00:42 of 00:50  
..Training support vector regressor...
```

Done.

```
model =  
    brisqueModel with properties:  
  
        Alpha: [35x1 double]  
        Bias: 56.2632  
    SupportVectors: [35x36 double]  
        Kernel: 'gaussian'
```

Scale: 0.2717

Read an image of a natural scene that has the same type of distortion as the training images. Display the image.

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



Calculate the BRISQUE score for the image using the custom model. Display the score.

```
brisqueI = brisque(I,model);  
fprintf('BRISQUE score for the image is %0.4f.\n',brisqueI)
```

BRISQUE score for the image is 78.7367.

## Input Arguments

### **A** — Input image

2-D grayscale image | 2-D RGB image

Input image, specified as a 2-D grayscale or RGB image.

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

### **model** — Custom model

`brisqueModel` object

Custom model trained on a set of quality-aware features, specified as a `brisqueModel` object. `model` is derived from natural scene statistics.

## Output Arguments

### **score** — No-reference image quality score

nonnegative scalar

No-reference image quality score, returned as a nonnegative scalar. The BRISQUE score is usually in the range [0, 100]. Lower values of `score` reflect better perceptual quality of image `A` with respect to the input `model`.

Data Types: `double`

## Algorithms

`brisque` predicts the BRISQUE score by using a support vector regression (SVR) model trained on an image database with corresponding differential mean opinion score (DMOS) values. The database contains images with known distortion such as compression artifacts, blurring, and noise, and it contains pristine versions of the distorted images. The image to be scored must have at least one of the distortions for which the model was trained.

## References

- [1] Mittal, A., A. K. Moorthy, and A. C. Bovik. "No-Reference Image Quality Assessment in the Spatial Domain." *IEEE Transactions on Image Processing*. Vol. 21, Number 12, December 2012, pp. 4695–4708.
- [2] Mittal, A., A. K. Moorthy, and A. C. Bovik. "Referenceless Image Spatial Quality Evaluation Engine." Presentation at the 45th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, November 2011.

## See Also

### Functions

fitbrisque | fitniqe | niqe | piqe

### Objects

brisqueModel

### Topics

"Image Quality Metrics"

### Introduced in R2017b



# brisqueModel

Blind/Referenceless Image Spatial Quality Evaluator (BRISQUE) model

## Description

A `brisqueModel` object encapsulates a model used to calculate the Blind/Referenceless Image Spatial Quality Evaluator (BRISQUE) perceptual quality score of an image. The object contains a support vector regressor (SVR) model.

## Creation

You can create a `brisqueModel` object using the following methods:

- `fitbrisque` — Train a BRISQUE model containing a custom trained support vector regressor (SVR) model. Use this function if you do not have a pretrained model.
- The `brisqueModel` function described here. Use this function if you have a pretrained SVR model, or if the default model is sufficient for your application.

## Syntax

```
m = brisqueModel
m = brisqueModel(alpha,bias,supportVectors,scale)
```

## Description

`m = brisqueModel` creates a BRISQUE model object with default property values that are derived from the LIVE IQA image database [1] [2].

`m = brisqueModel(alpha,bias,supportVectors,scale)` creates a custom BRISQUE model and sets the `Alpha` on page 1-0 , `Bias` on page 1-0 , `SupportVectors` on page 1-0 , and `Scale` on page 1-0 properties. You must provide all four arguments to create a custom model.

---

**Note** It is difficult to predict good property values without running an optimization routine. Use this syntax only if you are creating a `brisqueModel` object using a pretrained SVR model with known property values.

---

## Properties

### **Alpha — Coefficients obtained by solving dual problem**

*m*-by-1 numeric vector

Coefficients obtained by solving the dual problem, specified as an *m*-by-1 numeric vector. The length of Alpha must match the number of support vectors (the number of rows of `SupportVectors` on page 1-0 ).

Example: `rand(10,1)`

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

### **Bias — Bias term in SVM model**

43.4582 (default) | numeric scalar

Bias term in SVM model, specified as a numeric scalar.

Example: `47.4`

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

### **SupportVectors — Support vectors**

*m*-by-36 numeric vector

Support vectors, specified as an *m*-by-36 numeric vector. The number of rows, *m*, matches the length of Alpha on page 1-0 .

Example: `rand(10,36)`

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

### **Kernel — Kernel function**

'gaussian' (default)

This property is read-only.

Kernel function, specified as 'gaussian'.

**Scale — Kernel scale factor**

0.3210 (default) | numeric scalar

Kernel scale factor, specified as a numeric scalar. The scale factor divides predictor values in the SVR kernel.

Example: 0.25

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

## Examples

**Create BRISQUE Model Object with Default Properties**

```
model = brisqueModel
```

```
model =  
    brisqueModel with properties:  
  
        Alpha: [593x1 double]  
        Bias: 43.4582  
    SupportVectors: [593x36 double]  
        Kernel: 'gaussian'  
        Scale: 0.3210
```

**Create BRISQUE Model Object with Custom Properties**

Create a `brisqueModel` object using precomputed Alpha, Bias, SupportVectors, and Scale properties. Random initializations are shown for illustrative purposes only.

```
model = brisqueModel(rand(10,1),47,rand(10,36),0.25)
```

```
model =  
    brisqueModel with properties:  
  
        Alpha: [10x1 double]  
        Bias: 47  
    SupportVectors: [10x36 double]  
        Kernel: 'gaussian'
```

Scale: 0.2500

You can use the custom model to calculate the BRISQUE score for an image.

```
I = imread('lighthouse.png');  
score = brisque(I,model)
```

```
score = 47
```

## Algorithms

The support vector regressor (SVR) calculates regression scores for predictor matrix  $X$  as:

$$F = G(X, \text{SupportVectors}) \times \text{Alpha} + \text{Bias}$$

$G(X, \text{SupportVectors})$  is an  $n$ -by- $m$  matrix of kernel products for  $n$  rows in  $X$  and  $m$  rows in  $\text{SupportVectors}$ . The SVR has 36 predictors, which determine the number of columns in  $\text{SupportVectors}$ .

The SVR computes a kernel product between vectors  $x$  and  $z$  using  $\text{Kernel}(x/\text{Scale}, z/\text{Scale})$ .

## References

- [1] Mittal, A., A. K. Moorthy, and A. C. Bovik. "No-Reference Image Quality Assessment in the Spatial Domain." *IEEE Transactions on Image Processing*. Vol. 21, Number 12, December 2012, pp. 4695–4708.
- [2] Mittal, A., A. K. Moorthy, and A. C. Bovik. "Referenceless Image Spatial Quality Evaluation Engine." Presentation at the 45th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, November 2011.

## See Also

### Functions

`brisque` | `fitbrisque`

### **Objects**

CompactRegressionSVM | niqeModel

### **Topics**

“Image Quality Metrics”

“Train and Use a No-Reference Quality Assessment Model”

**Introduced in R2017b**

## **bwarea**

Area of objects in binary image

### **Syntax**

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

### **Description**

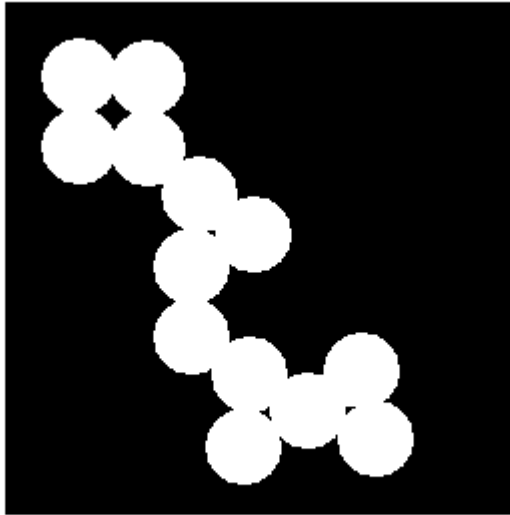
`total = bwarea(BW)` estimates the area of the objects in binary image `BW`. `total` is a scalar whose value corresponds roughly to the total number of on pixels in the image, but might not be exactly the same because different patterns of pixels are weighted differently.

### **Examples**

#### **Calculate Area of Objects in Binary Image**

Read a binary image and display it.

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



Calculate the area of objects in the image.

```
bwarea(BW)
```

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

## Input Arguments

### **BW** — Binary image

2-D numeric matrix | 2-D logical matrix

Binary image, specified as a 2-D numeric or logical matrix. For numeric input, any nonzero pixels are considered to be 1 (**true**).

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

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

## Output Arguments

### **total** — Estimated number of on pixels

numeric scalar

Estimated number of on pixels in binary image `BW`, returned as a numeric scalar.

Data Types: `double`

## 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)

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

`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)
BW2 = bwareafilt( ____,conn)
```

## Description

`BW2 = bwareafilt(BW,range)` extracts all connected components (objects) from the binary image `BW`, where the area of the objects is in the specified `range`, producing another binary image `BW2`. `bwareafilt` returns a binary image `BW2` containing only those objects that meet the criteria.

`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)` specifies whether to keep the `n` largest objects or the `n` smallest objects.

`BW2 = bwareafilt( ____,conn)` specifies the pixel connectivity that defines the objects.

## Examples

### Filter Binary Image by Area of Objects

Read image.

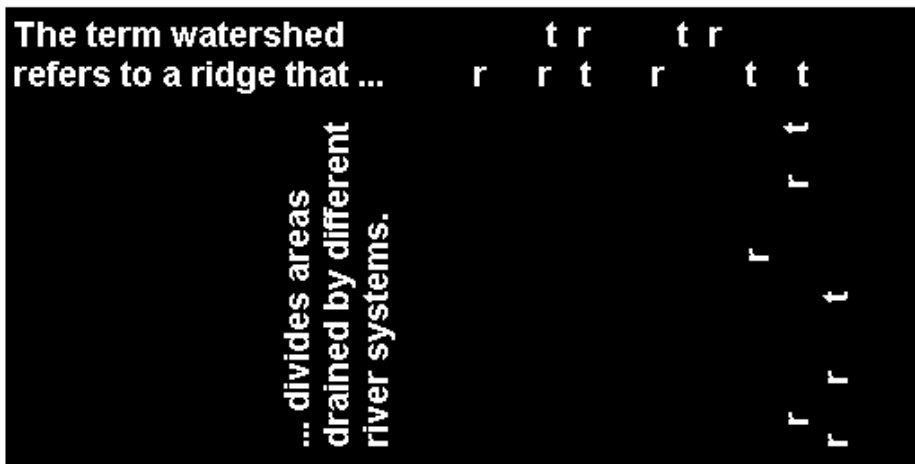
```
BW = imread('text.png');
```

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.

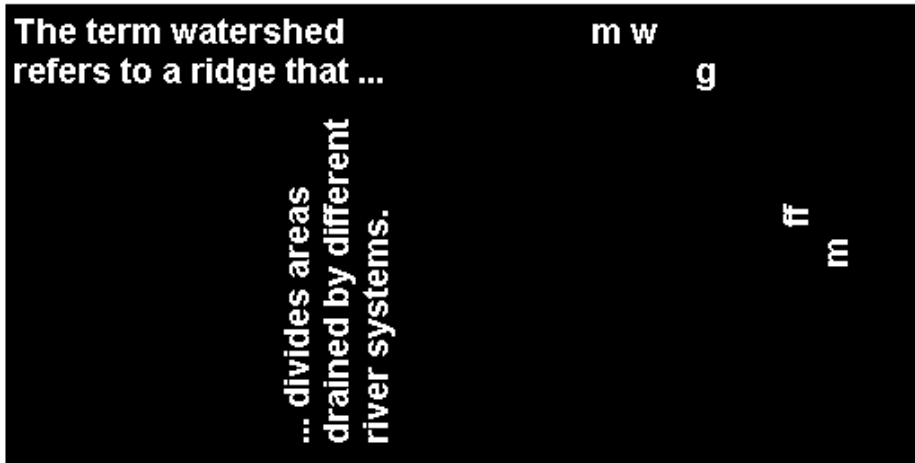
```
BW = imread('text.png');
```

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')
```



## 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 areas

2-by-1 numeric vector

Minimum and maximum values of the area, 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**

numeric scalar

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

Data Types: double

**keep — Size of objects to include**

'largest' (default) | 'smallest'

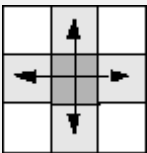
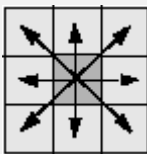
Size of objects to include in the output image, specified as 'largest' or 'smallest'. In the event of a tie for n-th place, `bwareafilt` includes only the first n objects.

Data Types: char | string

**conn — Pixel connectivity**

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

Pixel connectivity, specified as one of these values.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal, vertical, or diagonal direction.	

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

Data Types: double | logical

## Output Arguments

### **BW2 — Filtered image**

binary image

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

## See Also

[bwareaopen](#) | [bwconncomp](#) | [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`. 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.

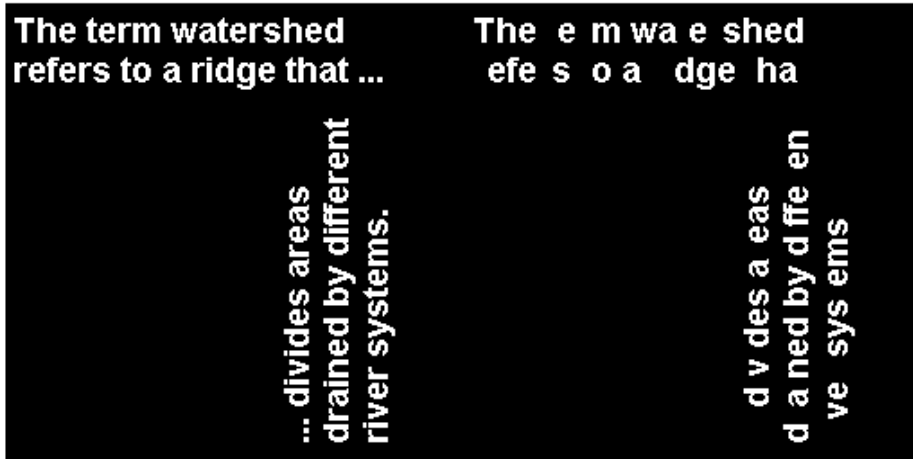
```
BW = imread('text.png');
```

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.

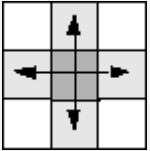
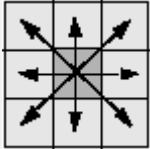
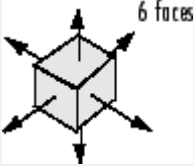
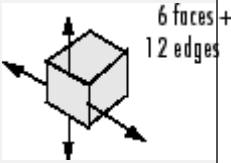
Example: BW2 = bwareaopen(BW, 50);

Data Types: double

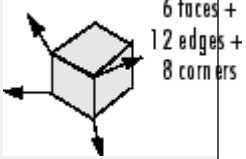
**conn — Pixel connectivity**

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal, vertical, or diagonal direction.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch. Two adjoining pixels are part of the same object if they are both on and are connected in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	
18-connected	Pixels are connected if their faces or edges touch. Two adjoining pixels are part of the same object if they are both on and are connected in <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	



Value	Meaning	
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected in</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	

For higher dimensions, `bwareaopen` uses the default value `conndef(ndims(BW), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

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

`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.

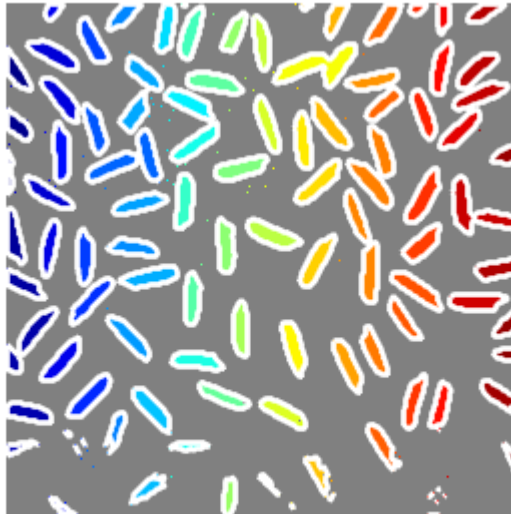
```
I = imread('rice.png');
```

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  
end
```



## Overlay Region Boundaries on Image and Annotate with Region Numbers

Read binary image into the workspace.

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

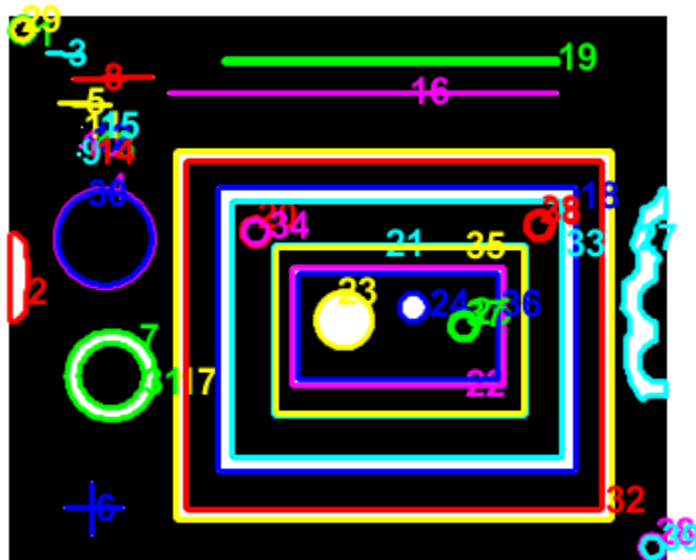
Calculate boundaries of regions in the image.

```
[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.

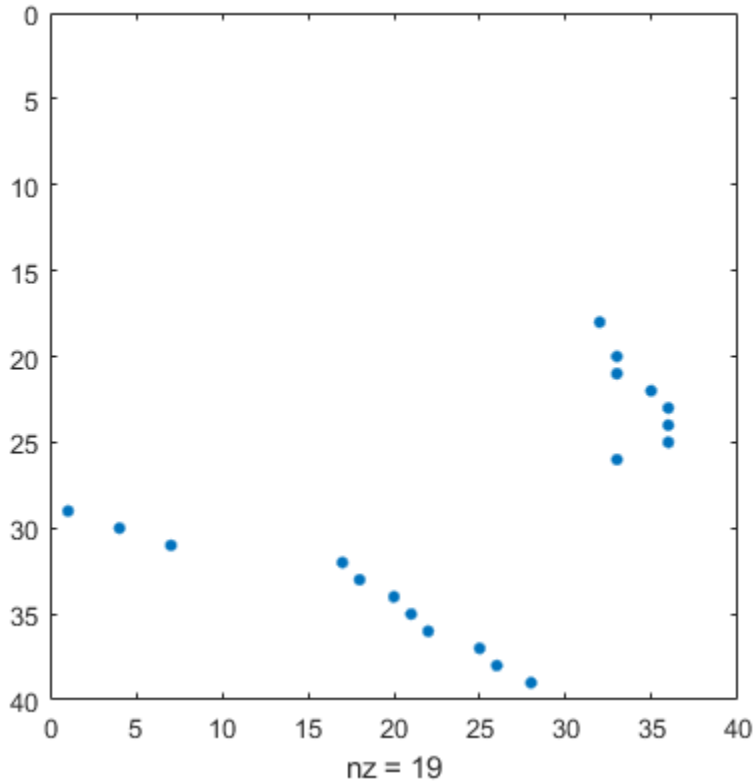
```
imshow(BW); hold on;
colors=['b' 'g' 'r' 'c' 'm' 'y'];
for k=1:length(B),
    boundary = B{k};
    cidx = mod(k,length(colors))+1;
    plot(boundary(:,2), boundary(:,1),...
         colors(cidx),'LineWidth',2);

    %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.

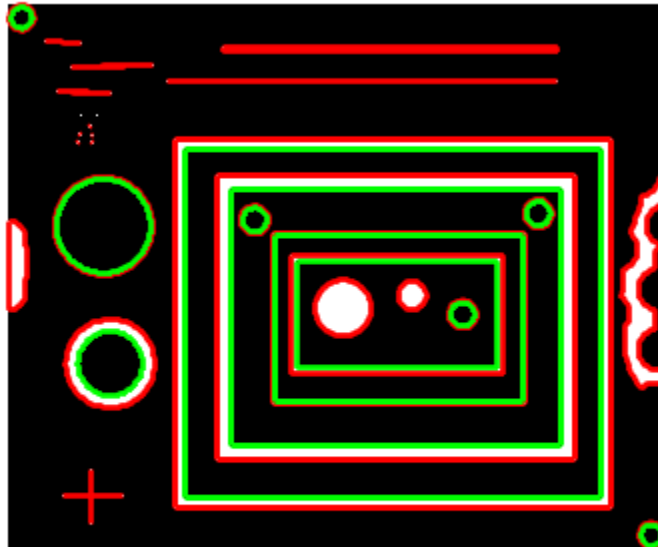
```
BW = imread('blobs.png');
```

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  
end
```



## Display Parent Boundaries in Red and Holes in Green

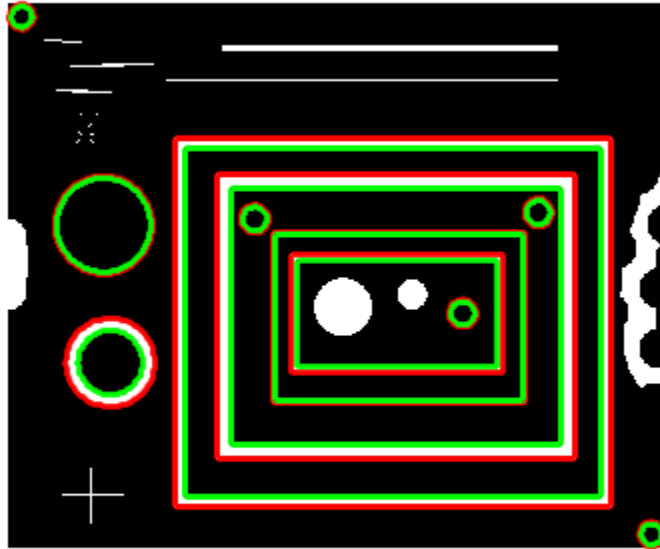
Read image into workspace.

```
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
```

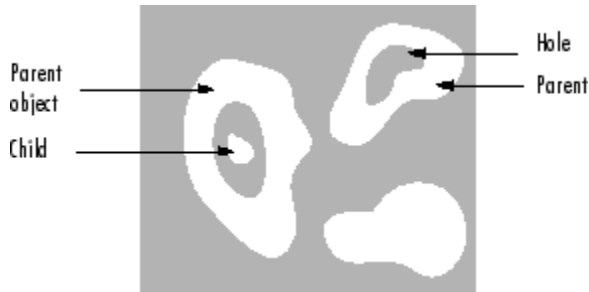


## Input Arguments

### **BW** — Input binary image

2-D numeric matrix | 2-D logical matrix

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

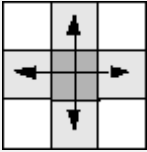
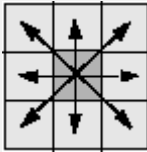


Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

**conn — Pixel connectivity**

8 (default) | 4

Pixel connectivity, specified as one of the values in this table.

Value	Meaning
<b>Two-Dimensional Connectivities</b>	
4-connected	<p>Pixels are connected if their edges touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal or vertical direction.</p> 
8-connected	<p>Pixels are connected if their edges or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal, vertical, or diagonal direction.</p> 

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` | `string`

## 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

2-D matrix of nonnegative integers

Label matrix of contiguous regions, returned as a 2-D matrix of nonnegative integers. 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.

Data Types: `double`

### **n** — Number of objects found

nonnegative integer

Number of objects found, returned as a nonnegative integer.

Data Types: `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 “Understand 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**

`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: {[5x1 double] [3x1 double]}
```

Calculate centroids of the objects in the array.

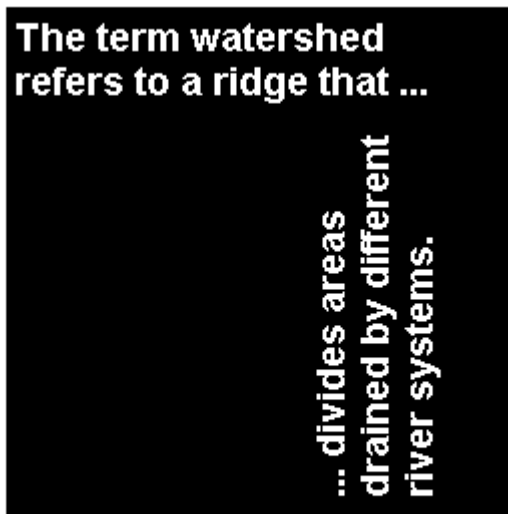
```
S = regionprops(CC, 'Centroid')
```

```
S = 2x1 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)

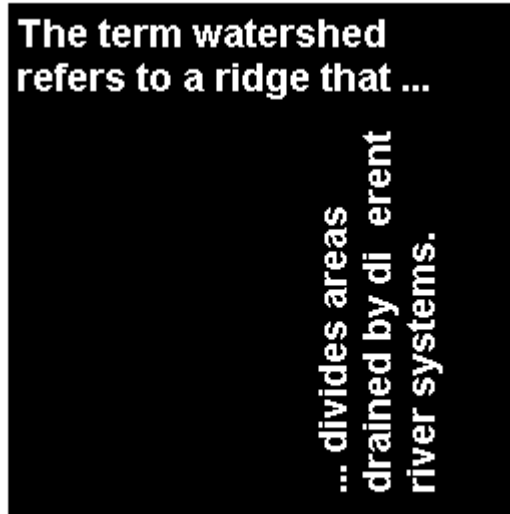
CC = struct with fields:
    Connectivity: 8
    ImageSize: [256 256]
    NumObjects: 88
    PixelIdxList: {1x88 cell}
```

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

numeric array | logical array

Input binary image, specified as a numeric or logical array of any dimension. For numeric input, any nonzero pixels are considered to be on.

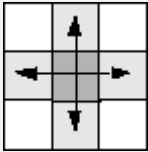
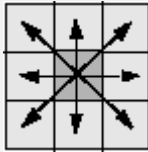
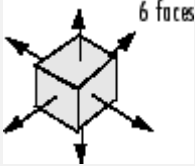
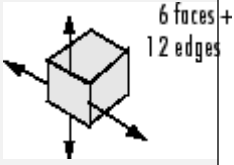
Example: `BW = imread('text.png');`

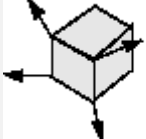
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### **conn** — Pixel connectivity

`4` | `8` | `6` | `18` | `26` | `3-by-3-by- ... -by-3` matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal, vertical, or diagonal direction.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch. Two adjoining pixels are part of the same object if they are both on and are connected in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	
18-connected	Pixels are connected if their faces or edges touch. Two adjoining pixels are part of the same object if they are both on and are connected in <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	

Value	Meaning	
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected in</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	 <p>6 faces + 12 edges + 8 corners</p>

For higher dimensions, `bwconncomp` uses the default value `conndef(ndims(BW), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

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 <i>k</i> -th element in the cell array is a vector containing the linear indices of the pixels in the <i>k</i> -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

`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.

## Examples

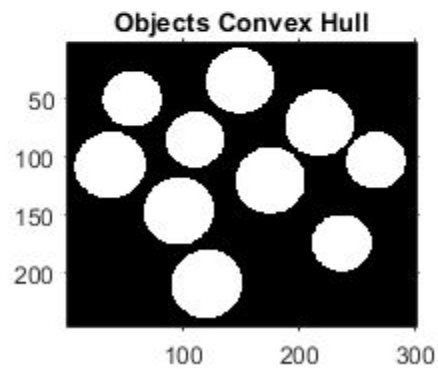
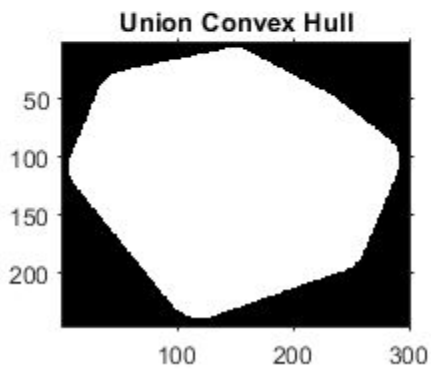
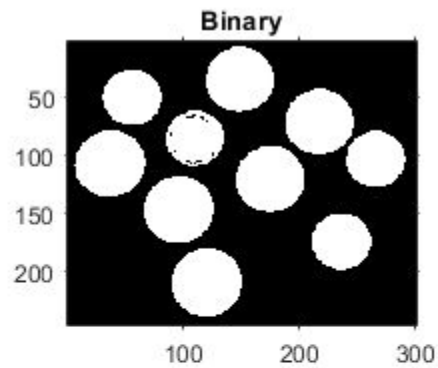
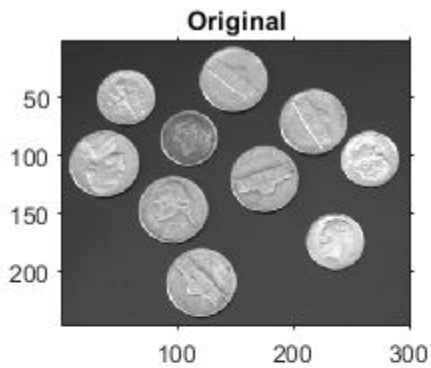
### Display Binary Convex Hull of Image

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');
```





## Input Arguments

### BW — Input binary image

2-D logical matrix

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

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:

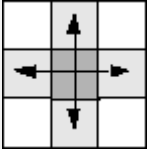
Value	Description
'union'	Compute the convex hull of all foreground objects, treating them as a single object
'objects'	Compute the convex hull of each connected component of BW individually. CH contains the convex hulls of each connected component.

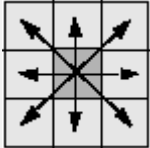
Data Types: `char` | `string`

### conn — Pixel connectivity

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

Pixel connectivity, specified as one of these values The conn parameter is only valid when the method is 'objects'.

Value	Meaning
<b>Two-Dimensional Connectivities</b>	
4-connected	Pixels are connected if their edges touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal or vertical direction. 

<b>Value</b>	<b>Meaning</b>
8-connected	Pixels are connected if their edges or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal, vertical, or diagonal direction. 

Connectivity can also be defined in a more general way by specifying a 3-by-3 matrix of 0s and 1s. The 1-valued elements define neighborhood locations relative to the center element of `conn`. The matrix 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 matrix

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

## 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)
```

### 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`.

You optionally can compute the Euclidean distance transform of a 2-D binary image using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

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

`[D,idx] = bwdist(BW,method)` computes the distance transform using an alternate distance metric, specified by `method`.

Only the 'euclidean' method is supported on a GPU.

### 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 = 5x5*

```
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 = 5x5 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 = 5x5 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 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)
```

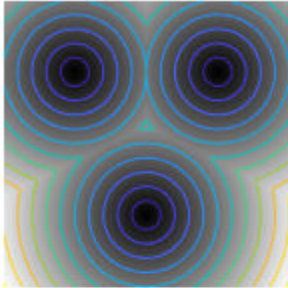
## Compare 2-D Distance Transforms for Supported Distance Methods

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.

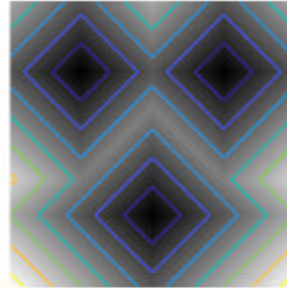
```
bw = zeros(200,200);
bw(50,50) = 1; bw(50,150) = 1; bw(150,100) = 1;
D1 = bwdist(bw, 'euclidean');
D2 = bwdist(bw, 'cityblock');
D3 = bwdist(bw, 'chessboard');
D4 = bwdist(bw, 'quasi-euclidean');
RGB1 = repmat(rescale(D1), [1 1 3]);
RGB2 = repmat(rescale(D2), [1 1 3]);
RGB3 = repmat(rescale(D3), [1 1 3]);
RGB4 = repmat(rescale(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)
```

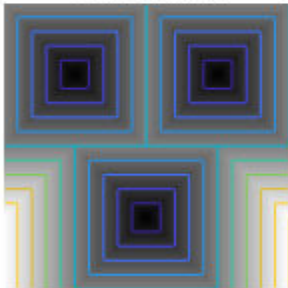
**Euclidean**



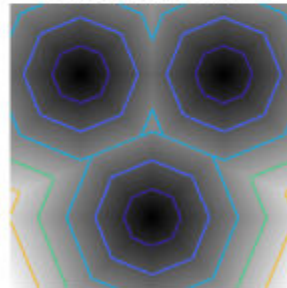
**City block**



**Chessboard**



**Quasi-Euclidean**

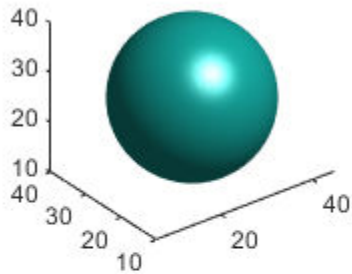
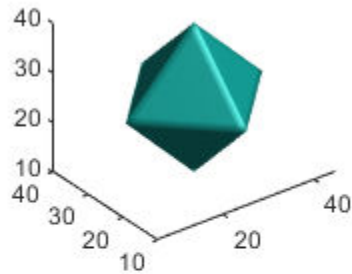
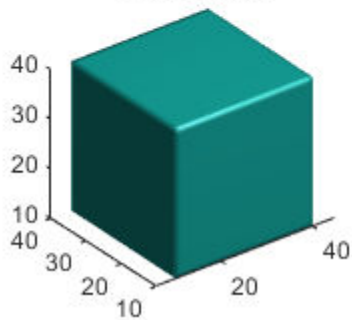
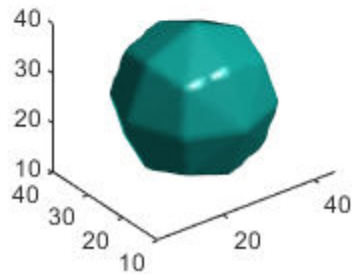


### Compare Isosurface Plots for Distance Transforms of 3-D Image

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')
```

**Euclidean****City block****Chessboard****Quasi-Euclidean**

## Input Arguments

### BW — Binary image

numeric array | logical array | gpuArray

Binary image, specified as a numeric or logical array of any dimension. For numeric input, any nonzero pixels are considered to be 1 (`true`).

To compute the Euclidean distance transform using a GPU, specify BW as a `gpuArray` that contains a 2-D numeric or logical matrix with fewer than  $2^{32}$  elements.

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

### method — Distance metric

'euclidean' (default) | 'chessboard' | 'cityblock' | 'quasi-euclidean'

Distance metric, specified as one of the these values.

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

Only the 'euclidean' method is supported on a GPU.



For more information, see “Distance Transform of a Binary Image”.

Data Types: `char` | `string`

## Output Arguments

### **D** — Distance array

numeric array | `gpuArray`

Distance, returned as a numeric array of the same size as `BW`. The value of each element is the distance between that pixel and the nearest nonzero pixel in `BW`, as defined by the distance metric, `method`.

If the Euclidean distance transform is computed using a GPU, then `D` is returned as a `gpuArray` containing a 2-D numeric matrix.

Data Types: `single`

### **idx** — Index array

numeric array | `gpuArray`

Index array, returned as a numeric array of the same size as `BW`. Each element of `idx` contains the linear index of the nearest nonzero pixel of `BW`. The class of `idx` depends on the number of elements in the input image, and is determined as follows.

Class	Range
'uint32'	$\text{numel}(\text{BW}) \leq 2^{32} - 1$
'uint64'	$\text{numel}(\text{BW}) \geq 2^{32}$

If the Euclidean distance transform is computed using a GPU, then `idx` is returned as a `gpuArray` containing a 2-D numeric matrix of data type `uint32`.

Data Types: `uint32` | `uint64`

## 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. [1]
- For cityblock, chessboard, and quasi-Euclidean distance transforms, `bwdist` uses the two-pass, sequential scanning algorithm. [2]
- The different distance measures are achieved by using different sets of weights in the scans, as described in [3].

## References

- [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.
- [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.
- [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 “Understand 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.

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Input images must be 2-D and have less than  $2^{32}$  elements.
- Euclidean is the only distance metric supported.

For more information, see “Image Processing on a GPU”.

## See Also

`bwulterode` | `watershed`

## Topics

“Distance Transform of a Binary Image”

**Introduced before R2006a**

## **bwdistgeodesic**

Geodesic distance transform of binary image

### **Syntax**

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

`D = bwdistgeodesic( ____,method)` computes the geodesic distance transform using an alternate distance metric, specified by `method`.

### **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 = 10x10 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

## Input Arguments

### **BW** — Binary image

real, nonsparse, numeric or logical array of any dimension

Binary image, specified as a real, nonsparse, numeric or logical array of any dimension. For numeric input, any nonzero pixels are considered to be on.

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

### **mask** — Seed locations

logical array

Seed locations, specified as a logical array of the same size as **BW**.

### **C, R** — Column or row coordinates of seed locations

vector of positive integers

Column or row coordinates of seed locations, specified as a vector of positive integers. Coordinate values are valid C,R subscripts in **BW**.

### **idx** — Linear indices of seed locations

vector of positive integers

Linear indices of seed locations, specified as a vector of positive integers.

### **method** — Distance metric

`'chessboard'` (default) | `'cityblock'` | `'quasi-euclidean'`

Distance metric, specified as one of the following.

Method	Description
<code>'chessboard'</code>	In 2-D, the chessboard distance between $(x_1, y_1)$ and $(x_2, y_2)$ is $\max(\text{abs}(x_1 - x_2), \text{abs}(y_1 - y_2))$
<code>'cityblock'</code>	In 2-D, the cityblock distance between $(x_1, y_1)$ and $(x_2, y_2)$ is $\text{abs}(x_1 - x_2) + \text{abs}(y_1 - y_2)$

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

Data Types: char | string

## Output Arguments

### D — Geodesic distances

numeric array

Geodesic distances, returned as a numeric array of the same size as BW.

Data Types: single

## 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

bwdist | graydist

Introduced in R2011b

## bweuler

Euler number of binary image

### Syntax

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

### Description

`eul = bweuler(BW,conn)` 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. `conn` specifies the connectivity. Objects are connected sets of on pixels, that is, pixels having a value of 1.

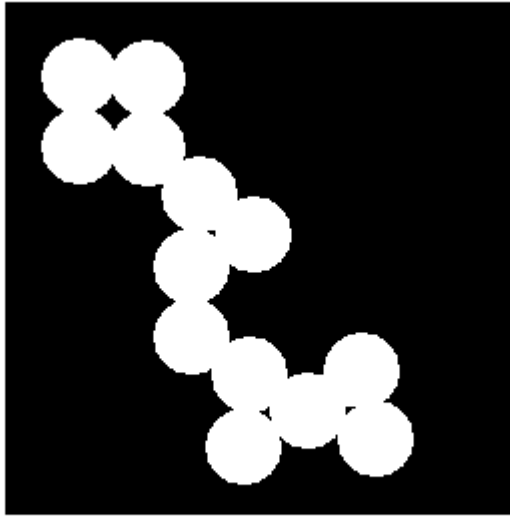
### Examples

#### 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** — Binary image

2-D real, nonsparse, numeric or logical matrix

Binary image, specified as a 2-D, real, nonsparse, numeric or logical matrix. For numeric input, any nonzero pixels are considered to be on.

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

### **conn — 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 = bweuler(BW,4);`

Data Types: `double`

## Output Arguments

### **eu1 — Euler number**

numeric scalar

Euler number, returned as a numeric scalar.

Data Types: `double`

## Algorithms

`bweuler` computes the Euler number by considering patterns of convexity and concavity in local 2-by-2 neighborhoods. See [2] on page 1-188 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 “Understand Code Generation with Image Processing Toolbox”.

### See Also

`bwmorph` | `bwperim`

**Introduced before R2006a**

## **bwhitmiss**

Binary hit-miss operation

### **Syntax**

```
BW2 = bwhitmiss(BW,SE1,SE2)
BW2 = bwhitmiss(BW,interval)
```

### **Description**

`BW2 = bwhitmiss(BW,SE1,SE2)` performs the hit-miss operation defined by the structuring elements `SE1` and `SE2`. The hit-miss operation preserves pixels in binary image `BW` whose neighborhoods match the shape of `SE1` and don't match the shape of `SE2`.

This syntax is equivalent to `imerode(BW,SE1) & imerode(~BW,SE2)`.

`BW2 = bwhitmiss(BW,interval)` performs the hit-miss operation defined in terms of a single array, called an *interval*. An interval is an array whose elements are 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.

This syntax is equivalent to `bwhitmiss(BW,interval==1,interval==-1)`.

### **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 = 6x6
```

```

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

```

## Input Arguments

### **BW** — Binary image

numeric array | logical array

Binary image, specified as a numeric or logical array of any dimension. For numeric input, any nonzero pixels are considered to be 1 (true).

### **SE1, SE2** — Structuring element

strel object | numeric array

Flat structuring element, specified as a `strel` object or a numeric matrix with values of 1 and 0. The neighborhoods of SE1 and SE2 should not have overlapping elements.

## **interval — Interval**

numeric array

Interval, specified as a numeric array with values of 1, 0, and -1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64`

## **Output Arguments**

### **BW2 — Processed binary image**

logical array

Processed binary image after the hit-miss operation, specified as a logical array of the same size as `BW`.

Data Types: `logical`

## **See Also**

`imdilate` | `imerode` | `strel`

**Introduced before R2006a**

# bwlabel

Label connected components in 2-D binary image

## Syntax

```
L = bwlabel(BW)
L = bwlabel(BW,conn)
[L,n] = bwlabel( ___ )
```

## Description

`L = bwlabel(BW)` returns the label matrix `L` that contains labels for the 8-connected objects found in `BW`.

You optionally can label connected components in a 2-D binary image using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

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

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

## 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 = 8x8
```

```
    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 = 4x2
```

```
    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 — Binary image

2-D numeric matrix | 2-D logical matrix | gpuArray

Binary image, specified as a 2-D numeric or logical matrix. For numeric input, any nonzero pixels are considered to be on.

To label connected components using a GPU, specify `BW` as a `gpuArray` that contains a 2-D numeric or logical matrix.

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

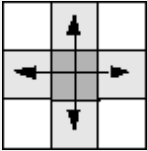
```
Example: BW = gpuArray(imread('text.png'));
```

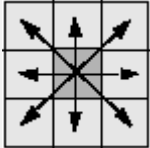
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### conn — Pixel connectivity

8 (default) | 4

Pixel connectivity, specified as one of these values.

Value	Meaning
<b>Two-Dimensional Connectivities</b>	
4-connected	Pixels are connected if their edges touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal or vertical direction. 

Value	Meaning
8-connected	<p>Pixels are connected if their edges or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal, vertical, or diagonal direction.</p> 

Data Types: `double` | `logical`

## Output Arguments

### L — Label matrix

matrix of nonnegative integers | `gpuArray`

Label matrix of contiguous regions, returned as matrix of nonnegative integers with the same size as `BW`. 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.

If connected components are labeled using a GPU, then `L` is returned as a `gpuArray` containing a matrix of nonnegative integers.

Data Types: `double`

### n — Number of connected objects

nonnegative integer

Number of connected objects in `BW`, returned as a nonnegative integer.

Data Types: `double`

## 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`.
- To extract features from a binary image using `regionprops` with default connectivity, just pass `BW` directly into `regionprops` by using the command `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.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

### See Also

`bwconncomp` | `bwlabeln` | `bwselect` | `label2rgb` | `labelmatrix` | `regionprops`

**Introduced before R2006a**

# bwlabeln

Label connected components in binary image

## Syntax

```
L = bwlabeln(BW)
L = bwlabeln(BW,conn)
[L,n] = bwlabeln( ___ )
```

## Description

`L = bwlabeln(BW)` returns a label matrix, `L`, containing labels for the connected components in `BW`.

`L = bwlabeln(BW,conn)` returns a label matrix, where `conn` specifies the connectivity.

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

## 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 =
BW(:,:,1) =
```

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

```
BW(:,:,2) =
```

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

```
BW(:,:,3) =
```

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

Label connected components in the image.

```
bwlabeln(BW)
```

```
ans =
```

```
ans(:,:,1) =
```

```
    1    1    0
    0    0    0
    2    0    0
```

```
ans(:,:,2) =
```

```
    0    1    0
    0    0    0
    0    2    0
```

```
ans(:,:,3) =
```

```
    0    1    1
    0    0    0
    0    0    2
```

## Input Arguments

### BW — Binary image

numeric array | logical array

Binary image, specified as a numeric or logical array of any dimension. For numeric input, any nonzero pixels are considered to be on.

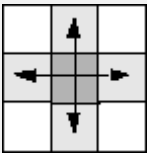
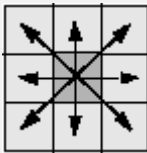
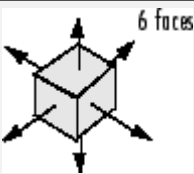
Example: `BW = imread('text.png');`

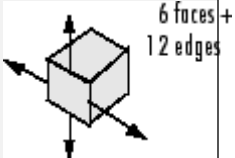
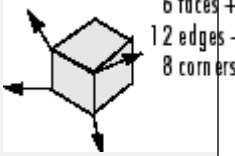
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### conn — Pixel connectivity

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal, vertical, or diagonal direction.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch. Two adjoining pixels are part of the same object if they are both on and are connected in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	

Value	Meaning	
18-connected	<p>Pixels are connected if their faces or edges touch. Two adjoining pixels are part of the same object if they are both on and are connected in</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	 <p>6 faces + 12 edges</p>
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected in</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	 <p>6 faces + 12 edges + 8 corners</p>

For higher dimensions, `bwlabeln` uses the default value `conndef(ndims(BW), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

Data Types: `double` | `logical`

## Output Arguments

### L — Label matrix

array of nonnegative integers



Label matrix, returned as an array of nonnegative integers with the same size as `BW`. 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.

Data Types: `double`

### **L — Label matrix**

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  $\max(L(:))$ . The zero-valued elements of `L` make up the background.

Data Types: `double`

### **n — Number of connected objects**

nonnegative integer

Number of connected objects in `BW`, returned as a nonnegative integer.

Data Types: `double`

## 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 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,n);  
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

`bwconncomp` | `bwlabel` | `label2rgb` | `labelmatrix` | `regionprops`

**Introduced before R2006a**

# bwlookup

Nonlinear filtering using lookup tables

## Syntax

```
J = bwlookup(I,lut)
```

## Description

`J = bwlookup(I,lut)` performs a 2-by-2 or 3-by-3 nonlinear neighborhood filtering operation on binary or grayscale image `I` and returns the results in output image `J`. 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 `J` at the targeted position.

You optionally can perform the filtering using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

## 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 = 16x1
```

```
0  
0  
0
```

```
0
0
0
0
0
0
0
0
0
0
:
```

Load a binary image.

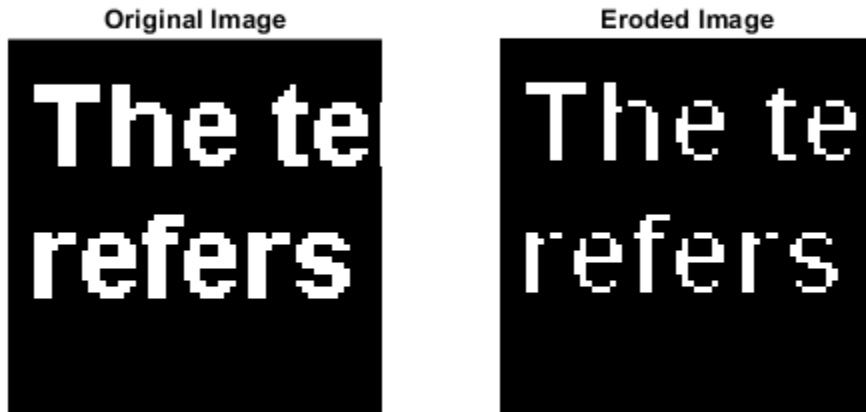
```
BW1 = imread('text.png');
```

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('Original Image')
h2 = subplot(1,2,2); imshow(BW2); axis off; title('Eroded Image')
% 16X zoom to see effects of erosion on text
set(h1,'Ylim',[1 64],'Xlim',[1 64]);
set(h2,'Ylim',[1 64],'Xlim',[1 64]);
```



### 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.

```
I1 = 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.

```
I2 = bwlookup(gpuArray(I1),lut);
```

Show zoomed before and after images.

```
figure;  
h1 = subplot(1,2,1); imshow(I1), axis off; title('I1')  
h2 = subplot(1,2,2); imshow(I2); axis off; title('I2')  
  
% 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

### **I** — Input image

binary image | grayscale image | `gpuArray`

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.

To perform the filtering using a GPU, specify `I` as a `gpuArray` that contains a grayscale or binary image.

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

### **lut** — Lookup table of output pixel values

16- or 512-element vector | `gpuArray`

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.

If you perform the filtering using a GPU, then you optionally can specify `lut` as a `gpuArray` that contains a 16- or 512-element vector.

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

## Output Arguments

### **J** — Output image

`binary image` | `grayscale image` | `gpuArray`

Output image, returned as a grayscale or binary image whose distribution of pixel values are determined by the content of the lookup table, `lut`. The output image `J` is the same size as the input image `I` and the same data type as `lut`.

If the filtering is performed using a GPU, then `J` is returned as a `gpuArray` that contains a grayscale or binary image.

## Algorithms

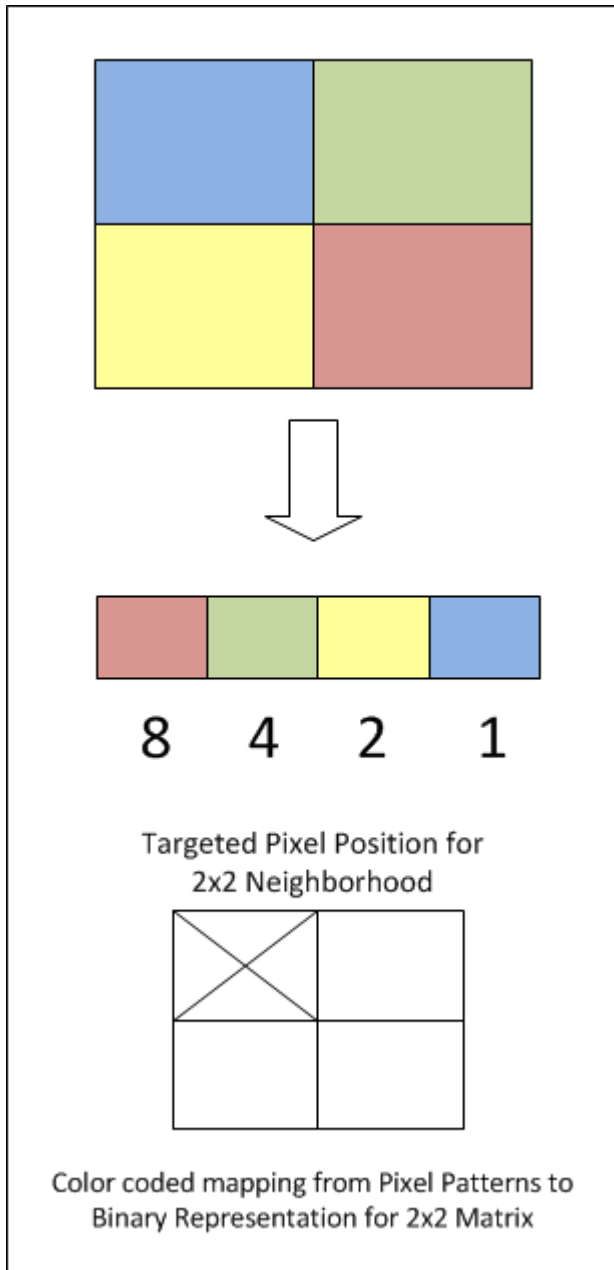
The first step in each iteration of the filtering operation performed by `bwlookup` entails computing the `index` into vector `lut` based on the binary pixel pattern of the neighborhood matrix on image `I`. The value in `lut` accessed at `index`, `lut(index)`, is inserted into output image `J` at the targeted pixel location. This results in image `J` being the same data type as vector `lut`.

Since there is a 1-to-1 correspondence in targeted pixel locations, image `J` is the same size as image `I`. If the targeted pixel location is on an edge of image `I` 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`.

### 2-by-2 Neighborhood Lookup

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 `I(1,1)`.

```
I = [1 0; 0 1]
```

```
I =
```

```
 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 `I(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 `I(2,2)` is red which maps to the Most Significant Bit (MSB) at position 3 in the 4-bit binary representation ( $2^3 = 8$ ).

```
% I(1,1): blue square; sets bit position 0 on right
% I(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(I,lut) % perform filtering
```

```
A =
```

```
246 32
161 231
```

A(1,1) does in fact equal 246.

## 3-by-3 Neighborhood Lookup

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 for 2-by-2 neighborhoods.

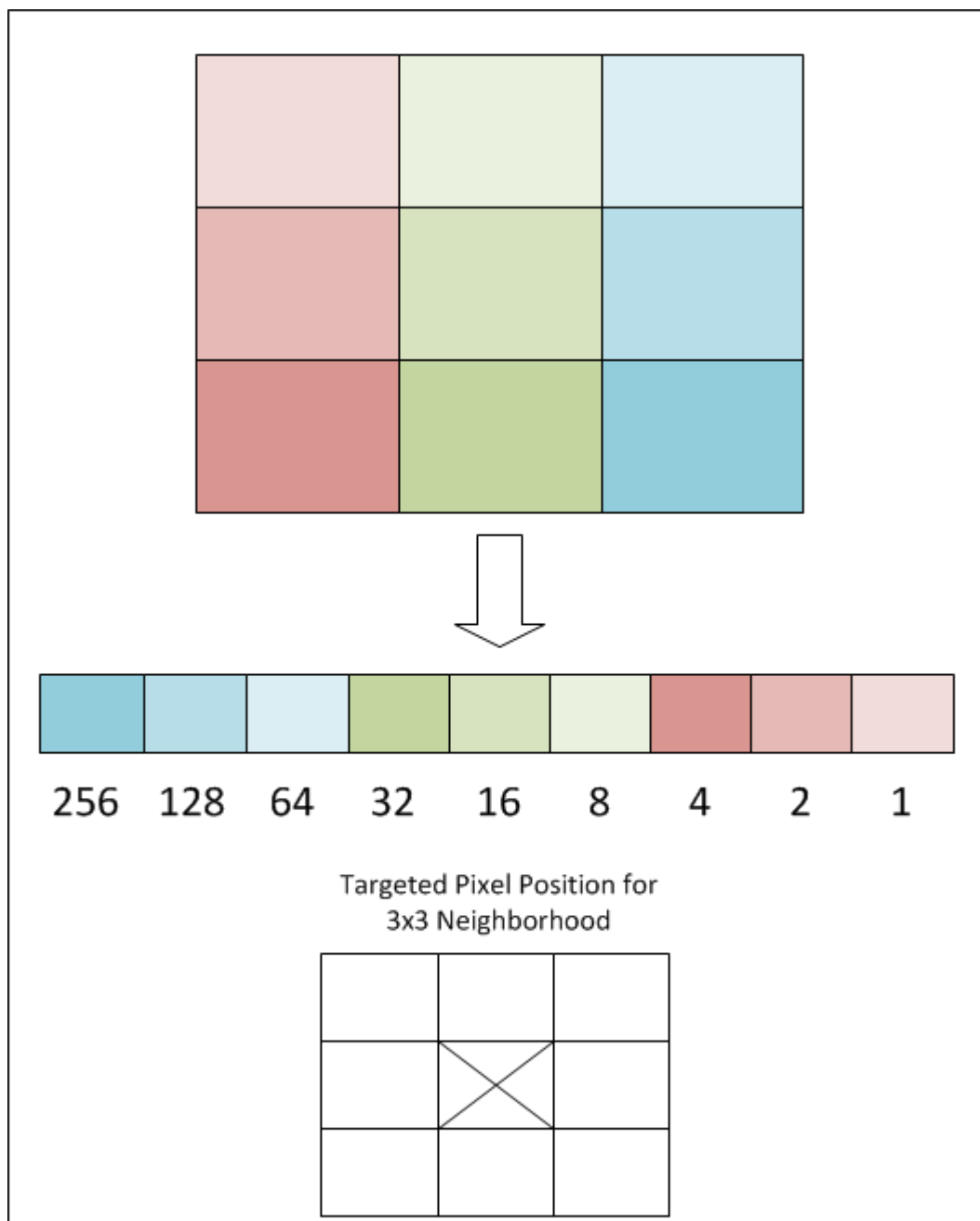


Figure 1: Color coded mapping from Pixel Patterns to Binary Representation for 3x3 Matrix

## 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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, specify an input image of class `logical`.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

### See Also

`makelut`

**Introduced in R2012b**

# bwmorph

Morphological operations on binary images

## Syntax

```
BW2 = bwmorph(BW,operation)
BW2 = bwmorph(BW,operation,n)
```

## Description

`BW2 = bwmorph(BW,operation)` applies a specific morphological operation to the binary image `BW`.

---

**Note** To perform morphological operations on a 3-D volumetric image, use `bwmorph3`.

---

You optionally can perform the morphological operation using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

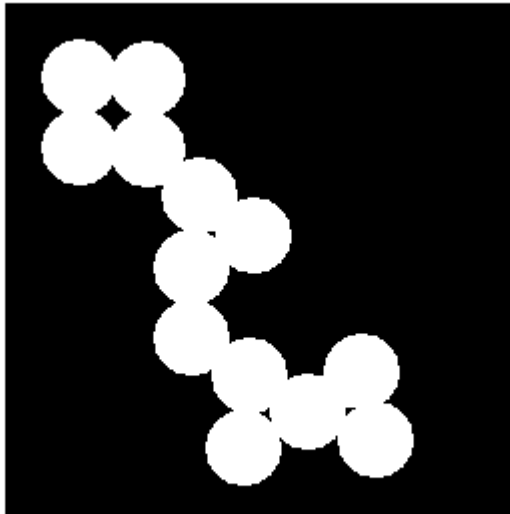
`BW2 = bwmorph(BW,operation,n)` applies the operation `n` times. `n` can be `Inf`, in which case the operation is repeated until the image no longer changes.

## Examples

### Perform Morphological Operations on Binary Image

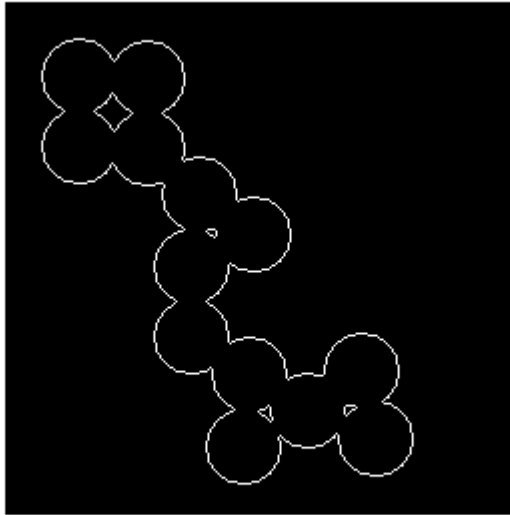
Read binary image and display it.

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



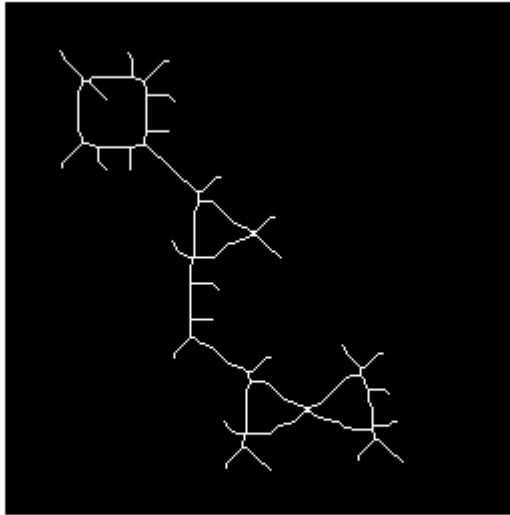
Remove interior pixels to leave an outline of the shapes.

```
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 | gpuArray

Input image, specified as a binary image. The input image can be numeric or logical, and must be 2-D, real and nonsparse.

To perform the morphological operation using a GPU, specify BW as a gpuArray that contains a binary image.

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

### operation — Morphological operation to perform

character vector | string scalar

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>

Operation	Description
'clean'	Removes isolated pixels (individual 1s that are surrounded by 0s), such as the center pixel in this pattern.  <pre>0 0 0 0 1 0 0 0 0</pre>
'close'	Performs morphological closing (dilation followed by erosion).
'diag'	Uses diagonal fill to eliminate 8-connectivity of the background. For example:  <pre>0 1 0      0 1 0 1 0 0 becomes 1 1 0 0 0 0      0 0 0</pre>
'endpoints'	Finds end points of skeleton. For example:  <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'	Fills isolated interior pixels (individual 0s that are surrounded by 1s), such as the center pixel in this pattern.  <pre>1 1 1 1 0 1 1 1 1</pre>
'hbreak'	Removes H-connected pixels. For example:  <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).

Operation	Description
'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.
'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.  When working with 3-D volumes, or when you want to prune a skeleton, use the <code>bwskel</code> function.
'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 1-222 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` | `string`

## **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`

## **Output Arguments**

### **BW2 — Output image**

binary image | `gpuArray`

Output image, returned as a binary image.

If the morphological operation is performed using a GPU, then `BW2` is returned as a `gpuArray` that contains a binary image.

Data Types: `logical`

## **Tips**

- To perform erosion or dilation, use the `imerode` or `imdilate` functions. If you want to duplicate the dilation or erosion performed by `bwmorph`, then 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 ( $n=\text{Inf}$ ), 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 "Understand Code Generation with Image Processing Toolbox".
- When generating code, the character vectors or string scalars specifying the operation must be a compile-time constant and, for best results, the input image must be of class `logical`.

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

## See Also

`bweuler` | `bwmorph3` | `bwperim` | `bwskel` | `imdilate` | `imerode`

**Introduced before R2006a**

## **bwmorph3**

Morphological operations on binary volume

### **Syntax**

```
J = bwmorph3(V,operation)
```

### **Description**

`J = bwmorph3(V,operation)` applies the morphological operation specified by the string or character vector `operation` to the binary volume `V`. `bwmorph3` returns the results of the operation in logical volume `J`, which has the same dimensions as the input volume.

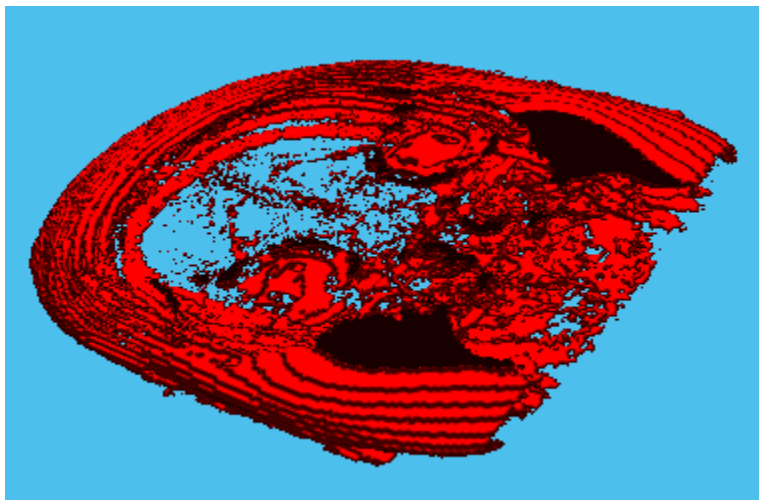
### **Examples**

#### **Compare the Clean and Majority Operations of `bwmorph3`**

Load 3-D MRI volumetric data and create a binary volume. Use the Volume Viewer app to view the volumetric data.

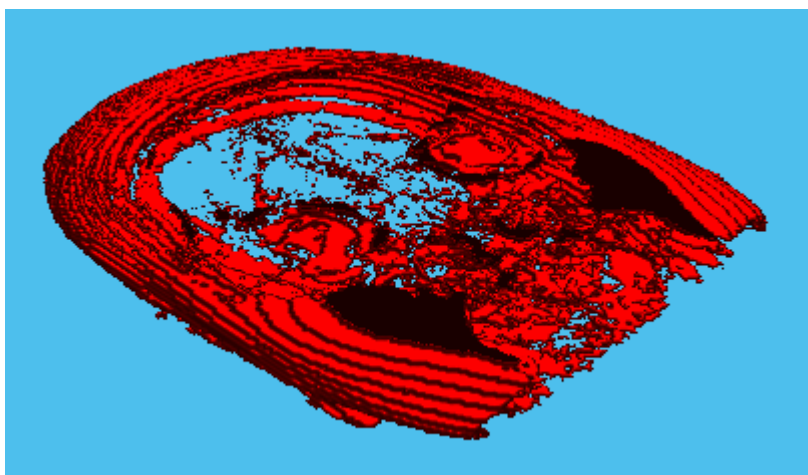
```
load mrystack;  
BW1 = mrystack > 127;
```





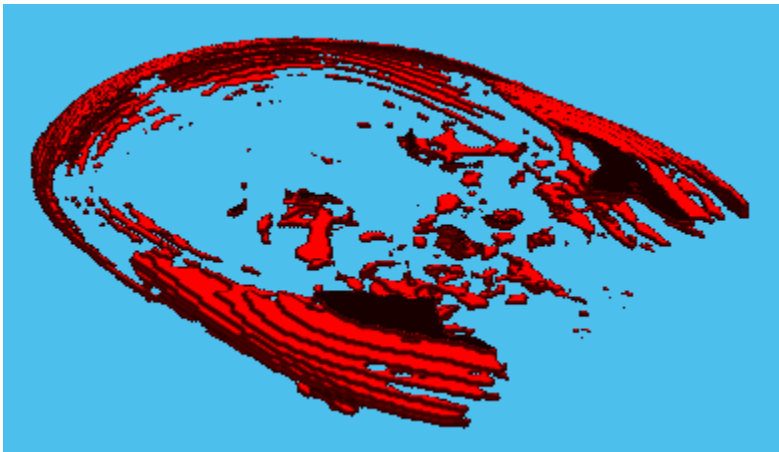
To remove voxels that are set to 1 and that are also surrounded by voxels set to 0, perform the 'clean' operation on the volumetric data. When determining which voxels to remove, the 'clean' operation considers 26 neighboring voxels. You can view the results using the Volume Viewer app.

```
BW2 = bwmorph3(BW1, 'clean');
```



For comparison, perform the 'majority' operation on the volumetric data. The 'majority' operation performs a similar task to the 'clean' operation but only retains voxels if more than half (the majority) of the voxels in the neighborhood of the target voxel are set to 1. When determining which voxels to retain, the 'majority' operation also considers 26 neighboring voxels. You can view the results using the Volume Viewer app.

```
BW3 = bwmorph3(BW1, 'majority');
```



### Illustrations of Morphological Operations

This example shows how each of the morphological operations supported by `bwmorph3` works on simple volumes.

Make a 9-by-9-by-3 cuboid of 0s that contains a 3-by-3-by-3 cube of 1s at its center.

```
innercube = ones(3,3,3);  
cube_center = padarray(innercube, [3 3], 0, 'both')
```

```
cube_center =  
cube_center(:,:,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 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

```

```
cube_center(:,:,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
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

```

```
cube_center(:,:,3) =
```

```

0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
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

```

### Turning Pixels Off with the Remove Operation

Set the center voxel of the inner cube to 0 using the 'remove' operation. This operation sets the value of any 'on' voxel completely surrounded by 'on' voxels to 'off'.

```
remove_center = bwmorph3(cube_center, 'remove')
```

```
remove_center = 9x9x3 logical array
remove_center(:,:,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 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
```

```
remove_center(:,:,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
0 0 0 1 1 1 0 0 0
0 0 0 1 0 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
```

```
remove_center(:,:,3) =
```

```
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
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
```

## Setting Pixels to On with the Fill Operation

Set the center voxel of the inner cube to 1 using the 'fill' operation. This operation sets the value of any 'off' voxel completely surrounded by 'on' voxels to 'on'.

```
fill_center = bwmorph3(remove_center, 'fill')
```

```
fill_center = 9x9x3 logical array
fill_center(:,:,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 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

```

```
fill_center(:,:,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
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

```

```
fill_center(:,:,3) =
```

```

0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
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

```

### Removing Unconnected Pixels with the Clean Operation

Use the 'clean' operation to remove any stray voxels that are set to 1 but are not connected to a component in the volume. The example creates a stray voxel by setting a random voxel on the second plane to 1 and then uses the 'clean' operation to remove it.

```
cube_center(2,2,2) = 1
```

```
cube_center =  
cube_center(:,:,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   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
```

```
cube_center(:,:,2) =
```

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

```
cube_center(:,:,3) =
```

```
  0   0   0   0   0   0   0   0   0  
  0   0   0   0   0   0   0   0   0  
  0   0   0   0   0   0   0   0   0  
  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
```

```
cube_cleaned = bwmorph3(cube_center, 'clean')
```

```
cube_cleaned = 9x9x3 logical array
```

```
cube_cleaned(:,:,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 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

```

```
cube_cleaned(:,:,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
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

```

```
cube_cleaned(:,:,3) =
```

```

0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
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

```

### Finding the Majority

Find the majority of the `cube_center` using the 'majority' operation. This operation retains a voxel only if more than half (the majority) of the voxels in the 26-connected neighborhood around the voxel are set to 1.

```
cube_major = bwmorph3(cube_center, 'majority')
```

```
cube_major = 9x9x3 logical array
cube_major(:,:,1) =
```

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

```
cube_major(:,:,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
0 0 0 0 1 0 0 0 0
0 0 0 1 1 1 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```

```
cube_major(:,:,3) =
```

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

## Creating a Shape Similar to a Skeleton

To illustrate the branch points and end points options, create another small matrix, this time with a linear shape, like a skeleton.



```
x1 = eye(5);
x2 = zeros(5);
x2(3,3) = 1;
x3 = x2;
shape = cat(3,x1,x2,x3)
```

```
shape =
shape(:,:,1) =
```

```
    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
```

```
shape(:,:,2) =
```

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

```
shape(:,:,3) =
```

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

### Finding End Points

Find the end points of the shape using the 'endpoints' operation. The shape has three end points, one at each end of the diagonal in the first plane and one at the end of the line through the center, on the third plane.

```
shape_endpts = bwmorph3(shape, 'endpoints')
```

```
shape_endpts = 5x5x3 logical array
shape_endpts(:,:,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  1
```

```
shape_endpts(:, :, 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
```

```
shape_endpts(:, :, 3) =
```

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

## Finding Branch Points

Find the branch points of the shape using the 'branchpoints' operation. The shape has a single branch point, where the diagonal line and the horizontal line meet.

```
shape_brpts = bwmorph3(shape, 'branchpoints')
```

```
shape_brpts = 5x5x3 logical array
```

```
shape_brpts(:, :, 1) =
```

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

```
shape_brpts(:, :, 2) =
```

```
0  0  0  0  0
```

```

0  0  0  0  0
0  0  1  0  0
0  0  0  0  0
0  0  0  0  0

```

```
shape_brpts(:, :, 3) =
```

```

0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0

```

## Input Arguments

### V — Input volume

real, nonsparse, numeric or logical array

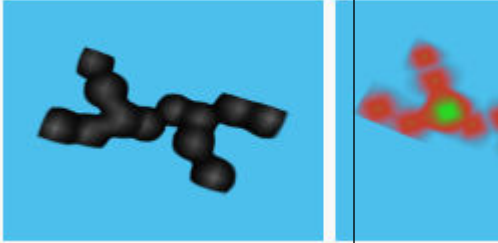
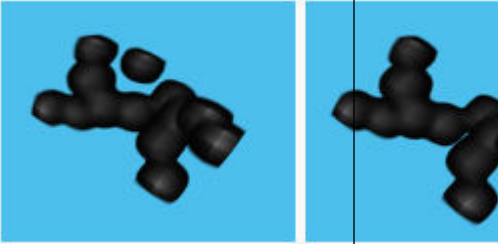
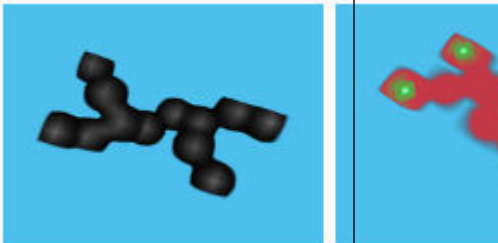
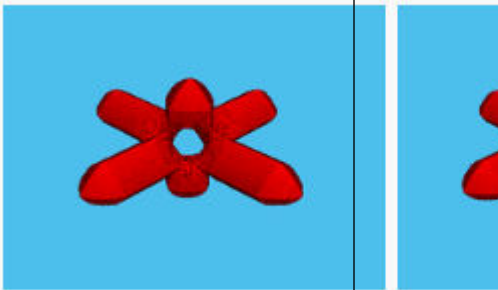
Input volume, specified as a real, nonsparse, numeric or logical array. `bwmorph3` accepts 1-D, 2-D, or 3-D arrays. If you specify 1-D or 2-D input arrays, `bwmorph3` performs the morphological operation as defined for a 3-D volume. If you want 2-D behavior, use `bwmorph` instead.

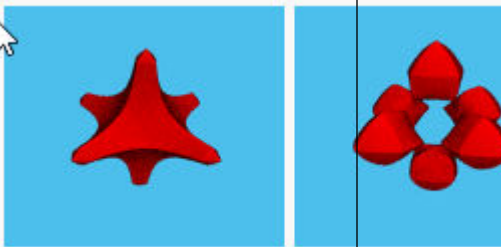
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### operation — Morphological operation to perform

character vector | string scalar

Morphological operation to perform, specified as one of the following character vectors or string scalar. For examples of these operations, see “Illustrations of Morphological Operations” on page 1-228.

Operation	Description	Illustration
'branchpoints'	<p>Find branch points of skeleton. Branch points are the voxels at the junction where multiple branches meet.</p> <p>To find branch points, the image must be skeletonized. To create a skeletonized image, use <code>bwskel</code>.</p>	
'clean'	<p>Remove isolated voxels, setting them to 0. An isolated voxel is an individual, 26-connected voxel that is set to 1 that are surrounded by voxels set to 0.</p>	
'endpoints'	<p>Find end points of skeleton. End points are voxels at the ends of branches.</p> <p>Note: To find end points, the image must be skeletonized. To create a skeletonized image, use <code>bwskel</code>.</p>	
'fill'	<p>Fill isolated interior voxels, setting them to 1. Isolated interior voxels are individual voxels that are set to 0 that are surrounded (6-connected) by voxels set to 1.</p>	

Operation	Description	Illustration
'majority'	Keep a voxel set to 1 if 14 or more voxels (the majority) in its 3-by-3-by-3, 26-connected neighborhood are set to 1; otherwise, set the voxel to 0.	See "Illustrations of Morphological Operations" on page 1-228.
'remove'	Remove interior voxels, setting it to 0. Interior voxels are individual voxels that are set to 1 that are surrounded (6-connected) by voxels set to 1.	

Data Types: char | string

## Output Arguments

### J — Output volume

logical array

Output volume, returned as a logical array.

## Tips

- To perform the morphological operations erosion or dilation on 3-D volumes, use the `imerode` or `imdilate` functions, specifying the structuring element `ones(3,3,3)`.
- To perform morphological closing, opening, top-hat filtering, or bottom-hat filtering on 3-D volumes, use the `imclose`, `imopen`, `imtophat`, or `imbothat` functions, specifying the structuring element `ones(3,3,3)`.

## See Also

`bwmorph` | `bwskel` | `imbothat` | `imclose` | `imdilate` | `imerode` | `imopen` | `imtophat`

## **Topics**

“Morphological Dilation and Erosion”

**Introduced in R2018a**

# bwpack

Pack binary image

## Syntax

```
BWP = bwpack(BW)
```

## Description

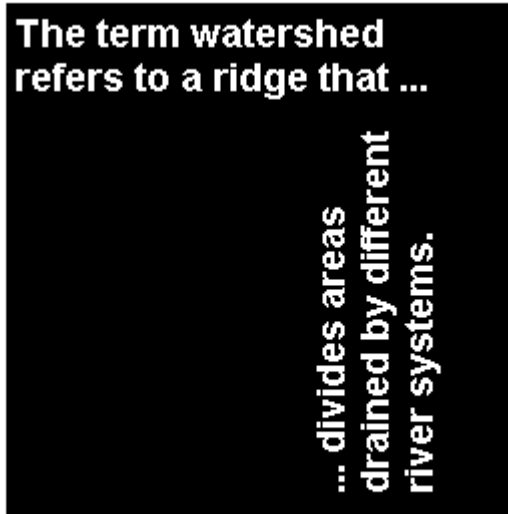
`BWP = bwpack(BW)` packs the binary image `BW` into the `uint32` array `BWP`, which is known as a *packed binary image*. Because each pixel value in the binary image has only two possible values, 1 and 0, `bwpack` can map each pixel to a single bit in the packed output image.

## 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)
```





## Input Arguments

### **BW** — Binary image

2-D numeric matrix | 2-D logical matrix

Binary image, specified as a 2-D numeric or logical matrix. For numeric input, any nonzero pixels are considered to be 1 (`true`).

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

## Output Arguments

### **BWP** — Packed binary image

numeric matrix

Packed binary image, returned as a numeric matrix of type `uint32`.

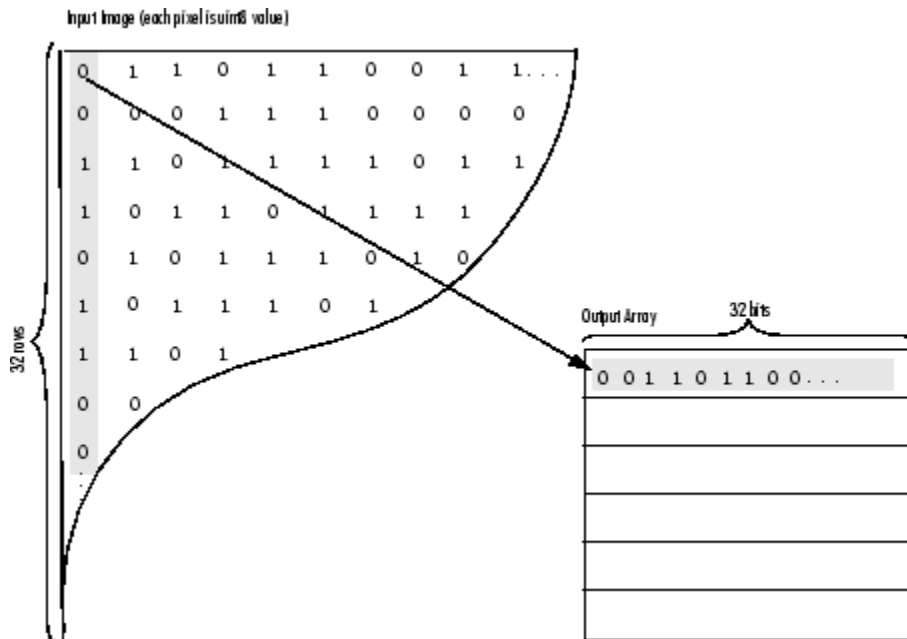
Data Types: `uint32`

### Tips

- 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, then the function uses a specialized routine to perform the operation faster.
- Use `bwunpack` to unpack packed binary images.

### Algorithms

`bwpack` processes the input 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.



## 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 “Understand Code Generation with Image Processing Toolbox”.

### 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.

`BW2 = bwperim(BW,conn)` specifies the pixel connectivity, `conn`.

## Examples

### Find Perimeter of Objects in Binary Image

Read binary image into workspace.

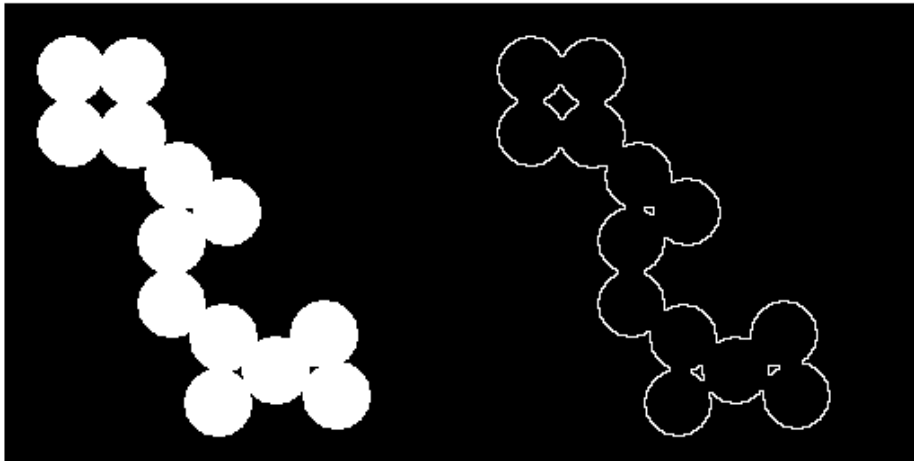
```
BW = imread('circles.png');
```

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

2-D numeric matrix | 2-D logical matrix

Input binary image, specified as a 2-D numeric or logical matrix.

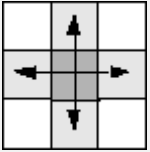
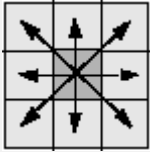
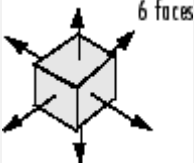
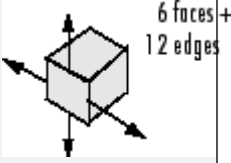
Example: `BW = imread('circles.png');`

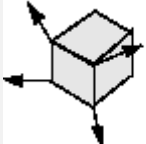
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### **conn** — Pixel connectivity

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 4 for 2-D images, and 6 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal, vertical, or diagonal direction.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch. Two adjoining pixels are part of the same object if they are both on and are connected in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	
18-connected	Pixels are connected if their faces or edges touch. Two adjoining pixels are part of the same object if they are both on and are connected in <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	

Value	Meaning	
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected in</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	 <p>6 faces + 12 edges + 8 corners</p>

For higher dimensions, `bwperim` uses the default value `conndef(ndims(BW), 'minimal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

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 “Understand 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

`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` whose value of property `attrib` is in the specified range. `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)` specifies whether to keep the `n` largest objects or the `n` smallest objects when sorted by property `attrib`.

`BW2 = bwpropfilt(BW,I,attrib,___)` sorts objects based on the intensity values in the grayscale image `I` and the property `attrib`.

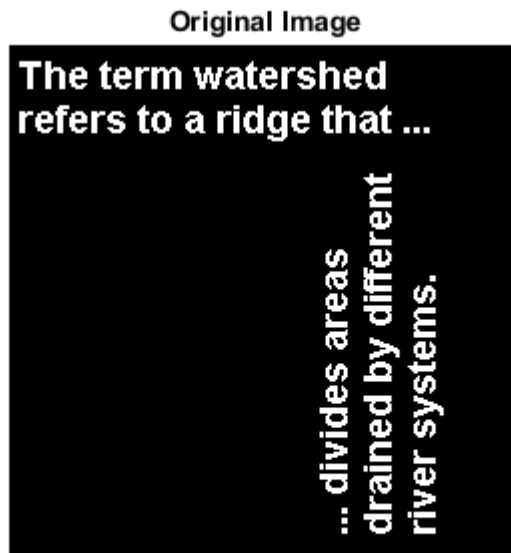
`BW2 = bwpropfilt(BW,___,conn)` specifies the pixel connectivity, `conn`.

### **Examples**

## Find Regions Without Holes

Read image and display it.

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

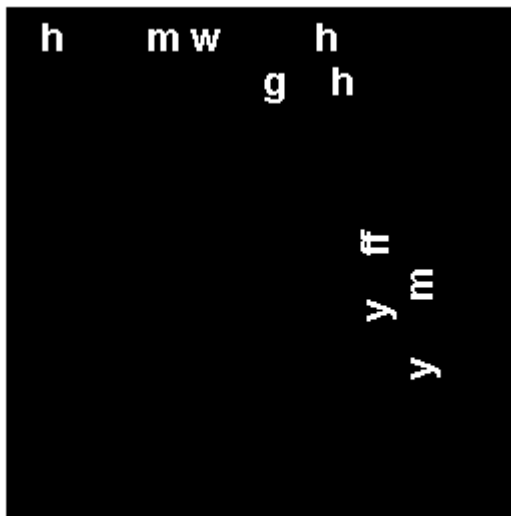
Read image.

```
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



## 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 | string scalar

Name of attribute on which to filter, specified as one of the following values. 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, then `attrib` can have one of these additional values.

MaxIntensity	MeanIntensity	MinIntensity
--------------	---------------	--------------

Data Types: `char` | `string`

### **range** — Minimum and maximum property values

2-by-1 numeric vector

Minimum and maximum property values, 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 return

positive integer

Number of object to return, specified as a positive integer.

Data Types: `double`

### **keep** — Objects to retain

'largest' (default) | 'smallest'

Objects to retain, specified as 'largest' or 'smallest'.

Data Types: `char` | `string`

### **I** — Marker image

grayscale image

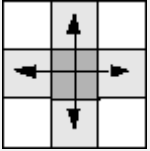
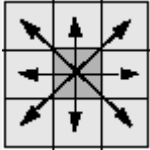
Marker image, specified as a grayscale image, the same size as the input binary image. Intensity values in the grayscale image define regions in the input binary image.

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

**conn — Pixel connectivity**

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

Pixel connectivity, specified as one of these values.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal, vertical, or diagonal direction.	

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

Data Types: `double` | `logical`

## Output Arguments

**BW2 — Filtered image**

binary image

Filtered image, returned as a binary image the same size as `BW`.

## 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.

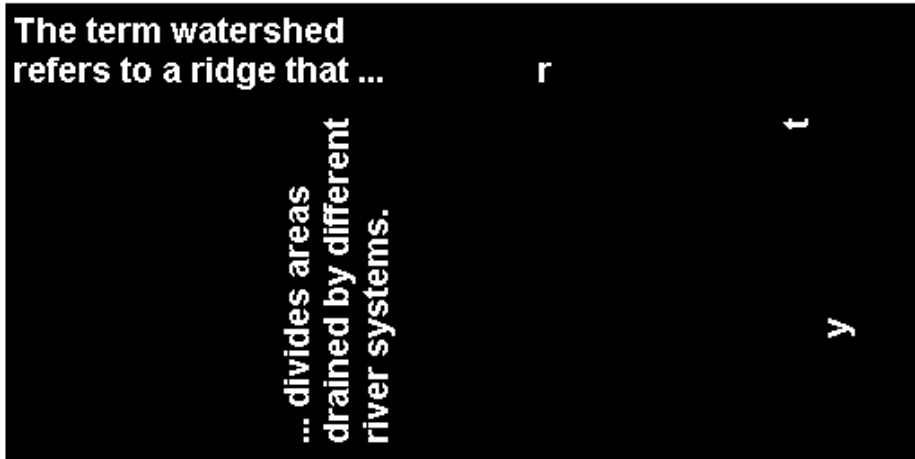
```
c = [43 185 212];  
r = [38 68 181];
```

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
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 “Understand 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

`bwlabel` | `bwselect3` | `grayconnected` | `imfill` | `regionfill` | `roipoly`

**Introduced before R2006a**

## bwselect3

Select objects in binary image

### Syntax

```
J = bwselect3(V,C,R,P)
J = bwselect3(X,Y,Z,V,Xi,Yi,Zi)
[J] = bwselect3( ____,N)
[J,idx] = bwselect3( ____)
[X,Y,Z,J,Xi,Yi,Zi] = bwselect3( ____)
[X,Y,Z,J,idx,Xi,Yi,Zi] = bwselect3( ____)
```

### Description

`J = bwselect3(V,C,R,P)` returns the binary volume `J` containing the objects that overlap the pixel location  $(R,C,P)$ . `R`, `C`, and `P` are scalars or equal-length vectors that specify the row, column, and plane index of the pixel location. Objects are connected sets of pixels with the value 1.

If you specify `R`, `C`, and `P` as vectors, `J` contains the set of objects overlapping with any of the pixels  $(R(k),C(k),P(k))$ , where  $k$  is an index into the vector.

`J = bwselect3(X,Y,Z,V,Xi,Yi,Zi)` uses the vectors `X`, `Y`, and `Z` to establish a nondefault spatial coordinate system for `V`. `Xi`, `Yi`, and `Zi` are scalars or equal-length vectors that specify pixel locations in this coordinate system.

`[J] = bwselect3( ____,N)` returns a binary volume where `N` specifies the connectivity used to define objects.

`[J,idx] = bwselect3( ____)` returns `idx`, a column vector of linear indices specifying the pixels belonging to the selected objects.

`[X,Y,Z,J,Xi,Yi,Zi] = bwselect3( ____)` returns the binary volume `J`, along with the `XData`, `YData`, and `ZData` of the output volume in `X`, `Y`, and `Z`. `Xi`, `Yi`, and `Zi` contain the specified spatial coordinates.

`[X,Y,Z,J,idx,Xi,Yi,Zi] = bwselect3( ___ )` returns the binary volume J, along with `idx`, a column vector of linear indices specifying the pixels belonging to the selected objects.

## Examples

### Find Objects in Volume

Load a volume and change its name to V.

```
load mrystack;  
V = mrystack;
```

Define a set of points in the volume.

```
C = [126 87 11];  
R = [34 120 20];  
P = [20 2 12];
```

Return a volume that contains objects that intersect with the points specified.

```
J = bwselect3(V,C,R,P);
```

## Input Arguments

### V — Input volume

nonsparse, logical or numeric 3-D array

Input volume, specified as a nonsparse, 3-D, logical or numeric array.

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

### R — Row index of object

numeric scalar | numeric vector

Row index of object, specified as a numeric scalar or vector. If you specify a vector, R must be the same length as C and P.



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

### **C — Column index of object**

numeric scalar | numeric vector

Column index of object, specified as a scalar or vector. If you specify a vector, C must be the same length as R and P.

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

### **P — Plane index of object**

numeric scalar | numeric vector

Plane index of object, specified as a scalar or vector. If you specify a vector, P must be the same length as R and C.

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

### **N — Connectivity**

26 (default) | 6 | 18

Connectivity, specified as 6, 18, or 26. Objects are connected sets of pixels with the value 1.

#### **Connectivities**

<b>Value</b>	<b>Connectivity</b>
6	6-connected objects (Face-Face)
18	18-connected objects (Face-Face and Edge-Edge)
26	26-connected objects (Face-Face, Edge-Edge, and Vertex-Vertex)

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

### **X — Limits of nondefault coordinate system in X direction**

vector

Limits of nondefault coordinate system in X direction, specified as a vector.

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

## **Y — Limits of nondefault coordinate system in Y direction**

vector

Limits of nondefault coordinate system in Y direction, specified as a vector.

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

## **Z — Limits of nondefault coordinate system in Z direction**

vector

Limits of nondefault coordinate system in Z direction, specified as a vector.

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

## **Xi — X-coordinate of location in nondefault coordinate system**

scalar | vector

X-coordinate of location in nondefault coordinate system, specified as a scalar or vector.

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

## **Yi — Y-coordinate of location in nondefault coordinate system**

scalar | vector

Y-coordinate of location in nondefault coordinate system, specified as a scalar or vector.

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

## **Zi — Z-coordinate of location in nondefault coordinate system**

scalar or vector

Z-coordinate of location in nondefault coordinate system, specified as a scalar or vector.

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

## Output Arguments

### **J — Output volume**

*N*-D logical array

Output volume, returned as an *N*-D logical array. **J** contains the set of objects overlapping with any of the pixels specified by **R**, **C**, and **P**, or **Xi**, **Yi**, and **Zi**.

### **idx — Linear indices of pixels belonging to selected objects**

vector

Linear indices of pixels belonging to the selected objects, returned as a vector.

### **X — Volume Xdata property**

vector

Volume Xdata property, returned as a vector.

### **Y — Volume Ydata property**

vector

Volume Ydata property, returned as a vector.

### **Z — Volume Zdata property**

vector

Volume Zdata property, returned as a vector.

### **Xi — X-coordinate of location in nondefault coordinate system**

scalar | vector

X-coordinate of location in nondefault coordinate system, returned as a scalar or vector.

### **Yi — Y-coordinate of location in nondefault coordinate system**

scalar | vector

Y-coordinate of location in nondefault coordinate system, returned as a scalar or vector.

### **Zi — Z-coordinate of location in nondefault coordinate system**

scalar | vector

Z-coordinate of location in nondefault coordinate system, returned as a scalar or vector.

## **See Also**

`bwlabel` | `bwselect` | `imfill` | `regionfill` | `roipoly`

**Introduced in R2017b**

# bwskel

Reduce all objects to lines in 2-D binary image or 3-D binary volume

## Syntax

```
B = bwskel(A)
B = bwskel(V)
B = bwskel( ____, 'MinBranchLength', N)
```

## Description

`B = bwskel(A)` reduces all objects in the 2-D binary image `A` to 1-pixel wide curved lines, without changing the essential structure of the image. This process, called skeletonization, extracts the centerline, preserving the topology and Euler number of the objects, which correspond to white regions in the image (logical true).

`B = bwskel(V)` returns the skeleton of a 3-D binary volume.

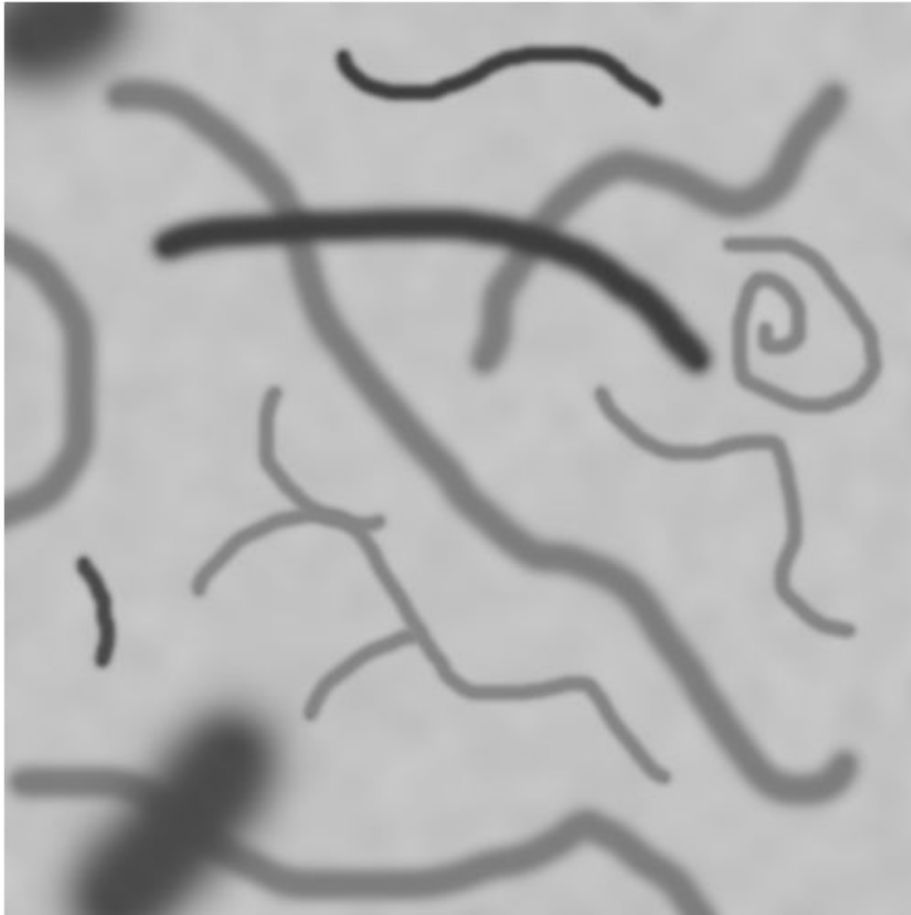
`B = bwskel( ____, 'MinBranchLength', N)` specifies the minimum branch length `N` of the skeleton. `bwskel` removes (prunes) all branches shorter than the specified length. `bwskel` calculates the length as the number of pixels in a branch using 8-connectivity for 2-D and 26-connectivity for 3-D. The default value of `'MinBranchLength'` is 0, which equates to no pruning.

## Examples

### Skeletonization of 2-D Grayscale Image

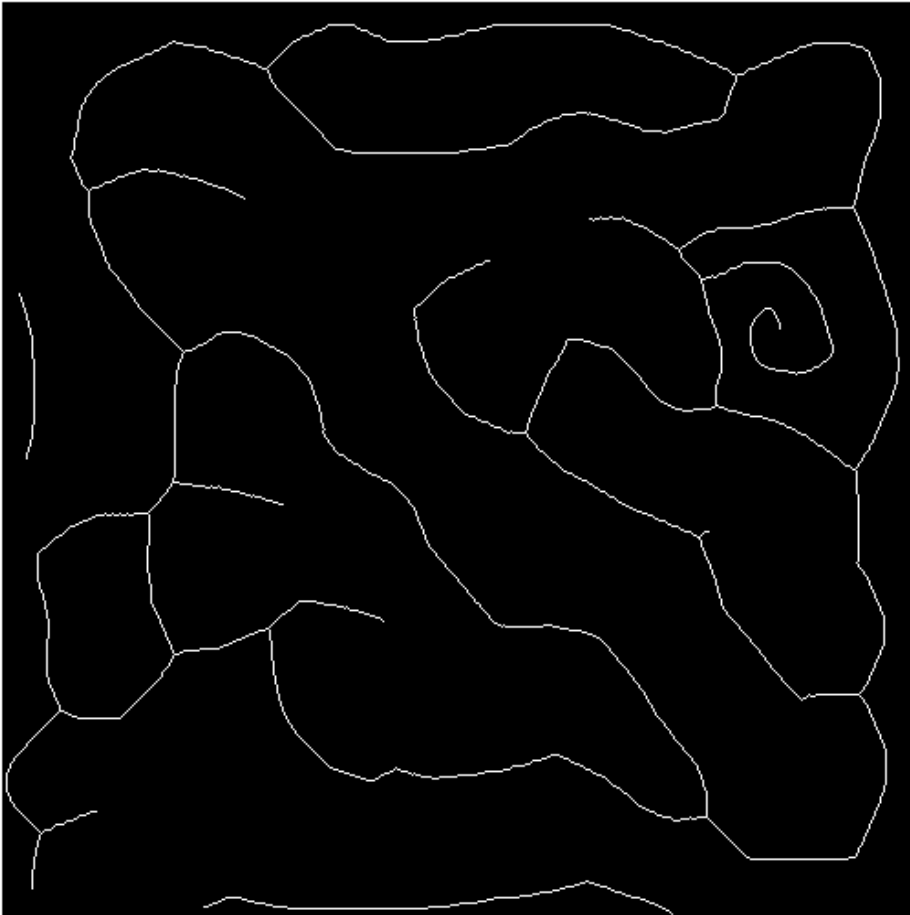
Read a 2-D grayscale image into the workspace and display it.

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



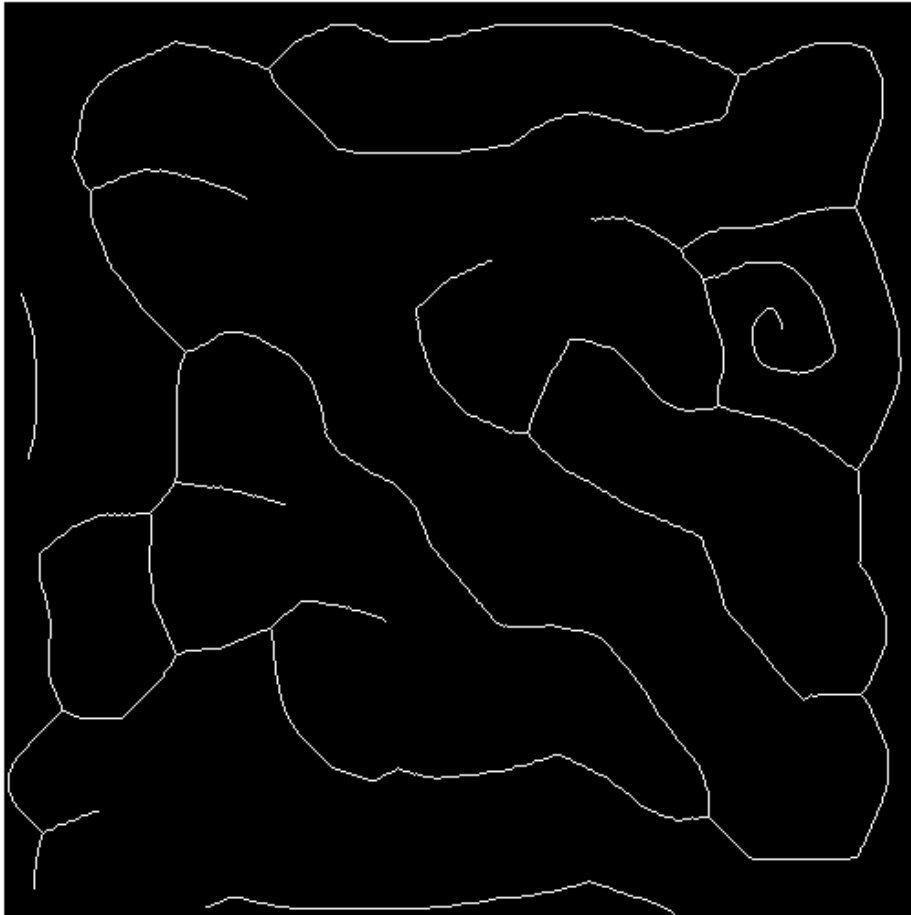
Perform skeletonization of the image using `bwskel` and view the result.

```
out = bwskel(imbinarize(I));  
figure  
imshow(out)
```



Prune small spurs that appear on the skeleton and view the result.

```
out2 = bwskel(imbinarize(I), 'MinBranchLength', 10);  
figure  
imshow(out2)
```



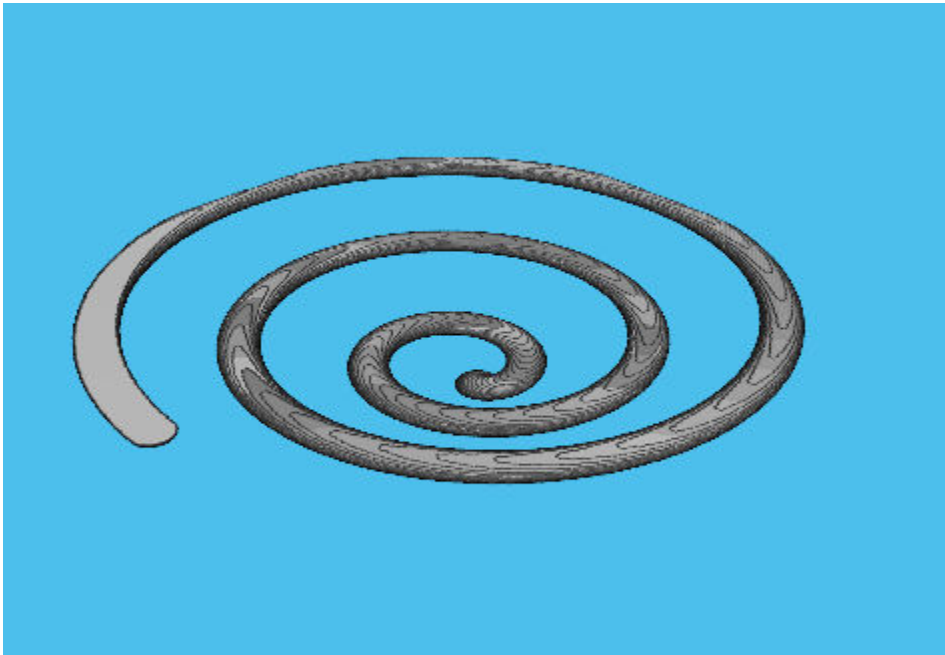
### **Skeletonize 3-D Volume**

Load a volumetric data set into the workspace. The name of the data set is `spiralVol`.



```
load spiralVol.mat;
```

You can view this data set using the Volume Viewer, `volumeViewer(spiralVol)`.



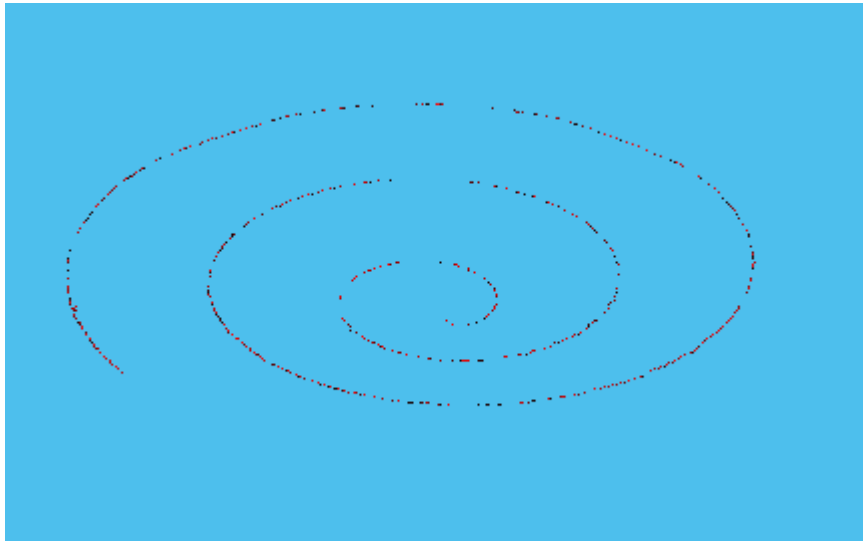
Convert the `spiralVol` data set to a binary format which is required by the `bwskel` function.

```
spiralVolLogical = imbinarize(spiralVol);
```

Skeletonize the spiral shape in the data set.

```
spiralVolSkel = bwskel(spiralVolLogical);
```

Use the Volume Viewer to view the skeletonized volume, `volumeViewer(spiralVolSkel)`.



## Input Arguments

### **A — Input binary image**

2-D logical array

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

Data Types: `logical`

### **V — Input 3-D binary volume**

nonsparse, 3-D logical array

Input binary volume, specified as a nonsparse, 3-D logical array.

Data Types: `logical`

## Output Arguments

### **B — Output image or volume**

2-D logical array | 3-D logical array

Output image or volume, returned as a 2-D or 3-D logical array, the same size as the input image or volume.

## Tips

- While both `bwskel` and `bwmorph` can skeletonize 2-D images, you might get different results using `bwmorph` than when using `bwskel`. Because they use different algorithms, the `bwskel` function uses 4-connectivity with 2-D images; `bwmorph` uses 8-connectivity.

## Algorithms

- The `bwskel` function uses the medial axis transform.

## References

- [1] Ta-Chih Lee, Rangasami L. Kashyap and Chong-Nam Chu *Building skeleton models via 3-D medial surface/axis thinning algorithms*. Computer Vision, Graphics, and Image Processing, 56(6):462-478, 1994.
- [2] Kerschnitzki, M, Kollmannsberger, P, Burghammer, M. et al., *Architecture of the osteocyte network correlates with bone material quality*. Journal of Bone and Mineral Research, 28(8):1837-1845, 2013.

## See Also

`bwmorph` | `bwmorph3`

## Topics

“Skeletonization”

**Introduced in R2018a**

## bwtraceboundary

Trace object in binary image

### Syntax

```
B = bwtraceboundary(BW,P,fstep)
B = bwtraceboundary(BW,P,fstep,conn)
B = bwtraceboundary(BW,P,fstep,conn,m,dir)
```

### Description

`B = bwtraceboundary(BW,P,fstep)` traces the outline of an object in binary image `BW`. Nonzero pixels belong to an object and zero-valued pixels constitute the background. `P` specifies the row and column coordinates of the point on the object boundary where you want the tracing to begin. `fstep` specifies the initial search direction for the next object pixel connected to `P`. `B` holds the row and column coordinates of the boundary pixels for the region.

`B = bwtraceboundary(BW,P,fstep,conn)` traces the boundary, where `conn` specifies the desired connectivity.

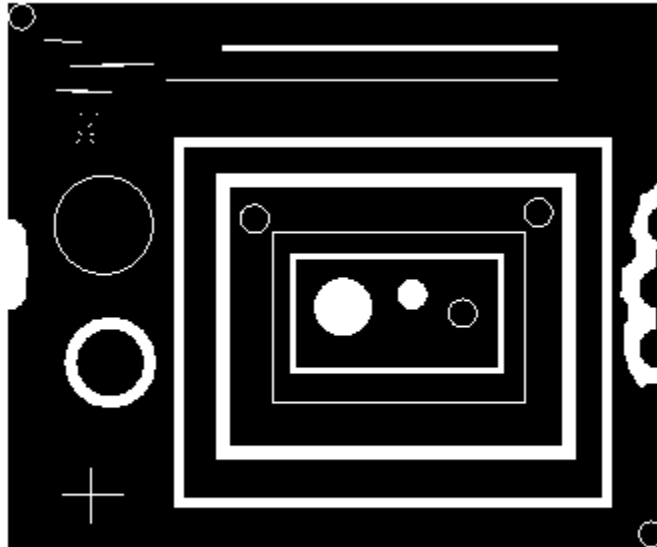
`B = bwtraceboundary(BW,P,fstep,conn,m,dir)` specifies `m`, the maximum number of boundary pixels to extract, and `dir`, the direction in which to trace the boundary. By default, `bwtraceboundary` identifies all the pixels on the boundary.

### Examples

#### Trace Boundary and Visualize Contours

Read image and display it.

```
BW = imread('blobs.png');
imshow(BW,[]);
```

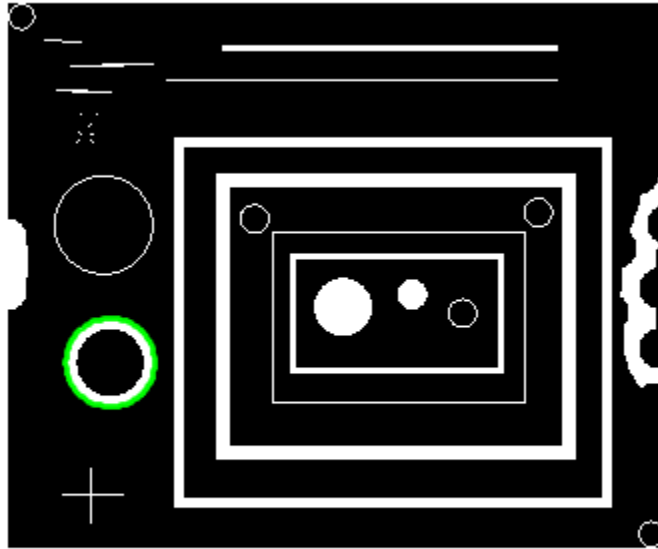


Pick an object in the image and trace the boundary. To select an object, specify a pixel on its boundary. This example uses the coordinates of a pixel on the boundary of the thick white circle, obtained through visual inspection using `impixelinfo`. The example specifies the initial search direction, 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);
```



## Input Arguments

### **BW** — Binary image

2-D numeric matrix | 2-D logical matrix

Binary image, specified as a 2-D numeric or logical matrix.

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

### **P** — Coordinates of starting point

2-element vector

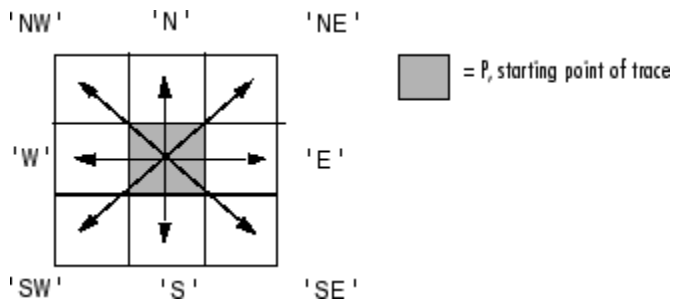
Coordinates of starting point on the object boundary where you want the tracing to begin, specified as a 2-element vector of the format `[row column]`.

Data Types: double

**fstep — Initial search direction**

'N' | 'NE' | 'E' | 'SE' | 'S' | 'SW' | 'W' | 'NW'

Initial search direction for the next object pixel connected to P, specified as a character vector or string scalar as depicted in the diagram.



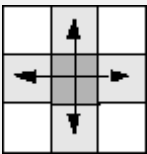
**Note** When the connectivity conn is 4, fstep is limited to the values 'N', 'E', 'S', and 'W'.

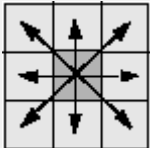
Data Types: char | string

**conn — Pixel connectivity**

8 (default) | 4

Pixel connectivity, specified as 8 or 4.

Value	Meaning
<b>Two-Dimensional Connectivities</b>	
4-connected	Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction. 

Value	Meaning
8-connected	<p>Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.</p> 

Data Types: double

**m — Maximum number of boundary pixels to extract**

Inf (default) | positive integer

Maximum number of boundary pixels to extract, specified as a positive integer. By default, m is Inf and `bwtraceboundary` identifies all the pixels on the boundary.

Data Types: double

**dir — Direction in which to trace boundary**

'clockwise' (default) | 'counterclockwise'

Direction in which to trace boundary, specified as 'clockwise' or 'counterclockwise'.

Data Types: char | string

## Output Arguments

**B — Row and column coordinates of boundary pixels**

q-by-2 matrix

Row and column coordinates of the boundary pixels for the region, returned as a q-by-2 matrix. Each row in B has the form [ row column ].

## Algorithms

The `bwtraceboundary` 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. For more information, see “Code Generation for Image Processing”.
- When generating code, the `dir`, `fstep`, and `conn` arguments must be compile-time constants.

### See Also

`bwboundaries` | `bwperim`

**Introduced before R2006a**

## **bwulterode**

Ultimate erosion

### **Syntax**

```
BW2 = bwulterode(BW)
BW2 = bwulterode(BW,method)
BW2 = bwulterode( ____,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`.

`BW2 = bwulterode(BW,method)` specifies the distance transform method.

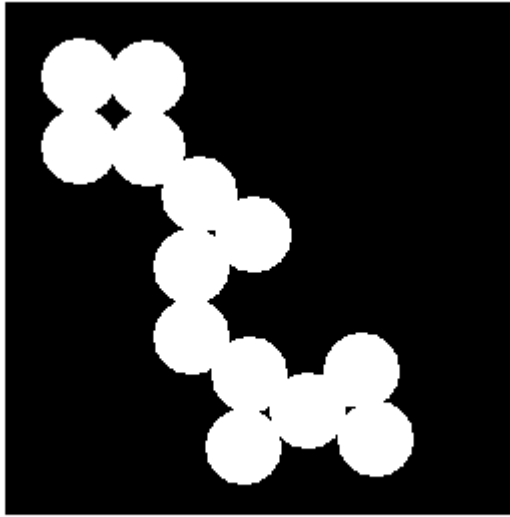
`BW2 = bwulterode( ____,conn)` specifies the pixel connectivity.

### **Examples**

#### **Perform Ultimate Erosion of Binary Image**

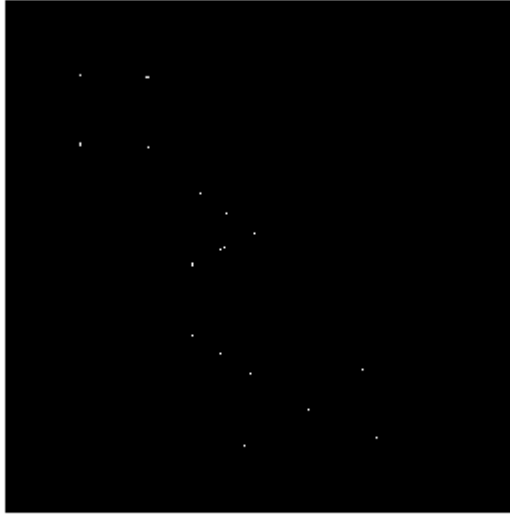
Read a binary image into the workspace and display it.

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



Perform the ultimate erosion of the image and display it.

```
ultimateErosion = bwulterode(originalBW);  
figure, imshow(ultimateErosion)
```



## Input Arguments

### **BW** — Binary image

numeric array | logical array

Binary image, specified as a numeric or logical array of any dimension. For numeric input, any nonzero pixels are considered to be 1 (true).

Example: `BW = imread('circles.png');`

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

### **method** — Distance transform method

'euclidean' (default) | 'quasi-euclidean' | 'cityblock' | 'chessboard'

Distance transform method, specified as one of the values in this table.

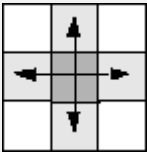
Method	Description
'chessboard'	In 2-D, the chessboard distance between $(x_1, y_1)$ and $(x_2, y_2)$ is $\max( x_1 - x_2 ,  y_1 - y_2 ).$
'cityblock'	In 2-D, the cityblock distance between $(x_1, y_1)$ and $(x_2, y_2)$ is $ x_1 - x_2  +  y_1 - y_2 $
'euclidean'	In 2-D, the Euclidean distance between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

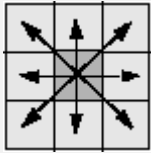
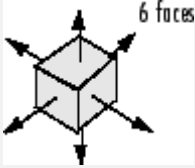
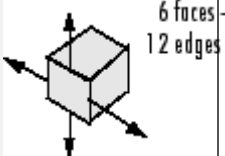
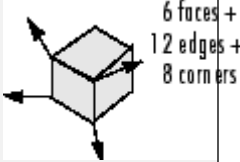
For more information, see “Distance Transform of a Binary Image”.

### conn — Pixel connectivity

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning
<b>Two-Dimensional Connectivities</b>	
4-connected	Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction. 

Value	Meaning	
8-connected	Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	
18-connected	Pixels are connected if their faces or edges touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	
26-connected	Pixels are connected if their faces, edges, or corners touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	

For higher dimensions, `bwulterode` uses the default value `conndef(ndims(BW), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

Data Types: `double` | `logical`

## Output Arguments

### **BW2 — Eroded image**

logical array

Eroded image, returned as a logical array of the same size as `BW`.

Data Types: `logical`

## See Also

`bwdist` | `conndef` | `imregionalmax`

## Topics

“Distance Transform of a Binary Image”

**Introduced before R2006a**

## **bwunpack**

Unpack binary image

### **Syntax**

```
BW = bwunpack(BWP,m)
```

### **Description**

`BW = bwunpack(BWP,m)` unpacks the packed binary image `BWP` into binary image `BW` with `m` rows.

### **Examples**

#### **Pack, Dilate, and Unpack Binary Image**

Read binary image into the workspace.

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



**The term watershed  
refers to a ridge that ...**

**... divides areas  
drained by different  
river systems.**

Pack the image.

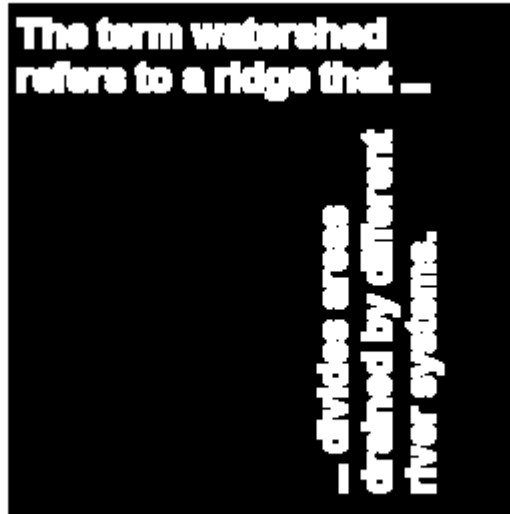
```
BWp = bwpack(BW);
```

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)
```



## Input Arguments

### **BWP — Packed binary image**

2-D numeric matrix

Packed binary image, specified as a 2-D numeric array of data type `uint32`.

Data Types: `uint32`

### **m — Number of image rows**

positive integer

Number of image rows, specified as a positive integer. The default value of `m` is `32*size(BWP,1)`.

Data Types: `uint32`

## Output Arguments

### **BW — Unpacked binary image**

m-by-n logical matrix

Unpacked binary image, returned as a logical matrix with m rows.

Data Types: `logical`

## Algorithms

When `bwunpack` unpacks `BWP`, the function 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.

## 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 “Understand 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.

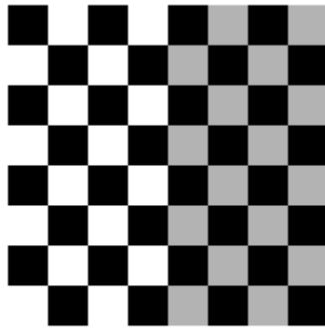
`I = checkerboard(n,p,q)` creates a rectangular checkerboard image where `p` specifies the number of rows of tiles and `q` specifies the number of columns of tiles. If you omit `q`, the number of columns defaults to `p` and the checkerboard is square. Each square has `n` pixels per side.

## 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)
```



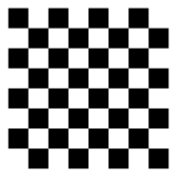
### Create Black and White Checkerboard

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  $2 \times p$  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  $2 \times q$  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

`fitgeotrans` | `imwarp`

**Introduced before R2006a**

## chromadapt

Adjust color balance of RGB image with chromatic adaptation

### Syntax

```
B = chromadapt(A,illuminant)
B = chromadapt(A,illuminant,Name,Value)
```

### Description

`B = chromadapt(A,illuminant)` adjusts the color balance of sRGB image `A` according to the scene illuminant. The illuminant must be in the same color space as the input image.

`B = chromadapt(A,illuminant,Name,Value)` adjusts the color balance of `A` using name-value pairs to control additional options.

### Examples

#### Color Balance Image by Specifying Gray Pixel

Read an image with a strong yellow color cast. Display the image, specifying an optional magnification to shrink the size of the displayed image.

```
A = imread('hallway.jpg');
figure
imshow(A,'InitialMagnification',25)
title('Original Image')
```

Original Image



Pick a pixel in the image that should look white or gray, such as a point on a pillar. Do not pick a saturated pixel, such as a point on the ceiling light.

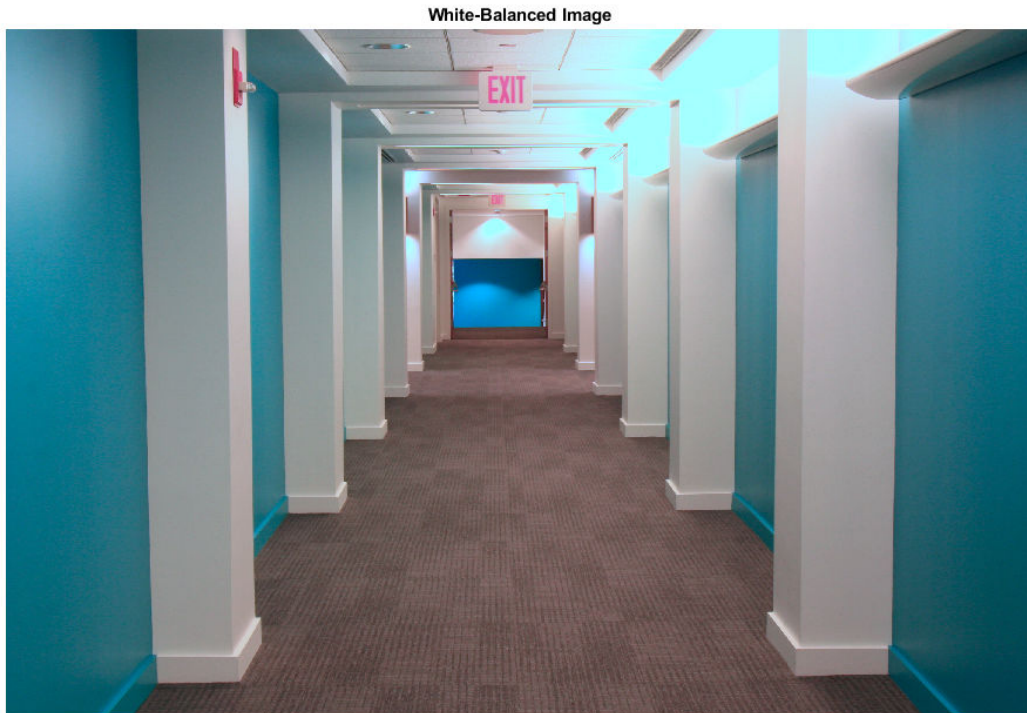
```
x = 2800;  
y = 1000;  
gray_val = [A(y,x,1) A(y,x,2) A(y,x,3)];
```

Use the selected color as reference for the scene illumination, and correct the white balance of the image.

```
B = chromadapt(A,gray_val);
```

Display the corrected image, setting an optional initial magnification.

```
figure
imshow(B, 'InitialMagnification', 25)
title('White-Balanced Image')
```



The pillars are now white as expected, and the rest of the image has no yellow tint.

### Color Balance Image in Linear RGB Color Space

Open an image file containing minimally processed linear RGB intensities.

```
A = imread('foosballraw.tiff');
```

The image data is the raw sensor data after correcting the black level and scaling to 16 bits per pixel. Interpolate the intensities to reconstruct color. The color filter array pattern is RGGB.

```
A = demosaic(A, 'rggb');
```

Display the image. Because the image is in linear RGB color space, apply gamma correction so the image appears correctly on the screen. To shrink the image so that it appears fully on the screen, set the optional initial magnification to a value less than 100

```
A_sRGB = lin2rgb(A);  
figure  
imshow(A_sRGB, 'InitialMagnification', 25)  
title('Original Image')
```

Original Image



The image has a ColorChecker chart in the scene. To get the color of the ambient light, pick a pixel on one of the neutral patches of the chart.

```
x = 1510;  
y = 1250;  
light_color = [A(y,x,1) A(y,x,2) A(y,x,3)]
```

```
light_color = 1x3 uint16 row vector
```

```
    7361    14968    10258
```

The intensity of the red channel is lower than the intensity of the other two channels, which indicates the light is bluish green.

Balance the color channels of the image. Use the 'ColorSpace' option to specify that the image and the illuminant are expressed in linear RGB.

```
B = chromadapt(A,light_color,'ColorSpace','linear-rgb');
```

Display the corrected image, applying gamma correction and setting the initial magnification.

```
B_sRGB = lin2rgb(B);  
figure  
imshow(B_sRGB,'InitialMagnification',25)  
title('White-Balanced Image')
```

White-Balanced Image



Confirm that the gray patch has been color balanced.

```
patch_color = [B(y,x,1) B(y,x,2) B(y,x,3)]
```

```
patch_color = 1x3 uint16 row vector
```

```
13010 13010 13010
```

The three color channels in the color-balanced gray patch have similar intensities, as expected.

## Input Arguments

### **A** — Input RGB image

real, nonsparse, *m*-by-*n*-by-3 array

Input RGB image, specified as a real, nonsparse, *m*-by-*n*-by-3 array.

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

### **illuminant** — Scene illuminant

real, nonempty, 3-element vector

Scene illuminant, specified as a real, nonempty, 3-element vector. The illuminant must be in the same color space as the input image, **A**.

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 quotes. You can specify several name and value pair arguments in any order as **Name1**, **Value1**, . . . , **NameN**, **ValueN**.

Example: `I2 = chromadapt(I,uint8([22 97 118]),'ColorSpace','linear-rgb')` adjusts the color balance of an image, **I**, in linear RGB color space.

### **ColorSpace** — Color space

'srgb' (default) | 'adobe-rgb-1998' | 'linear-rgb'

Color space of the input image and illuminant, specified as the comma-separated pair consisting of 'ColorSpace' and 'srgb', 'adobe-rgb-1998', or 'linear-rgb'. Use the 'linear-rgb' option to adjust the color balance of an RGB image whose intensities are linear.

Data Types: `char` | `string`

### **Method** — Chromatic adaptation method

'bradford' (default) | 'vonkries' | 'simple'

Chromatic adaptation method used to scale the RGB values in **A**, specified as the comma-separated pair consisting of 'Method' and one of:



- 'bradford'—Scale using the Bradford cone response model
- 'vonkries'—Scale using the von Kries cone response model
- 'simple'—Scale using the illuminant

Data Types: `char` | `string`

## Output Arguments

### **B** — Color-balanced RGB image

*m*-by-*n*-by-3 array

Color-balanced RGB image, returned as an *m*-by-*n*-by-3 array. B has the same data type as A.

## References

- [1] Lindbloom, Bruce. Chromatic Adaptation. [http://www.brucelindbloom.com/index.html?Eqn\\_ChromAdapt.html](http://www.brucelindbloom.com/index.html?Eqn_ChromAdapt.html).

## See Also

`colorangle` | `illumgray` | `illumpca` | `illumwhite` | `whitepoint`

**Introduced in R2017b**

## col2im

Rearrange matrix columns into blocks

### Syntax

```
A = col2im(B,[m n],[M N])  
A = col2im(B,[m n],[M N],'sliding')  
A = col2im(B,[m n],[M N],'distinct')
```

### Description

`A = col2im(B,[m n],[M N])` or

`A = col2im(B,[m n],[M N],'sliding')` rearranges the row vector `B` into neighborhoods of size `m-by-n` to create the matrix `A` of size `(M-m+1)-by-(N-n+1)`.

The row vector `B` is usually the result of processing the output of `im2col(...,'sliding')` using a column compression function, such as `sum`.

`A = col2im(B,[m n],[M N],'distinct')` rearranges each column of matrix `B` into a distinct `m-by-n` block to create the matrix `A` of size `M-by-N`.

For example, if `B` consists of column vectors `Bi(:)` with length `m*n`, arranged as `B = [B1(:) B2(:) B3(:) B4(:)]`, then `A = [B1 B3; B2 B4]` where each block `Bi` has size `m-by-n`.

### Examples

#### Rearrange Matrix Values into Row-wise Orientation

Create a matrix.

```
B = reshape(uint8(1:25),[5 5])'
```

```
B = 5x5 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 = 5x5 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 = 5x5 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
```

## Input Arguments

### **B — Image blocks**

matrix | row vector

Image blocks, specified as one of the following.

- For distinct block processing,  $B$  is a numeric or logical matrix with  $m*n$  rows. Each column corresponds to one block.
- For sliding neighborhood processing,  $B$  is a numeric or logical row vector of size  $1\text{-by-}(M-m+1)*(N-n+1)$ .

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

### **[m n] — Block size**

2-element vector of positive integers

Block size, specified as a 2-element vector of positive integers.  $m$  is the number of rows and  $n$  is the number of columns in each block.  $m*n$  must be equal to the number of rows of  $B$ .

Data Types: `double`

### **[M N] — Image size**

2-element vector of positive integers

Image size, specified as a 2-element vector of positive integers.  $M$  is the number of rows and  $N$  is the number of columns in the image.

Data Types: `double`

## Output Arguments

### **A — Reconstructed image**

numeric matrix

Reconstructed image, returned as a numeric matrix of size  $M\text{-by-}N$  for distinct block processing, or  $(M-m+1)\text{-by-}(N-n+1)$  for sliding block processing.  $A$  has the same data type as  $B$ .

## See Also

`blockproc` | `colfilt` | `im2col` | `nlfilter` | `reshape`

**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. `colfilt` zero-pads `A`, if necessary.

`B = colfilt(A,[m n],[mblock nblock],block_type,fun)` subdivides `A` into regions of size `m`-by-`n` blocks to save memory. Note that the result of the operation does not change when using the `[mblock nblock]` argument.

For example, if `[mblock nblock]` is `[3 4]` and the size of each block is 16-by-16 pixels, then `colfilt` subdivides the image into regions of size 48-by-64 pixels and processes each region separately.

`B = colfilt(A,'indexed',___)` processes `A` as an indexed image, padding with 0s if the class of `A` is `uint8`, `uint16`, or `logical`, and padding with 1s otherwise.

## 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**

## Input Arguments

### **A — Image**

array

Image, specified as an array of any class supported by `fun`.

### **[m n] — Block size**

2-element vector of positive integers

Block size, specified as a 2-element vector of positive integers. `m` is the number of rows and `n` is the number of columns in each block.

### **[mblock nblock] — Block group size**

2-element vector of positive integers

Block group size, specified as a 2-element vector of positive integers. `mblock` is the number of blocks in the group in the vertical direction, and `nblock` is the number of blocks in the group in the horizontal direction.

**block\_type — Block type**

'sliding' | 'distinct'

Block type, specified as 'sliding' for sliding neighborhoods or 'distinct' for distinct blocks.

Data Types: char | string

**fun — Function handle**

handle

Function handle, specified as a handle. The input and output arguments to this function depend on the value of `block_type`. For more information, see “Algorithms” on page 1-314.

For more information about function handles, see “Create Function Handle” (MATLAB).

## Output Arguments

**B — Filtered image**

numeric matrix

Filtered image, returned as a numeric matrix.

## Algorithms

The algorithm that `colfilt` uses to process images depends on the value of `block_type`.

Value	Description
'distinct'	<ul style="list-style-type: none"><li>• First, <code>colfilt</code> rearranges each <math>m</math>-by-<math>n</math> block of <math>A</math> into a column in a temporary matrix by using the <code>im2col</code> function.</li><li>• Next, <code>colfilt</code> applies the function <code>fun</code> to this temporary matrix. <code>fun</code> must return a matrix the same size as the temporary matrix.</li><li>• Finally, <code>colfilt</code> rearranges the columns of the matrix returned by <code>fun</code> into <math>m</math>-by-<math>n</math> distinct blocks, by using the <code>col2im</code> function.</li></ul>



Value	Description
'sliding'	<ul style="list-style-type: none"> <li>• First, <code>colfilt</code> rearranges each <math>m</math>-by-<math>n</math> neighborhood of <math>A</math> into a column in a temporary matrix by using the <code>im2col</code> function.</li> <li>• Next, <code>colfilt</code> applies the function <code>fun</code> to this temporary matrix. <code>fun</code> must return a row vector containing a single value for each column in the temporary matrix. (Column compression functions such as <code>sum</code> return the appropriate type of output.)</li> <li>• Finally, <code>colfilt</code> reshapes the vector returned by <code>fun</code> into a matrix the same size as <math>A</math>, by using the <code>reshape</code> function.</li> </ul>

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.

## See Also

`blockproc` | `col2im` | `im2col` | `nlfilter` | `reshape`

## Topics

“Use Column-wise Processing to Speed Up Sliding Neighborhood or Distinct Block Operations”

“Border Padding Behavior in Sliding Neighborhood Operations”

“Anonymous Functions” (MATLAB)

“Parameterizing Functions” (MATLAB)

“Create Function Handle” (MATLAB)

**Introduced before R2006a**

## colorangle

Angle between two RGB vectors

### Syntax

```
angle = colorangle(rgb1,rgb2)
```

### Description

`angle = colorangle(rgb1,rgb2)` computes the angle in degrees between two RGB vectors.

### Examples

#### Compare Accuracy of Illuminant Estimation Algorithms

Open a test image. The image is the raw data captured with a Canon EOS 30D digital camera after correcting the black level and scaling the intensities to 16 bits per pixel. No demosaicing, white balancing, color enhancement, noise filtering, or gamma correction has been applied.

```
A = imread('foosballraw.tiff');
```

Interpolate using the `demosaic` function to obtain a color image. The color filter array pattern is RGGB.

```
A_demosaiced = demosaic(A,'rggb');
```

The image contains a ColorChecker chart. Specify the ground truth illuminant, which was calculated in advance using the neutral patches of the chart.

```
illuminant_groundtruth = [0.0717 0.1472 0.0975];
```

To avoid skewing the estimation of the illuminant, exclude the ColorChecker chart by creating a mask.

```
mask = true(size(A_demosaiced,1), size(A_demosaiced,2));  
mask(920:1330,1360:1900) = false;
```

Run three different illuminant estimation algorithms: `illumwhite`, `illumgray`, and `illumpca`.

```
illuminant_whitepatch = illumwhite(A_demosaiced, 'Mask', mask);  
illuminant_grayworld = illumgray(A_demosaiced, 'Mask', mask);  
illuminant_pca = illumpca(A_demosaiced, 'Mask', mask);
```

Compare each estimation against the ground truth by calculating the angle between each estimated illuminant and the ground truth using the `colorangle` function. The smaller the angle, the better the estimation. The magnitude of the estimation does not matter because only the direction of the illuminant is used to white-balance an image with chromatic adaptation.

```
angle_whitepatch = colorangle(illuminant_whitepatch, illuminant_groundtruth)
```

```
angle_whitepatch = 5.0921
```

```
angle_grayworld = colorangle(illuminant_grayworld, illuminant_groundtruth)
```

```
angle_grayworld = 5.1036
```

```
angle_pca = colorangle(illuminant_pca, illuminant_groundtruth)
```

```
angle_pca = 5.0134
```

The value of `angle_pca` is smallest, indicating that the PCA illuminant estimation algorithm is closest to the ground truth illumination for this image.

## Input Arguments

### **rgb1** — First RGB vector

3-element numeric vector

First RGB vector, specified as a 3-element numeric vector.

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

### **rgb2** — Second RGB vector

3-element numeric vector

Second RGB vector, specified as a 3-element numeric vector.

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

## Output Arguments

### **angle** — Angle between RGB vectors

numeric scalar

Angle between RGB vectors, returned as a numeric scalar.

Data Types: `double`

## Definitions

### Angular Error

Angular error is a useful metric to evaluate the estimation of an illuminant against the ground truth. The smaller the angle between the ground truth illuminant and the estimated illuminant, the better the estimate.

## See Also

`chromadapt` | `illumgray` | `illumpca` | `illumwhite` | `whitepoint`

**Introduced in R2017b**

# 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

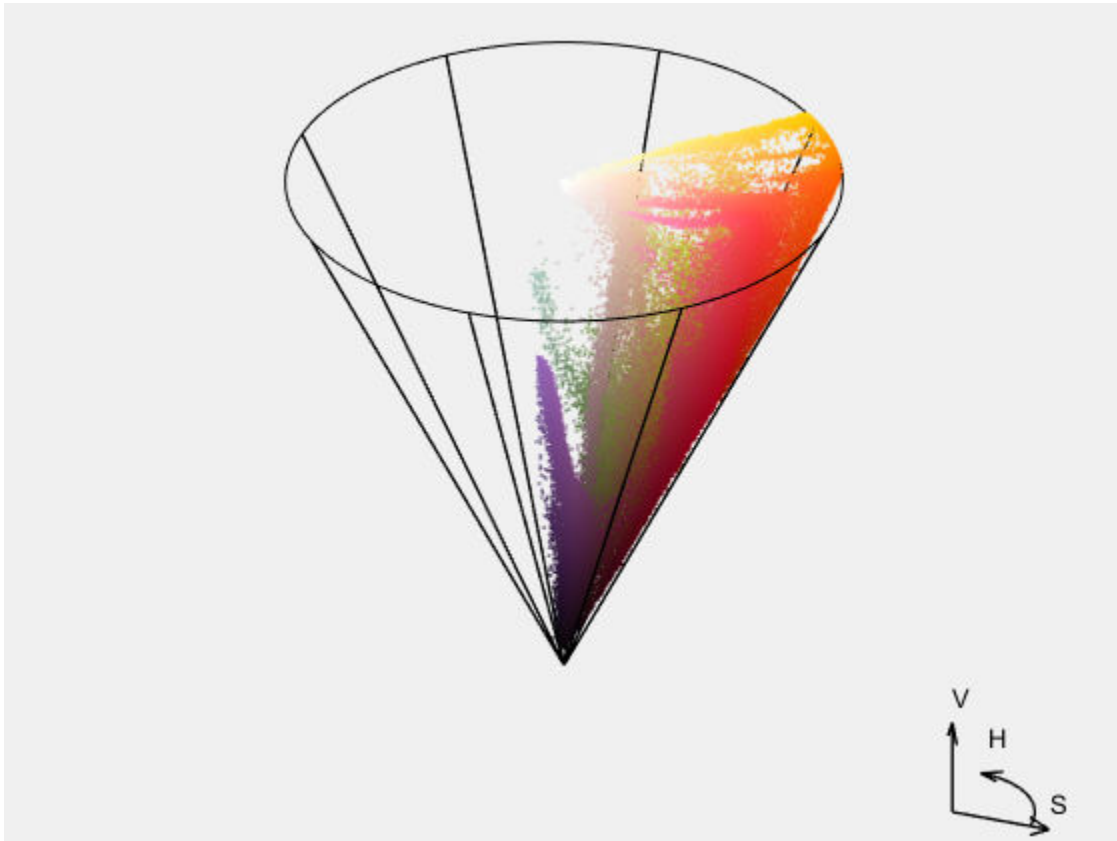
### View 3D Color Gamut of RGB Image in HSV Color Space

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 | string

## 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 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

### **hPanel — 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 pixel 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 = 3×3
```

```
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 = 3×3
```

```
 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 =
```

```
ans(:,:,1) =
```

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

```
ans(:,:,2) =
```

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

```
ans(:,:,3) =
```

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

## Input Arguments

**num\_dims** — Number of dimensions

positive integer

Number of dimensions, specified as a positive integer.

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 `'minimal'` or `'maximal'`

Value	Description
<code>'minimal'</code>	Defines a neighborhood whose neighbors are touching the central element on an (N-1)-dimensional surface, for the N-dimensional case.
<code>'maximal'</code>	Defines a neighborhood including neighbors that touch the central element in any way; it is <code>ones ( repmat ( 3 , 1 , NUM_DIMS ) )</code> .

Data Types: `char` | `string`

## Output Arguments

### **conn** — Pixel connectivity

3-by-3-by...-3 logical array

Pixel connectivity, returned as a 3-by-3-...-by-3 logical array. `conn` is symmetric about its center element. See “Specifying Custom Connectivities” for more information.

## 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**

---

# contains

Determine if image contains points in world coordinate system

## Syntax

```
TF = contains(R,xWorld,yWorld)
TF = contains(R,xWorld,yWorld,zWorld)
```

## Description

`TF = contains(R,xWorld,yWorld)` returns a logical array `TF`. Each element `TF(k)` is true if and only if the corresponding point `(xWorld(k),yWorld(k))` falls within the bounds of an image associated with 2-D spatial referencing object `R`.

`TF = contains(R,xWorld,yWorld,zWorld)` indicates whether each point falls within the bounds of an image associated with 3-D spatial referencing object `R`.

## Examples

### Check If Coordinates Fall Within 2-D Image Bounds

Read a 2-D image into the workspace.

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

Create an `imref2d` spatial referencing object associated with the image.

```
R = imref2d(size(I))
```

```
R =  
  imref2d with 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]
```

Check if certain world coordinates are in the image.

```
res = contains(R,[5 8 8],[5 10 257])
```

```
res = 1x3 logical array
```

```
    1    1    0
```

This result indicates that the points (5,5) and (8,10) are within the image bounds, and that the point (8, 257) is outside the image bounds. This conclusion is consistent with the `XWorldLimits` and `YWorldLimits` properties of the spatial referencing object `R`.

### Check If Coordinates Fall Within 3-D Image Bounds

Read a 3-D image into the workspace. This image consists of 27 frames of 128-by-128 pixel images.

```
load mri;
D = squeeze(D);
```

Create an `imref3d` spatial referencing object associated with the image.

```
R = imref3d(size(D))
```

```
R =
  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]

```

Check if certain 3-D world coordinates are in the image.

```
res = contains(R,[5 6 6 8],[5 10 10 257],[1 27.5 28 1])
```

*res = 1x4 logical array*

```
1 1 0 0
```

This result indicates that the points (5,5,1) and (6,10,27.5) are within the image bounds. The points (6,10,28) and (8,257,1) are outside the image bounds. This conclusion is consistent with the `XWorldLimits`, `YWorldLimits`, and `ZWorldLimits` properties of the spatial referencing object `R`.

## Input Arguments

### **R** — Spatial referencing object

`imref2d` or `imref3d` object

Spatial referencing object, specified as an `imref2d` or `imref3d` object. `R` is associated with an image.

### **xWorld** — Coordinates along the x-dimension in the world coordinate system

numeric scalar or vector

Coordinates along the x-dimension in the world coordinate system, specified as a numeric scalar or vector.

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

### **yWorld** — Coordinates along the y-dimension in the world coordinate system

numeric scalar or vector

Coordinates along the y-dimension in the world coordinate system, specified as a numeric scalar or vector. `yWorld` is the same length as `xWorld`.

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

**`zWorld` — Coordinates along the z-dimension in the world coordinate system**  
numeric scalar or vector

Coordinates along the z-dimension in the world coordinate system, specified as a numeric scalar or vector. `zWorld` is the same length as `xWorld` and `yWorld`.

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

## Output Arguments

**TF — Flag indicating whether coordinates exist within the bounds of the image**  
logical scalar or vector

Flag indicating whether coordinates exist within the bounds of the image, returned as a logical scalar or vector. TF is the same length as the input coordinate vectors `xWorld`, `yWorld`, and (when relevant) `zWorld`.

Data Types: `logical`

## See Also

`imref2d` | `imref3d`

**Introduced in R2013a**



# 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.

## Examples

### Create a Convolution Matrix

Show that, for 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)`

Description of first code block

```
H = ones(3,3)/9; % averaging filter 3-by-3
M = 5;
X = magic(M);
T = convmtx2(H,M,M);
Y1 = reshape(T*X(:), size(H)+[5 5]-1)
```

`Y1 = 7×7`

1.8889	4.5556	4.6667	3.6667	2.6667	2.5556	1.6667
4.4444	7.6667	8.5556	6.5556	6.7778	5.8889	3.4444

```
4.8889  8.7778  11.1111  10.8889  12.8889  10.5556  5.8889
4.1111  6.6667  11.0000  13.0000  15.0000  10.6667  4.5556
2.7778  6.7778  13.1111  15.1111  14.8889  8.5556  3.7778
2.3333  5.6667  10.5556  10.7778  8.7778  3.8889  1.3333
1.2222  3.2222  6.0000  5.0000  4.0000  1.2222  1.0000
```

```
Y2 = conv2(X,H)
```

```
Y2 = 7×7
```

```
1.8889  4.5556  4.6667  3.6667  2.6667  2.5556  1.6667
4.4444  7.6667  8.5556  6.5556  6.7778  5.8889  3.4444
4.8889  8.7778  11.1111  10.8889  12.8889  10.5556  5.8889
4.1111  6.6667  11.0000  13.0000  15.0000  10.6667  4.5556
2.7778  6.7778  13.1111  15.1111  14.8889  8.5556  3.7778
2.3333  5.6667  10.5556  10.7778  8.7778  3.8889  1.3333
1.2222  3.2222  6.0000  5.0000  4.0000  1.2222  1.0000
```

```
isequal(Y1,Y2) % They are the same.
```

```
ans = logical
      0
```

## Input Arguments

### **H** — Input matrix

numeric array

Input matrix, specified as a numeric array.

Data Types: double

### **m** — Rows in convolution matrix

numeric scalar

Rows in convolution matrix, specified as a numeric scalar.

Data Types: double

**n — Columns in convolution matrix**

numeric scalar

Columns in convolution matrix, specified as a numeric scalar.

Data Types: double

**[m n] — Dimensions of convolution matrix**

numeric scalar

Dimensions of convolution matrix, specified as a two-element vector of the form [m n], where m is the number of rows and n is the number of columns.

Data Types: double

## Output Arguments

**T — Convolution matrix**

numeric array

Convolution matrix, returned as a numeric array. The output matrix T is of class `sparse`. The number of nonzero elements in T is no larger than `prod(size(H))*m*n`.

## See Also

`conv2` | `convmtx`

**Introduced before R2006a**

## corner

Find corner points in image

---

**Note** `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 their coordinates 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.

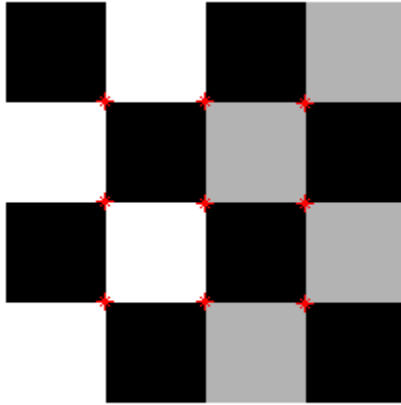
`C = corner( ____,Name,Value)` specifies parameters and corresponding values that control various aspects of the corner detection algorithm.

## Examples

### Find Corner Points in Checkerboard Image

```
I = checkerboard(50,2,2);
C = corner(I);
```

```
imshow(I)
hold on
plot(C(:,1),C(:,2), 'r*');
```



## Input Arguments

### **I** — Grayscale or binary image

*m*-by-*n* numeric matrix

Grayscale or binary image, specified as an *m*-by-*n* numeric matrix.

### **method** — Corner detection algorithm

'Harris' (default) | 'MinimumEigenvalue'

Corner detection method, specified as 'Harris' for the Harris corner detector, or 'MinimumEigenvalue' for Shi & Tomasi's minimum eigenvalue method.

### **N** — Maximum number of corners

200 (default) | positive integer

Maximum number of corners that the corner function can return, specified as a positive integer.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `corner(I, 'QualityLevel', 0.2)` specifies the minimum quality level of corners in image `I` as `0.2`.

### **FilterCoefficients** — Filter coefficients

numeric vector

Filter coefficients for the separable smoothing filter, specified as the comma-separated pair consisting of `'FilterCoefficients'` and a numeric vector. The vector, `V`, must have odd length and a minimum length of 3. The outer product, `V*V'`, gives the full filter kernel. The default filter coefficients are given by `fspecial('gaussian', [5 1], 1.5)`.

### **QualityLevel** — Minimum accepted quality

0.01 (default) | numeric scalar

Minimum accepted quality of corners, specified as the comma-separated pair consisting of `'QualityLevel'` and a numeric scalar in the range (0, 1). For a quality level `Q`, the toolbox rejects candidate corners with corner metric values less than  $Q * \max(\text{corner metric})$ . Use larger values of `Q` to remove erroneous corners.

### **SensitivityFactor** — Sensitivity factor

0.04 (default) | numeric scalar

Sensitivity factor used in the Harris detection algorithm, specified as the comma-separated pair consisting of `'SensitivityFactor'` and a numeric scalar in the range (0, 0.25). The smaller the sensitivity factor, the more likely the algorithm is to detect sharp corners. Use this parameter with the `'Harris'` method only.

## Output Arguments

### **C** — Coordinates of corner points

*p*-by-2 matrix

*x* and *y* coordinates of the corner points detected in image `I`, returned as a *p*-by-2 matrix.

Data Types: double

## 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

---

**Note** `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. The outer product, <code>V*V'</code> , 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> .



Parameter	Description
'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');
```

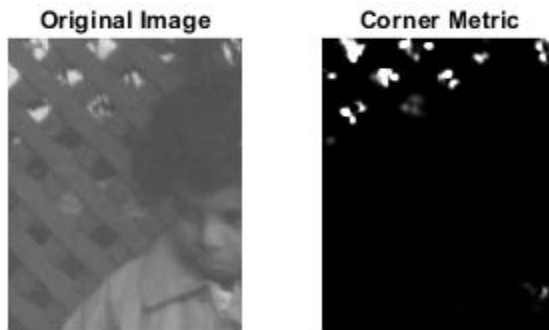


Generate a corner metric matrix.

```
C = cornermetric(I);
```

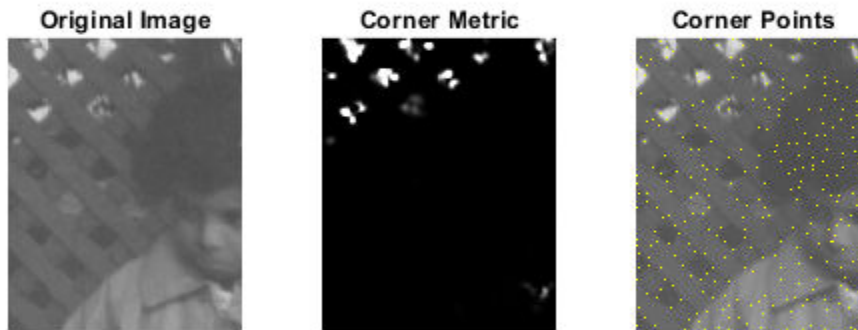
Adjust the corner metric for viewing.

```
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');
```



## 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

`edge`

**Introduced in R2008b**

## corr2

2-D correlation coefficient

### Syntax

```
R = corr2(A,B)
```

### Description

`R = corr2(A,B)` returns the 2-D correlation coefficient `R` between arrays `A` and `B`.

You optionally can compute the correlation coefficient using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

### 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
```

## Input Arguments

### A — First input array

numeric array | logical array | gpuArray

First input array, specified as a numeric or logical array.

To perform the computation using a GPU, specify A as a `gpuArray` that contains a numeric or logical array.

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

### B — Second input array

numeric array | logical array | gpuArray

Second input array, specified as a numeric or logical array. B has the same size as the first input array, A.

To perform the computation using a GPU, specify B as a `gpuArray` that contains a numeric or logical array.

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

## Output Arguments

### R — Correlation coefficient

numeric scalar | gpuArray

Correlation coefficient, returned as a numeric scalar.

If the correlation coefficient is computed using a GPU, then R is returned as a `gpuArray` containing a numeric scalar.

Data Types: `double`

## 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  $\bar{B} = \text{mean2}(B)$ .

## Extended Capabilities

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

### See Also

`corrcoef` | `std2`

**Introduced before R2006a**



## cp2tform

Infer spatial transformation from control point pairs

---

**Note** 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.

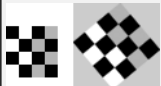
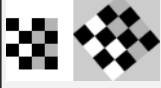
### **fixedPoints**




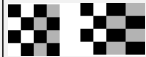

*m*-by-2, double matrix containing the *x*- and *y*-coordinates of control points in the fixed image.

### **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	

Transformation Type	Description	Minimum Number of Control Point Pairs	Example
'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 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,

$$[u \ v] = [x \ y \ 1] * T_{inv}$$

$T_{inv}$  is a 3-by-2 matrix. Solve for the six elements of  $T_{inv}$ :

```
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,

$$[up \ vp \ wp] = [x \ y \ w] * T_{inv}$$

where

$$u = up/wp$$

$$v = vp/wp$$

$T_{inv}$  is a 3-by-3 matrix.

Assuming

$$T_{inv} = \begin{bmatrix} A & D & G; \\ B & E & H; \\ C & F & I \end{bmatrix};$$
$$u = (Ax + By + C)/(Gx + Hy + I)$$
$$v = (Dx + Ey + F)/(Gx + Hy + I)$$

Solve for the nine elements of  $T_{inv}$ :

```
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,

$$[u \ v] = [1 \ x \ y \ x*y \ x^2 \ y^2] * Tinv$$

Both `u` and `v` are second-order polynomials of `x` and `y`. Each second-order polynomial has six terms. To specify all coefficients, `Tinv` 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] * Tinv
```

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]  
* Tinv
```

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);
```

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')
```



```
figure; imshow(moving)
hold on
plot(movingPoints(:,1),movingPoints(:,2),'xw')
title('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 = 2x2
```

```
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')
```



## 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

`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.

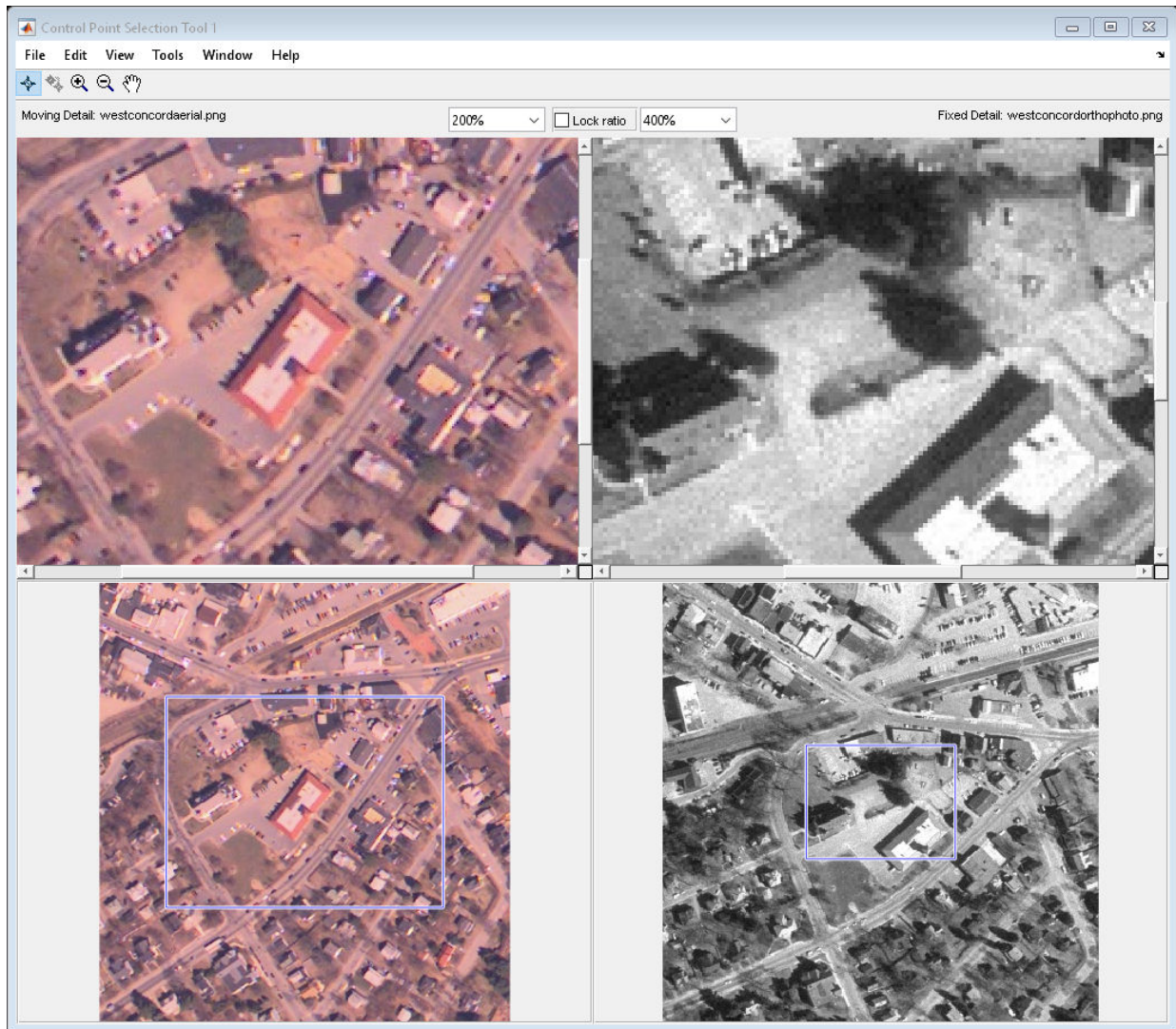
## Examples

### 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');
```

# 1 Functions — Alphabetical List



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 that brings the moving image into alignment with the fixed image. Specify the control points you selected and the type of transformation you want.

```
t_concord = fitgeotrans(aerial_points,ortho_points,'projective');
```

Next use `imwarp` to perform the transformation. By defining a spatial referencing object from `ortho` and specifying the object as the `'OutputView'`, the registered image has a size and location matching `ortho`.

```
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')
```

Finally, display the transformed image over the original orthophoto to see how well the registration succeeded.

```
figure, imshowpair(aerial_registered,ortho,'blend')
```

## Input Arguments

### **moving** — Input image to be aligned

grayscale image | truecolor image | binary image | character vector | string

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` | `char` | `string`

### **fixed** — Reference image

grayscale image | truecolor image | binary image | character vector | string

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` | `char` | `string`

### **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 a 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 the spatial transformation to the new point. This transformation generates the predicted point.



- 4 Display the predicted point.

## 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` structure. 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
```

The two sets of points are identical, which indicates that all points in the stored set of points belong to valid control point pairs.

## 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

`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` contains the discrete cosine transform coefficients  $B(k_1,k_2)$ .

`B = dct2(A,m,n)` and

`B = dct2(A,[m n])` pad the matrix `A` with 0s to size `m`-by-`n` before applying the transformation. If `m` or `n` is smaller than the corresponding dimension of `A`, then `dct2` crops `A` before the transformation.

### 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.

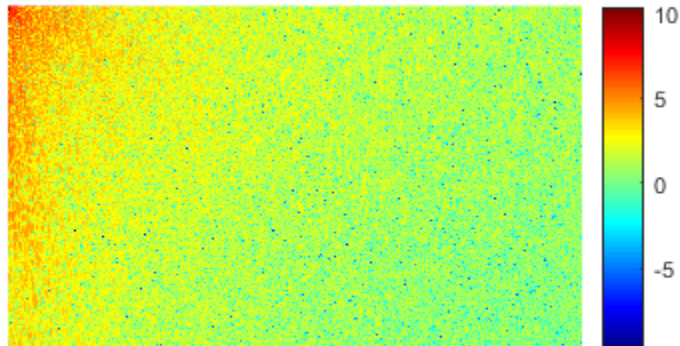
```
RGB = imread('autumn.tif');
I = rgb2gray(RGB);
```

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)



## Input Arguments

### **A** — Input matrix

2-D numeric matrix

Input matrix, specified as a 2-D numeric matrix.

### **m** — Number of image rows

`size(A,1)` (default) | positive integer

Number of image rows, specified as a positive integer. `dct2` pads image `A` with 0s or truncates image `A` so that it has `m` rows. By default, `m` is equal to `size(A,1)`.

### **n** — Number of image columns

`size(A,2)` (default) | positive integer

Number of image columns, specified as a positive integer. `dct2` pads image `A` with 0s or truncates image `A` so that it has `n` columns. By default, `n` is equal to `size(A,2)`.

## Output Arguments

### **B** — Transformed matrix

`m`-by-`n` numeric matrix

Transformed matrix using a two-dimensional discrete cosine transform, returned as an m-by-n numeric matrix.

Data Types: double

## Definitions

### Discrete Cosine Transform

The discrete cosine transform (DCT) is closely related to the discrete Fourier transform. It is a separable linear transformation; that is, the two-dimensional transform is equivalent to a one-dimensional DCT performed along a single dimension followed by a one-dimensional DCT in the other dimension. The definition of the two-dimensional DCT for an input image A and output image B is

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

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 A, respectively.



## Tips

- 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.
- To invert the DCT transformation, use `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

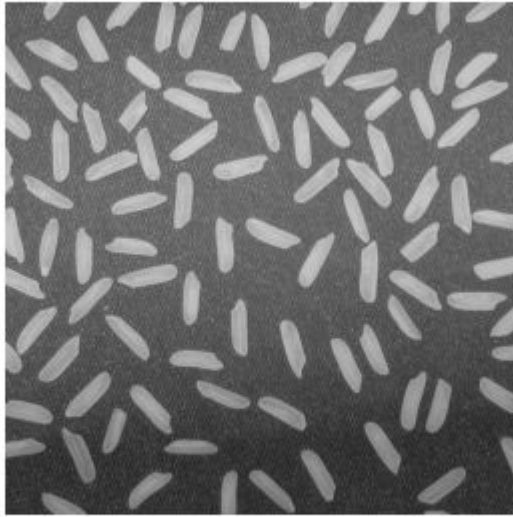
`D = dctmtx(n)` returns the  $n$ -by- $n$  discrete cosine transform (DCT) matrix, which you can use to perform a 2-D DCT on an image.

### Examples

#### Calculate Discrete Cosine Transform Matrix

Read an image into the workspace and cast it to class `double`.

```
A = im2double(imread('rice.png'));  
imshow(A)
```



Calculate the 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';  
imshow(dct)
```



## Input Arguments

**n** — Size of DCT matrix

positive integer

Size of DCT matrix, specified as a positive integer.

Data Types: double

## Output Arguments

**D** — DCT matrix

numeric matrix

DCT matrix, returned as a numeric matrix of size n-by-n.

Data Types: double

## Tips

- If you have an  $n$ -by- $n$  image,  $A$ , then  $D*A$  is the DCT of the columns of  $A$  and  $D'*A$  is the inverse DCT of the columns of  $A$ .
- The two-dimensional DCT of  $A$  can be computed as  $D*A*D'$ . This computation is sometimes faster than using `dct2`, especially if you are computing a large number of small DCTs, because  $D$  needs to be determined only once.

For example, in JPEG compression, the DCT of each 8-by-8 block is computed. To perform this computation, use `dctmtx` to determine  $D$ , and then calculate each DCT using  $D*A*D'$  (where  $A$  is each 8-by-8 block). This is faster than calling `dct2` for each individual block.

## See Also

`dct2`

**Introduced before R2006a**

## decompose

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 structuring element SE. SEQ is equivalent to SE, but the elements of SEQ cannot be decomposed further.

### 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 = 5x5
```

```
 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
```

### 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: [11x11 double]
Dimensionality: 2
```

Obtain the decomposition of the structuring element.

```
seq = decompose(se)
```

```
seq =
```

```
1x8 offsetstrel array with properties:
```

```
    Offset
Dimensionality
```

## Input Arguments

### **SE — Structuring element**

`strel` or `offsetstrel` object

Structuring element, specified as a `strel` or `offsetstrel` object.

## Output Arguments

### **SEQ — Sequence of structuring elements that approximate the desired shape**

array of `strel` or `offsetstrel` objects

Sequence of structuring elements that approximate the desired shape, returned as an array of `strel` or `offsetstrel` objects.

## See Also

**Introduced before R2006a**



# deconvblind

Deblur image using blind deconvolution

## Syntax

```
[J,psfr] = deconvblind(I,psfi)
[J,psfr] = deconvblind(I,psfi,iter)
[J,psfr] = deconvblind(I,psfi,iter,dampar)
[J,psfr] = deconvblind(I,psfi,iter,dampar,weight)
[J,psfr] = deconvblind(I,psfi,iter,dampar,weight,readout)
[J,psfr] = deconvblind( ____,fun)
```

## Description

`[J,psfr] = deconvblind(I,psfi)` deconvolves image `I` using the maximum likelihood algorithm and an initial estimate of the point-spread function (PSF), `psfi`. The `deconvblind` function returns both the deblurred image `J` and a restored PSF, `psfr`.

To improve the restoration, `deconvblind` supports several optional parameters, described below. Use `[]` as a placeholder if you do not specify an intermediate parameter.

`[J,psfr] = deconvblind(I,psfi,iter)` specifies the number of iterations, `iter`.

`[J,psfr] = deconvblind(I,psfi,iter,dampar)` controls noise amplification by suppressing iterations for pixels that deviate a small amount compared to the noise, specified by the damping threshold `dampar`. By default, no damping occurs.

`[J,psfr] = deconvblind(I,psfi,iter,dampar,weight)` specifies which pixels in the input image `I` are considered in the restoration. 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,psfr] = deconvblind(I,psfi,iter,dampar,weight,readout)` specifies the additive noise (e.g., background, foreground noise) and variance of the read-out camera noise, readout.

`[J,psfr] = deconvblind(____,fun)`, where `fun` is a handle to a function that describes additional constraints on the PSF. `fun` is called at the end of each iteration. For more information about function handles, see “Create Function Handle” (MATLAB).

## 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));
```

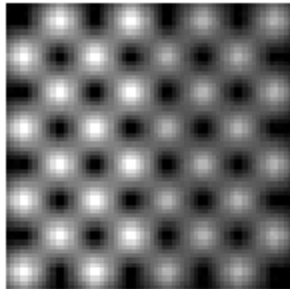
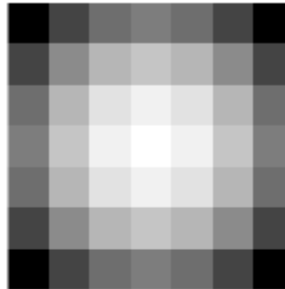
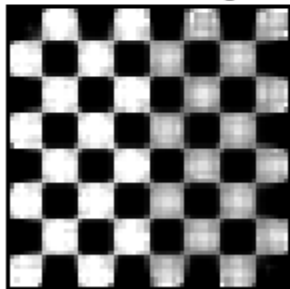
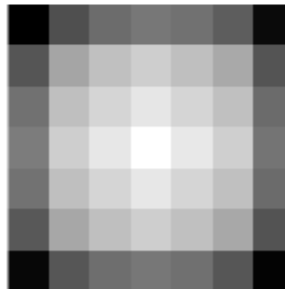
Perform blind deconvolution.

```
[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**

## Input Arguments

### **I** — Blurry image

numeric array | cell array

Blurry image, specified as a numeric array of any dimension. You can also specify the image as a cell array to enable interrupted iterations. For more information, see “Tips” on page 1-391.

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

## **psfi** — Initial estimate of PSF

numeric array

Initial estimate of PSF, specified as a numeric array. The PSF restoration is affected strongly by the size of the initial guess `psfi` and less by the values it contains. For this reason, specify an array of 1s as your `psfi`.

You can also specify `psfi` as a cell array to enable interrupted iterations. For more information, see “Tips” on page 1-391.

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

## **iter** — Number of iterations

10 (default) | positive integer

Number of iterations, specified as a positive integer.

Data Types: `double`

## **dampar** — Threshold for damping

0 (default) | numeric scalar

Threshold for damping, specified as a numeric scalar. Damping occurs for pixels whose deviation between iterations is less than the threshold. `dampar` has the same data type as `I`.

## **weight** — Weight of each pixel

numeric array

Weight value of each pixel, specified as a numeric array with values in the range [0, 1]. `weight` has the same size as the input image, `I`. By default, all elements in `weight` have the value 1, so all pixels are considered equally in the restoration.

Data Types: `double`

## **readout** — Noise

0 (default) | numeric scalar | numeric array

Noise, specified as a numeric scalar or numeric array. The value of `readout` corresponds to the additive noise (such as noise from the foreground and background) and the variance of the read-out camera noise. `readout` has the same data type as `I`.

**fun — Function handle**

handle

Function handle, specified as a handle. `fun` must accept the PSF as its first argument. The function must return one argument: a PSF that is the same size as the original PSF and that satisfies the positivity and normalization constraints.

## Output Arguments

**J — Deblurred image**

numeric array | 1-by-4 cell array

Deblurred image, returned as a numeric array or a 1-by-4 cell array. `J` (or `J{1}` when `J` is a cell array) has the same data type as `I`. For more information about returning `J` as a cell array for interrupted iterations, see “Tips” on page 1-391.

**psfr — Restored PSF**

array of positive numbers | 1-by-4 cell array

Restored PSF, returned as an array of positive numbers or a 1-by-4 cell array. `psfr` has the same size as the initial estimate of the PSF, `psfi`, and it is normalized so the sum of elements is 1. For more information about returning `psfr` as a cell array for interrupted iterations, see “Tips” on page 1-391.

Data Types: double

## Tips

- 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, `psfi`, as cell arrays: `{I}` and `{psfi}`. When you do, the `deconvblind` function returns the output image `J` and the restored point-spread function, `psfr`, 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.

- 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, psfi)` before calling `deconvblind`.

## References

- [1] D.S.C. Biggs and M. Andrews, *Acceleration of iterative image restoration algorithms*, Applied Optics, Vol. 36, No. 8, 1997.
- [2] R.J. Hanisch, R.L. White, and R.L. Gilliland, *Deconvolutions of Hubble Space Telescope Images and Spectra*, Deconvolution of Images and Spectra, Ed. P.A. Jansson, 2nd ed., Academic Press, CA, 1997.
- [3] Timothy J. Holmes, et al, *Light Microscopic Images Reconstructed by Maximum Likelihood Deconvolution*, Handbook of Biological Confocal Microscopy, Ed. James B. Pawley, Plenum Press, New York, 1995.

## See Also

`deconvlucy` | `deconvreg` | `deconvwnr` | `edgetaper` | `imnoise` | `otf2psf` | `padarray` | `psf2otf`

## Topics

“Deblurring Images Using the Blind Deconvolution Algorithm”

“Image Deblurring”

“Deblur with the Blind Deconvolution Algorithm”

**Introduced before R2006a**

# deconvlucy

Deblur image using Lucy-Richardson method

## Syntax

```
J = deconvlucy(I,psf)
J = deconvlucy(I,psf,iter)
J = deconvlucy(I,psf,iter,dampar)
J = deconvlucy(I,psf,iter,dampar,weight)
J = deconvlucy(I,psf,iter,dampar,weight,readout)
J = deconvlucy(I,psf,iter,dampar,weight,readout,subsample)
```

## Description

`J = deconvlucy(I,psf)` restores image `I` that was degraded by convolution with a point-spread function (PSF), `psf`, and possibly by additive noise. The algorithm is based on maximizing the likelihood that the resulting image `J` is an instance of the original image `I` under Poisson statistics.

To improve the restoration, `deconvlucy` supports several optional parameters, described below. Use `[]` as a placeholder if you do not specify an intermediate parameter.

`J = deconvlucy(I,psf,iter)` specifies the number of iterations, `iter`.

`J = deconvlucy(I,psf,iter,dampar)` controls noise amplification by suppressing iterations for pixels that deviate a small amount compared to the noise, specified by the damping threshold `dampar`. By default, no damping occurs.

`J = deconvlucy(I,psf,iter,dampar,weight)` specifies which pixels in the input image `I` are considered in the restoration. 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 = deconvlucy(I,psf,iter,dampar,weight,readout)` specifies the additive noise (e.g., background, foreground noise) and variance of the read-out camera noise, readout.

`J = deconvlucy(I,psf,iter,dampar,weight,readout,subsample)` uses subsampling when the PSF is given on a grid that is `subsample` times finer than the image.

## 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);
```

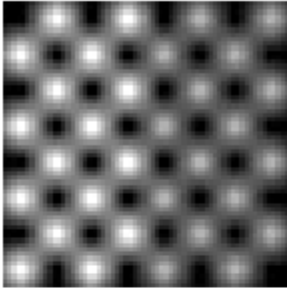
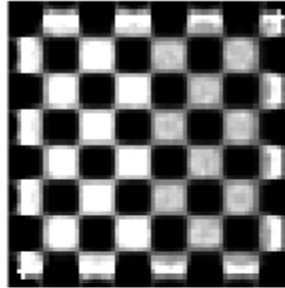
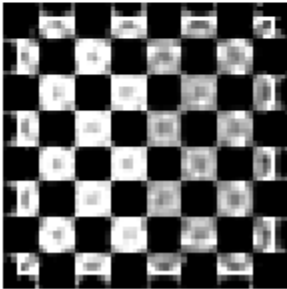
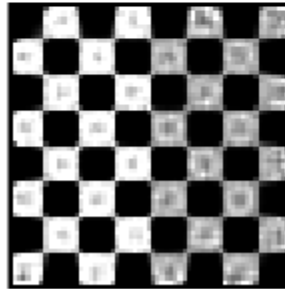
Create a weight array and call `deconvlucy` using several optional parameters.

```
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)**

## Input Arguments

### **I** – Blurry image

numeric array | cell array

Blurry image, specified as a numeric array of any dimension. You can also specify the image as a cell array to enable interrupted iterations. For more information, see “Tips” on page 1-397.

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

## **psf — PSF**

numeric array

PSF, specified as a numeric array. You can also specify `psf` as a cell array to enable interrupted iterations. For more information, see “Tips” on page 1-397.

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

## **iter — Number of iterations**

10 (default) | positive integer

Number of iterations, specified as a positive integer.

Data Types: `double`

## **dampar — Threshold for damping**

0 (default) | numeric scalar

Threshold for damping, specified as a numeric scalar. Damping occurs for pixels whose deviation between iterations is less than the threshold. `dampar` has the same data type as `I`.

## **weight — Weight of each pixel**

numeric array

Weight value of each pixel, specified as a numeric array with values in the range [0, 1]. `weight` has the same size as the input image, `I`. By default, all elements in `weight` have the value 1, so all pixels are considered equally in the restoration.

Data Types: `double`

## **readout — Noise**

0 (default) | numeric scalar | numeric array

Noise, specified as a numeric scalar or numeric array. The value of `readout` corresponds to the additive noise (such as noise from the foreground and background) and the variance of the read-out camera noise. `readout` has the same data type as `I`.

## **subsample — Subsampling**

1 (default) | positive scalar

Subsampling, specified as a positive scalar.

Data Types: `double`

## Output Arguments

### J — Deblurred image

numeric array | 1-by-4 cell array

Deblurred image, returned as a numeric array or a 1-by-4 cell array. J (or J{1} when J is a cell array) has the same data type as I. For more information about returning J as a cell array for interrupted iterations, see “Tips” on page 1-397.

## Tips

- You can use `deconvlucy` to perform a deconvolution that starts where a previous deconvolution stopped. To use this feature, pass the input image I and the PSF, `psf`, as cell arrays: `{I}` and `{psf}`. When you do, the `deconvlucy` function returns the output image J as a cell array, which can then be passed as the input array 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.

- 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`.
- `deconvlucy` converts the PSF to `double` without normalization.

## References

- [1] D.S.C. Biggs and M. Andrews, *Acceleration of iterative image restoration algorithms*, Applied Optics, Vol. 36, No. 8, 1997.
- [2] R.J. Hanisch, R.L. White, and R.L. Gilliland, *Deconvolutions of Hubble Space Telescope Images and Spectra*, Deconvolution of Images and Spectra, Ed. P.A. Jansson, 2nd ed., Academic Press, CA, 1997.

## **See Also**

deconvlucy | deconvreg | deconvwnr | edgetaper | otf2psf | padarray | psf2otf

## **Topics**

“Deblurring Images Using the Lucy-Richardson Algorithm”

“Image Deblurring”

“Deblur with the Lucy-Richardson Algorithm”

**Introduced before R2006a**

# deconvreg

Deblur image using regularized filter

## Syntax

```
J = deconvreg(I,psf)
J = deconvreg(I,psf,np)
J = deconvreg(I,psf,np,lrange)
J = deconvreg(I,psf,np,lrange,regop)
[J,lagra] = deconvreg( ___ )
```

## 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), `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.

`J = deconvreg(I,psf,np)` specifies the additive noise power, `np`.

`J = deconvreg(I,psf,np,lrange)` specifies the range, `lrange`, where the search for the optimal solution is performed. The algorithm finds an optimal Lagrange multiplier `lagra` within the `lrange` range.

`J = deconvreg(I,psf,np,lrange,regop)` constrains the deconvolution using regularization operator `regop`. The default regularization operator is the Laplacian operator, to retain the image smoothness.

`[J,lagra] = deconvreg( ___ )` outputs the value of the Lagrange multiplier, `lagra` in addition to the restored image, `J`.

## 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.

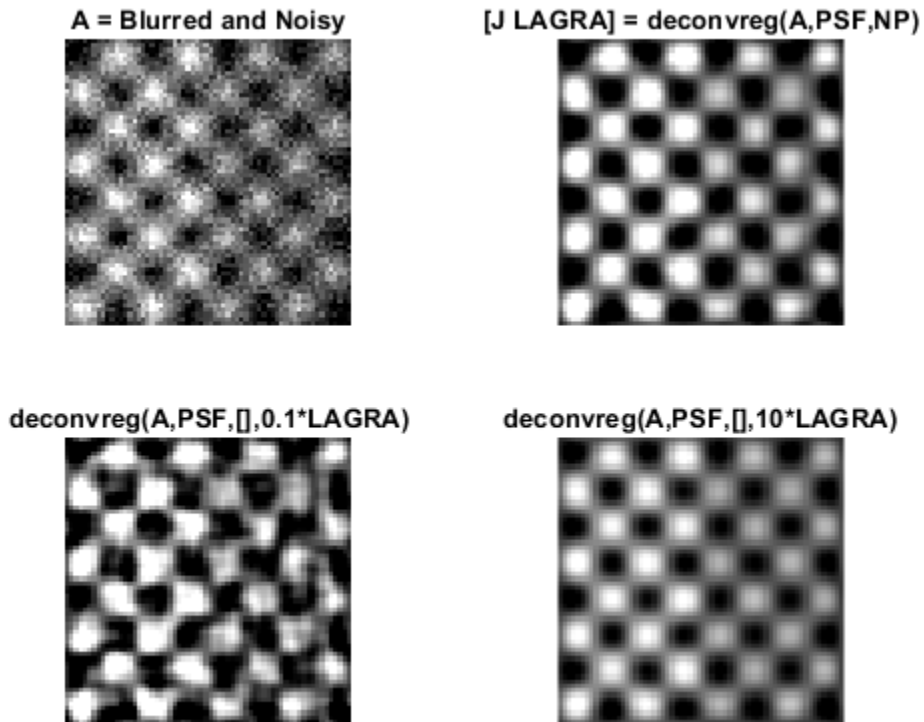
```
PSF = fspecial('gaussian',7,10);  
V = .01;  
BlurredNoisy = imnoise(imfilter(I,PSF),'gaussian',0,V);  
NOISEPOWER = V*prod(size(I));
```

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)');
```



## Input Arguments

### **I** – Blurry image

numeric array

Blurry image, specified as a numeric array of any dimension.

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

### **psf** – PSF

numeric array

PSF, specified as a numeric array.

Data Types: `double`

### **np — Noise power**

0 (default) | numeric scalar

Noise power, specified as a numeric scalar.

Data Types: `double`

### **lrange — Search range**

[1e-9 1e9] (default) | numeric scalar | 2-element numeric vector

Search range, specified as a numeric scalar or a 2-element numeric vector. If `lrange` is a scalar, the algorithm assumes that `lagra` is equal to `lrange`; the `np` value is then ignored.

Data Types: `double`

### **regop — Regularization operator**

numeric array

Regularization operator, specified as a numeric array. The `regop` array dimensions must not exceed the dimensions of the image, `I`. Any nonsingleton dimensions must correspond to the nonsingleton dimensions of `psf`.

Data Types: `double`

## Output Arguments

### **J — Deblurred image**

numeric array

Deblurred image, returned as a numeric array. `J` has the same data type as `I`.

### **lagra — Lagrange multiplier**

numeric scalar

Lagrange multiplier, returned as a numeric scalar.



## Tips

- The output image  $J$  could exhibit ringing introduced by the discrete Fourier transform used in the algorithm. To reduce the ringing, use  $I = \text{edgetaper}(I, \text{psf})$  before calling `deconvreg`.

## See Also

`deconvblind` | `deconvlucy` | `deconvwnr` | `edgetaper` | `otf2psf` | `padarray` | `psf2otf`

## Topics

“Deblurring Images Using a Regularized Filter”

“Image Deblurring”

“Deblur with a Regularized Filter”

**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`. `psf` is the point-spread function (PSF) with which `I` was convolved. `nsr` is the noise-to-signal power ratio of the additive noise. 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.

### 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')
```

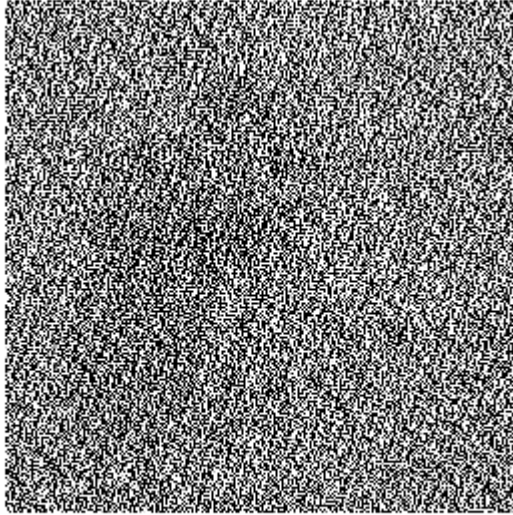
### Simulate Blur and Noise



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



## Input Arguments

### **I** — Blurry image

numeric array

Blurry image, specified as a numeric array of any dimension.

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

### **psf** — PSF

numeric array

PSF, specified as a numeric array.

Data Types: `double`

## **nsr — Noise-to-signal ratio**

positive scalar

Noise-to-signal ratio, specified as a positive scalar or numeric array of the same size as the image, `I`. If `nsr` is an array, then it represents the spectral domain. Specifying `0` for the `nsr` is equivalent to creating an ideal inverse filter.

Data Types: `double`

## **ncorr — Autocorrelation function of the noise**

numeric array

Autocorrelation function of the noise, specified as a numeric array of any size or dimension, not exceeding the original image.

- If the dimensionality of `ncorr` matches the dimensionality of the image `I`, then the values correspond to the autocorrelation within each dimension.
- If `ncorr` is a vector and `psf` is also a vector, then the values in `ncorr` represent the autocorrelation function in the first dimension.
- If `ncorr` is a vector and `psf` is an array, then the 1-D autocorrelation function is extrapolated by symmetry to all non-singleton dimensions of `psf`.
- If `ncorr` is a scalar, then the value represents the power of the image noise.

Data Types: `double`

## **icorr — Autocorrelation function of the image**

numeric array

Autocorrelation function of the image, specified as a numeric array of any size or dimension, not exceeding the original image.

- If the dimensionality of `icorr` matches the dimensionality of the image `I`, then the values correspond to the autocorrelation within each dimension.
- If `icorr` is a vector and `psf` is also a vector, then the values in `icorr` represent the autocorrelation function in the first dimension.
- If `icorr` is a vector and `psf` is an array, then the 1-D autocorrelation function is extrapolated by symmetry to all non-singleton dimensions of `psf`.
- If `icorr` is a scalar, then the value represents the power of the image noise.

Data Types: `double`



## Output Arguments

### **J** — Deblurred image

numeric array

Deblurred image, returned as a numeric array. J has the same data type as I.

## Tips

- 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 `deconvwnr`.

## References

- [1] "Digital Image Processing", R. C. Gonzalez & R. E. Woods, Addison-Wesley Publishing Company, Inc., 1992.

## See Also

`deconvblind` | `deconvlucy` | `deconvreg` | `edgetaper` | `otf2psf` | `padarray` | `psf2otf`

## Topics

"Deblurring Images Using a Wiener Filter"

"Image Deblurring"

"Deblur with the Wiener Filter"

**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 RGB or multispectral image `A` and returns the result in `S`. The mean and variance in each band of `S` are the same as in `A`.

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)` uses name-value pairs to control aspects of the decorrelation stretch, such as the target mean and standard deviation of each band.

### 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**

RGB image | multispectral image

Image to be enhanced, specified as an RGB image or multispectral image of size  $m$ -by- $n$ -by- $n$ Bands. For an RGB image,  $n$ Bands = 3.

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 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) | `'covariance'`

Decorrelation method, specified as the comma-separated pair consisting of `'Mode'` and of the following values.

- `'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` | `string`

### TargetMean — Target mean values

real scalar | vector of length `nBands`

Target mean values of the output bands, specified as the comma-separated pair consisting of `'TargetMean'` and a real scalar or vector of length `nBands`. By default, `TargetMean` is an 1-by-`nBands` vector containing the sample mean of each band, which preserves the band-wise means before and after the decorrelation stretch.

`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 a reasonable value. If values need to be clamped to the standard range of the input/output image class, it can impact the results.

Data Types: `double`

### TargetSigma — Target standard deviation values

positive scalar | vector of length `nBands`

Target standard deviation values of the output bands, specified as the comma-separated pair consisting of `'TargetSigma'` and a positive scalar or vector of length `nBands`. By

default, `TargetSigma` is an 1-by-nBands vector containing the sample standard deviation of each band, which preserves the band-wise variance before and after the decorrelation stretch. The target standard deviation is ignored for uniform (zero-variance) bands.

`TargetSigma` must be class `double`, but uses the same values as the pixels in the input image. For example, if `A` is of class `uint8`, then `50.0` would be a reasonable value.

Data Types: `double`

### **Tol — Linear contrast stretch**

numeric scalar | 2-element numeric vector

Linear contrast stretch following the decorrelation stretch, specified as the comma-separated pair consisting of `'Tol'` and a numeric scalar or 2-element numeric vector of class `double`. Specifying a value of `Tol` overrides the value of `TargetMean` or `TargetSigma`. If you do not specify `Tol`, then by default `decorrstretch` does not perform linear contrast stretch.

`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.

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 — Decorrelation stretched image

numeric array

Decorrelation stretched image, returned as a numeric array of the same size and class as the input image, 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 255 or 65535, respectively). In these cases, `decorstretch` clamps its output to the supported range.
- For class `double`, `decorstretch` 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:

$$b = T * (a - m) + m\_target.$$

`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:

$$SIGMA(k,k) = \text{sqrt}(Cov(k,k), k = 1, \dots, nBands).$$

$Cov$ ,  $SIGMA$ , 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

$$Corr = \text{inv}(SIGMA) * Cov * \text{inv}(SIGMA).$$

- 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:

$$S(k,k) = 1 / \text{sqrt}(LAMBDA(k,k)).$$

Finally, matrix  $T$  is computed from either

$$T = SIGMA\_target V S V' \text{inv}(SIGMA) \text{ (correlation-based method)}$$

or

$$T = SIGMA\_target V S V' \text{ (covariance-based method)}.$$

The two methods yield identical results if the band variances are uniform.

Substituting  $T$  into the expression for  $b$ :



$$b = m\_target + SIGMA\_target V S V' inv(SIGMA) * (a - m)$$

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

`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 'bgr' .

```
I = imread('mandi.tif');  
J = demosaic(I,'bgr');  
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				

Pattern	2-by-2 Sensor Alignment				
'rggb'	<table border="1"><tbody><tr><td>Red</td><td>Green</td></tr><tr><td>Green</td><td>Blue</td></tr></tbody></table>	Red	Green	Green	Blue
Red	Green				
Green	Blue				

Data Types: char | string

## 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**

## denoiseImage

Denoise image using deep neural network

### Syntax

```
B = denoiseImage(A,net)
```

### Description

`B = denoiseImage(A,net)` estimates denoised image `B` from noisy image `A` using a denoising deep neural network specified by `net`.

This function requires that you have Deep Learning Toolbox™.

### Examples

#### Remove Image Noise Using Pretrained Neural Network

Retrieve the pretrained denoising convolutional neural network, 'DnCNN'.

```
net = denoisingNetwork('DnCNN');
```

Load a grayscale image into the workspace, then create a noisy version of the image. Display the two images.

```
I = imread('cameraman.tif');  
noisyI = imnoise(I,'gaussian',0,0.01);  
figure  
imshowpair(I,noisyI,'montage');  
title('Original Image (left) and Noisy Image (right)')
```



Original Image (left) and Noisy Image (right)



Remove noise from the noisy image, and display the result.

```
denoisedI = denoiseImage(noisyI, net);  
figure  
imshow(denoisedI)  
title('Denoised Image')
```

**Denoised Image**



## Input Arguments

### **A — Noisy image**

2-D image | stack of 2-D images

Noisy image, specified as a single 2-D image or a stack of 2-D images. A can be:

- A 2-D grayscale image with size  $m$ -by- $n$ .
- A 2-D multichannel image with size  $m$ -by- $n$ -by- $c$ , where  $c$  is the number of image channels. For example,  $c$  is 3 for RGB images, and 4 for four-channel images such as RGB images with an infrared channel.
- A stack of equally-sized 2-D images. In this case, A has size  $m$ -by- $n$ -by- $c$ -by- $p$ , where  $p$  is the number of images in the stack.

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

**net — Denoising deep neural network**

SeriesNetwork object

Denoising deep neural network, specified as a SeriesNetwork object. The network should be trained to handle images with the same channel format as A.

## Output Arguments

**B — Denoised image**

2-D image | stack of 2-D images

Denoised image, returned as a single 2-D image or a stack of 2-D images. B has the same size and data type as A.

## See Also

denoisingImageDatastore | denoisingNetwork | dnCNNLayers

## Topics

“Train and Apply Denoising Neural Networks”

**Introduced in R2017b**

# denoisingImageDatstore

Denoising image datastore

## Description

Use a `denoisingImageDatstore` object to generate batches of noisy image patches and corresponding noise patches from images in an `ImageDatstore`. The patches are used to train a denoising deep neural network.

This object requires that you have Deep Learning Toolbox.

---

**Note** When you use a denoising image datastore as a source of training data, the datastore adds random noise to the image patches for each epoch, so that each epoch uses a slightly different data set. The actual number of training images at each epoch is increased by a factor of `PatchesPerImage`. The noisy image patches and corresponding noise patches are not stored in memory.

---

## Creation

## Syntax

```
dnimds = denoisingImageDatstore(imds)
dnimds = denoisingImageDatstore(imds,Name,Value)
```

## Description

`dnimds = denoisingImageDatstore(imds)` creates a denoising image datastore, `dnimds` using images from image datastore `imds`. To generate noisy image patches, the denoising image datastore randomly crops pristine images from `imds` then adds zero-mean Gaussian white noise with a standard deviation of `0.1` to the image patches.

`dnimds = denoisingImageDatstore(imds,Name,Value)` uses name-value pairs to specify the two-dimensional image patch size or to set the `PatchesPerImage`,

`GaussianNoiseLevel`, `ChannelFormat`, and `DispatchInBackground` properties. You can specify multiple name-value pairs. Enclose each argument or property name in quotes.

For example, `denoisingImageDatastore(imds, 'PatchesPerImage', 40)` creates a denoising image datastore and randomly generates 40 noisy patches from each image in the image datastore, `imds`.

## Input Arguments

### **imds** — Images with labels for classification problems

ImageDatastore object

Images, specified as an `ImageDatastore` object with categorical labels. You can store data in `ImageDatastore` for only classification problems.

`ImageDatastore` allows batch-reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, prefetching does not happen.

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

Example: `denoisingImageDatastore(imds, 'patchSize', 48)` creates a denoising image datastore that has a square patch size of 48 pixels.

### **patchSize** — Patch size

50 (default) | scalar | 2-element vector

Patch size, specified as the comma-separated pair consisting of `'patchSize'` and a scalar or 2-element vector with positive integer values. This argument sets the first two elements of the `PatchSize` property.

- If `'patchSize'` is a scalar, then the patches are square.
- If `'patchSize'` is a 2-element vector of the form `[r c]`, then the first element specifies the number of rows in the patch, and the second element specifies the number of columns.

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

## Properties

### **ChannelFormat** — Channel format

'grayscale' (default) | 'rgb'

Channel format, specified as 'grayscale' or 'rgb'.

Data Types: char

### **DispatchInBackground** — Dispatch observations in background

false (default) | true

Dispatch observations in the background during training, prediction, and classification, specified as `false` or `true`. To use background dispatching, you must have Parallel Computing Toolbox. If `DispatchInBackground` is `true` and you have Parallel Computing Toolbox, then `denoisingImageDatastore` asynchronously reads patches, adds noise, and queues patch pairs.

### **GaussianNoiseLevel** — Gaussian noise standard deviation

0.1 (default) | scalar | 2-element vector

Gaussian noise standard deviation as a fraction of the image class maximum, specified as a scalar or 2-element vector with values in the range [0, 1].

- If `GaussianNoiseLevel` is a scalar, then the standard deviation of the added zero-mean Gaussian white noise is identical for all image patches.
- If `GaussianNoiseLevel` is a 2-element vector, then it specifies a range of standard deviations [*stdmin* *stdmax*]. The standard deviation of the added zero-mean Gaussian white noise is unique for each image patch, and is randomly sampled from a uniform distribution with the range [*stdmin* *stdmax*].

Data Types: single | double

### **MiniBatchSize** — Number of observations in each batch

positive integer

This property is read-only.

Number of observations that are returned in each batch. For training, prediction, or classification, the `MiniBatchSize` property is set to the mini-batch size defined in `trainingOptions`.

**NumObservations — Total number of observations in the datastore**

positive integer

This property is read-only.

Total number of observations in the denoising image datastore. The number of observations is the length of one training epoch.

**PatchesPerImage — Number of random patches per image**

512 (default) | positive integer

Number of random patches per image, specified as a positive integer.

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

**PatchSize — Patch size**

[50 50 1] (default) | 3-element vector of positive integers

This property is read-only.

Patch size, specified as a 3-element vector of positive integers. If you create a denoising image datastore by specifying a 'patchSize' name-value pair argument, then the first two elements of the PatchSize property are set according to the value of the patchSize argument.

The ChannelFormat property determines the third element of the PatchSize property.

- If ChannelFormat is 'Grayscale', then all color images are converted to grayscale and the third element of PatchSize is 1.
- If ChannelFormat is 'RGB', then grayscale images are replicated to simulate an RGB image and the third element of PatchSize is 3.

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

**Object Functions**

hasdata	Determine if data is available to read
partitionByIndex	Partition denoisingImageDatastore according to indices
preview	Subset of data in datastore
read	Read data from denoisingImageDatastore
readall	Read all data in datastore

readByIndex	Read data specified by index from denoisingImageDatastore
reset	Reset datastore to initial state
shuffle	Shuffle data in denoisingImageDatastore

## Examples

### Create Denoising Image Datastore

Get an image datastore. The datastore in this example contains color images.

```
setDir = fullfile(toolboxdir('images'),'imdata');  
imds = imageDatastore(setDir,'FileExtensions',{' .jpg'});
```

Create a `denoisingImageDatastore` object that creates many patches from each image in the image datastore, and adds Gaussian noise to the patches. Set the optional `PatchesPerImage`, `PatchSize`, `GaussianNoiseLevel`, and `ChannelFormat` properties of the `denoisingImageDatastore` using name-value pairs. When you set the `ChannelFormat` property to `'grayscale'`, the `denoisingImageDatastore` converts all color images to grayscale.

```
dnds = denoisingImageDatastore(imds,...  
    'PatchesPerImage',512,...  
    'PatchSize',50,...  
    'GaussianNoiseLevel',[0.01 0.1],...  
    'ChannelFormat','grayscale')  
  
dnds =  
    denoisingImageDatastore with properties:  
  
        PatchesPerImage: 512  
        PatchSize: [50 50 1]  
    GaussianNoiseLevel: [0.0100 0.1000]  
        ChannelFormat: 'grayscale'  
        MiniBatchSize: 128  
        NumObservations: 18944  
    DispatchInBackground: 0
```



## Tips

- Training a deep neural network for a range of Gaussian noise standard deviations is a much more difficult problem than training a network for a single Gaussian noise standard deviation. You should create more patches compared to a single noise level case, and training might take more time.
- To visualize the data in a denoising image datastore, you can use the `preview` function, which returns a subset of data in a table. The `input` variable contains the noisy image patches and the `response` variable contains the corresponding noise patches. Visualize all of the noisy image patches or noise patches in the same figure by using the `montage` function. For example, this code displays data in a denoising image datastore called `dnimds`.

```
minibatch = preview(dnimds);  
montage(minibatch.input)  
figure  
montage(minibatch.response)
```

- Each time images are read from the denoising image datastore, a different random amount of Gaussian noise is added to each image.

## See Also

`denoiseImage` | `denoisingNetwork` | `dnCNNLayers` | `trainNetwork`

## Topics

“Train and Apply Denoising Neural Networks”

**Introduced in R2018a**

## partitionByIndex

Partition `denoisingImageDatastore` according to indices

### Syntax

```
dnimds2 = partitionByIndex(dnimds,ind)
```

### Description

`dnimds2 = partitionByIndex(dnimds,ind)` partitions a subset of observations in a denoising image datastore, `dnimds`, into a new datastore, `dnimds2`. The desired observations are specified by indices, `ind`.

### Input Arguments

**dnimds** — Denoising image datastore

`denoisingImageDatastore`

Denoising image datastore, specified as a `denoisingImageDatastore` object.

**ind** — Indices

vector of positive integers

Indices of observations, specified as a vector of positive integers.

### Output Arguments

**dnimds2** — Output datastore

`denoisingImageDatastore` object

Output datastore, returned as a `denoisingImageDatastore` object containing a subset of files from `dnimds`.

## See Also

denoisingImageDatastore | read | readByIndex | readall

**Introduced in R2018a**

## read

Read data from `denoisingImageDatastore`

## Syntax

```
data = read(dnims)
[data,info] = read(dnims)
```

## Description

`data = read(dnims)` returns a batch of data from a denoising image datastore, `dnims`. Subsequent calls to the `read` function continue reading from the endpoint of the previous call.

`[data,info] = read(dnims)` also returns information about the extracted data, including metadata, in `info`.

## Input Arguments

### **dnims** — Denoising image datastore

`denoisingImageDatastore`

Denoising image datastore, specified as a `denoisingImageDatastore` object. The datastore specifies a `MiniBatchSize` number of observations in each batch, and a `numObservations` total number of observations.

## Output Arguments

### **data** — Output data

table

Output data, returned as a table with `MiniBatchSize` number of rows.

For the last batch of data in the datastore `dnimds`, if `numObservations` is not cleanly divisible by `MiniBatchSize`, then `read` returns a partial batch containing all the remaining observations in the datastore.

### **info – Information about read data**

structure array

Information about read data, returned as a structure array. The structure array can contain the following fields.

Field Name	Description
<code>CurrentFileIndices</code>	Current read index of the denoising image datastore.

## **See Also**

`denoisingImageDatastore` | `matlab.io.datastore.read` | `readByIndex` | `readall`

**Introduced in R2018a**

## readByIndex

Read data specified by index from `denoisingImageDatastore`

### Syntax

```
data = readByIndex(dnims,ind)
[data,info] = readByIndex(dnims,ind)
```

### Description

`data = readByIndex(dnims,ind)` returns a subset of observations from a denoising image datastore, `dnims`. The desired observations are specified by indices, `ind`.

`[data,info] = readByIndex(dnims,ind)` also returns information about the observations, including metadata, in `info`.

### Input Arguments

**dnims** — Denoising image datastore

`denoisingImageDatastore`

Denoising image datastore, specified as a `denoisingImageDatastore` object.

**ind** — Indices

vector of positive integers

Indices of observations, specified as a vector of positive integers.

### Output Arguments

**data** — Observations from datastore

table

Observations from the datastore, returned as a table with `length(ind)` number of rows.

**info — Information about read data**

structure array

Information about read data, returned as a structure array. The structure array can contain the following fields.

Field Name	Description
CurrentFileIndices	Numeric vector containing the indices of all read files of the denoising image datastore.

**See Also**

denoisingImageDatastore | partitionByIndex | read | readall

**Introduced in R2018a**

## shuffle

Shuffle data in `denoisingImageDatastore`

### Syntax

```
dnimds2 = shuffle(dnimds)
```

### Description

`dnimds2 = shuffle(dnimds)` returns a `denoisingImageDatastore` object containing a random ordering of the data from denoising image datastore `dnimds`.

### Input Arguments

**dnimds** — Denoising image datastore

`denoisingImageDatastore`

Denoising image datastore, specified as a `denoisingImageDatastore` object.

### Output Arguments

**dnimds2** — Output datastore

`denoisingImageDatastore` object

Output datastore, returned as a `denoisingImageDatastore` object containing randomly ordered files from `dnimds`.

### See Also

`denoisingImageDatastore` | `read` | `readByIndex` | `readall`



**Introduced in R2018a**

## denoisingImageSource

(To be removed) Create denoising image datastore

---

**Note** `denoisingImageSource` will be removed in a future release. Create a denoising image datastore using the `denoisingImageDatastore` function instead.

---

### Syntax

```
dnimds = denoisingImageSource(imds)
dnimds = denoisingImageSource(imds,Name,Value)
```

### Description

`dnimds = denoisingImageSource(imds)` creates a denoising image datastore, `dnimds`, that generates pairs of randomly cropped pristine and noisy image patches from images in image datastore `imds`.

`dnimds = denoisingImageSource(imds,Name,Value)` sets properties on page 1-432 of the denoising image datastore using name-value pairs. You can specify multiple name-value pairs. Enclose each argument name in quotes.

### Examples

#### Create Denoising Image Source

Get an image datastore. This datastore contains RGB images.

```
setDir = fullfile(toolboxdir('images'),'imdata');
imds = imageDatastore(setDir,'FileExtensions',{' .jpg'});
```

Create a `denoisingImageSource` object. The image source creates many patches from each image in the datastore, and adds Gaussian noise to the patches. Set the optional

PatchesPerImage, PatchSize, GaussianNoiseLevel, and ChannelFormat properties of the denoisingImageSource using name-value pairs.

```
source = denoisingImageSource(imds,...
    'PatchesPerImage',512,...
    'PatchSize',50,...
    'GaussianNoiseLevel',[0.01 0.1],...
    'ChannelFormat','RGB')

source =
    denoisingImageDatastore with properties:

        PatchesPerImage: 512
           PatchSize: [50 50 3]
    GaussianNoiseLevel: [0.0100 0.1000]
           ChannelFormat: 'rgb'
           MiniBatchSize: 128
           NumObservations: 18944
    DispatchInBackground: 0
```

You can use the `denoisingImageSource` to train a custom image denoising network. However, to do this, the image source must be created using grayscale images. Modify this example to point to your own image datastore containing grayscale images, and create a new denoising image source. Specify the `ChannelFormat` property of the image source as `'Grayscale'`. Provide your new image source and `dnCNNLayers` to `trainNetwork`.

## Input Arguments

### **imds** — Images with labels for classification problems

ImageDatastore object

Images, specified as an ImageDatastore object with categorical labels. You can store data in ImageDatastore for only classification problems.

ImageDatastore allows batch-reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, prefetching does not happen.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `denoisingImageSource(imds, 'patchSize', 48)` creates a denoising image datastore that has a square patch size of 48 pixels.

### **patchSize** — Patch size

50 (default) | scalar | 2-element vector

Patch size, specified as the comma-separated pair consisting of `'patchSize'` and a scalar or 2-element vector with positive integer values. This argument sets the first two elements of the `PatchSize` on page 1-0 property of the returned denoising image datastore, `dnimds`.

- When `'patchSize'` is a scalar, the patches are square
- When `'patchSize'` is a 2-element vector of the form  $[r\ c]$ , the first element specifies the number of rows in the patch, and the second element specifies the number of columns

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

### **PatchesPerImage** — Number of random patches per image

512 (default) | positive integer

Number of random patches per image, specified as the comma-separated pair consisting of `'PatchesPerImage'` and a positive integer. This argument sets the `PatchesPerImage` on page 1-0 property of the returned denoising image datastore, `dnimds`.

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

### **GaussianNoiseLevel** — Gaussian noise standard deviation

0.1 (default) | scalar | 2-element vector

Gaussian noise standard deviation as a fraction of the image class maximum, specified as the comma-separated pair consisting of `'GaussianNoiseLevel'` and a scalar or 2-element vector with values in the range  $[0, 1]$ . This argument sets the `GaussianNoiseLevel` on page 1-0 property of the returned denoising image datastore, `dnimds`.

- If `GaussianNoiseLevel` is a scalar, then the standard deviation of the added zero-mean Gaussian white noise is identical for all image patches.
- If `GaussianNoiseLevel` is a 2-element vector, then it specifies a range of standard deviations [*stdmin stdmax*]. The standard deviation of the added zero-mean Gaussian white noise is unique for each image patch, and is randomly sampled from a uniform distribution with the range [*stdmin stdmax*].

Data Types: `single` | `double`

### **ChannelFormat — Channel format**

`'Grayscale'` (default) | `'RGB'`

Channel format, specified as the comma-separated pair consisting of `'ChannelFormat'` and `'Grayscale'` or `'RGB'`. This argument sets the `ChannelFormat` on page 1-0 property of the returned denoising image datastore, `dnimds`.

Data Types: `char`

### **BackgroundExecution — Preprocess training patches in parallel**

`false` (default) | `true`

Preprocess training patches in parallel, specified as the comma-separated pair consisting of `'BackgroundExecution'` and `true` or `false`. This argument sets the `DispatchInBackground` on page 1-0 property of the returned denoising image datastore, `dnimds`. If `BackgroundExecution` is `true` and you have Parallel Computing Toolbox, then the denoising image datastore asynchronously reads patches, adds noise, and queues patch pairs.

Data Types: `char`

## **Output Arguments**

### **dnimds — Denoising image datastore**

`denoisingImageDatastore` object

Denoising image datastore, returned as an `denoisingImageDatastore` object.

## **See Also**

`denoisingImageDatastore`

**Introduced in R2017b**

# denoisingNetwork

Get image denoising network

## Syntax

```
net = denoisingNetwork(modelName)
```

## Description

`net = denoisingNetwork(modelName)` returns a pretrained image denoising deep neural network specified by `modelName`.

This function requires that you have Deep Learning Toolbox.

## Examples

### Get Pretrained Image Denoising Network

Get the pretrained image denoising convolutional neural network, 'DnCNN'.

```
net = denoisingNetwork('DnCNN')
```

```
net =  
  SeriesNetwork with properties:  
    Layers: [59x1 nnet.cnn.layer.Layer]
```

See `denoiseImage` for an example of how to denoise an image using the pretrained network.

## Input Arguments

### **modelName** — Name of neural network

'DnCNN'

Name of pretrained denoising deep neural network, specified as the character vector 'DnCNN'. This is the only pretrained denoising network currently available, and it is trained for grayscale images only.

Data Types: `char` | `string`

## Output Arguments

### **net** — Denoising deep neural network

`SeriesNetwork` object

Pretrained denoising deep neural network, returned as a `SeriesNetwork` object.

## References

- [1] Zhang, K., W. Zuo, Y. Chen, D. Meng, and L. Zhang. "Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising." *IEEE Transactions on Image Processing*. Vol. 26, Number 7, Feb. 2017, pp. 3142-3155.

## See Also

`denoiseImage` | `denoisingImageDatastore` | `dnCNNLayers`

## Topics

"Train and Apply Denoising Neural Networks"

**Introduced in R2017b**



# dice

Sørensen-Dice similarity coefficient for image segmentation

## Syntax

```
similarity = dice(BW1,BW2)
similarity = dice(L1,L2)
similarity = dice(C1,C2)
```

## Description

`similarity = dice(BW1,BW2)` computes the Sørensen-Dice similarity coefficient between binary images `BW1` and `BW2`.

`similarity = dice(L1,L2)` computes the Dice index for each label in label images `L1` and `L2`.

`similarity = dice(C1,C2)` computes the Dice index for each category in categorical images `C1` and `C2`.

## Examples

### Compute Dice Similarity Coefficient for Binary Segmentation

Read an image with an object to segment. Convert the image to grayscale, and display the result.

```
A = imread('hands1.jpg');
I = rgb2gray(A);
figure
imshow(I)
title('Original Image')
```

Original Image



Use active contours (snakes) to segment the hand.

```
mask = false(size(I));  
mask(25:end-25,25:end-25) = true;  
BW = activecontour(I, mask, 300);
```

Read in the ground truth segmentation.

```
BW_groundTruth = imread('hands1-mask.png');
```

Compute the Dice index of the active contours segmentation against the ground truth.

```
similarity = dice(BW, BW_groundTruth);
```

Display the masks on top of each other. Colors indicate differences in the masks.

```
figure  
imshowpair(BW, BW_groundTruth)  
title(['Dice Index = ' num2str(similarity)])
```



### Compute Dice Similarity Coefficient for Multi-Region Segmentation

This example shows how to segment an image into multiple regions. The example then computes the Dice similarity coefficient for each region.

Read an image with several regions to segment.

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

Create scribbles for three regions that distinguish their typical color characteristics. The first region classifies the yellow flower. The second region classifies the green stem and leaves. The last region classifies the brown dirt in two separate patches of the image. Regions are specified by a 4-element vector, whose elements indicate the x- and y-coordinate of the upper left corner of the ROI, the width of the ROI, and the height of the ROI.

```
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;

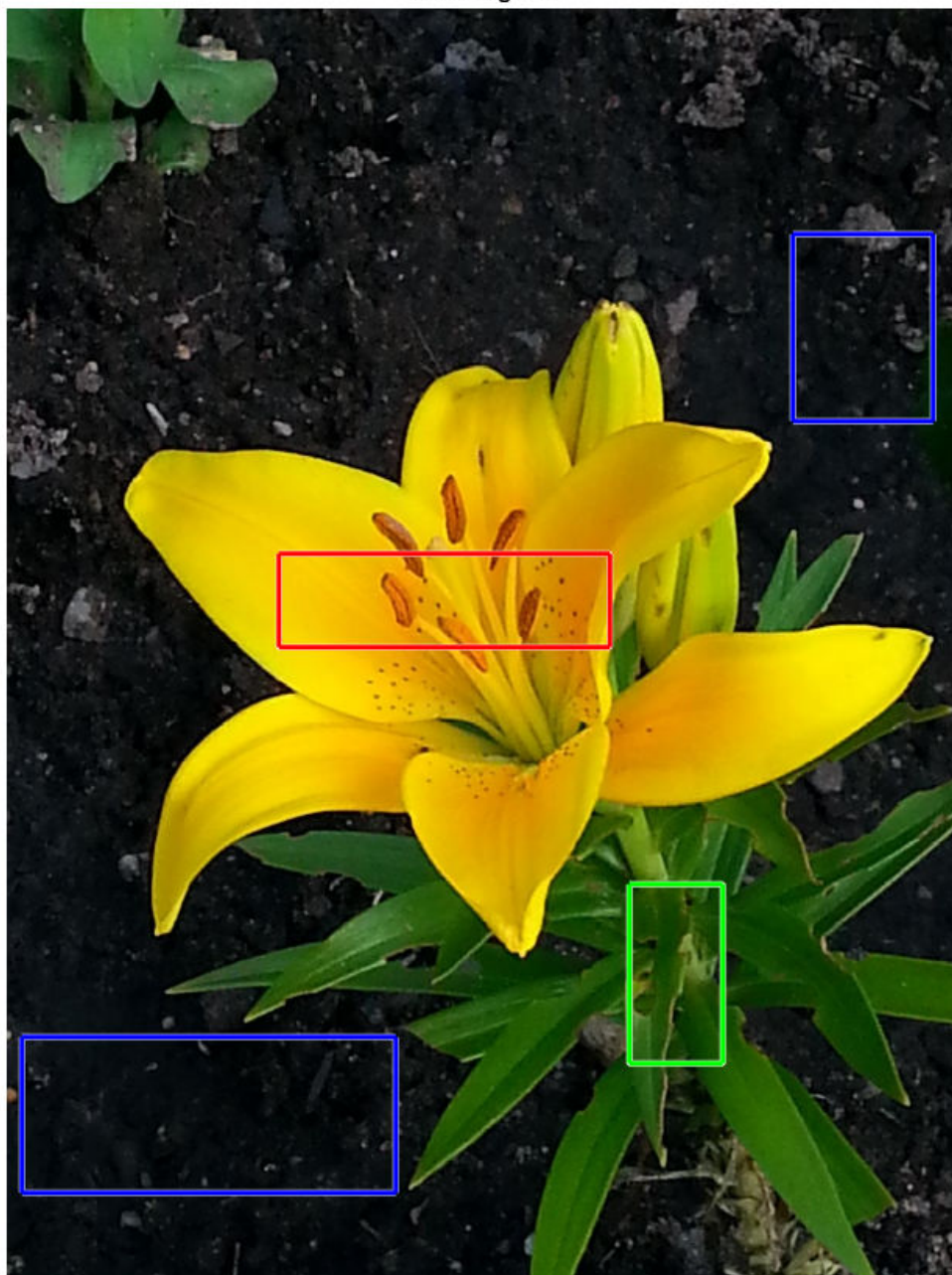
region2 = [800 1124 120 230];
BW2 = false(size(RGB,1),size(RGB,2));
BW2(region2(2):region2(2)+region2(4),region2(1):region2(1)+region2(3)) = true;

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)) = t
BW3(region3(2,2):region3(2,2)+region3(2,4),region3(2,1):region3(2,1)+region3(2,3)) = t
```

Display the seed regions on top of the image.

```
figure
imshow(RGB)
hold on
visboundaries(BW1, 'Color', 'r');
visboundaries(BW2, 'Color', 'g');
visboundaries(BW3, 'Color', 'b');
title('Seed Regions')
```

Seed Regions



Segment the image into three regions using geodesic distance-based color segmentation.

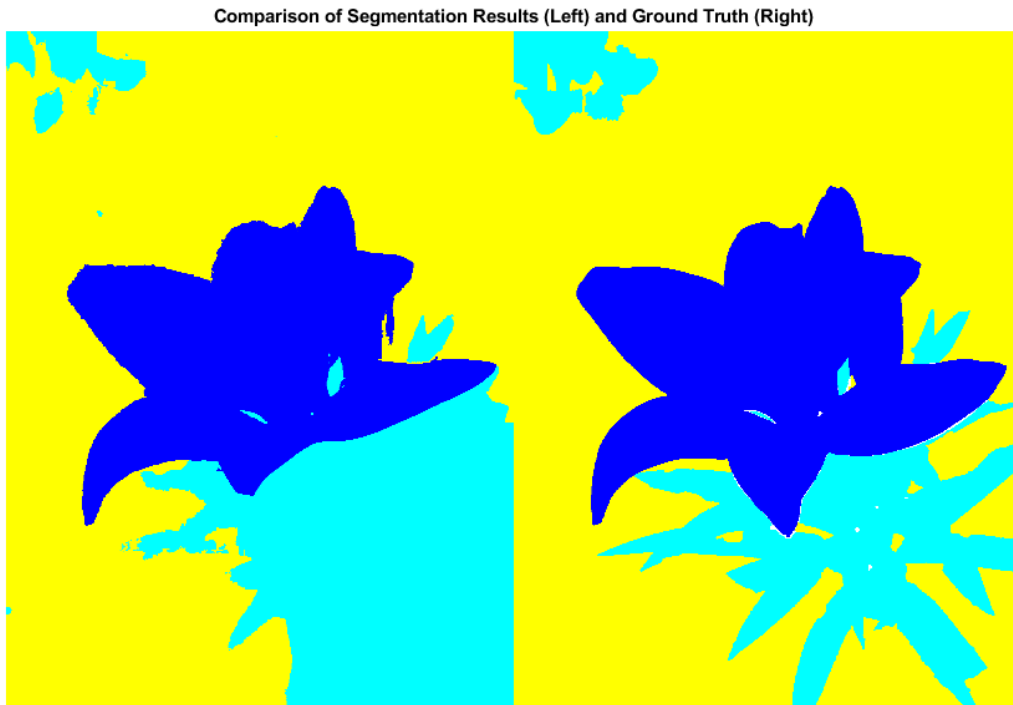
```
L = imseggeodesic(RGB,BW1,BW2,BW3,'AdaptiveChannelWeighting',true);
```

Load a ground truth segmentation of the image.

```
L_groundTruth = double(imread('yellowlily-segmented.png'));
```

Visually compare the segmentation results with the ground truth.

```
figure  
imshowpair(label2rgb(L),label2rgb(L_groundTruth),'montage')  
title('Comparison of Segmentation Results (Left) and Ground Truth (Right)')
```



Compute the Dice similarity index for each segmented region.

```
similarity = dice(L, L_groundTruth)
```

```
similarity = 3x1
    0.9396
    0.7247
    0.9139
```

The Dice similarity index is noticeably smaller for the second region. This result is consistent with the visual comparison of the segmentation results, which erroneously classifies the dirt in the lower right corner of the image as leaves.

## Input Arguments

### **BW1 — First binary image**

logical array

First binary image, specified as a logical array of any dimension.

Data Types: logical

### **BW2 — Second binary image**

logical array

Second binary image, specified as a logical array of the same size as BW1.

Data Types: logical

### **L1 — First label image**

array of nonnegative integers

First label image, specified as an array of nonnegative integers, of any dimension.

Data Types: double

### **L2 — Second label image**

array of nonnegative integers

Second label image, specified as an array of nonnegative integers, of the same size as L1.

Data Types: double

### **C1 — First categorical image**

categorical array

First categorical image, specified as a `categorical` array of any dimension.

Data Types: `category`

## **C2 — Second categorical image**

`categorical` array

Second categorical image, specified as a `categorical` array of the same size as C1.

Data Types: `category`

## **Output Arguments**

### **similarity — Dice similarity coefficient**

numeric scalar | numeric vector

Dice similarity coefficient, returned as a numeric scalar or numeric vector with values in the range [0, 1]. A `similarity` of 1 means that the segmentations in the two images are a perfect match. If the input arrays are:

- binary images, `similarity` is a scalar.
- label images, `similarity` is a vector, where the first coefficient is the Dice index for label 1, the second coefficient is the Dice index for label 2, and so on.
- categorical images, `similarity` is a vector, where the first coefficient is the Dice index for the first category, the second coefficient is the Dice index for the second category, and so on.

Data Types: `double`

## **Definitions**

### **Dice Similarity Coefficient**

The Dice similarity coefficient of two sets  $A$  and  $B$  is expressed as:

$$\text{dice}(A,B) = 2 * |\text{intersection}(A,B)| / (|A| + |B|)$$

where  $|A|$  represents the cardinal of set  $A$ . The Dice index can also be expressed in terms of true positives ( $TP$ ), false positives ( $FP$ ) and false negatives ( $FN$ ) as:



$$\text{dice}(A,B) = 2 * TP / ( 2 * TP + FP + FN)$$

The Dice index is related to the Jaccard index according to:

$$\text{dice}(A,B) = 2 * \text{jaccard}(A,B) / (1 + \text{jaccard}(A,B) )$$

## See Also

[bfscore](#) | [jaccard](#)

**Introduced in R2017b**

## dicomanon

Anonymize DICOM file

### Syntax

```
dicomanon(file_in,file_out)
dicomanon( __ , 'keep', fields)
dicomanon( __ , 'update', attributes)
dicomanon( __ , Name, Value)
```

### 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`. This syntax is useful for keeping metadata that does not uniquely identify the patient but is useful for diagnostic purposes (such as `PatientAge` and `PatientSex`).

---

**Note** Keeping certain fields might compromise patient confidentiality.

---

`dicomanon( __ , 'update', attributes)` modifies the confidential data and updates particular confidential data listed in `attributes`. Use this syntax to preserve the Study/Image hierarchy or to replace a specific value with a more generic property (such as removing `PatientBirthDate` but keeping a computed `PatientAge`).

`dicomanon( __ , Name, Value)` uses name-value pairs to provide additional options to the parser.

### 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
```

## Input Arguments

### **file\_in** — Name of DICOM file to read

character vector | string scalar

Name of DICOM file to read, specified as a character vector or string scalar.

Data Types: char | string

### **file\_out** — Name of anonymized DICOM file to write

character vector | string scalar

Name of anonymized DICOM file to write, specified as a character vector or string scalar.

Data Types: char | string

### **fields** — Names of fields to preserve

cell array

Names of the fields to preserve, specified as a cell array of field names.

## **attributes — Names of the attributes to preserve**

structure

Names of the attributes to preserve, specified as a structure whose fields are attribute names. The structure values are the attribute values to preserve.

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `dicomanon('CT-MON02-16-ankle.dcm', 'CT-MON02-16-ankle_anon.dcm', 'UseVRHeuristic', false)`

## **WritePrivate — Write nonstandard attributes to the anonymized file**

false (default) | true

Write nonstandard attributes to the anonymized file, specified as the comma-separated pair consisting of 'WritePrivate' and false or true.

When set to true, then `dicomanon` includes private extensions in the file, which could compromise patient confidentiality.

Data Types: logical

## **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 comma-separated pair consisting of 'UseVRHeuristic' and true or false.

When 'UseVRHeuristic' is true (the default), then `dicomanon` instructs the parser to use a heuristic to help read certain noncompliant files which switch value representation (VR) modes incorrectly. A small number of compliant files will not be read correctly. `dicomanon` displays a warning if the heuristic is employed. Set 'UseVRHeuristic' to false to read these compliant files. Compliant files are always written.

Data Types: logical

## Tips

- For information about the fields that will be modified or removed, see DICOM Supplement 55 from <https://www.dicomstandard.org/>.

## See Also

`dicomdict` | `dicomdisp` | `dicominfo` | `dicomlookup` | `dicomread` | `dicomuid` | `dicomwrite`

**Introduced before R2006a**

## dicomCollection

Gather details about related series of DICOM files

### Syntax

```
collection = dicomCollection(directory)
collection = dicomCollection(directory, 'IncludeSubfolders', TF)
collection = dicomCollection(DICOMDIR)
```

### Description

`collection = dicomCollection(directory)` gathers details about the DICOM files contained in `directory` and returns them in the table `collection`. The `dicomCollection` function aggregates details by DICOM series, using the value of the `SeriesInstanceUID` metadata field in each file to determine series membership. A DICOM series is a logically related set of images from an imaging operation.

`collection = dicomCollection(directory, 'IncludeSubfolders', TF)` recursively searches for DICOM files below `directory` when `TF` is true (the default). When `TF` is false, `dicomCollection` only within `directory`.

`collection = dicomCollection(DICOMDIR)` gathers details about the DICOM files referenced in the DICOM directory file `DICOMDIR`.

### Examples

#### Gather Details from DICOM Files in Sample Image Folder

Gather information about the DICOM files in the Image Processing Toolbox sample image folder.

```
details = dicomCollection(fullfile(matlabroot, 'toolbox/images/imdata'))
```

```
details=5×14 table
```

```
StudyDateTime
```

```
SeriesDateTime
```

```
PatientName
```

```
PatientS
```

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

### Gather Details About DICOM Files from DICOMDIR File

Gather information about DICOM files in a folder from a DICOMDIR file.

```
details = dicomCollection(fullfile(matlabroot, 'toolbox/images/imdata/DICOMDIR'))
```

details=4x14 table

	StudyDateTime	SeriesDateTime	PatientName	PatientSex	Modal
s1	30-Apr-1993 11:27:24	' '	"Anonymized"	" "	"C"
s2	30-Jan-1994 11:25:01	' '	"Anonymized"	" "	"U"
s3	03-Oct-2011 19:18:11	' '	" "	" "	"M"
s4	03-Oct-2011 19:18:11	' '	" "	" "	"M"

## Input Arguments

### directory — Folder containing DICOM files

string scalar | character vector

Name of a folder containing DICOM files, specified as a string scalar or character vector.

Example: `details = dicomCollection(fullfile(matlabroot, 'toolbox/images/imdata'))`

Data Types: char | string

### DICOMDIR — DICOM directory file

character vector | string scalar

DICOM directory file, specified as a string scalar or character vector.

A DICOM directory file (DICOMDIR) is a special DICOM file that serves as a directory to a collection of DICOM files stored on removable media, such as CD/DVD ROMs. When devices write DICOM files to removable media, they typically write a DICOMDIR file on the disk to serve as a list of the disk contents.

```
Example: details = dicomCollection(fullfile(matlabroot,'toolbox/  
images/imdata/DICOMDIR'))
```

Data Types: char | string

## Output Arguments

### **collection** — Metadata from DICOM files

table

Metadata from DICOM files, returned as a table. The `dicomCollection` function aggregates the information by DICOM series.

## See Also

**DICOM Browser** | `dicominfo` | `dicomread` | `dicomreadVolume`

**Introduced in R2017b**



# 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 string scalar or 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` | `dicomdisp` | `dicominfo` | `dicomlookup` | `dicomread` | `dicomuid` | `dicomwrite`

**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 string scalar or 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

# 1 Functions — Alphabetical List

---

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	[DER3
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	[1993
0000486	0	(0008,0021)	DA	10 bytes	- SeriesDate	[1993
0000504	0	(0008,0023)	DA	10 bytes	- ContentDate	[1993
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	[CT0
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	*Bin
0001060	0	(0028,0120)	US	2 bytes	- PixelPaddingValue	*Bin
0001070	0	(0028,1050)	DS	4 bytes	- WindowCenter	[1024
0001082	0	(0028,1051)	DS	4 bytes	- WindowWidth	[4095
0001094	0	(0028,1052)	DS	6 bytes	- RescaleIntercept	[-1024
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 | string scalar

Name of DICOM file, specified as a string scalar or character vector .

Data Types: char | string

## 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 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) | string scalar | character vector

Name of DICOM data dictionary, specified as a string scalar or 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 | string

### **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

`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( ____,Name,Value)
```

## Description

`info = dicominfo(filename)` reads the metadata from the compliant Digital Imaging and Communications in Medicine (DICOM) file, `filename`.

`info = dicominfo(filename,'dictionary',D)` reads the DICOM message by using the data dictionary file, `D`.

`info = dicominfo( ____,Name,Value)` provides additional options to the parser using `Name,Value` pairs. You can specify multiple name-value pairs.

## 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 12:06:43'
    FileSize: 525436
    Format: 'DICOM'
    FormatVersion: 3
```

```
        Width: 512
        Height: 512
        BitDepth: 16
        ColorType: 'grayscale'
FileMetaInformationGroupLength: 192
  FileMetaInformationVersion: [2x1 uint8]
  MediaStorageSOPClassUID: '1.2.840.10008.5.1.4.1.1.7'
  MediaStorageSOPInstanceUID: '1.2.840.113619.2.1.2411.1031152382.365.1.736169244'
  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: '1.2.840.113619.2.1.2411.1031152382.365.1.736169244'
  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: [1x1 struct]
  StationName: 'CT010C0'
  StudyDescription: 'RT ANKLE'
PhysicianReadingStudy: [1x1 struct]
  OperatorName: [1x1 struct]
  ManufacturerModelName: 'GENESIS_ZEUS'
  PatientGroupLength: 18
  PatientName: [1x1 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
```

## Input Arguments

### **filename** — Name of DICOM file

character vector | string scalar

Name of DICOM file, specified as a character vector or string scalar.

Data Types: char | string

### **D** — Data dictionary file

'dicom-dict.mat' | character vector | string scalar

Data dictionary file, specified as a character vector or string scalar. The file in D must be on the MATLAB search path. The default file is `dicom-dict.mat`.

Data Types: char | string

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `dicominfo('CT-MON02-16-ankle.dcm', 'UseVRHeuristic', false)`

## **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 comma-separated pair consisting of 'UseVRHeuristic' and `true` or `false`.

When set to `true` (the default), `dicomread` uses a heuristic to help read certain noncompliant DICOM files that switch VR modes incorrectly. `dicomread` displays a warning if the heuristic is used. A small number of compliant files will not be read correctly. Set `UseVRHeuristic` to `false` to read these compliant files.

Data Types: `logical`

## **UseDictionaryVR — Conform data types to data dictionary**

`false (default) | true`

Conform data types in `info` 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 `UseDictionaryVR` is `false` (the default), `dicominfo` issues a warning when they do not agree. Specify `UseDictionaryVR` as `true` when the warning is issued and providing `info` to `dicomwrite` causes errors.

Data Types: `logical`

## **Output Arguments**

### **info — DICOM metadata**

`struct`

DICOM metadata, returned as a `struct`.

## **See Also**

`dicomanon` | `dicomdict` | `dicomdisp` | `dicomlookup` | `dicomread` | `dicomuid` | `dicomwrite`

**Introduced before R2006a**

## dicomlookup

Find attribute in DICOM data dictionary

### Syntax

```
nameOut = dicomlookup(group,element)
[groupOut,elementOut] = dicomlookup(name)
```

### Description

`nameOut = dicomlookup(group,element)` looks into the current DICOM data dictionary for the attribute with the specified `group` and `element` tags. `dicomlookup` returns the name of the attribute.

`[groupOut,elementOut] = dicomlookup(name)` looks into the current DICOM data dictionary for the attribute specified by name and returns the `group` and `element` tags associated with the attribute.

### 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'
```

## Input Arguments

### **group** — DICOM group tag

positive integer decimal | character vector | string scalar

DICOM group tag, specified as a positive integer decimal number or a character vector or string scalar that contains a hexadecimal value.

Example: 40

Example: '7FE0' or "7FE0"

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

### **element** — DICOM element tag

positive integer decimal | character vector | string scalar

DICOM group tag, specified as a positive integer decimal number or a character vector or string scalar that contains a hexadecimal value. `element` and `group` must represent the same type of value:

- If `group` is a positive integer, then `element` is also a positive integer.

If `group` is a character vector or string scalar that contains a hexadecimal value, then `element` is either a character vector or a string scalar that contains a hexadecimal value.

Example: 4

Example: '0010' or "0010"

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

### **name** — DICOM attribute name

character vector | string scalar

DICOM attribute name, specified as a character vector or string scalar.

Example: 'PhotometricInterpretation' or "PhotometricInterpretation"

Data Types: char | string

## Output Arguments

### **groupOut** — Returned DICOM group tag

positive integer decimal

Returned DICOM group tag, returned as a positive integer decimal number.

Data Types: double

### **elementOut** — Returned DICOM element tag

positive integer decimal

Returned DICOM group tag, returned as a positive integer decimal number.

Data Types: double

### **nameOut** — Returned DICOM attribute name

character vector

Returned DICOM attribute name, returned as a character vector.

Data Types: char

## See Also

dicomanon | dicomdict | dicomdisp | dicominfo | dicomread | dicomuid | dicomwrite

**Introduced in R2006b**

## dicomread

Read DICOM image

### Syntax

```
X = dicomread(filename)
X = dicomread(info)
X = dicomread( ____, 'frames', f)
X = dicomread( ____, Name, Value)

[X, cmap] = dicomread( ____ )
[X, cmap, alpha] = dicomread( ____ )
[X, cmap, alpha, overlays] = dicomread( ____ )
```

### Description

`X = dicomread(filename)` reads the image data from the compliant Digital Imaging and Communications in Medicine (DICOM) file `filename`. To read a group of DICOM files that contain a series of images that comprise a volume, use `dicomreadVolume`.

`X = dicomread(info)` reads DICOM image data from the message referenced in the DICOM metadata structure `info`.

`X = dicomread( ____, 'frames', f)` reads only the frames specified by `f` from the image.

`X = dicomread( ____, Name, Value)` reads DICOM image data using `Name, Value` pairs to configure the parser.

`[X, cmap] = dicomread( ____ )` also returns the colormap, `cmap`.

`[X, cmap, alpha] = dicomread( ____ )` also returns `alpha`, an alpha channel matrix for `X`.

`[X, cmap, alpha, overlays] = dicomread( ____ )` also returns any overlays from the DICOM file.

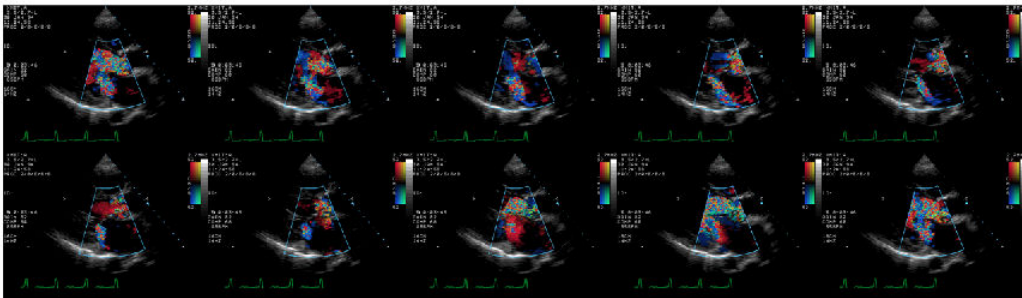


## Examples

### 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, []);
```



## Input Arguments

### **filename** — Name of DICOM file

character vector | string scalar

Name of DICOM file, specified as a character vector or string scalar.

Data Types: `char` | `string`

### **info** — DICOM metadata

struct

DICOM metadata, specified as a structure. The `info` structure is produced by the `dicominfo` function.

### **f** — Frames to read

'all' (default) | integer | vector of integers

Frames to read, specified as an integer scalar, a vector of integers, or 'all'. When `f` is numeric, `dicomread` reads only the specified frame numbers from the image. By default, `dicomread` reads all frames of the DICOM 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `dicomread('CT-MON02-16-ankle.dcm', 'UseVRHeuristic', false)`

### **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 comma-separated pair consisting of 'UseVRHeuristic' and `true` or `false`.

When set to `true` (the default), `dicomread` uses a heuristic to help read certain noncompliant DICOM files that switch VR modes incorrectly. `dicomread` displays a

warning if the heuristic is used. A small number of compliant files will not be read correctly. Set `UseVRHeuristic` to `false` to read these compliant files.

Data Types: `logical`

## Output Arguments

### **X — DICOM image**

*m*-by-*n* matrix | *m*-by-*n*-by-3 array | 4-D array

DICOM image, returned as one of the following.

- An *m*-by-*n* matrix representing a single-frame grayscale image or an indexed image
- An *m*-by-*n*-by-3 array representing a single-frame truecolor (RGB) image
- A 4-D array representing a multiframe image.

Data Types: `int8` | `int16` | `uint8` | `uint16`

### **cmap — Color map**

*c*-by-3 matrix | `[]`

Color map associated with image X.

- If X is an indexed image, then `cmap` is returned as *c*-by-3 matrix. There are *c* colors in the color map, each represented by a red, green, and blue pixel value.
- If X is a grayscale or true-color image, then `cmap` is empty (`[]`).

Data Types: `double`

### **alpha — Alpha channel matrix**

*m*-by-*n* matrix of nonnegative integers | 4-D array of nonnegative integers

Alpha channel matrix for image X, returned as an array of nonnegative integers. The values of `alpha` are 0 if the pixel is opaque; otherwise they are row indices into `cmap`. The RGB value in `cmap` 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. `alpha` has the same data type as X.

Data Types: `int8` | `int16` | `uint8` | `uint16`

## **overlays — Overlays**

binary *m*-byn matrix | binary 4-D array | []

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, then `overlays` is a 4-D multiframe image. If no overlays are in the file, then `overlays` is empty ( []).

Data Types: `logical`

## **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` | `dicomreadVolume` | `dicomuid` | `dicomwrite`

**Introduced before R2006a**

## dicomreadVolume

Construct volume from directory of DICOM images

### Syntax

```
[V,spatial,dim] = dicomreadVolume(source)
[V,spatial,dim] = dicomreadVolume(sourcetable)
[V,spatial,dim] = dicomreadVolume(sourcetable, rowname)
```

### Description

`[V,spatial,dim] = dicomreadVolume(source)` loads the 4-D DICOM volume `V` from `source`, which can be one of the following:

- Name of a folder containing DICOM files
- String array of filenames comprising the volume
- Cell array of character vectors containing filenames

`spatial` is a structure describing the location, resolution, and orientation of slices in the volume. `dim` specifies which real-world dimension (`X = 1`, `Y = 2`, `Z = 3`) has the largest amount of offset from the previous slice.

`dicomreadVolume` is useful when working with DICOM volumes because it reads the volumetric image data from each DICOM files, identifies the correct ordering of the images, and constructs 4-D volume from the data.

`[V,spatial,dim] = dicomreadVolume(sourcetable)` loads the volume from the `sourcetable`, which is a table returned by `dicomCollection`. The `sourcetable` argument must contain only one row.

`[V,spatial,dim] = dicomreadVolume(sourcetable, rowname)` loads the volume with the specified `rowname` from the multirow table `sourcetable` returned by `dicomCollection`. Use this syntax when `sourcetable` contains multiple rows.

## Examples

### Read Volume Data from DICOM Files

Read volume data from a sample folder of DICOM files.

```
[V,s,d] = dicomreadVolume(fullfile(matlabroot, 'toolbox/images/imdata/dog'));
```

## Input Arguments

### **source** — Volume data folder or files

string | character vector | string array | cell array of character vectors

Volume data folder or files, specified as a string scalar, string array, character vector, or cell array of character vectors.

Data Types: char | string | cell

### **sourcetable** — Collection of DICOM file metadata

table

Collection of DICOM file metadata, specified as a table returned by `dicomCollection`.

Data Types: table

### **rowname** — Name of table row to load

string | character vector

Name of table row to load, specified as a string scalar or character vector. The row is one of the rows in the multirow table returned by `dicomCollection`.

Data Types: char | string

## Output Arguments

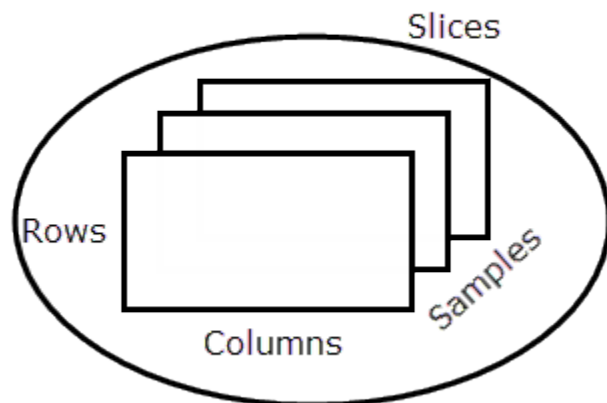
### **V** — 4-D DICOM volume

numeric array

4-D DICOM volume, returned as a numeric array.

The dimensions of `V` are `[rows, columns, samples, slices]` where `samples` is the number of color channels per voxel. For example, grayscale volumes have one sample, and RGB volumes have three samples. Use the `squeeze` function to remove any singleton dimensions, such as when `samples` is 1.

## DICOM 4-D Volume



### **spatial** — Location, resolution, and orientation

structure

Location, resolution, and orientation of slices in the volume, specified as a structure with the following fields. For more information, see part 3 of the DICOM standard, section C.7.6.2.

#### **Spatial Structure**

Fields	Description
PatientPositions	( <code>x,y,z</code> ) triplet of the first pixel in each slice, measured in millimeters from the origin of the scanner's coordinate system
PixelSpacings	Distance between neighboring rows and columns within each slice, in millimeters
PatientOrientations	Pair of direction cosine triplets of the rows and columns for each slice of the image



**dim — Dimension with largest offset from the previous slice**

1 | 2 | 3

Dimension with largest offset from the previous slice, returned as a numeric scalar 1, 2, or 3, where X = 1, Y = 2, and Z = 3.

**See Also****DICOM Browser** | [dicomCollection](#) | [dicominfo](#) | [dicomread](#)**Introduced in R2017b**

## **dicomuid**

Generate DICOM unique identifier

### **Syntax**

UID = dicomuid

### **Description**

UID = dicomuid returns a new DICOM unique identifier, UID, as a character vector.

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,cmap,filename)
dicomwrite( ____,meta_struct)
dicomwrite( ____,info)
dicomwrite( ____, 'ObjectType',IOD)
dicomwrite( ____, 'SOPClassUID',UID)
dicomwrite( ____,Name,Value)
status = dicomwrite( ____)
```

## Description

`dicomwrite(X,filename)` writes the binary, grayscale, or truecolor image `X` to the file `filename`, where `filename` specifies the name of the Digital Imaging and Communications in Medicine (DICOM) file to create.

`dicomwrite(X,cmap,filename)` writes the indexed image `X` with colormap `cmap`.

`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.

`dicomwrite( ____, 'ObjectType',IOD)` writes a file containing the necessary metadata for a particular type of DICOM Information Object (IOD). For the supported 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. For more information, see [Tips](#) on page 1-499.

`dicomwrite( ____, 'SOPClassUID', UID)` writes a file containing the necessary metadata for a particular type IOD, specified using a DICOM Unique Identifier (UID).

`dicomwrite( ____, Name, Value)` writes to a DICOM file using `Name`, `Value` pairs to affect how the file is written. You can specify multiple name-value pairs.

You can also use `Name`, `Value` pairs to specify optional metadata to write to the DICOM file. 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. Enclose each attribute name in quotes.

`status = dicomwrite( ____, info)` 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 to the `dicomwrite` function.

## Examples

### Write Data to DICOM File

Read a CT image from the sample DICOM file included with the toolbox.

```
X = dicomread('CT-MON02-16-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-MON02-16-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');
```

## Input Arguments

### **X — DICOM image**

*m*-by-*n* matrix | *m*-by-*n*-by-3 array | 4-D array

DICOM image, specified as one of the following.

- An *m*-by-*n* matrix representing a single-frame grayscale image or an indexed image
- An *m*-by-*n*-by-3 array representing a single-frame truecolor (RGB) image
- A 4-D array representing a multiframe image.

Data Types: `int8` | `int16` | `uint8` | `uint16`

### **cmap — Color map**

*c*-by-3 matrix | []

Color map associated with indexed image *X*, specified as *c*-by-3 matrix. There are *c* colors in the color map, each represented by a red, green, and blue pixel value.

Data Types: `double`

### **filename — Name of DICOM file**

character vector | string scalar

Name of DICOM file to write to, specified as a specified as a character vector or string scalar.

Data Types: `char` | `string`

### **meta\_struct — Optional metadata or file options**

struct

Optional metadata or file options, specified as a 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.

### **info — Metadata produced by `dicominfo` function**

struct

Metadata produced by `dicominfo` function, specified as a structure.

## **IOD — DICOM Information Object**

'Secondary Capture Image Storage' (default) | 'CT Image Storage' | 'MR Image Storage'

DICOM Information Object, specified as 'Secondary Capture Image Storage', 'CT Image Storage', or 'MR Image Storage'.

Data Types: char | string

## **UID — DICOM Unique Identifier**

character vector | string scalar

DICOM Unique Identifier corresponding to an IOD, specified as a character vector or string scalar.

Data Types: char | string

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'CompressionMode', 'JPEG lossless'

## **CompressionMode — Compression mode**

'None' (default) | 'JPEG lossless' | 'JPEG lossy' | ...

Compression mode to use when storing the image, specified as the comma-separated pair consisting of 'CompressionMode' and one of the following.

- 'None'
- 'JPEG lossless'
- 'JPEG lossy'
- 'JPEG2000 lossy'
- 'JPEG2000 lossless'
- 'RLE'

## **CreateMode — Method used for creating data**

'Create' (default) | 'Copy'

Method used for creating data to put in the new file, specified as the comma-separated pair consisting of 'CreateMode' and one of the following.

- 'Create' — Verify input values and generate missing data values.
- 'Copy' — Copy all values from the input and do not generate missing values.

For help selecting a creation method, see Tips on page 1-499.

### **Dictionary — Name of DICOM data dictionary**

character vector | string scalar

Name of DICOM data dictionary, specified as the comma-separated pair consisting of 'Dictionary' and a character vector or string scalar. The default file is `dicom-dict.mat`.

### **Endian — Byte ordering**

'Little' (default) | 'Big'

Byte ordering of the file, specified as the comma-separated pair consisting of 'Endian' and 'Little' or 'Big'.

---

**Note** If VR is set to 'Explicit', then Endian must be 'Big'. `dicomwrite` ignores this value if `CompressionMode` or `TransferSyntax` is set.

---

### **MultiframeSingleFile — Write multiframe image to one file**

true (default) | false

Write multiframe image to one file, specified as the comma-separated pair consisting of 'MultiframeSingleFile' and true or false. When true, one file is created regardless of how many frames X contains. When false, one file is written for each frame in the image.

Data Types: logical

### **TransferSyntax — Transfer syntax**

character vector | string scalar

Transfer syntax, specified as the comma-separated pair consisting of 'TransferSyntax' and a character vector or string scalar. `TransferSyntax` is a UID that encodes values for the Endian, VR, and `CompressionMode` options.

---

**Note** If you specify a transfer syntax, then `dicomwrite` ignores any values specified for the `Endian`, `VR`, and `CompressionMode` options.

---

## **UseMetadataBitDepths — Preserve metadata values**

`false` (default) | `true`

Preserve the metadata values `'BitStored'`, `'BitsAllocated'`, and `'HighBit'`, specified as the comma-separated pair consisting of `'UseMetadataBitDepths'` and `false` or `true`. When `true`, `dicomwrite` preserves existing values. When `false` (default), `dicomwrite` computes these values based on the datatype of the pixel data. When `CreateMode` is `'Create'`, `dicomwrite` ignores this field.

Data Types: `logical`

## **VR — Write two-letter value representation (VR) code to file**

`'implicit'` (default) | `'explicit'`

Write two-letter value representation (VR) code to file, specified as the comma-separated pair consisting of `'VR'` and one of the following.

- `'implicit'` — Infer from data dictionary.
- `'explicit'` — Write VR to file.

---

**Note** If you specify the `Endian` value as `'Big'`, then you must specify `VR` as `'explicit'`.

---

## **WritePrivate — Write private data to file**

`false` (default) | `true`

Write private data to file, specified as the comma-separated pair consisting of `'WritePrivate'` and `false` or `true`.

Data Types: `logical`

# **Output Arguments**

## **status — Status of attributes**

`struct` | []



Status of attributes, returned as a structure. `status` contains information about the metadata and the descriptions used to generate the DICOM file. If no metadata was specified, `dicomwrite` returns an empty matrix (`[]`).

The `status` structure 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.

## 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 (for example, 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 IODs. 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**

# displayChart

Display Imatest® eSFR chart with overlaid regions of interest

## Syntax

```
displayChart(chart)
displayChart(chart,Name,Value)
```

## Description

`displayChart(chart)` displays an Imatest® eSFR chart with overlaid rectangles indicating the slanted edge, gray patch, and color patch ROIs.

`displayChart(chart,Name,Value)` displays an eSFR chart with additional parameters controlling aspects of the chart display.

## Examples

### Display Color Patch ROIs on an eSFR Chart

Read an image of an eSFR chart into the workspace.

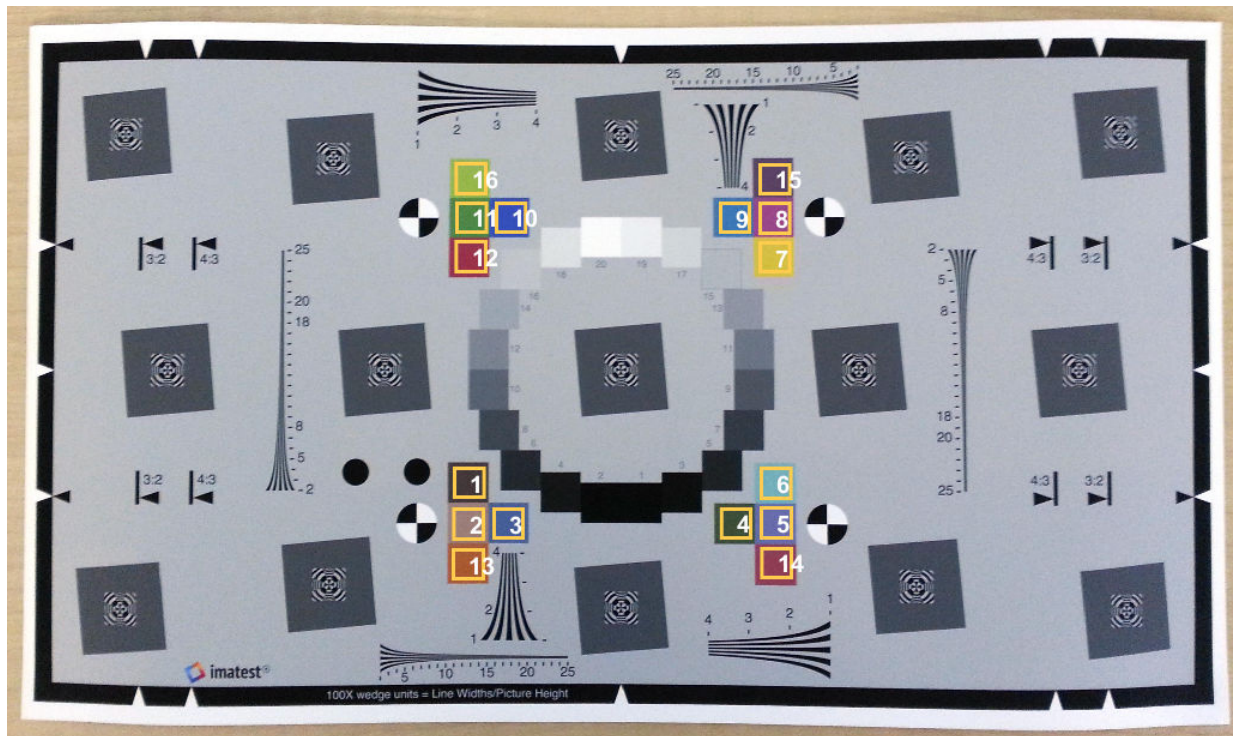
```
I = imread('eSFRTestImage.jpg');
```

Create an `esfrChart` object that stores information about the test chart.

```
chart = esfrChart(I);
```

Display only the color patch ROIs. To accomplish this, turn off the display of slanted edge ROIs, gray patch ROIs, and registration points.

```
displayChart(chart,'displayEdgeROIs',false,'displayGrayROIs',false,'displayRegistrationROIs',false);
```



## Input Arguments

### chart — eSFR chart

esfrChart object

eSFR chart, specified as an esfrChart object.

## 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 quotes. You can specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

Example: `displayChart(myChart, 'displayEdgeROIs', false)` turns off the overlay of slanted edge ROIs.

### **displayEdgeROIs — Display slanted edge ROIs**

`true (default) | false`

Display slanted edge ROIs, specified as the comma-separated pair consisting of 'displayEdgeROIs' and true or false. When `displayEdgeROIs` is true, the 60 slanted-edge bounding boxes are overlaid on the image in pale yellow.

Data Types: `logical`

### **displayGrayROIs — Display gray patch ROIs**

`true (default) | false`

Display gray patch ROIs, specified as the comma-separated pair consisting of 'displayGrayROIs' and true or false. When `displayGrayROIs` is true, the 20 gray patch bounding boxes are overlaid on the image in blue.

Data Types: `logical`

### **displayColorROIs — Display color patch ROIs**

`true (default) | false`

Display color patch ROIs, specified as the comma-separated pair consisting of 'displayColorROIs' and true or false. When `displayColorROIs` is true, the 16 color patch bounding boxes are overlaid on the image in dark yellow.

Data Types: `logical`

### **displayRegistrationPoints — Display registration points**

`true (default) | false`

Display registration points, specified as the comma-separated pair consisting of 'displayRegistrationPoints' and true or false. When `displayRegistrationPoints` is true, the four registration points are indicated with a red diamond overlay.

Data Types: `logical`

### **Parent — Axes handle of displayed image object**

`axes handle`

Axes handle of the displayed image object, specified as the comma-separated pair consisting of 'Parent' and an axes handle. Parent specifies the parent of the image object created by `displayChart`.

## See Also

### Functions

`measureChromaticAberration` | `measureColor` | `measureIlluminant` |  
`measureNoise` | `measureSharpness`

### Objects

`esfrChart`

**Introduced in R2017b**

# displayColorPatch

Display visual color reproduction as color patches

## Syntax

```
displayColorPatch(colorTable)
displayColorPatch(colorTable,Name,Value)
```

## Description

`displayColorPatch(colorTable)` displays measured and reference colors, `colorTable`, for color patch regions of interest (ROIs) in a test chart. The measured color values are displayed as squares surrounded by a thick boundary of the corresponding reference color.

`displayColorPatch(colorTable,Name,Value)` displays measured color values with additional parameters to control aspects of the display.

## Examples

### Display Color Patch Diagram from Color Accuracy Measurements

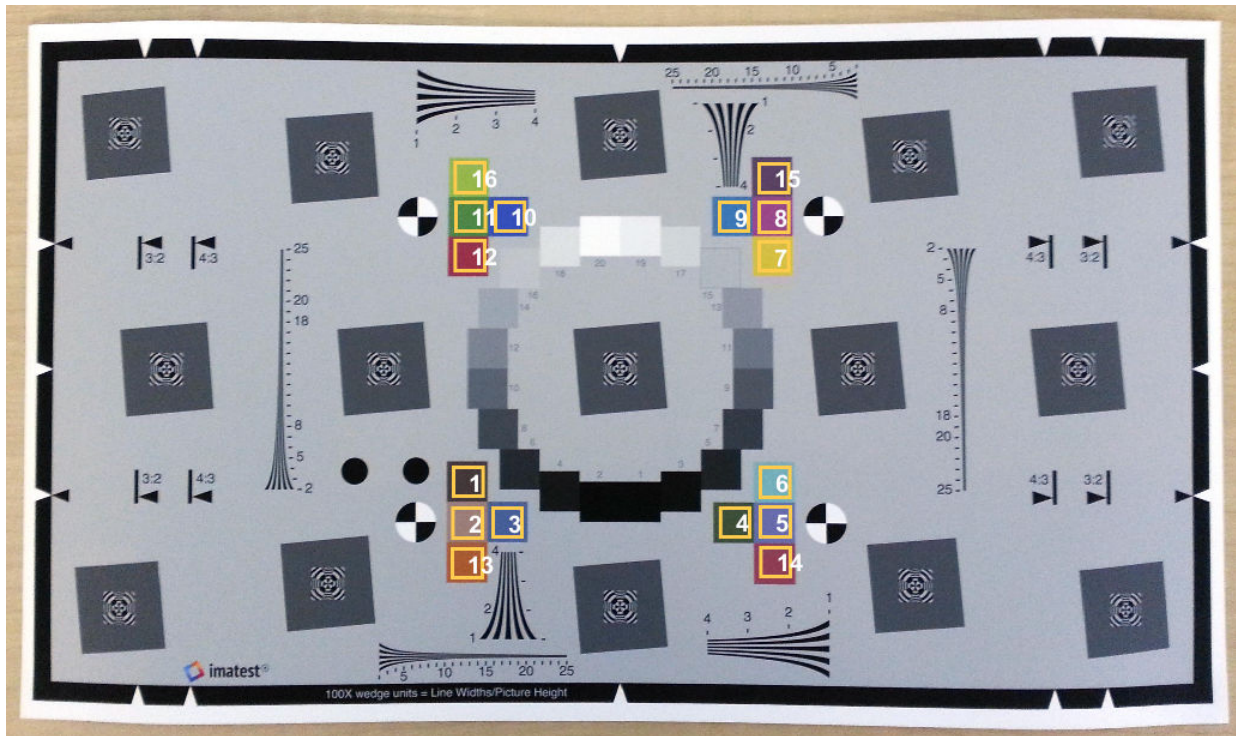
This example shows how to display the color patch diagram from measurements of color accuracy on an Imatest® eSFR chart.

Read an image of an eSFR chart into the workspace.

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

Create an `esfrChart` object. Display the chart, highlighting the 16 color patches.

```
chart = esfrChart(I);
displayChart(chart,'displayEdgeROIs',false, ...
    'displayGrayROIs',false,'displayRegistrationPoints',false)
```



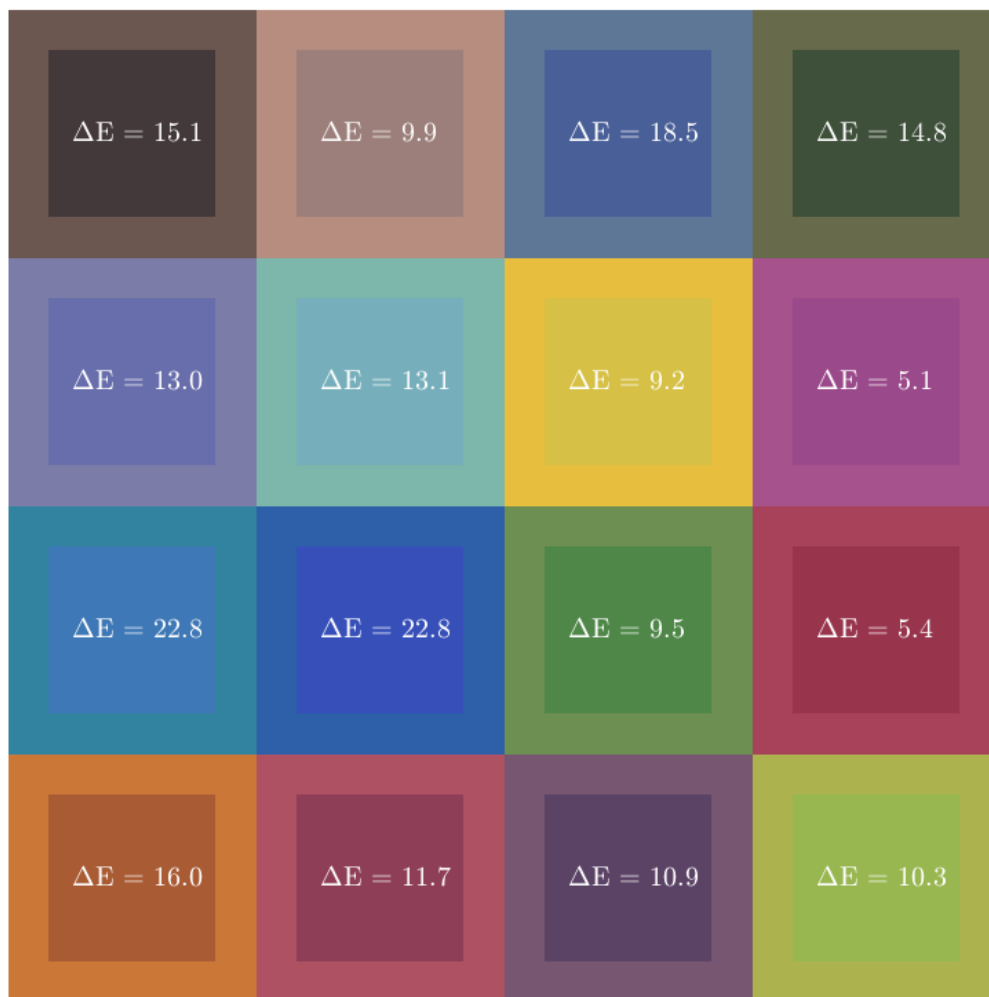
Measure the color in all color patch ROIs.

```
colorTable = measureColor(chart);
```

Display the color accuracy measurements without the ROI index overlay. Each square color patch is the measured color, and the thick surrounding border is the reference color for that ROI. The color accuracy measurement is displayed as  $\Delta E$ , the Euclidean distance between measured and reference colors in CIE 1976  $L^*a^*b^*$  color space. More accurate colors have a smaller  $\Delta E$ .

```
displayColorPatch(colorTable, 'displayROIIndex', false)
```





## Input Arguments

### colorTable — Color values

color table

Color values in each color patch, specified as an  $m$ -by-8 color table, where  $m$  is the number of patches. The eight columns represent these variables:

Variable	Description
ROI	Index of the sampled ROI. The value of ROI is an integer in the range [1, 16]. The indices match the ROI numbers displayed by <code>displayChart</code> .
Measured_R	Mean value of red channel pixels in an ROI. <code>Measured_R</code> is a scalar of the same data type as <code>chart.Image</code> , which can be of type <code>single</code> , <code>double</code> , <code>uint8</code> , or <code>uint16</code> .
Measured_G	Mean value of green channel pixels in an ROI. <code>Measured_G</code> is a scalar of the same data type as <code>chart.Image</code> .
Measured_B	Mean value of blue channel pixels in an ROI. <code>Measured_B</code> is a scalar of the same data type as <code>chart.Image</code> .
Reference_L	Reference L* value corresponding to the ROI. <code>Reference_L</code> is a scalar of type <code>double</code> .
Reference_a	Reference a* value corresponding to the ROI. <code>Reference_a</code> is a scalar of type <code>double</code> .
Reference_b	Reference b* value corresponding to the ROI. <code>Reference_b</code> is a scalar of type <code>double</code> .
Delta_E	Euclidean color distance between the measured and reference color values, as outlined in CIE 1976.

To obtain a color table, use the `measureColor` function.

### 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `displayColorPatch(myColorTable, 'displayROIIndex', false)` turns off the display of the ROI indices.

### **displayROIIndex — Display ROI index labels**

true (default) | false

Display ROI index labels, specified as the comma-separated pair consisting of 'displayROIIndex' and true or false. When `displayROIIndex` is true, then `displayColorPatch` overlays color patch ROI index labels on the displayed color patches. The indices match the ROI numbers displayed by `displayChart`.

Data Types: logical

### **displayDeltaE — Display Delta\_E values**

true (default) | false

Display Delta\_E values, specified as the comma-separated pair consisting of 'displayDeltaE' and true or false. When `displayDeltaE` is true, `displayColorPatch` overlays the values of Delta\_E on the displayed color patches.

Data Types: logical

### **Parent — Axes handle of displayed image object**

axes handle

Axes handle of the displayed image object, specified as the comma-separated pair consisting of 'Parent' and an axes handle. Parent specifies the parent of the image object created by `displayColorPatch`.

## **See Also**

### **Functions**

`displayChart` | `measureColor` | `plotChromaticity`

### **Objects**

`esfrChart`

**Introduced in R2017b**

## dnCNNLayers

Get denoising convolutional neural network layers

### Syntax

```
layers = dnCNNLayers  
layers = dnCNNLayers(Name, Value)
```

### Description

`layers = dnCNNLayers` returns layers of the denoising convolutional neural network (DnCNN) for grayscale images.

This function requires that you have Deep Learning Toolbox.

`layers = dnCNNLayers(Name, Value)` returns layers of the denoising convolutional neural network with additional name-value parameters specifying network architecture.

### Examples

#### Get Layers of Image Denoising Network

Get layers of the image denoising convolutional neural network, 'DnCNN'. Request the default number of layers, which returns 20 convolution layers.

```
layers = dnCNNLayers
```

```
layers =  
1x59 Layer array with layers:
```

1	'InputLayer'	Image Input	50x50x1 images
2	'Conv1'	Convolution	64 3x3x1 convolutions with str
3	'ReLU1'	ReLU	ReLU
4	'Conv2'	Convolution	64 3x3x64 convolutions with str
5	'BNorm2'	Batch Normalization	Batch normalization with 64 ch

6	'ReLU2'	ReLU	ReLU
7	'Conv3'	Convolution	64 3x3x64 convolutions with str
8	'BNorm3'	Batch Normalization	Batch normalization with 64 cha
9	'ReLU3'	ReLU	ReLU
10	'Conv4'	Convolution	64 3x3x64 convolutions with str
11	'BNorm4'	Batch Normalization	Batch normalization with 64 cha
12	'ReLU4'	ReLU	ReLU
13	'Conv5'	Convolution	64 3x3x64 convolutions with str
14	'BNorm5'	Batch Normalization	Batch normalization with 64 cha
15	'ReLU5'	ReLU	ReLU
16	'Conv6'	Convolution	64 3x3x64 convolutions with str
17	'BNorm6'	Batch Normalization	Batch normalization with 64 cha
18	'ReLU6'	ReLU	ReLU
19	'Conv7'	Convolution	64 3x3x64 convolutions with str
20	'BNorm7'	Batch Normalization	Batch normalization with 64 cha
21	'ReLU7'	ReLU	ReLU
22	'Conv8'	Convolution	64 3x3x64 convolutions with str
23	'BNorm8'	Batch Normalization	Batch normalization with 64 cha
24	'ReLU8'	ReLU	ReLU
25	'Conv9'	Convolution	64 3x3x64 convolutions with str
26	'BNorm9'	Batch Normalization	Batch normalization with 64 cha
27	'ReLU9'	ReLU	ReLU
28	'Conv10'	Convolution	64 3x3x64 convolutions with str
29	'BNorm10'	Batch Normalization	Batch normalization with 64 cha
30	'ReLU10'	ReLU	ReLU
31	'Conv11'	Convolution	64 3x3x64 convolutions with str
32	'BNorm11'	Batch Normalization	Batch normalization with 64 cha
33	'ReLU11'	ReLU	ReLU
34	'Conv12'	Convolution	64 3x3x64 convolutions with str
35	'BNorm12'	Batch Normalization	Batch normalization with 64 cha
36	'ReLU12'	ReLU	ReLU
37	'Conv13'	Convolution	64 3x3x64 convolutions with str
38	'BNorm13'	Batch Normalization	Batch normalization with 64 cha
39	'ReLU13'	ReLU	ReLU
40	'Conv14'	Convolution	64 3x3x64 convolutions with str
41	'BNorm14'	Batch Normalization	Batch normalization with 64 cha
42	'ReLU14'	ReLU	ReLU
43	'Conv15'	Convolution	64 3x3x64 convolutions with str
44	'BNorm15'	Batch Normalization	Batch normalization with 64 cha
45	'ReLU15'	ReLU	ReLU
46	'Conv16'	Convolution	64 3x3x64 convolutions with str
47	'BNorm16'	Batch Normalization	Batch normalization with 64 cha
48	'ReLU16'	ReLU	ReLU
49	'Conv17'	Convolution	64 3x3x64 convolutions with str

50	'BNorm17'	Batch Normalization	Batch normalization with 64 channels
51	'ReLU17'	ReLU	ReLU
52	'Conv18'	Convolution	64 3x3x64 convolutions with stride 2
53	'BNorm18'	Batch Normalization	Batch normalization with 64 channels
54	'ReLU18'	ReLU	ReLU
55	'Conv19'	Convolution	64 3x3x64 convolutions with stride 2
56	'BNorm19'	Batch Normalization	Batch normalization with 64 channels
57	'ReLU19'	ReLU	ReLU
58	'Conv20'	Convolution	1 3x3x64 convolutions with stride 2
59	'FinalRegressionLayer'	Regression Output	mean-squared-error

You can train a custom image denoising network by providing these layers and a `denoisingImageDatastore` to `trainNetwork`.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'NetworkDepth', 15`

#### **NetworkDepth** — Number of convolution layers

20 (default) | positive integer

Number of convolution layers, specified as a positive integer with value greater than or equal to 3.

Example: 15

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

## Output Arguments

#### **Layers** — Network layers

vector of `Layer` objects

Denoising convolutional neural network layers, returned as a vector of `Layer` objects.

## Tips

- The DnCNN network can detect noise and other high-frequency image artifacts. For example, you can train the DnCNN network to increase image resolution or remove JPEG compression artifacts. The example “JPEG Image Deblocking Using Deep Learning” shows how to train a DnCNN to reduce JPEG compression artifacts in an image.

## References

- [1] Zhang, K., W. Zuo, Y. Chen, D. Meng, and L. Zhang. "Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising." *IEEE Transactions on Image Processing*. Vol. 26, Issue 7, 2017, pp. 3142-3155.

## See Also

`denoiseImage` | `denoisingImageDatastore` | `denoisingNetwork` | `trainNetwork`

## Topics

“JPEG Image Deblocking Using Deep Learning”  
“Train and Apply Denoising Neural Networks”

**Introduced in R2017b**

## dpxinfor

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 | string scalar

Name of a DPX file, specified as a string scalar or 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 | string

## Output Arguments

### **metadata** — Information about the DPX image data

structure

Information about the DPX image data, returned as a structure.

## 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 | string scalar

Name of a DPX file, specified as a string scalar or 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` | `string`

## 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

`dpxinfo`

**Introduced in R2015b**

# drawassisted

Create freehand ROI on image with assistance from image edges

## Syntax

```
h = drawassisted
h = drawassisted(hImage)
h = drawassisted( ____,Name,Value)
```

## Description

`h = drawassisted` begins interactive placement of a freehand region-of-interest (ROI) on the current axes. The `drawassisted` function uses the edges in the underlying image to "assist" the drawing process. The function returns `h`, a handle to an `images.roi.AssistedFreehand` object. You can modify the ROI interactively using your mouse. The ROI also supports a right-click context menu that controls aspects of its appearance and behavior.

`h = drawassisted(hImage)` begins interactive placement of a freehand ROI on the image specified by `hImage`.

`h = drawassisted( ____,Name,Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs. Unspecified name-value pairs are set to default values.

During interactive placement, click and release to place the first waypoint. A live segment of the freehand boundary follows the mouse cursor, snapping to nearby edges in the image. Waypoints are placed automatically at regular intervals, or by a manual left-click. To delete the last waypoint, use the **Backspace** key. To override the assisted segment, click and drag it. To close the region, double-click, right-click or click the first waypoint. To reshape the freehand region after the shape has been drawn, click the waypoints. No assistance is available in this mode.

You add additional waypoints after drawing via the context menu or by double-clicking on the edge of the freehand ROI. Remove a waypoint by right-clicking on that waypoint and selecting **Remove Waypoint** from the context menu.

## Examples

### Alpha Blend Source ROI into Target Image

Read an image into the workspace and display it.

```
figure
im = imread('peppers.png');
imshow(im)
```

Draw an assisted freehand ROI.

```
h = drawassisted();
```

Create a mask of the ROI you drew.

```
bw = createMask(h);
```

Create an alpha matrix that specifies the transparency of the source image at each pixel.

```
alphamat = imguifilter(single(bw),im,'DegreeOfSmoothing',2);
```

Obtain a target image, and resize the source image and the alpha matrix.

```
target = imread('fabric.png');
alphamat = imresize(alphamat,[size(target,1),size(target,2)]);
im = imresize(im,[size(target,1),size(target,2)]);
```

Alpha blend the source ROI into the target image.

```
fused = single(im).*alphamat + (1-alphamat).*single(target);
fused = uint8(fused);
imshow(fused)
```

## Input Arguments

### **hImage** — Image object on which to draw ROI

image object

Image object on which to draw ROI, specified as an image object.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `h = drawassisted('LineWidth',5);`

### Closed — Close freehand ROI

`true` (default) | `false`

Close the freehand ROI, specified the comma-separated pair consisting of `'Closed'` and the logical value `true` or `false`. If `true` (default), `drawassisted` closes the ROI by connecting the last waypoint drawn to the first waypoint drawn.

### Color — ROI color

`blue` (default) | `ColorSpec` value

ROI color, specified as the comma-separated pair consisting of `'Color'` and a MATLAB `ColorSpec` (`Color Specification`) value.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

### Deletable — ROI can be interactively deleted via context menu

`true` (default) | `false`

ROI can be interactively deleted via a context menu, specified as the comma-separated pair consisting of `'Deletable'` and a logical value of `true` or `false`. When the value is `true` (default), you can delete the ROI via the context menu. To disable this context menu item, set `'Deletable'` to `false`. When the value is set to `false`, you can still delete the ROI by calling the `delete` function, specifying the handle to the ROI as the input.

### FaceAlpha — Transparency of ROI face

`0.2` (default) | scalar

Transparency of ROI face, specified as the comma-separated pair consisting of `'FaceAlpha'` and a scalar value in the range `[0, 1]`. When the value is `1`, the ROI face is fully opaque. When the value is `0`, the ROI face is completely transparent.

**FaceSelectable — ROI face can capture clicks**`true (default) | false`

ROI face can capture clicks, specified as the comma-separated pair consisting of 'FaceSelectable' and the value `true` or `false`. When `true` (default), the ROI face can capture clicks. When `false`, the ROI face cannot capture clicks.

**HandleVisibility — Visibility of ROI handle in Children property of parent**`'on' (default) | 'off' | 'callback'`

Visibility of the ROI handle in the Children property of the parent, specified as the comma-separated pair consisting of 'HandleVisibility' and one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**Image — Image on which to draw ROI**`handle to Image object`

Image on which to draw ROI, specified as the comma-separated pair consisting of 'Image' and a handle to an Image object.

**InteractionsAllowed — Interactivity of ROI**`'all' (default) | 'none' | 'translate' | 'reshape'`

Interactivity of the ROI, specified as the comma-separated pair consisting of 'InteractionsAllowed' and one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.



Value	Description
'translate'	ROI can be translated (moved) within the drawing area but not reshaped. This value is not available with the <code>drawPoint</code> function.
'reshape'	ROI can be reshaped within the drawing area but not translated. This value is not available with the <code>drawPoint</code> function and the <code>drawRectangle</code> function.

**Label — ROI label**

' ' (default) | character vector | string

ROI label, specified as the comma-separated pair consisting of 'Label' and a character vector or string. By default, this property is set to an empty array ( ' ') and the ROI has no label.

**LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as the comma-separated pair consisting of 'LineWidth' and a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Parent — ROI parent**

Axes object

ROI parent, specified as the comma-separated pair consisting of 'Parent' and an Axes object. A UIAxes cannot be the parent of the ROI.

**Position — Position of ROI**

*n*-by-2 array

Position of the ROI, specified as the comma-separated pair consisting of 'Position' and an *n*-by-2 array of the form [x1 y1; ...; xn yn] where *n* is the number of vertices in the ROI. Each row specifies the position of a vertex in the ROI.

**Selected — Selection state of ROI**

false (default) | true

Selection state of the ROI, specified as the comma-separated pair consisting of 'Selected' and true or false. You can also set this value interactively. For example,

clicking the ROI selects it and sets this value to `true`. Similarly, if you **Ctrl-click** the ROI, deselects it and sets this value to `false`.

**SelectedColor — Color of ROI when selected**

'none' (default) | colorspec

Color of the ROI when selected, specified as the comma-separated pair consisting of 'SelectedColor' and a MATLAB ColorSpec (Color Specification). If 'SelectedColor' is set to 'none', then the value of Color defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**Smoothing — Smooth edge of freehand ROI**

1 (default) | nonnegative scalar

Smooth the edge of the freehand ROI using the standard deviation of the Gaussian smoothing kernel, specified as the comma-separated pair consisting of 'Smoothing' and a nonnegative scalar. The function uses this 'Smoothing' value to filter the x and y coordinates of the freehand ROI. The function defines the filter size as  $2*\text{ceil}(2*\text{Smoothing}) + 1$ .

**StripeColor — Color of ROI stripe**

'none' (default) | ColorSpec value

Color of the ROI stripe, specified as the comma-separated pair consisting of 'StripeColor' and a MATLAB ColorSpec (Color Specification) value. If you specify 'StripeColor', the ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**Tag — Tag to associate with ROI**

character vector | string scalar

Tag to associate with the ROI, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

### **UIContextMenu — Context menu**

ContextMenu object

Context menu, specified as the comma-separated pair consisting of 'UIContextMenu' and a `uicontextmenu` object. Specify this name-value pair to display a custom context menu when you right-click on the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

### **UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as the comma-separated pair consisting of 'UserData' and any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

### **Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as the comma-separated pair consisting of 'Visible' and one of the values in this table.

Value	Description
'on'	Specify this value to display the ROI (default).
'off'	Specify this value to hide the ROI without deleting it. You still can access the properties of an invisible ROI.

### **Waypoints — Control points to reshape the ROI**

1 (default) |  $n$ -by-1 logical vector

Control points to reshape the ROI, specified as the comma-separated pair consisting of 'Waypoints' and an  $n$ -by-1 logical vector, the same length as `Position`. Each element of the `Waypoints` vector indicates if the corresponding point in the `Position` array is a waypoint. Positions that are waypoints are set to `true` in `Waypoints`. Dragging a waypoint modifies the ROI between the specified waypoint and its immediate neighboring

waypoints. If empty, `drawfreehand` automatically generates `Waypoints` at locations of increased curvature.

## Output Arguments

### **h** — Assisted Freehand ROI

`images.roi.AssistedFreehand` object

Assisted Freehand ROI, returned as an `images.roi.AssistedFreehand` object.

## See Also

`drawcircle` | `drawellipse` | `drawline` | `drawpoint` | `drawpolygon` | `drawpolyline` | `drawrectangle` | `images.roi.AssistedFreehand`

**Introduced in R2018b**

# drawcircle

Create customizable circular ROI

## Syntax

```
h = drawcircle
h = drawcircle(ax)
h = drawcircle( ____, Name, Value)
```

## Description

`h = drawcircle` begins interactive placement of a circular region-of-interest (ROI) on the current axes. The function returns a handle to an `images.roi.Circle` object. You can modify the ROI interactively using your mouse. The ROI also supports a right-click context menu that controls aspects of its appearance and behavior.

`h = drawcircle(ax)` begins interactive placement of a circular ROI on the axes specified by `ax`.

`h = drawcircle( ____, Name, Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs. Unspecified name-value pairs are set to the default values.

During interactive placement, click and drag to draw the circular ROI. After the shape has been drawn, click and drag the waypoints to reshape the ROI.

## Examples

### Create Black Circular ROI

Read an image into the workspace and display it.

```
figure
imshow(imread('baby.jpg'))
```

Begin interactive placement of a black circular ROI, labeling it "My Circle."

```
h = drawcircle('Color','k','Label','My Circle');
```

Disable the ability to delete the ROI using the context menu.

```
h.Deletable = false;
```

### **Create Circular ROI Programmatically**

Read an image into the workspace and display it.

```
I = imread('baby.jpg');  
figure  
imshow(I)
```



Draw a circular ROI on the image, Use the 'Center' name-value pair to specify the location of the circle and the 'Radius' name-value pair to specify its size. Set the edge of the circle to be striped by specifying the 'StripeColor' name-value pair.

```
h = drawcircle('Center',[1000,1000],'Radius',500,'StripeColor','red');
```





Parent axes of the ROI, specified as an Axes object.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `h = drawcircle('Color','k','Label','My Circle');`

### Center — Center of ROI

`[x y]`

Center of the ROI, specified as the comma-separated pair consisting of `'Center'` and a 1-by-2 array of the form `[x y]`. The values `x` and `y` are the coordinates of the center point of the ROI. The value of this property changes automatically when you draw or move the ROI.

### Color — ROI color

`blue` (default) | `ColorSpec` value

ROI color, specified as the comma-separated pair consisting of `'Color'` and a MATLAB `ColorSpec` (`Color Specification`) value.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

### Deletable — ROI can be interactively deleted via context menu

`true` (default) | `false`

ROI can be interactively deleted via a context menu, specified as the comma-separated pair consisting of `'Deletable'` and a logical value of `true` or `false`. When the value is `true` (default), you can delete the ROI via the context menu. To disable this context menu item, set `'Deletable'` to `false`. When the value is set to `false`, you can still delete the ROI by calling the `delete` function, specifying the handle to the ROI as the input.

### DrawingArea — Area of axes in which you can interactively place ROI

`'auto'` (default) | `'unlimited'` | `[x,y,w,h]`

Area of the axes in which you can interactively place the ROI, specified as the comma-separated pair consisting of 'DrawingArea' and one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangle beginning at (x,y), with width w and height h.

#### FaceAlpha — Transparency of ROI face

0.2 (default) | scalar

Transparency of ROI face, specified as the comma-separated pair consisting of 'FaceAlpha' and a scalar value in the range [0, 1]. When the value is 1, the ROI face is fully opaque. When the value is 0, the ROI face is completely transparent.

#### FaceSelectable — ROI face can capture clicks

true (default) | false

ROI face can capture clicks, specified as the comma-separated pair consisting of 'FaceSelectable' and the value true or false. When true (default), the ROI face can capture clicks. When false, the ROI face cannot capture clicks.

#### HandleVisibility — Visibility of ROI handle in Children property of parent

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as the comma-separated pair consisting of 'HandleVisibility' and one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.

Value	Description
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**InteractionsAllowed — Interactivity of ROI**

'all' (default) | 'none' | 'translate' | 'reshape'

Interactivity of the ROI, specified as the comma-separated pair consisting of 'InteractionsAllowed' and one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.
'translate'	ROI can be translated (moved) within the drawing area but not reshaped. This value is not available with the <code>drawPoint</code> function.
'reshape'	ROI can be reshaped within the drawing area but not translated. This value is not available with the <code>drawPoint</code> function and the <code>drawRectangle</code> function.

**Label — ROI label**

' ' (default) | character vector | string

ROI label, specified as the comma-separated pair consisting of 'Label' and a character vector or string. By default, this property is set to an empty array ( ' ' ) and the ROI has no label.

**LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as the comma-separated pair consisting of 'LineWidth' and a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Parent — ROI parent**

Axes object

ROI parent, specified as the comma-separated pair consisting of 'Parent' and an Axes object. A UIAxes cannot be the parent of the ROI.

**Radius — Radius of circle**

nonnegative numeric scalar

Radius of the circle, specified as the comma-separated pair consisting of 'Radius' and a nonnegative numeric scalar. The value of this property changes automatically when you draw or move the ROI.

**Selected — Selection state of ROI**

false (default) | true

Selection state of the ROI, specified as the comma-separated pair consisting of 'Selected' and true or false. You can also set this value interactively. For example, clicking the ROI selects it and sets this value to true. Similarly, if you **Ctrl-click** the ROI, deselects it and sets this value to false.

**SelectedColor — Color of ROI when selected**

'none' (default) | colorspec

Color of the ROI when selected, specified as the comma-separated pair consisting of 'SelectedColor' and a MATLAB ColorSpec (Color Specification). If 'SelectedColor' is set to 'none', then the value of Color defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**StripeColor — Color of ROI stripe**

'none' (default) | ColorSpec value

Color of the ROI stripe, specified as the comma-separated pair consisting of 'StripeColor' and a MATLAB ColorSpec (Color Specification) value. If you specify 'StripeColor', the ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

## **Tag — Tag to associate with ROI**

character vector | string scalar

Tag to associate with the ROI, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

## **UIContextMenu — Context menu**

ContextMenu object

Context menu, specified as the comma-separated pair consisting of 'UIContextMenu' and a `uicontextmenu` object. Specify this name-value pair to display a custom context menu when you right-click on the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

## **UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as the comma-separated pair consisting of 'UserData' and any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

## **Vertices — Locations of points on the edge of ROI**

*n*-by-2 array

Locations of points on the edge of the ROI, returned as the comma-separated pair consisting of 'Vertices' and an *n*-by-2 array.

This parameter is read-only.

## **Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as the comma-separated pair consisting of 'Visible' and one of the values in this table.

Value	Description
'on'	Specify this value to display the ROI (default).
'off'	Specify this value to hide the ROI without deleting it. You still can access the properties of an invisible ROI.

## Output Arguments

### **h** — Circular ROI

`images.roi.Circle` object

Circular ROI, returned as an `images.roi.Circle` object.

## See Also

`drawassisted` | `drawellipse` | `drawfreehand` | `drawline` | `drawpoint` | `drawpolygon` | `drawpolyline` | `drawrectangle` | `images.roi.Circle`

**Introduced in R2018b**

## drawellipse

Create customizable elliptical ROI

### Syntax

```
h = drawellipse  
h = drawellipse(ax)  
h = drawellipse( ____, Name, Value)
```

### Description

`h = drawellipse` begins interactive placement of an elliptical region-of-interest (ROI) on the current axes. The function returns a handle to an `images.roi.Ellipse` object. You can modify the ROI interactively using your mouse. The ROI supports a context menu that controls aspects of its appearance and behavior.

`h = drawellipse(ax)` begins interactive placement of an ROI on the axes specified by `ax`.

`h = drawellipse( ____, Name, Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs. Unspecified name-value pairs are set to the default value.

During interactive placement, click and drag to draw the ROI. After the shape has been drawn, click and drag the waypoints to reshape the ROI.

### Examples

#### Create Red Elliptical ROI

Read an image into the workspace and display it.



```
figure  
imshow(imread('baby.jpg'))
```

Begin interactive placement of a red elliptical ROI.

```
h = drawellipse('Color', 'r');
```

Add a label to the ROI after placement.

```
h.Label = 'My Ellipse'
```

### **Create Elliptical ROI Programmatically**

Read an image into the workspace and display it.

```
I = imread('baby.jpg');  
figure  
imshow(I)
```



Draw an elliptical ROI on the image. Use the 'Center' name-value pair to specify the location of the ellipse and the 'SemiAxes' name-value pair to specify the shape of the ellipse. Set the edge of the ellipse to be a striped red line by specifying the 'StripeColor' name-value pair.

```
h = drawellipse('Center',[1000,1000],'SemiAxes',[500,250],'StripeColor','r');
```



## Input Arguments

### **ax** — Parent axes of ROI

Axes object

Parent axes of the ROI, specified as an Axes object.

### **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `h = drawellipse('Color','k','Label','My Ellipse');`

### **AspectRatio** — Aspect ratio of ellipse

nonnegative numeric scalar

Aspect ratio of the ellipse, specified as the comma-separated pair consisting of `'AspectRatio'` and a nonnegative numeric scalar, calculated as `SemiAxes(1)/SemiAxes(2)`. This value changes automatically when you draw or resize the ROI, or change the value of the `SemiAxes` property.

### **Center** — Center of ROI

`[x y]`

Center of the ROI, specified as the comma-separated pair consisting of `'Center'` and a 1-by-2 array of the form `[x y]`. The values `x` and `y` are the coordinates of the center point of the ROI. The value of this property changes automatically when you draw or move the ROI.

### **Color** — ROI color

blue (default) | `ColorSpec` value

ROI color, specified as the comma-separated pair consisting of `'Color'` and a MATLAB `ColorSpec` (Color Specification) value.

Example: `'green'`

Example: 'g'

Example: [0 1 0]

**Deletable — ROI can be interactively deleted via context menu**

true (default) | false

ROI can be interactively deleted via a context menu, specified as the comma-separated pair consisting of 'Deletable' and a logical value of true or false. When the value is true (default), you can delete the ROI via the context menu. To disable this context menu item, set 'Deletable' to false. When the value is set to false, you can still delete the ROI by calling the delete function, specifying the handle to the ROI as the input.

**DrawingArea — Area of axes in which you can interactively place ROI**

'auto' (default) | 'unlimited' | [x,y,w,h]

Area of the axes in which you can interactively place the ROI, specified as the comma-separated pair consisting of 'DrawingArea' and one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangle beginning at (x,y), with width w and height h.

**FaceAlpha — Transparency of ROI face**

0.2 (default) | scalar

Transparency of ROI face, specified as the comma-separated pair consisting of 'FaceAlpha' and a scalar value in the range [0, 1]. When the value is 1, the ROI face is fully opaque. When the value is 0, the ROI face is completely transparent.

**FaceSelectable — ROI face can capture clicks**

true (default) | false

ROI face can capture clicks, specified as the comma-separated pair consisting of 'FaceSelectable' and the value `true` or `false`. When `true` (default), the ROI face can capture clicks. When `false`, the ROI face cannot capture clicks.

### **FixedAspectRatio – Aspect ratio remains constant**

`false` (default) | `true`

Aspect ratio remains constant during interaction, specified as the comma-separated pair consisting of 'FixedAspectRatio' and the logical values `true` or `false`. When this value is `true`, the ROI maintains the aspect ratio when you draw or resize the ellipse. When this value is `false` (default), you can change the aspect ratio when drawing or resizing the ellipse. You can change this value by using the default context menu.

### **HandleVisibility – Visibility of ROI handle in Children property of parent**

`'on'` (default) | `'off'` | `'callback'`

Visibility of the ROI handle in the Children property of the parent, specified as the comma-separated pair consisting of 'HandleVisibility' and one of the values in this table.

Value	Description
<code>'on'</code>	The object handle is always visible (default).
<code>'off'</code>	The object handle is hidden at all times.
<code>'callback'</code>	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

### **InteractionsAllowed – Interactivity of ROI**

`'all'` (default) | `'none'` | `'translate'` | `'reshape'`

Interactivity of the ROI, specified as the comma-separated pair consisting of 'InteractionsAllowed' and one of the values in this table.

Value	Description
<code>'all'</code>	The ROI is fully interactable (default).
<code>'none'</code>	The ROI is not interactable, and no drag points are visible.

<b>Value</b>	<b>Description</b>
'translate'	ROI can be translated (moved) within the drawing area but not reshaped. This value is not available with the <code>drawPoint</code> function.
'reshape'	ROI can be reshaped within the drawing area but not translated. This value is not available with the <code>drawPoint</code> function and the <code>drawRectangle</code> function.

**Label — ROI label**`' '` (default) | character vector | string

ROI label, specified as the comma-separated pair consisting of 'Label' and a character vector or string. By default, this property is set to an empty array ( ' ') and the ROI has no label.

**LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as the comma-separated pair consisting of 'LineWidth' and a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Parent — ROI parent**

Axes object

ROI parent, specified as the comma-separated pair consisting of 'Parent' and an Axes object. A UIAxes cannot be the parent of the ROI.

**RotationAngle — Angle around center of ellipse** $\theta$  (default) | nonnegative numeric scalar

Angle around the center of the ellipse, specified as the comma-separated pair consisting of 'RotationAngle' and as a nonnegative numeric scalar. The angle is measured in degrees in a clockwise direction. This value changes automatically when you draw or move the ROI.

**Selected — Selection state of ROI**

false (default) | true



Selection state of the ROI, specified as the comma-separated pair consisting of 'Selected' and `true` or `false`. You can also set this value interactively. For example, clicking the ROI selects it and sets this value to `true`. Similarly, if you **Ctrl-click** the ROI, deselects it and sets this value to `false`.

### **SelectedColor — Color of ROI when selected**

'none' (default) | `colormap`

Color of the ROI when selected, specified as the comma-separated pair consisting of 'SelectedColor' and a MATLAB `ColorSpec` (Color Specification). If 'SelectedColor' is set to 'none', then the value of `Color` defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example: [0 1 0]

### **SemiAxes — Lengths of semiaxes of ellipse**

[`semiaxis1` `semiaxis2`]

Lengths of the semiaxis of the ellipse, specified as the comma-separated pair consisting of 'SemiAxes' and a 1-by-2 array of the form [`semiaxis1` `semiaxis2`]. The `drawellipse` function assigns the length of the semiaxis that is closest to the x direction to `semiaxis1`. Note however that the shape and orientation of the ellipse can change through interaction. The value of this property changes automatically when you draw or reshape the ROI.

### **StripeColor — Color of ROI stripe**

'none' (default) | `ColorSpec` value

Color of the ROI stripe, specified as the comma-separated pair consisting of 'StripeColor' and a MATLAB `ColorSpec` (Color Specification) value. If you specify 'StripeColor', the ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

### **Tag — Tag to associate with ROI**

character vector | string scalar

Tag to associate with the ROI, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

**UIContextMenu — Context menu**

ContextMenu object

Context menu, specified as the comma-separated pair consisting of 'UIContextMenu' and a `uicontextmenu` object. Specify this name-value pair to display a custom context menu when you right-click on the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

**UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as the comma-separated pair consisting of 'UserData' and any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

**Vertices — Locations of points on edge of ROI**

N-by-2 array

Locations of points on the edge of the ROI, returned as the comma-separated pair consisting of 'Vertices' and an *n*-by-2 array.

This property is read-only.

**Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as the comma-separated pair consisting of 'Visible' and one of the values in this table.

Value	Description
'on'	Specify this value to display the ROI (default).
'off'	Specify this value to hide the ROI without deleting it. You still can access the properties of an invisible ROI.

## Output Arguments

### **h — Elliptical ROI**

`images.roi.Ellipse` object

Elliptical ROI, returned as an `images.roi.Ellipse` object.

## See Also

`drawassisted` | `drawcircle` | `drawfreehand` | `drawline` | `drawpoint` | `drawpolygon` | `drawpolyline` | `drawrectangle` | `images.roi.Ellipse`

**Introduced in R2018b**

## drawfreehand

Create customizable freehand ROI

### Syntax

```
h = drawfreehand  
h = drawfreehand(ax)  
h = drawfreehand( ____, Name, Value)
```

### Description

`h = drawfreehand` begins interactive placement of a freehand region-of-interest (ROI) on the current axes. The function returns a handle to an `images.roi.Freehand` object. You can modify the ROI interactively using your mouse. The ROI object also supports a context menu that controls aspects of its appearance and behavior.

`h = drawfreehand(ax)` begins interactive placement of an ROI in the axes specified by `ax`.

`h = drawfreehand( ____, Name, Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs. Unspecified name-value pairs are set to the default value.

During interactive placement, click and drag to draw the ROI shape. After the shape has been drawn, click waypoints to reshape the corresponding region of the freehand.

You can add additional waypoints after drawing via a context menu or by double-clicking the edge of the ROI. To remove a waypoint, right-click the waypoint and select **Remove Waypoint** from the context menu.

### Examples

## Create Freehand ROI That Is Not Selectable

Read an image into the workspace and display it.

```
figure
imshow(imread('baby.jpg'))
```

Draw a freehand ROI.

```
h = drawfreehand();
```

Fill in the face of the freehand ROI and disable the ability to select it.

```
h.FaceAlpha = 1;
h.FaceSelectable = false;
```

## Input Arguments

### ax — Parent axes of ROI

Axes object

Parent axes of the ROI, specified as an Axes object.

### 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `h = drawfreehand('LineWidth',5);`

### Closed — Close freehand ROI

true (default) | false

Close the freehand ROI, specified as the comma-separated pair consisting of 'Closed' and the logical value `true` or `false`. If `true` (default), `drawfreehand` closes the ROI by connecting the last waypoint drawn to the first waypoint drawn.

### Color — ROI color

blue (default) | ColorSpec value

ROI color, specified as the comma-separated pair consisting of 'Color' and a MATLAB ColorSpec (Color Specification) value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**Deletable — ROI can be interactively deleted via context menu**

true (default) | false

ROI can be interactively deleted via a context menu, specified as the comma-separated pair consisting of 'Deletable' and a logical value of true or false. When the value is true (default), you can delete the ROI via the context menu. To disable this context menu item, set 'Deletable' to false. When the value is set to false, you can still delete the ROI by calling the delete function, specifying the handle to the ROI as the input.

**DrawingArea — Area of axes in which you can interactively place ROI**

'auto' (default) | 'unlimited' | [x,y,w,h]

Area of the axes in which you can interactively place the ROI, specified as the comma-separated pair consisting of 'DrawingArea' and one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangle beginning at (x,y), with width w and height h.

**FaceAlpha — Transparency of ROI face**

0.2 (default) | scalar

Transparency of ROI face, specified as the comma-separated pair consisting of 'FaceAlpha' and a scalar value in the range [0, 1]. When the value is 1, the ROI face is fully opaque. When the value is 0, the ROI face is completely transparent.

**FaceSelectable – ROI face can capture clicks**

true (default) | false

ROI face can capture clicks, specified as the comma-separated pair consisting of 'FaceSelectable' and the value true or false. When true (default), the ROI face can capture clicks. When false, the ROI face cannot capture clicks.

**HandleVisibility – Visibility of ROI handle in Children property of parent**

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as the comma-separated pair consisting of 'HandleVisibility' and one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**InteractionsAllowed – Interactivity of ROI**

'all' (default) | 'none' | 'translate' | 'reshape'

Interactivity of the ROI, specified as the comma-separated pair consisting of 'InteractionsAllowed' and one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.
'translate'	ROI can be translated (moved) within the drawing area but not reshaped. This value is not available with the drawPoint function.

Value	Description
'reshape'	ROI can be reshaped within the drawing area but not translated. This value is not available with the <code>drawPoint</code> function and the <code>drawRectangle</code> function.

**Label — ROI label**`' '` (default) | character vector | string

ROI label, specified as the comma-separated pair consisting of 'Label' and a character vector or string. By default, this property is set to an empty array ( ' ' ) and the ROI has no label.

**LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as the comma-separated pair consisting of 'LineWidth' and a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Multiclick — Control freehand drawing style during interactive placement**`true` (default) | `false`

Control the freehand drawing style during interactive placement, specified as the comma-separated pair consisting of 'Multiclick' and the logical scalar `true` or `false`. When the value is `false` (default), a single click and drag gesture completes the freehand. When `true`, multiple click and drag gestures can be combined with straight edges to make a more complex freehand shape.

**Parent — ROI parent**

Axes object

ROI parent, specified as the comma-separated pair consisting of 'Parent' and an Axes object. A UIAxes cannot be the parent of the ROI.

**Position — Position of the freehand ROI**`[]` (default) | *n*-by-2 array

Position of the freehand ROI, specified as an *n*-by-2 array, [*x*<sub>1</sub> *y*<sub>1</sub>; ...; *x*<sub>*n*</sub> *y*<sub>*n*</sub>] where each row specifies the position of a vertex of the freehand ROI.



**Selected — Selection state of ROI**`false (default) | true`

Selection state of the ROI, specified as the comma-separated pair consisting of 'Selected' and `true` or `false`. You can also set this value interactively. For example, clicking the ROI selects it and sets this value to `true`. Similarly, if you **Ctrl-click** the ROI, deselects it and sets this value to `false`.

**SelectedColor — Color of ROI when selected**`'none' (default) | colorspec`

Color of the ROI when selected, specified as the comma-separated pair consisting of 'SelectedColor' and a MATLAB `ColorSpec` (Color Specification). If 'SelectedColor' is set to 'none', then the value of `Color` defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**Smoothing — Smooth edge of freehand ROI**`1 (default) | nonnegative scalar`

Smooth the edge of the freehand ROI using the standard deviation of the Gaussian smoothing kernel, specified as the comma-separated pair consisting of 'Smoothing' and a nonnegative scalar. The function uses this 'Smoothing' value to filter the  $x$  and  $y$  coordinates of the freehand ROI. The function defines the filter size as  $2*\text{ceil}(2*\text{Smoothing}) + 1$ .

**StripeColor — Color of ROI stripe**`'none' (default) | ColorSpec value`

Color of the ROI stripe, specified as the comma-separated pair consisting of 'StripeColor' and a MATLAB `ColorSpec` (Color Specification) value. If you specify 'StripeColor', the ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**Tag — Tag to associate with ROI**

character vector | string scalar

Tag to associate with the ROI, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

**UIContextMenu — Context menu**

ContextMenu object

Context menu, specified as the comma-separated pair consisting of 'UIContextMenu' and a `uicontextmenu` object. Specify this name-value pair to display a custom context menu when you right-click on the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

**UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as the comma-separated pair consisting of 'UserData' and any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

**Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as the comma-separated pair consisting of 'Visible' and one of the values in this table.

Value	Description
'on'	Specify this value to display the ROI (default).
'off'	Specify this value to hide the ROI without deleting it. You still can access the properties of an invisible ROI.

**Waypoints — Control points to reshape the ROI**1 (default) | *n*-by-1 logical vector

Control points to reshape the ROI, specified as the comma-separated pair consisting of 'Waypoints' and an *n*-by-1 logical vector, the same length as `Position`. Each element of the `Waypoints` vector indicates if the corresponding point in the `Position` array is a

waypoint. Positions that are waypoints are set to `true` in `Waypoints`. Dragging a waypoint modifies the ROI between the specified waypoint and its immediate neighboring waypoints. If empty, `drawfreehand` automatically generates `Waypoints` at locations of increased curvature.

## Output Arguments

### **h** — Freehand ROI

`images.roi.Freehand` object

Freehand ROI, returned as an `images.roi.Freehand` object.

## See Also

`drawassisted` | `drawcircle` | `drawellipse` | `drawline` | `drawpoint` | `drawpolygon` | `drawpolyline` | `drawrectangle` | `images.roi.Freehand`

**Introduced in R2018b**

## drawline

Create customizable linear ROI

### Syntax

```
h = drawline
h = drawline(ax)
h = drawline( ____, Name, Value)
```

### Description

`h = drawline` begins interactive placement of a linear region-of-interest (ROI) on the current axes. The function returns a handle to an `images.roi.Line` object. You can modify the ROI interactively using your mouse. The ROI object also supports a context menu that controls aspects of its appearance and behavior.

`h = drawline(ax)` creates an instance of the ROI in the axes specified by `ax`.

`h = drawline( ____, Name, Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs. Unspecified name-value pairs are set to the default value.

### Examples

#### Create Linear ROI That Turns Yellow When Selected

Read an image into the workspace and display it.

```
figure
imshow(imread('baby.jpg'))
```

Draw a linear ROI on the image. Use the `'SelectedColor'` name-value pair to specify the color of the ROI when selected.

```
h = drawline('SelectedColor','yellow');
```

Make the ROI turn yellow when selected.

```
h.Selected = true;
```

### **Create Linear ROI Programmatically**

Read an image into the workspace and display it.

```
I = imread('baby.jpg');  
figure  
imshow(I)
```



Draw a linear ROI over the image. Use the 'Position' name-value pair to specify the location and length of the linear ROI. Set the line to be striped red by specifying the 'StripeColor' name-value pair.

```
h = drawline('Position',[500 500;500 1500],'StripeColor','r');
```





Parent axes of the ROI, specified as an Axes object.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `h = drawline('LineWidth',4);`

### Color — ROI color

blue (default) | `ColorSpec` value

ROI color, specified as the comma-separated pair consisting of `'Color'` and a MATLAB `ColorSpec` (`Color Specification`) value.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

### Deletable — ROI can be interactively deleted via context menu

true (default) | false

ROI can be interactively deleted via a context menu, specified as the comma-separated pair consisting of `'Deletable'` and a logical value of `true` or `false`. When the value is `true` (default), you can delete the ROI via the context menu. To disable this context menu item, set `'Deletable'` to `false`. When the value is set to `false`, you can still delete the ROI by calling the `delete` function, specifying the handle to the ROI as the input.

### DrawingArea — Area of axes in which you can interactively place ROI

'auto' (default) | 'unlimited' | `[x,y,w,h]`

Area of the axes in which you can interactively place the ROI, specified as the comma-separated pair consisting of `'DrawingArea'` and one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).

Value	Description
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangle beginning at (x,y), with width <i>w</i> and height <i>h</i> .

**HandleVisibility — Visibility of ROI handle in Children property of parent**

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as the comma-separated pair consisting of 'HandleVisibility' and one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**InteractionsAllowed — Interactivity of ROI**

'all' (default) | 'none' | 'translate' | 'reshape'

Interactivity of the ROI, specified as the comma-separated pair consisting of 'InteractionsAllowed' and one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.

Value	Description
'translate'	ROI can be translated (moved) within the drawing area but not reshaped. This value is not available with the drawPoint function.
'reshape'	ROI can be reshaped within the drawing area but not translated. This value is not available with the drawPoint function and the drawRectangle function.

**Label — ROI label**

' ' (default) | character vector | string

ROI label, specified as the comma-separated pair consisting of 'Label' and a character vector or string. By default, this property is set to an empty array ( ' ' ) and the ROI has no label.

**LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as the comma-separated pair consisting of 'LineWidth' and a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Parent — ROI parent**

Axes object

ROI parent, specified as the comma-separated pair consisting of 'Parent' and an Axes object. A UIAxes cannot be the parent of the ROI.

**Position — Position of ROI**

2-by-2 array

Position of the ROI, specified as a 2-by-2 array of the form [x1 y1; x2 y2]. Each row specifies the respective end-point of the linear ROI. This value changes automatically when you draw or move the line.

**Selected — Selection state of ROI**

false (default) | true

Selection state of the ROI, specified as the comma-separated pair consisting of 'Selected' and true or false. You can also set this value interactively. For example,

clicking the ROI selects it and sets this value to `true`. Similarly, if you **Ctrl-click** the ROI, deselects it and sets this value to `false`.

**SelectedColor — Color of ROI when selected**

'none' (default) | colorspec

Color of the ROI when selected, specified as the comma-separated pair consisting of 'SelectedColor' and a MATLAB ColorSpec (Color Specification). If 'SelectedColor' is set to 'none', then the value of Color defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**StripeColor — Color of ROI stripe**

'none' (default) | ColorSpec value

Color of the ROI stripe, specified as the comma-separated pair consisting of 'StripeColor' and a MATLAB ColorSpec (Color Specification) value. If you specify 'StripeColor', the ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**Tag — Tag to associate with ROI**

character vector | string scalar

Tag to associate with the ROI, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

**UIContextMenu — Context menu**

ContextMenu object

Context menu, specified as the comma-separated pair consisting of 'UIContextMenu' and a `uicontextmenu` object. Specify this name-value pair to display a custom context menu when you right-click on the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

**UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as the comma-separated pair consisting of 'UserData' and any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

**Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as the comma-separated pair consisting of 'Visible' and one of the values in this table.

Value	Description
'on'	Specify this value to display the ROI (default).
'off'	Specify this value to hide the ROI without deleting it. You still can access the properties of an invisible ROI.

**Output Arguments****h — Linear ROI**`images.roi.Line` object

Linear ROI, returned as an `images.roi.Line` object.

**See Also**

`drawassisted` | `drawcircle` | `drawellipse` | `drawfreehand` | `drawpoint` | `drawpolygon` | `drawpolyline` | `drawrectangle` | `images.roi.Line`

**Introduced in R2018b**

## drawpoint

Create customizable point ROI

### Syntax

```
h = drawpoint
h = drawpoint(ax, ___)
h = drawpoint( ___, Name, Value)
```

### Description

`h = drawpoint` begins interactive placement of a point region-of-interest (ROI) on the current axes. The function returns a handle to an `image.roi.Point` object. You can modify the ROI interactively using your mouse. The ROI object also supports a context menu that controls aspects of its appearance and behavior.

`h = drawpoint(ax, ___)` begins interactive placement of an ROI in the axes specified by `ax`.

`h = drawpoint( ___, Name, Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs. Unspecified name-value pairs are set to the default value.

### Examples

#### Create Point ROI Interactively

Read image into the workspace and display it.

```
figure
imshow(imread('baby.jpg'))
```

Draw a point ROI on the image.

```
h = drawpoint();
```

Store some information about the ROI in the Tag property.

```
h.Tag = 'My Point';
```

### **Create Point ROI Programmatically**

Read image into the workspace and display it.

```
I = imread('baby.jpg');  
figure  
imshow(I)
```





Draw a point ROI on the image, using named parameters to specify the location.

```
h = drawpoint('Position',[500 500]);
```



1-572

## Input Arguments

Parent axes of the ROI, specified as an Axes object.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `h = drawpoint('Position',[1000 1000]);`

### Color — ROI color

blue (default) | `ColorSpec` value

ROI color, specified as the comma-separated pair consisting of `'Color'` and a MATLAB `ColorSpec` (`Color Specification`) value.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

### Deletable — ROI can be interactively deleted via context menu

true (default) | false

ROI can be interactively deleted via a context menu, specified as the comma-separated pair consisting of `'Deletable'` and a logical value of `true` or `false`. When the value is `true` (default), you can delete the ROI via the context menu. To disable this context menu item, set `'Deletable'` to `false`. When the value is set to `false`, you can still delete the ROI by calling the `delete` function, specifying the handle to the ROI as the input.

### DrawingArea — Area of axes in which you can interactively place ROI

'auto' (default) | 'unlimited' | `[x,y,w,h]`

Area of the axes in which you can interactively place the ROI, specified as the comma-separated pair consisting of `'DrawingArea'` and one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).

Value	Description
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangle beginning at (x,y), with width <i>w</i> and height <i>h</i> .

**HandleVisibility — Visibility of ROI handle in Children property of parent**

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as the comma-separated pair consisting of 'HandleVisibility' and one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**InteractionsAllowed — Interactivity of ROI**

'all' (default) | 'none' | 'translate' | 'reshape'

Interactivity of the ROI, specified as the comma-separated pair consisting of 'InteractionsAllowed' and one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.

Value	Description
'translate'	ROI can be translated (moved) within the drawing area but not reshaped. This value is not available with the drawPoint function.
'reshape'	ROI can be reshaped within the drawing area but not translated. This value is not available with the drawPoint function and the drawRectangle function.

**Label — ROI label**

' ' (default) | character vector | string

ROI label, specified as the comma-separated pair consisting of 'Label' and a character vector or string. By default, this property is set to an empty array ( ' ') and the ROI has no label.

**LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as the comma-separated pair consisting of 'LineWidth' and a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Parent — ROI parent**

Axes object

ROI parent, specified as the comma-separated pair consisting of 'Parent' and an Axes object. A UIAxes cannot be the parent of the ROI.

**Position — Position of ROI**

1-by-2 array

Position of the ROI, specified as the comma-separated pair consisting of 'Position' and a 1-by-2 array of the form [x y]. The values x and y specify the x- and y-coordinates of the ROI. This value changes automatically when you draw or move the point.

**Selected — Selection state of ROI**

false (default) | true

Selection state of the ROI, specified as the comma-separated pair consisting of 'Selected' and true or false. You can also set this value interactively. For example,

clicking the ROI selects it and sets this value to `true`. Similarly, if you **Ctrl-click** the ROI, deselects it and sets this value to `false`.

**SelectedColor — Color of ROI when selected**

'none' (default) | colorspec

Color of the ROI when selected, specified as the comma-separated pair consisting of 'SelectedColor' and a MATLAB ColorSpec (Color Specification). If 'SelectedColor' is set to 'none', then the value of Color defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**StripeColor — Color of ROI stripe**

'none' (default) | ColorSpec value

Color of the ROI stripe, specified as the comma-separated pair consisting of 'StripeColor' and a MATLAB ColorSpec (Color Specification) value. If you specify 'StripeColor', the ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**Tag — Tag to associate with ROI**

character vector | string scalar

Tag to associate with the ROI, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

**UIContextMenu — Context menu**

ContextMenu object

Context menu, specified as the comma-separated pair consisting of 'UIContextMenu' and a `uicontextmenu` object. Specify this name-value pair to display a custom context menu when you right-click on the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

**UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as the comma-separated pair consisting of 'UserData' and any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

**Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as the comma-separated pair consisting of 'Visible' and one of the values in this table.

Value	Description
'on'	Specify this value to display the ROI (default).
'off'	Specify this value to hide the ROI without deleting it. You still can access the properties of an invisible ROI.

## Output Arguments

**h — Point ROI**`images.roi.Point` object

Point ROI, returned as an `images.roi.Point` object.

## See Also

`drawassisted` | `drawcircle` | `drawellipse` | `drawfreehand` | `drawline` | `drawpolygon` | `drawpolyline` | `drawrectangle` | `images.roi.Point`

**Introduced in R2018b**

## drawpolygon

Create customizable polygonal ROI

### Syntax

```
h = drawpolygon  
h = drawpolygon(ax)  
h = drawpolygon( ____, Name, Value)
```

### Description

`h = drawpolygon` begins interactive placement of a polygonal region-of-interest (ROI) on the current axes. The function returns a handle to an `images.roi.Polygon` object. You can modify the ROI interactively using your mouse. The ROI object also supports a context menu that controls aspects of its appearance and behavior.

`h = drawpolygon(ax)` begins interactive placement of an ROI on the axes specified by `ax`.

`h = drawpolygon( ____, Name, Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs. Unspecified name-value pairs are set to the default value.

During interactive placement, click to place polygon vertices. Double-click to add a vertex and finish initial placement of the polygon. Right-click to finish initial placement of the polygon.

You can add additional vertices after drawing via a context menu or by double-clicking the edge of the polygonal ROI. To remove a vertex, right-click the vertex and select **Delete Vertex** from the context menu.

### Examples



### Create Selectable Transparent Polygon

Read an image into the workspace and display it.

```
figure  
imshow(imread('baby.jpg'))
```

Draw a polygonal ROI on the image.

```
h = drawpolygon();
```

Use the 'FaceAlpha' name-value pair to make the face of the ROI transparent and use the 'FaceSelectable' name-value pair to make the ROI clickable.

```
h.FaceAlpha = 0;  
h.FaceSelectable = true;
```

### Create Polygonal ROI Programmatically

Read image into the workspace and display it.

```
I = imread('baby.jpg');  
figure  
imshow(I)
```



Draw a polygonal ROI over the image, using the `Position` parameter to specify the location of vertices.

```
my_vertices = [500 500;400 600;400 700;500 800;600 800;700 700; 700 600];  
h = drawpolygon('Position',my_vertices);
```



Parent axes of the ROI, specified as an Axes object.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `h = drawpolygon('FaceAlpha',0,'FaceSelectable',1);`

### Color — ROI color

blue (default) | `ColorSpec` value

ROI color, specified as the comma-separated pair consisting of `'Color'` and a MATLAB `ColorSpec` (Color Specification) value.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

### Deletable — ROI can be interactively deleted via context menu

true (default) | false

ROI can be interactively deleted via a context menu, specified as the comma-separated pair consisting of `'Deletable'` and a logical value of `true` or `false`. When the value is `true` (default), you can delete the ROI via the context menu. To disable this context menu item, set `'Deletable'` to `false`. When the value is set to `false`, you can still delete the ROI by calling the `delete` function, specifying the handle to the ROI as the input.

### DrawingArea — Area of axes in which you can interactively place ROI

'auto' (default) | 'unlimited' | `[x,y,w,h]`

Area of the axes in which you can interactively place the ROI, specified as the comma-separated pair consisting of `'DrawingArea'` and one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).

Value	Description
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangle beginning at (x,y), with width w and height h.

**FaceAlpha — Transparency of ROI face**

0.2 (default) | scalar

Transparency of ROI face, specified as the comma-separated pair consisting of 'FaceAlpha' and a scalar value in the range [0,1]. When the value is 1, the ROI face is fully opaque. When the value is 0, the ROI face is completely transparent.

**FaceSelectable — ROI face can capture clicks**

true (default) | false

ROI face can capture clicks, specified as the comma-separated pair consisting of 'FaceSelectable' and the value true or false. When true (default), the ROI face can capture clicks. When false, the ROI face cannot capture clicks.

**HandleVisibility — Visibility of ROI handle in Children property of parent**

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as the comma-separated pair consisting of 'HandleVisibility' and one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**InteractionsAllowed – Interactivity of ROI**

'all' (default) | 'none' | 'translate' | 'reshape'

Interactivity of the ROI, specified as the comma-separated pair consisting of 'InteractionsAllowed' and one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.
'translate'	ROI can be translated (moved) within the drawing area but not reshaped. This value is not available with the <code>drawPoint</code> function.
'reshape'	ROI can be reshaped within the drawing area but not translated. This value is not available with the <code>drawPoint</code> function and the <code>drawRectangle</code> function.

**Label – ROI label**

' ' (default) | character vector | string

ROI label, specified as the comma-separated pair consisting of 'Label' and a character vector or string. By default, this property is set to an empty array ( ' ') and the ROI has no label.

**LineWidth – Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as the comma-separated pair consisting of 'LineWidth' and a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Parent – ROI parent**

Axes object

ROI parent, specified as the comma-separated pair consisting of 'Parent' and an Axes object. A UIAxes cannot be the parent of the ROI.

**Position — Position of ROI***n*-by-2 array

Position of the ROI, specified as the comma-separated pair consisting of 'Position' and an *n*-by-2 array, of the form [x1 y1; ...; xn yn]. Each row specifies the position of a vertex of the polygonal ROI. This value changes automatically when you draw or move the ROI.

**Selected — Selection state of ROI**

false (default) | true

Selection state of the ROI, specified as the comma-separated pair consisting of 'Selected' and true or false. You can also set this value interactively. For example, clicking the ROI selects it and sets this value to true. Similarly, if you **Ctrl-click** the ROI, deselects it and sets this value to false.

**SelectedColor — Color of ROI when selected**

'none' (default) | colorspec

Color of the ROI when selected, specified as the comma-separated pair consisting of 'SelectedColor' and a MATLAB ColorSpec (Color Specification). If 'SelectedColor' is set to 'none', then the value of Color defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**StripeColor — Color of ROI stripe**

'none' (default) | ColorSpec value

Color of the ROI stripe, specified as the comma-separated pair consisting of 'StripeColor' and a MATLAB ColorSpec (Color Specification) value. If you specify 'StripeColor', the ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]



**Tag — Tag to associate with ROI**

character vector | string scalar

Tag to associate with the ROI, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

**UIContextMenu — Context menu**

ContextMenu object

Context menu, specified as the comma-separated pair consisting of 'UIContextMenu' and a `uicontextmenu` object. Specify this name-value pair to display a custom context menu when you right-click on the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

**UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as the comma-separated pair consisting of 'UserData' and any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

**Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as the comma-separated pair consisting of 'Visible' and one of the values in this table.

Value	Description
'on'	Specify this value to display the ROI (default).
'off'	Specify this value to hide the ROI without deleting it. You still can access the properties of an invisible ROI.

**Output Arguments****h — Polygonal ROI**`images.roi.Polygon` object

Polygonal ROI, returned as an `images.roi.Polygon` object.

## **See Also**

`drawassisted` | `drawcircle` | `drawellipse` | `drawfreehand` | `drawline` |  
`drawpoint` | `drawpolyline` | `drawrectangle` | `images.roi.Polygon`

**Introduced in R2018b**

# drawpolyline

Create customizable polyline ROI

## Syntax

```
h = drawpolyline
h = drawpolyline(ax, ___ )
h = drawpolyline( ___ ,Name,Value)
```

## Description

`h = drawpolyline` begins interactive placement of a polyline region-of-interest (ROI) on the current axes. The function returns a handle to a `images.roi.Polyline` object. You can modify the ROI interactively using your mouse. The ROI object also supports a context menu that controls aspects of its appearance and behavior.

`h = drawpolyline(ax, ___ )` begins interactive placement of an instance of the ROI in the axes specified by `ax`.

`h = drawpolyline( ___ ,Name,Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs. Unspecified name-value pairs are set to the default value.

During interactive placement, click to place polyline vertices. Double-click to add a vertex and finish initial placement of the polyline. Right-click to finish initial placement of the polyline.

You can add additional vertices after drawing via context menu or by double-clicking the polyline edge. To remove a vertex, right-click the vertex and select **Delete Vertex** from the context menu.

## Examples

### **Create Polyline ROI Interactively**

Read an image into the workspace and display it.

```
figure  
imshow(imread('baby.jpg'))
```

Draw the polyline ROI on the image.

```
h = drawpolyline();
```

Use the 'LineWidth' name-value pair to make the width of the edge of the ROI thinner than the default width.

```
h.LineWidth = 1;
```

### **Create Polygonal ROI Programmatically**

Read image into the workspace and display it.

```
I = imread('baby.jpg');  
figure  
imshow(I)
```



Draw a polygonal ROI over the image, using named parameters to specify the location and shape. The example also specifies that the edge of the polygon is a striped.

```
h = drawpolyline('Position',[500 500;400 600;400 700;500 800;600 800;700 700; 700 600])
```



Parent axes of the ROI, specified as an Axes object.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `h = drawpolyline('LineWidth',1);`

### Color — ROI color

blue (default) | `ColorSpec` value

ROI color, specified as the comma-separated pair consisting of `'Color'` and a MATLAB `ColorSpec` (`Color Specification`) value.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

### Deletable — ROI can be interactively deleted via context menu

true (default) | false

ROI can be interactively deleted via a context menu, specified as the comma-separated pair consisting of `'Deletable'` and a logical value of `true` or `false`. When the value is `true` (default), you can delete the ROI via the context menu. To disable this context menu item, set `'Deletable'` to `false`. When the value is set to `false`, you can still delete the ROI by calling the `delete` function, specifying the handle to the ROI as the input.

### DrawingArea — Area of axes in which you can interactively place ROI

'auto' (default) | 'unlimited' | `[x,y,w,h]`

Area of the axes in which you can interactively place the ROI, specified as the comma-separated pair consisting of `'DrawingArea'` and one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).



Value	Description
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangle beginning at (x,y), with width w and height h.

### HandleVisibility – Visibility of ROI handle in Children property of parent

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as the comma-separated pair consisting of 'HandleVisibility' and one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

### InteractionsAllowed – Interactivity of ROI

'all' (default) | 'none' | 'translate' | 'reshape'

Interactivity of the ROI, specified as the comma-separated pair consisting of 'InteractionsAllowed' and one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.

Value	Description
'translate'	ROI can be translated (moved) within the drawing area but not reshaped. This value is not available with the <code>drawPoint</code> function.
'reshape'	ROI can be reshaped within the drawing area but not translated. This value is not available with the <code>drawPoint</code> function and the <code>drawRectangle</code> function.

**Label — ROI label**

' ' (default) | character vector | string

ROI label, specified as the comma-separated pair consisting of 'Label' and a character vector or string. By default, this property is set to an empty array ( ' ') and the ROI has no label.

**LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as the comma-separated pair consisting of 'LineWidth' and a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Parent — ROI parent**

Axes object

ROI parent, specified as the comma-separated pair consisting of 'Parent' and an Axes object. A UIAxes cannot be the parent of the ROI.

**Position — Position of the polyline ROI**

*n*-by-2 array

Position of the polyline ROI, specified as the comma-separated pair consisting of 'Position' and an *n*-by-2 array, of the form [*x*<sub>1</sub> *y*<sub>1</sub>; ...; *x*<sub>*n*</sub> *y*<sub>*n*</sub>], where *n* represents the number of vertices. Each row specifies the position of a vertex of the polyline ROI.

**Selected — Selection state of ROI**

false (default) | true

Selection state of the ROI, specified as the comma-separated pair consisting of 'Selected' and true or false. You can also set this value interactively. For example,

clicking the ROI selects it and sets this value to `true`. Similarly, if you **Ctrl-click** the ROI, deselects it and sets this value to `false`.

### **SelectedColor — Color of ROI when selected**

'none' (default) | colorspec

Color of the ROI when selected, specified as the comma-separated pair consisting of 'SelectedColor' and a MATLAB ColorSpec (Color Specification). If 'SelectedColor' is set to 'none', then the value of Color defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example: [0 1 0]

### **StripeColor — Color of ROI stripe**

'none' (default) | ColorSpec value

Color of the ROI stripe, specified as the comma-separated pair consisting of 'StripeColor' and a MATLAB ColorSpec (Color Specification) value. If you specify 'StripeColor', the ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

### **Tag — Tag to associate with ROI**

character vector | string scalar

Tag to associate with the ROI, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

### **UIContextMenu — Context menu**

ContextMenu object

Context menu, specified as the comma-separated pair consisting of 'UIContextMenu' and a `uicontextmenu` object. Specify this name-value pair to display a custom context menu when you right-click on the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

**UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as the comma-separated pair consisting of 'UserData' and any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

**Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as the comma-separated pair consisting of 'Visible' and one of the values in this table.

Value	Description
'on'	Specify this value to display the ROI (default).
'off'	Specify this value to hide the ROI without deleting it. You still can access the properties of an invisible ROI.

## Output Arguments

**h — Polyline ROI**`images.roi.Polyline` object

Polyline ROI, returned as an `images.roi.Polyline` object.

## See Also

`drawassisted` | `drawcircle` | `drawellipse` | `drawfreehand` | `drawline` | `drawpoint` | `drawpolygon` | `drawrectangle` | `images.roi.Polyline`

**Introduced in R2018b**

# drawrectangle

Create customizable rectangular ROI

## Syntax

```
h = drawrectangle  
h = drawrectangle(ax)  
h = drawrectangle( ____, Name, Value)
```

## Description

`h = drawrectangle` begins interactive placement of a rectangular region-of-interest (ROI) on the current axes. The function returns a handle to an `images.roi.Rectangle` object. You can modify the ROI interactively using your mouse. The ROI object also supports a context menu that controls aspects of its appearance and behavior.

`h = drawrectangle(ax)` begins interactive placement of the ROI in the axes specified by `ax`.

`h = drawrectangle( ____, Name, Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs. Unspecified name-value pairs are set to the default value.

During interactive placement, click and drag to draw the ROI. After the shape has been drawn, click and drag the waypoints to reshape the ROI.

## Examples

### Create Rectangular ROI Inside Rectangular ROI

Read an image into the workspace and display it.

```
figure  
imshow(imread('baby.jpg'))
```

Begin interactive placement of the rectangular ROI on the image.

```
h1 = drawrectangle('Label', 'OuterRectangle', 'Color', [1 0 0]);
```

Draw another rectangular ROI and specify that the new rectangle must be drawn inside the first rectangular ROI.

```
h2 = drawrectangle('Label', 'InnerRectangle', 'DrawingArea', h1.Position);
```

## **Create Rectangular ROI Programmatically**

Read image into the workspace and display it.

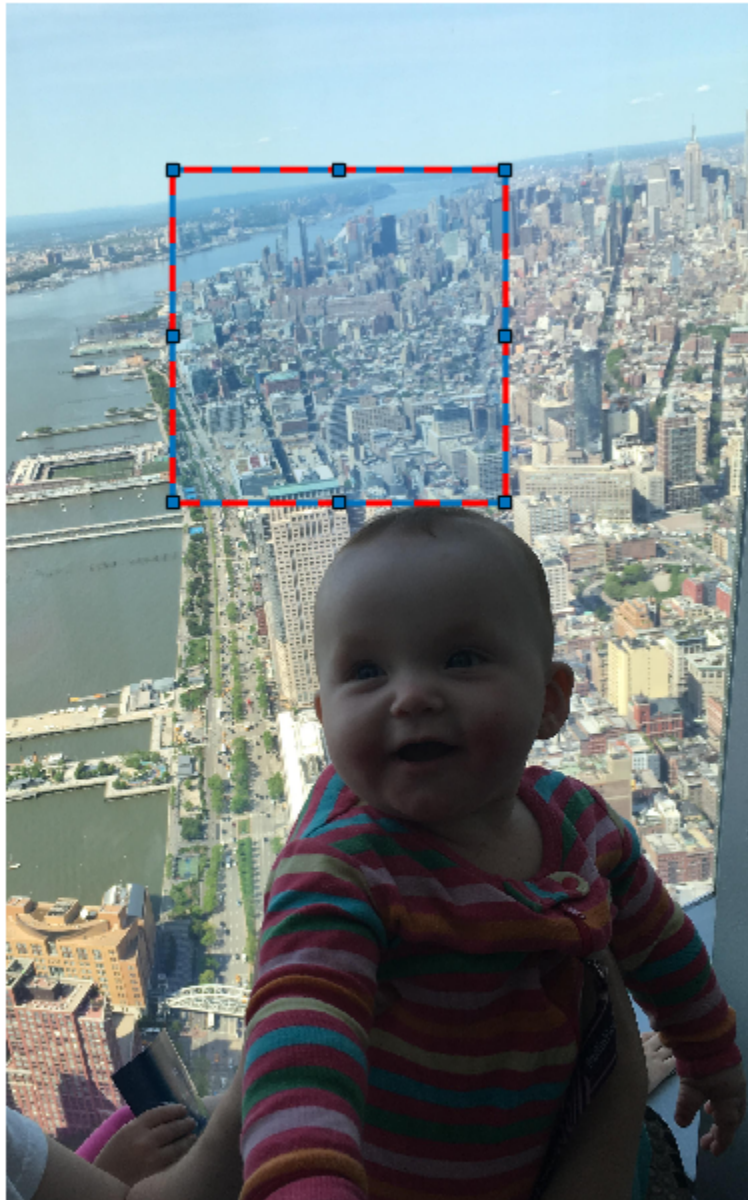
```
I = imread('baby.jpg');  
figure  
imshow(I)
```



Draw a rectangular ROI over the image, using named parameters to specify the location and size of the rectangle. The example also specifies that the edge of the rectangle is a striped line.

```
h = drawrectangle('Position',[500,500,1000,1000], 'StripeColor','r');
```





Parent axes of the ROI, specified as an Axes object.

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `h1 = drawrectangle('LineWidth',5);`

### **AspectRatio — Aspect ratio of rectangle**

nonnegative numeric scalar

Aspect ratio of the rectangle, specified as the comma-separated pair consisting of `'AspectRatio'` and a nonnegative numeric scalar, calculated as `height/width`. This value changes automatically when you draw or resize the ROI.

### **Color — ROI color**

blue (default) | `ColorSpec` value

ROI color, specified as the comma-separated pair consisting of `'Color'` and a MATLAB `ColorSpec` (Color Specification) value.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

### **Deletable — ROI can be interactively deleted via context menu**

true (default) | false

ROI can be interactively deleted via a context menu, specified as the comma-separated pair consisting of `'Deletable'` and a logical value of `true` or `false`. When the value is `true` (default), you can delete the ROI via the context menu. To disable this context menu item, set `'Deletable'` to `false`. When the value is set to `false`, you can still delete the ROI by calling the `delete` function, specifying the handle to the ROI as the input.

### **DrawingArea — Area of axes in which you can interactively place ROI**

'auto' (default) | 'unlimited' | `[x,y,w,h]`

Area of the axes in which you can interactively place the ROI, specified as the comma-separated pair consisting of `'DrawingArea'` and one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangle beginning at (x,y), with width w and height h.

### FaceAlpha — Transparency of ROI face

0.2 (default) | scalar

Transparency of ROI face, specified as the comma-separated pair consisting of 'FaceAlpha' and a scalar value in the range [0, 1]. When the value is 1, the ROI face is fully opaque. When the value is 0, the ROI face is completely transparent.

### FaceSelectable — ROI face can capture clicks

true (default) | false

ROI face can capture clicks, specified as the comma-separated pair consisting of 'FaceSelectable' and the value true or false. When true (default), the ROI face can capture clicks. When false, the ROI face cannot capture clicks.

### FixedAspectRatio — Aspect ratio remains constant

false (default) | true

Aspect ratio remains constant during interaction, specified as the comma-separated pair consisting of 'FixedAspectRatio' and the logical values true or false. When true, the ROI maintains the aspect ratio when you draw or resize the rectangle. When false (default), you can change the aspect ratio when drawing or resizing the rectangle. You can change the state of this property using the default context menu.

### HandleVisibility — Visibility of ROI handle in Children property of parent

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as the comma-separated pair consisting of 'HandleVisibility' and one of the values in this table.

<b>Value</b>	<b>Description</b>
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**InteractionsAllowed — Interactivity of ROI**

'all' (default) | 'none' | 'translate' | 'reshape'

Interactivity of the ROI, specified as the comma-separated pair consisting of 'InteractionsAllowed' and one of the values in this table.

<b>Value</b>	<b>Description</b>
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.
'translate'	ROI can be translated (moved) within the drawing area but not reshaped. This value is not available with the <code>drawPoint</code> function.
'reshape'	ROI can be reshaped within the drawing area but not translated. This value is not available with the <code>drawPoint</code> function and the <code>drawRectangle</code> function.

**Label — ROI label**

' ' (default) | character vector | string

ROI label, specified as the comma-separated pair consisting of 'Label' and a character vector or string. By default, this property is set to an empty array ( ' ' ) and the ROI has no label.

**LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as the comma-separated pair consisting of 'LineWidth' and a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Parent — ROI parent**

Axes object

ROI parent, specified as the comma-separated pair consisting of 'Parent' and an Axes object. A UIAxes cannot be the parent of the ROI.

**Position — Position of ROI**

1-by-4 array

Position of the ROI, specified as the comma-separated pair consisting of 'Position' and a 1-by-4 array of the form [xmin, ymin, width, height]. xmin and ymin specify the location of the upper left corner of the rectangle. width and height specify the extend to the rectangle in two dimensions. This value changes automatically when you draw or move the ROI.

**Rotatable — Ability of ROI to be rotated**

false (default) | true

Ability of the ROI to be rotated, specified as the comma-separated pair consisting of 'Rotatable' and the logical value true or false. When the value is false (default), the rectangle cannot be rotated. When the value is true, you can rotate the rectangle by clicking near the markers at the corners.

**RotationAngle — Angle around center of rectangle**

0 (default) | nonnegative numeric scalar

Angle around the center of the rectangle, specified as the comma-separated pair consisting of 'RotationAngle' and a nonnegative numeric scalar. The angle is measured in degrees in a clockwise direction. The value of this property changes automatically when you draw or move the ROI. The value of RotationAngle does not impact the values in Position. The Position property represents the rectangle prior to any rotation. When the rectangle is rotated, use the Vertices property to determine the location of the rotated rectangle.

**Selected — Selection state of ROI**

false (default) | true

Selection state of the ROI, specified as the comma-separated pair consisting of 'Selected' and true or false. You can also set this value interactively. For example,

clicking the ROI selects it and sets this value to `true`. Similarly, if you **Ctrl-click** the ROI, deselects it and sets this value to `false`.

**SelectedColor — Color of ROI when selected**

'none' (default) | colorspec

Color of the ROI when selected, specified as the comma-separated pair consisting of 'SelectedColor' and a MATLAB ColorSpec (Color Specification). If 'SelectedColor' is set to 'none', then the value of Color defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**StripeColor — Color of ROI stripe**

'none' (default) | ColorSpec value

Color of the ROI stripe, specified as the comma-separated pair consisting of 'StripeColor' and a MATLAB ColorSpec (Color Specification) value. If you specify 'StripeColor', the ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**Tag — Tag to associate with ROI**

character vector | string scalar

Tag to associate with the ROI, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

**UIContextMenu — Context menu**

ContextMenu object

Context menu, specified as the comma-separated pair consisting of 'UIContextMenu' and a `uicontextmenu` object. Specify this name-value pair to display a custom context menu when you right-click on the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

**UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as the comma-separated pair consisting of 'UserData' and any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

**Vertices — Locations of points on edge of ROI***n*-by-2 array

Locations of points on edge of the ROI, returned as the comma-separated pair consisting of 'Vertices' and an *n*-by-2 array.

This property is read-only.

**Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as the comma-separated pair consisting of 'Visible' and one of the values in this table.

Value	Description
'on'	Specify this value to display the ROI (default).
'off'	Specify this value to hide the ROI without deleting it. You still can access the properties of an invisible ROI.

## Output Arguments

**h — Rectangular ROI**`images.roi.Rectangle` object

Rectangular ROI, returned as an `images.roi.Rectangle` object.

## See Also

`drawassisted` | `drawcircle` | `drawellipse` | `drawfreehand` | `drawline` | `drawpoint` | `drawpolygon` | `drawpolyline` | `images.roi.Rectangle`

**Introduced in R2018b**



# edge

Find edges in intensity image

## Syntax

```
BW = edge(I)
BW = edge(I,method)
BW = edge(I,method,threshold)
BW = edge(I,method,threshold,direction)
BW = edge( ____, 'nothinning')
BW = edge(I,method,threshold,sigma)
BW = edge(I,method,threshold,h)
```

```
[BW,threshOut] = edge( ____ )
[BW,threshOut,Gv,Gh] = edge( ____ )
```

## Description

`BW = edge(I)` returns a binary image `BW` containing 1s where the function finds edges in the input image `I` and 0s elsewhere. By default, `edge` uses the Sobel edge detection method.

You optionally can find edges using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`BW = edge(I,method)` detects edges in image `I` using the edge-detection algorithm specified by `method`.

The 'Canny' and 'approxcanny' methods are not supported on a GPU.

`BW = edge(I,method,threshold)` returns all edges that are stronger than `threshold`.

`BW = edge(I,method,threshold,direction)` specifies the orientation of edges to detect. The Sobel and Prewitt methods can detect edges in the vertical direction, horizontal direction, or both. The Roberts method can detect edges at angles of 45° from

horizontal, 135° from horizontal, or both. This syntax is valid only when method is 'Sobel', 'Prewitt', or 'Roberts'.

`BW = edge( ____, 'nothinning' )` skips the edge-thinning stage, which can improve performance. This syntax is valid only when method is 'Sobel', 'Prewitt', or 'Roberts'.

`BW = edge(I,method,threshold,sigma)` specifies `sigma`, the standard deviation of the filter. This syntax is valid only when method is 'log' or 'Canny'.

`BW = edge(I,method,threshold,h)` detects edges using the 'zerocross' method with a filter, `h`, that you specify. This syntax is valid only when method is 'zerocross'.

`[BW,threshOut] = edge( ____ )` also returns the threshold value.

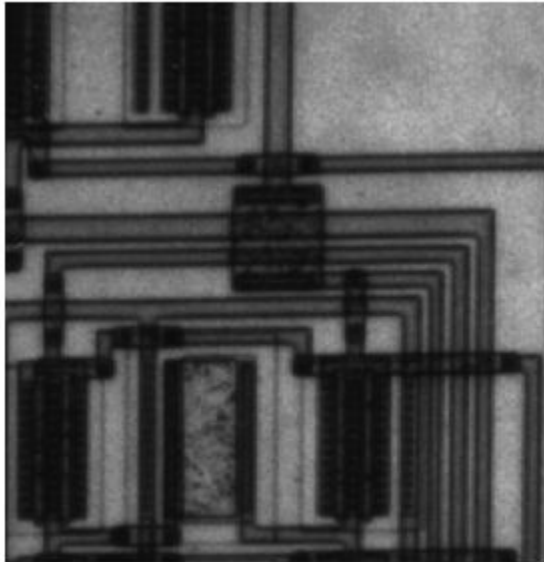
`[BW,threshOut,Gv,Gh] = edge( ____ )` also returns the directional gradient magnitudes. For the Sobel and Prewitt methods, `Gv` and `Gh` correspond to the vertical and horizontal gradients. For the Roberts methods, `Gv` and `Gh` correspond to the gradient at angles of 45° and 135° from horizontal, respectively. This syntax is valid only when method is 'Sobel', 'Prewitt', or 'Roberts'.

## 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.

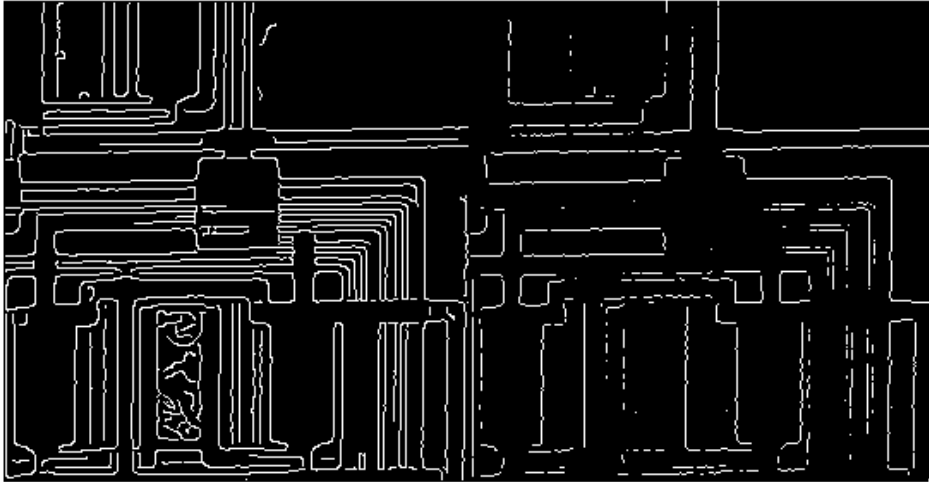
```
BW1 = edge(I, 'Canny');
```

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 a grayscale image, creating a `gpuArray`.

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

Find edges using the Prewitt method.

```
BW = edge(I, 'prewitt');
```

Display the results.

```
figure, imshow(BW)
```

## Input Arguments

### I — Input image

grayscale image | binary image | gpuArray

Input image, specified as a 2-D grayscale or binary image.

For the 'approxcanny' method, images of datatype `single` or `double` must be normalized in the range [0 1].

To find edges using a GPU, specify I as a `gpuArray` that contains a grayscale or binary image.

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

### method — Edge detection method

'Sobel' (default) | 'Prewitt' | 'Roberts' | 'log' | 'zerocross' | 'Canny' | 'approxcanny'

Edge detection method, specified as one of the following.

Method	Description
'Sobel'	Finds edges at those points where the gradient of the image I is maximum, using the Sobel approximation to the derivative.
'Prewitt'	Finds edges at those points where the gradient of I is maximum, using the Prewitt approximation to the derivative.
'Roberts'	Finds edges at those points where the gradient of I is maximum, using the Roberts approximation to the derivative.
'log'	Finds edges by looking for zero-crossings after filtering I with a Laplacian of Gaussian (LoG) filter.
'zerocross'	Finds edges by looking for zero-crossings after filtering I with a filter that you specify, h

Method	Description
'Canny'	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.
'approxcanny'	Finds edges using an approximate version of the Canny edge detection algorithm that provides faster execution time at the expense of less precise detection. Floating point images are expected to be normalized in the range [0 1].

---

**Note** The 'Canny' and 'approxcanny' methods are not supported on a GPU.

---

### **threshold — Sensitivity threshold**

numeric scalar | 2-element vector | []

Sensitivity threshold, specified as a numeric scalar for any method, or a 2-element vector for the 'Canny' and 'approxcanny' methods. `edge` ignores all edges that are not stronger than `threshold`. For more information about this parameter, see “Algorithm” on page 1-619.

- If you do not specify `threshold`, or if you specify an empty array (`[]`), then `edge` chooses the value or values automatically.
- For the 'log' and 'zerocross' methods, if you specify the threshold value `0`, then the output image has closed contours because it includes all the zero-crossings in the input image.
- The 'Canny' and 'approxcanny' methods use two thresholds. `edge` disregards all edges with edge strength below the lower threshold, and preserves all edges with edge strength above the higher threshold. You can specify `threshold` as a 2-element vector of the form `[low high]` with `low` and `high` values in the range `[0 1]`. You can also specify `threshold` as a numeric scalar, which `edge` assigns to the higher threshold. In this case, `edge` uses `threshold*0.4` as the lower threshold.

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'`. The `direction` argument is only valid when the method is `'Sobel'`, `'Prewitt'`, or `'Roberts'`.

---

**Note** If you select the Roberts method, then the `'horizontal'` direction actually detects edges at an angle of 135° from horizontal, and the `'vertical'` direction detects edges at an angle of 45° from horizontal.

---

Data Types: `char` | `string`

**h — Filter**

matrix

Filter, specified as a matrix. The `h` argument is supported by the `'zerocross'` method only.

Data Types: `double`

**sigma — Standard deviation of the filter**

scalar

Standard deviation of the filter, specified as a scalar. The `sigma` argument is supported by the `'Canny'` and `'log'` methods only.

Method	Description
<code>'Canny'</code>	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> .
<code>'log'</code> (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 <code>n-by-n</code> , where <code>n=ceil(sigma*3)*2+1</code> .

Data Types: `double`

## Output Arguments

### **BW — Output binary image**

logical array | gpuArray

Output binary image, returned as a logical array of the same size as I, with 1s where the function finds edges in I and 0s elsewhere.

If you use a GPU to find the edges, then BW is returned as a gpuArray that contains a logical array.

### **threshOut — Threshold value used in the computation**

numeric scalar | 2-element vector | [] | gpuArray

Threshold value used in the computation, returned as a 2-element vector for the 'Canny' method, an empty vector ([]) for the 'approxcanny' method, or a numeric scalar for all other edge detection methods.

If you use a GPU to find the edges, then threshOut is returned as a gpuArray that contains a numeric scalar.

### **Gv — Vertical gradient**

numeric array | gpuArray

Vertical gradient, returned as a numeric array of the same size as I.

---

**Note** If you select the Roberts method, then edge returns the gradient calculated at an angle of 45° from horizontal.

---

If you use a GPU to find the edges, then Gv is returned as a gpuArray that contains a numeric array.

### **Gh — Horizontal gradient**

numeric array | gpuArray

Horizontal gradient, returned as a numeric array of the same size as I.

---

**Note** If you select the Roberts method, then edge returns the gradient calculated at an angle of 135° from horizontal.

---



If you use a GPU to find the edges, then `Gh` is returned as a `gpuArray` that contains a numeric array.

## Tips

- 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', ___)`

## Algorithms

- 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 to detect fewer edge pixels, or lower to detect more edge pixels.

## 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 “Understand 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, then edge displays an image. This syntax is not supported with code generation.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Canny method is not supported on the GPU.

For more information, see “Image Processing on a GPU”.

### See Also

`fspecial` | `imgradient` | `imgradientxy`

**Introduced before R2006a**

## edge3

Find edges in 3-D intensity volume

### Syntax

```
BW = edge3(V, 'approxcanny', thresh)
BW = edge3(V, 'approxcanny', thresh, sigma)
BW = edge3(V, 'Sobel', thresh)
BW = edge3(V, 'Sobel', thresh, 'nothinning')
```

### Description

`BW = edge3(V, 'approxcanny', thresh)` returns the edges found in the intensity or a binary volume `V` using the approximate Canny method. The approximate Canny method finds edges by looking for local maxima of the gradient of `V`. `edge3` calculates the gradient using the derivative of a Gaussian smoothed volume.

For the approximate Canny method, `thresh` is a two-element vector in which the first element is the low threshold, and the second element is the high threshold, [`lowthresh highthresh`]. If you specify a scalar for `thresh`, `edge3` uses this value for the high threshold and  $0.4 * \text{thresh}$  for the low threshold.

The approximate Canny method uses two thresholds to detect strong and weak edges, and includes the weak edges in the output only if they are connected to strong edges. This method is more likely than the Sobel method to detect true weak edges.

`BW = edge3(V, 'approxcanny', thresh, sigma)` returns the edges found in the intensity or binary volume `V`, where `sigma` is a scalar that specifies the standard deviation of the Gaussian smoothing filter. `sigma` can also be a 1-by-3 vector, [`SigmaX, SigmaY, SigmaZ`], specifying different standard deviations in each direction. For anisotropic volumes that have different scales in each direction, use multiple `sigma` values. By default, `sigma` is  $\sqrt{2}$  and is isotropic. `edge3` chooses the size of the filter automatically, based on `sigma`.

`BW = edge3(V, 'Sobel', thresh)` accepts an intensity or a binary volume `V` and returns a binary volume `BW`, that is the same size as `V`, with 1s where the function finds edges in `V` and 0s elsewhere.

The Sobel method finds edges using the Sobel approximation to the derivative. It returns edges at those points where the gradient of `V` is maximum.

`thresh` is a scalar that specifies the sensitivity threshold for the Sobel method. `edge3` ignores all edges that are not stronger than `thresh`.

`BW = edge3(V, 'Sobel', thresh, 'nothinning')` speeds up the operation of the algorithm by skipping the additional edge-thinning stage. By default, or when `'thinning'` is specified, the algorithm applies edge thinning.

## Examples

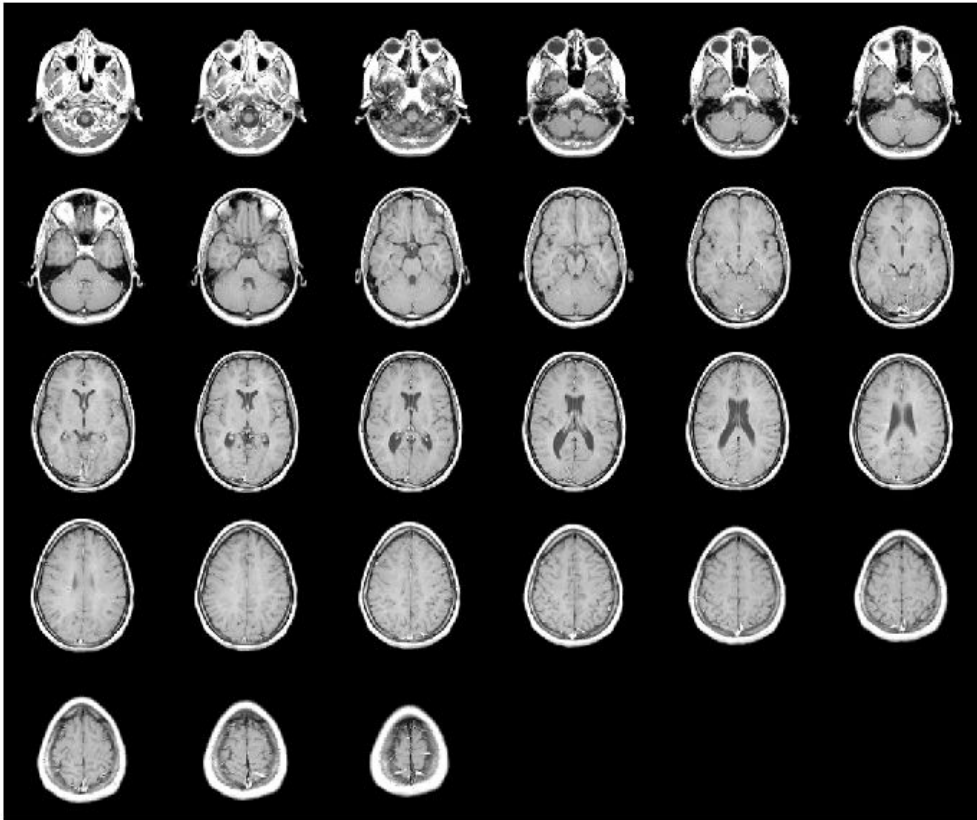
### Find Edges of MRI Volume using Approximate Canny Method

Load volumetric data and remove any singleton dimensions.

```
load mri
V = squeeze(D);
```

Visualize original image.

```
montage(reshape(V, size(D)), map);
```

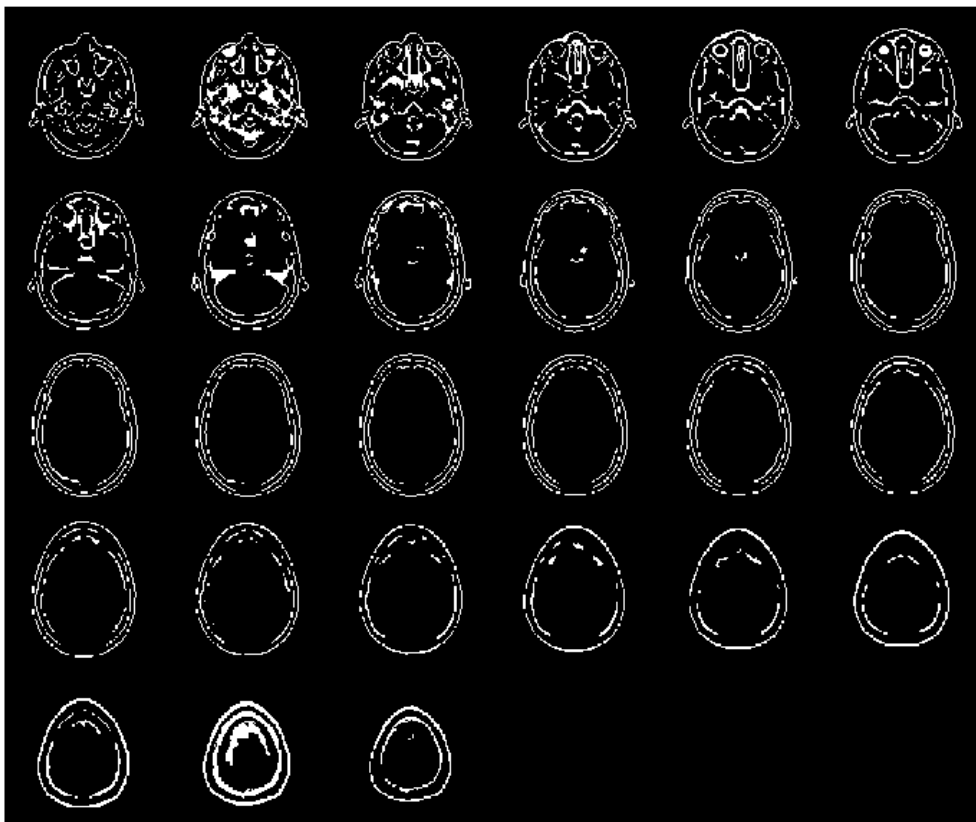


Detect edges in the volume.

```
BW = edge3(V, 'aproxcanny', 0.6);
```

Visualize the detected edges. You can also view the result using the Volume Viewer app.

```
montage(reshape(BW, size(D)))
```



## Input Arguments

### **V** — Input volume

nonsparse 3-D numeric array

Input volume, specified as a nonsparse 3-D numeric array.

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

### **thresh — Sensitivity threshold**

scalar | 1-by-2 numeric vector

Sensitivity threshold, specified as a scalar or, for approximate Canny, a 1-by-2 numeric vector of the form [`lowthresh` `hightresh`]. If you specify a scalar, `edge3` uses this value for the high threshold and  $0.4 * \text{thresh}$  for the low threshold.

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

### **sigma — Standard deviation of the Gaussian filter**

scalar | 1-by-3 numeric vector

Standard deviation of the Gaussian filter, specified as a scalar or a 1-by-3 numeric vector of the form [`SigmaX` `SigmaY` `SigmaZ`].

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

## **Output Arguments**

### **BW — Binary volume containing 1s indicating edges and 0s elsewhere**

3-D numeric array

Binary volume containing 1s indicating edges and 0s elsewhere, returned as a 3-D numeric array, the same size as `V`.

## **See Also**

`edge`

**Introduced in R2017b**

## 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 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.

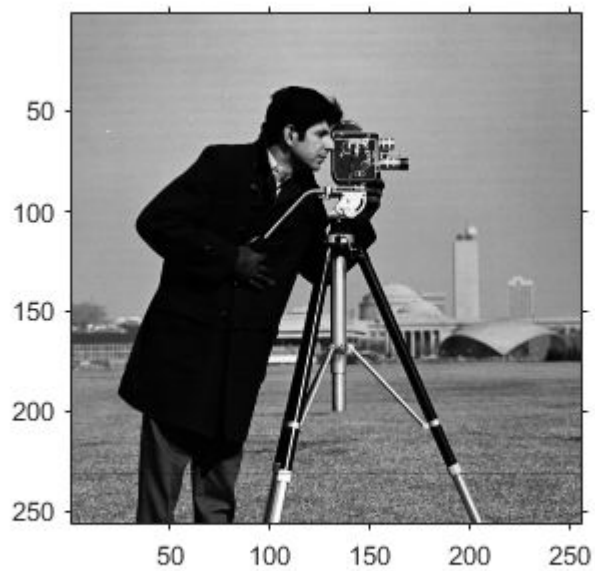
The `edgetaper` function reduces the ringing effect in image deblurring methods that use the discrete Fourier transform, such as `deconvwnr`, `deconvreg`, and `deconvlucy`.

### Examples

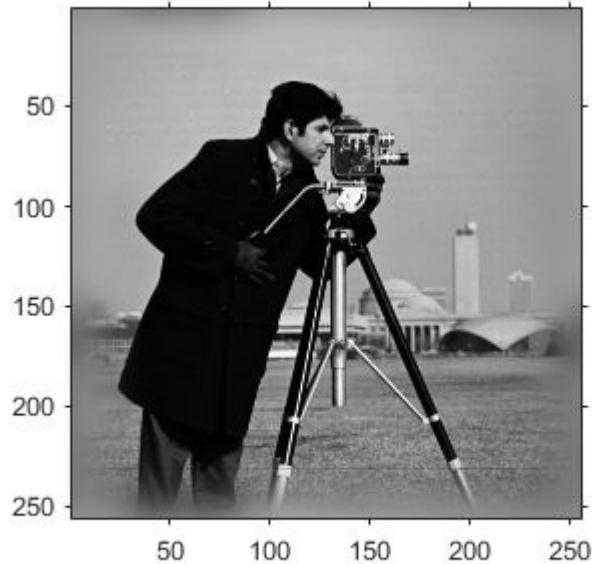
#### Blur the Edges of an Image

```
original = imread('cameraman.tif');  
PSF = fspecial('gaussian',60,10);  
edgesTapered = edgetaper(original,PSF);  
figure, imshow(original,[]);
```





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



## Input Arguments

### **I — Input image**

numeric array

Input image, specified as a numeric array.

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

### **PSF — Point spread function**

numeric array

Point spread function, specified as a numeric array. The size of the PSF cannot exceed half of the image size in any dimension.

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

## Output Arguments

### **J — Weighted sum of original image and its blurred version**

numeric array

Weighted sum of original image and its blurred version, returned as a numeric array the same size and class as I. 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.

## 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`.

### Examples

#### Calculate Entropy of Grayscale Image

Read image into the workspace.

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

Calculate the entropy.

```
J = entropy(I)
```

```
J = 6.9439
```

### Input Arguments

#### **I** — Grayscale image

real, nonsparse numeric array

Grayscale image, specified as a real, nonsparse numeric array. `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.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `logical`

## Output Arguments

### **e** — Entropy

numeric scalar

Entropy of image `I`, returned as a numeric scalar.

Data Types: `double`

## Definitions

### Entropy

Entropy is a statistical measure of randomness that can be used to characterize the texture of the input image.

Entropy is defined as  $-\sum(p \cdot \log_2(p))$ , where `p` contains the normalized histogram counts returned from `imhist`.

## Tips

- By default, `entropy` uses two bins for logical arrays and 256 bins for `uint8`, `uint16`, or `double` arrays. `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.

## 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**

`entropyfilt` | `imhist`

**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`.

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.

## 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.

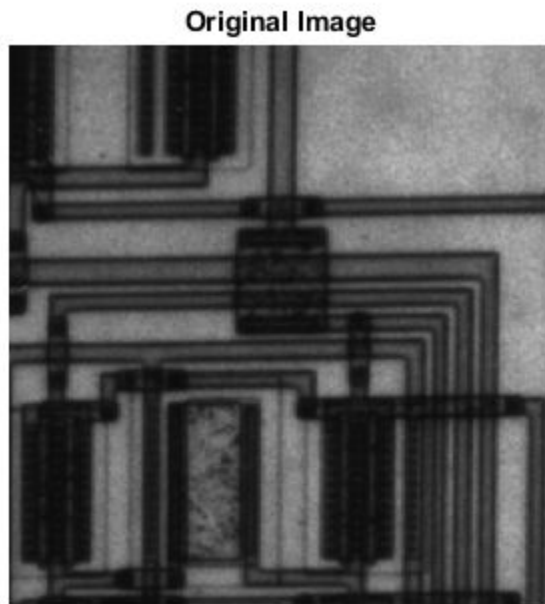
```
I = imread('circuit.tif');
```

Perform entropy filtering using `entropyfilt`.

```
J = entropyfilt(I);
```

Show the original image and the processed image.

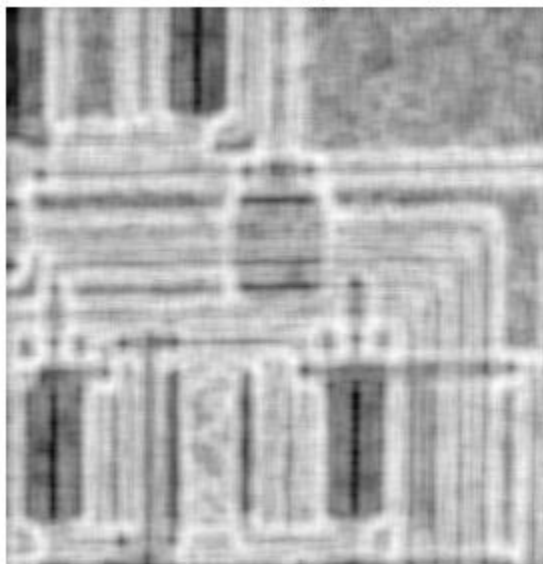
```
imshow(I)  
title('Original Image')
```



```
figure  
imshow(J,[])  
title('Result of Entropy Filtering')
```



### Result of Entropy Filtering



## Input Arguments

### **I** — Image to be filtered

real, nonsparse numeric array

Image to be filtered, specified as a real, nonsparse numeric array. 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.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `logical`

### **nhood** — Neighborhood

`true(9)` (default) | multidimensional, logical or numeric array containing zeros and ones

Neighborhood, specified as a multidimensional, logical or numeric array containing zeros and ones. The size of `nhood` 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 other shapes, such as a disk, use the `strel` function to create a structuring element object of the desired shape. Then, extract the neighborhood from the structuring element object's `neighborhood` property.

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

## Output Arguments

### J — Filtered image

numeric array

Filtered image, returned as a numeric array the same size as the input image and of class `double`.

## Algorithms

`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.

## 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`

## Topics

“What Is Image Filtering in the Spatial Domain?”

**Introduced before R2006a**

# esfrChart

Imatest® edge spatial frequency response (eSFR) test chart

## Description

An `esfrChart` object stores the positions and measurements of regions of interest of the Enhanced Imatest® edge spatial frequency response (eSFR) test chart.

The Enhanced eSFR test chart is an extended version of the ISO 12233:2014 standard test chart [2].

## Creation

## Syntax

```
chart = esfrChart(A)
chart = esfrChart(A,'Sensitivity',s)
chart = esfrChart(A,'RegistrationPoints',p)
```

## Description

`chart = esfrChart(A)` creates an `esfrChart` object and sets the `Image` on page 1-0 property from input image `A`.

`chart = esfrChart(A,'Sensitivity',s)` creates an `esfrChart` object, using sensitivity `s` during chart import.

`chart = esfrChart(A,'RegistrationPoints',p)` creates an `esfrChart` object and sets the `RegistrationPoints` on page 1-0 property from points in argument `p`.

## Input Arguments

### **s** — Sensitivity

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

Sensitivity of chart detection, specified as a numeric scalar in the range [0, 1]. If you set a high sensitivity value, the `esfrChart` model detects more points of interest with which to register the test chart image.

Data Types: `single` | `double`

## Properties

### **Image** — Test chart image

*m-by-n-by-3* RGB image

Test chart image, specified as an *m-by-n-by-3* RGB image.

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

### **SlantedEdgeROIs** — Position and intensity values of slanted edges

60-by-1 vector of structures

Position and intensity values of slanted edges, specified as a 60-by-1 vector of structures. Each element in the vector corresponds to one ROI and contains the following fields:

Field	Description
ROI	A 1-by-4 vector specifying the spatial extent of the ROI. The vector has the form [X Y Width Height]. X and Y are the coordinates of the top-left corner of the ROI. Width and Height are the width and height of the ROI, in pixels. ROI is of data type <code>double</code> .
ROIIntensity	Array of intensity values within the ROI, in RGB format. The array has dimensions Height-by-Width-by-3. The data type of ROIIntensity matches the data type of the Image on page 1-0 property.

### **GrayROIs** — Position and intensity values of gray patches

20-by-1 vector of structures

Position and intensity values of gray patches, specified as a 20-by-1 vector of structures. Each element in the vector corresponds to one ROI and contains the following fields:

Field	Description
ROI	A 1-by-4 vector specifying the spatial extent of the ROI. The vector has the form [X Y Width Height]. X and Y are the coordinates of the top-left corner of the ROI. Width and Height are the width and height of the ROI, in pixels. ROI is of data type double.
ROIIntensity	Array of intensity values within the ROI, in RGB format. The array has dimensions Height-by-Width-by-3. The data type of ROIIntensity matches the data type of the Image on page 1-0 property.

### **ColorROIs — Position and intensity values of color patches**

16-by-1 vector of structures

Position and intensity values of color patches, specified as a 16-by-1 vector of structures. Each element in the vector corresponds to one ROI and contains the following fields:

Field	Description
ROI	A 1-by-4 vector specifying the spatial extent of the ROI. The vector has the form [X Y Width Height]. X and Y are the coordinates of the top-left corner of the ROI. Width and Height are the width and height of the ROI, in pixels. ROI is of data type double.
ROIIntensity	Array of intensity values within the ROI, in RGB format. The array has dimensions Height-by-Width-by-3. The data type of ROIIntensity matches the data type of the Image on page 1-0 property.

### **RegistrationPoints — Position of registration points**

4-by-2 numeric matrix

Position of registration points used to orient the image, specified as a 4-by-2 numeric matrix. The four rows correspond to the top-left, top-right, bottom-right, and bottom-left registration points, respectively. The two columns represent pixel coordinates in [x, y] format.

Data Types: double

### ReferenceGrayLab — Reference values of gray ROIs

20-by-3 numeric matrix

Reference values of gray ROIs in the CIE 1976 L\*a\*b\* color space, specified as a 20-by-3 numeric matrix. The three columns contain the L\*, a\*, and b\* values of the gray patches, respectively. The rows contain the reference intensities of the 20 gray ROIs, in the same sequential order.

---

**Note** The `esfrChart` object includes default CIE 1976 L\*a\*b\* values for the gray ROIs. However, the actual reference values can vary depending on several factors, such as print quality.

---

Data Types: double

### ReferenceColorLab — Reference values of color ROIs

16-by-3 numeric matrix

Reference values of color ROIs in the CIE 1976 L\*a\*b\* color space, specified as a 16-by-3 numeric matrix. The three columns contain the L\*, a\*, and b\* values of the color patches, respectively. The rows contain the reference intensities of the 16 color ROIs, in the same sequential order.

---

**Note** The `esfrChart` object includes default CIE 1976 L\*a\*b\* values for the color ROIs. However, the actual reference values can vary depending on several factors, such as print quality. Accurate reference color values result in more faithful color reproduction measurements.

---

Data Types: double

## Object Functions

<code>measureSharpness</code>	Measure spatial frequency response using <code>Imatest®</code> eSFR chart
<code>measureChromaticAberration</code>	Measure chromatic aberration at slanted edges using <code>Imatest®</code> eSFR chart
<code>measureNoise</code>	Measure noise using <code>Imatest®</code> eSFR chart
<code>measureColor</code>	Measure color reproduction using <code>Imatest®</code> eSFR chart

measureIlluminant  
displayChart

Measure scene illuminant using Imatest® eSFR chart  
Display Imatest® eSFR chart with overlaid regions of interest

## Examples

### Create an eSFR Chart Object from a Test Image

Read an image of an eSFR chart into the workspace. Display the image.

```
I = imread('eSFRTestImage.jpg');  
figure  
imshow(I)  
title('Captured Image of eSFR Chart')  
text(size(I,2),size(I,1)+15, ...  
      ['Chart courtesy of Imatest',char(174)], 'FontSize',10, 'HorizontalAlignment', 'right')
```

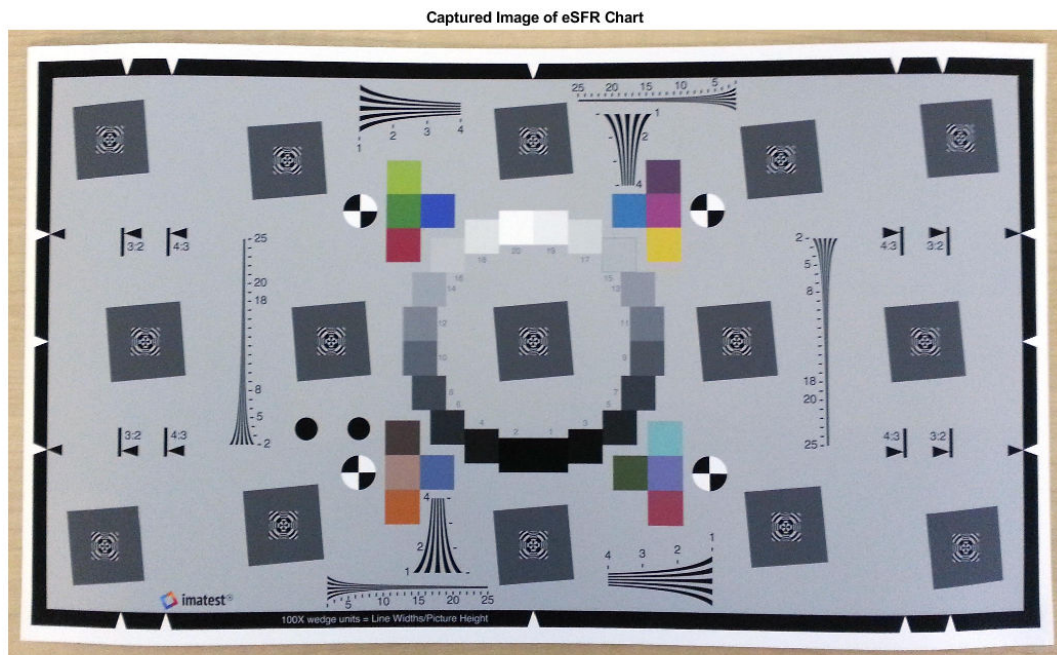


Chart courtesy of Imatest®



Linearize the image. The displayed chart will appear darker because the image no longer has gamma correction.

```
I_lin = rgb2lin(I);
```

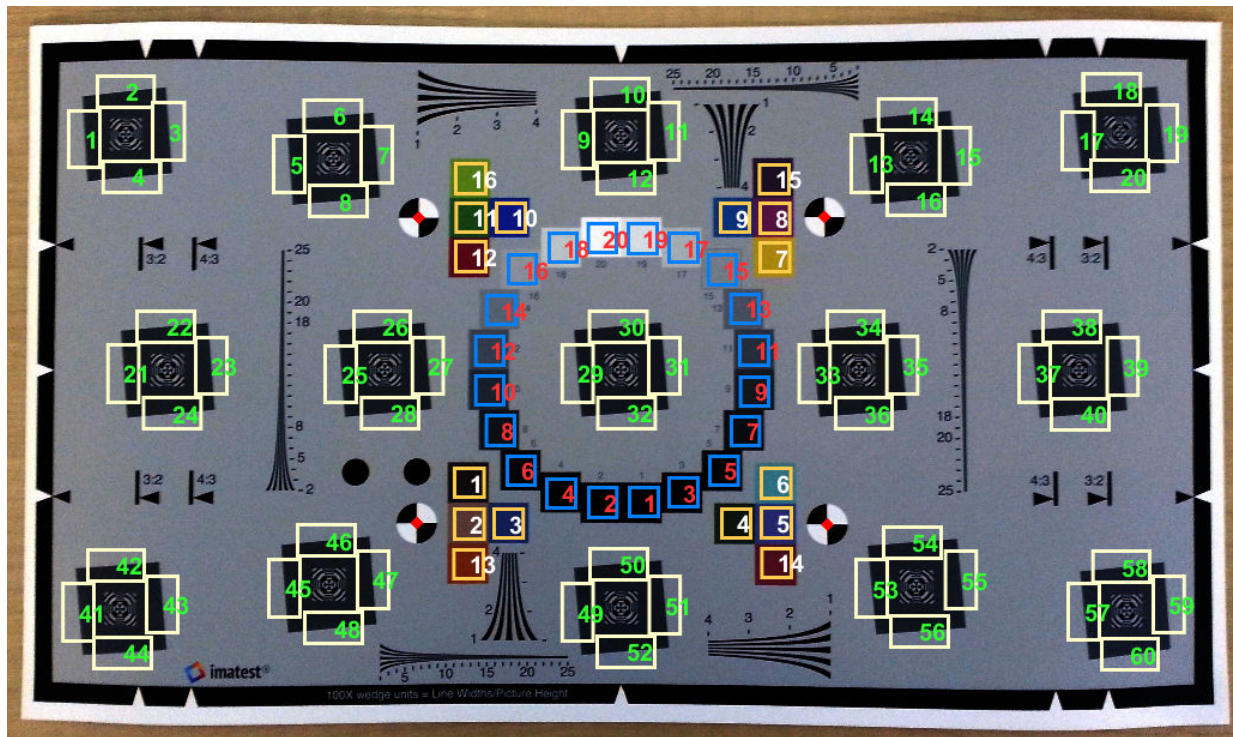
Create an `esfrChart` object using the linearized chart image. Specify the sensitivity that the `esfrChart` model uses to detect the points with which to register the chart image.

```
chart = esfrChart(I_lin, 'Sensitivity', 0.6)
```

```
chart =  
  esfrChart with properties:  
      Image: [1836x3084x3 uint8]  
  SlantedEdgeROIs: [60x1 struct]  
      GrayROIs: [20x1 struct]  
      ColorROIs: [16x1 struct]  
  RegistrationPoints: [4x2 double]  
  ReferenceGrayLab: [20x3 double]  
  ReferenceColorLab: [16x3 double]
```

Display the imported eSFR chart. Regions of interest (ROI) are highlighted and labeled.

```
displayChart(chart)
```



The chart is imported correctly. All 60 slanted edge ROIs (labeled with green numbers) are visible and centered on appropriate edges. 20 gray patch ROIs (labeled in red) and 16 color patch ROIs (labeled in white) are visible and are contained within the boundary of each patch.

## Create eSFR Chart Object Using Specified Registration Points

Create an `esfrChart` object by specifying the coordinates of the four registration points. Registration points are located at the center of the black-and-white checkered circles.

Read an image of an eSFR chart into the workspace.

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

Display the image and configure it to collect four registration points.

```
figure
imshow(I)
[X, Y] = ginput(4);
```

Click the registration points in this order: top-left, top-right, bottom-right, bottom-left.

Create an `esfrChart` object, specifying the four registration points. Display the imported eSFR chart. Regions of interest are highlighted and labeled. The registration points appear in red.

```
chart = esfrChart(I, 'RegistrationPoints', [X, Y]);
displayChart(chart);
```

## Tips

- For accurate and reliable results, acquire an image of the test chart according to standard specifications outlined in the ISO standard and by the manufacturer [1] [2]. As a simple guideline, align the chart horizontally on a light background. Cover over 90% of the field of view with the chart, but ensure that the top and bottom edges of the chart are still visible. For reliable measurements, set the minimum image width to at least 500 pixels.
- You can capture an image of the eSFR test chart at the full 16:9 aspect ratio, or at an aspect ratio of 3:2 or 4:3, as specified on the chart.
- To ensure that the chart is properly imported, visually verify the test chart image using the `displayChart` function.

## References

[1] ISO 12233:2014. "Photography - Electronic still picture imaging - Resolution and spatial frequency responses." *International Organization for Standardization; ISO/TC 42 Photography*. URL: <https://www.iso.org/standard/59419.html>.

[2] *Using eSFR ISO Part 1*. URL: [https://www.imatest.com/docs/esfriso\\_instructions](https://www.imatest.com/docs/esfriso_instructions).

## See Also

`displayColorPatch` | `plotChromaticity` | `plotSFR`

## **Topics**

*“Anatomy of an eSFR Chart”*

*“Evaluate Quality Metrics on eSFR Test Chart”*

**Introduced in R2017b**

## fan2para

Convert fan-beam projections to parallel-beam

### Syntax

```
P = fan2para(F,D)
P = fan2para(F,D,Name,Value)
[P,parallel_sensor_positions,parallel_rotation_angles] = fan2para(
___)
```

### Description

`P = fan2para(F,D)` converts the fan-beam data `F` to the parallel-beam data `P`. Each column of `F` contains the fan-beam data at one rotation angle. `D` is the distance from the fan-beam vertex to the center of rotation.

`P = fan2para(F,D,Name,Value)` uses name-value pairs to control aspects of the data conversion. Argument names can be abbreviated, and case does not matter.

`[P,parallel_sensor_positions,parallel_rotation_angles] = fan2para( ___)` returns the parallel-beam sensor locations in `parallel_sensor_positions` and rotation angles in `parallel_rotation_angles`.

### Examples

#### Recover Parallel-beam Data from Fan-beam Data

Create synthetic parallel-beam data.

```
ph = phantom(128);
```

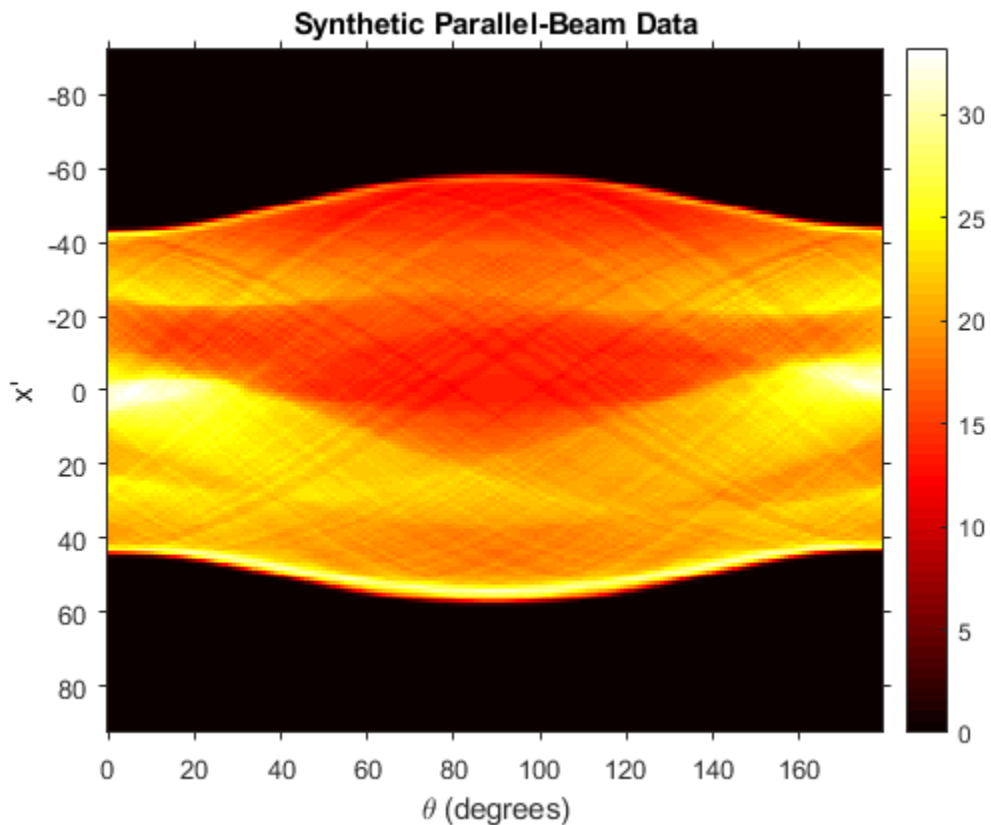
Calculate the parallel beam transform and display it.

```
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);
```

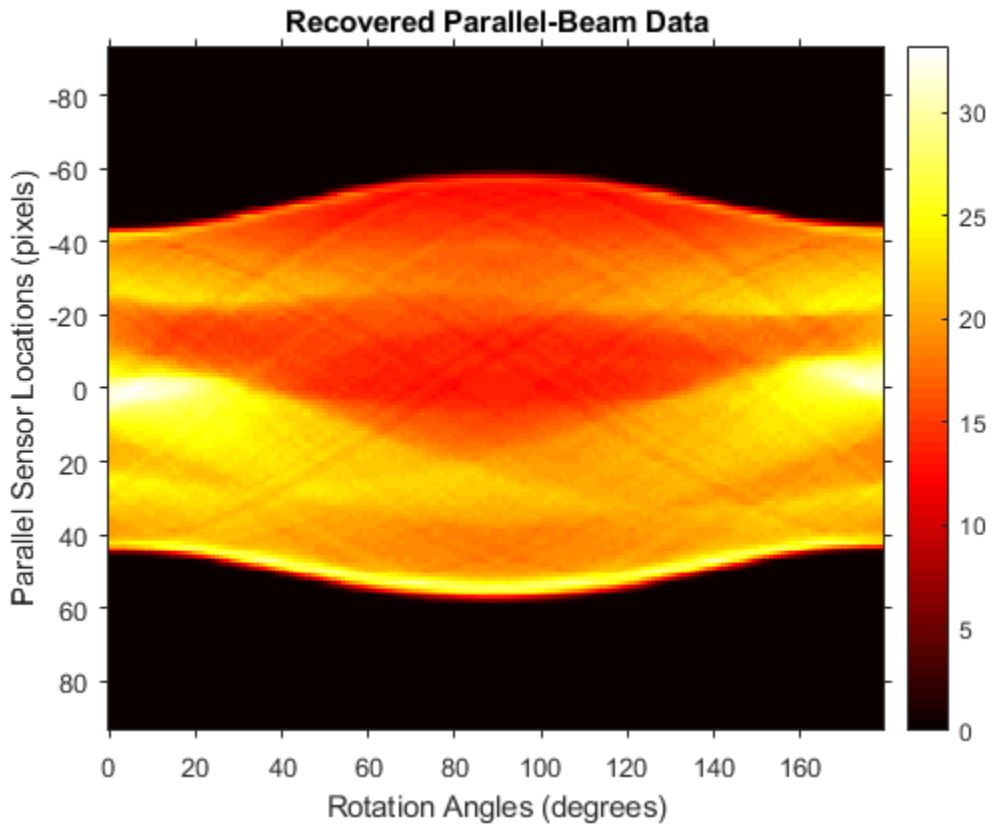
Recover original parallel-beam data.

```

[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

```



## Input Arguments

### **F — Fan-beam projection data**

numeric matrix

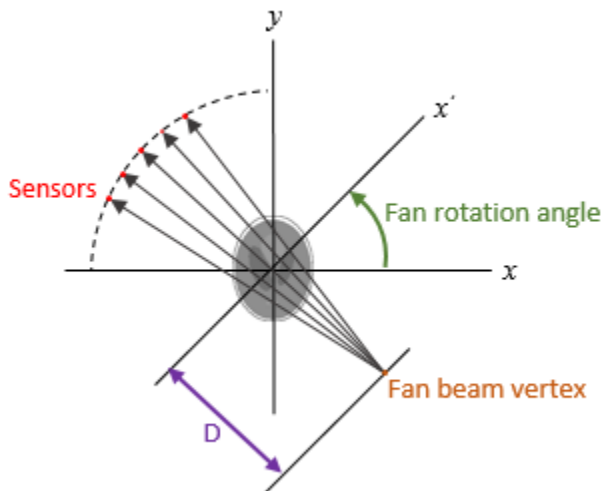
Fan-beam projection data, specified as a numeric matrix. Each column of  $F$  contains the fan-beam data at one rotation angle. The number of columns indicates the number of fan-beam rotation angles and the number of rows indicates the number of fan-beam sensors.

Data Types: `double` | `single`

### **D — Distance from fan beam vertex to center of rotation**

positive number

Distance in pixels from the fan beam vertex to the center of rotation, specified as a positive number. `fan2para` assumes that the center of rotation is the center point of the projections, which is defined as `ceil(size(F,1)/2)`. The figure illustrates  $D$  in relation to the fan-beam vertex for one fan-beam projection.



Data Types: `double` | `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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `P = fan2para(F,D,'FanRotationIncrement',5)`

### FanCoverage — Range of fan-beam rotation

'cycle' (default) | 'minimal'

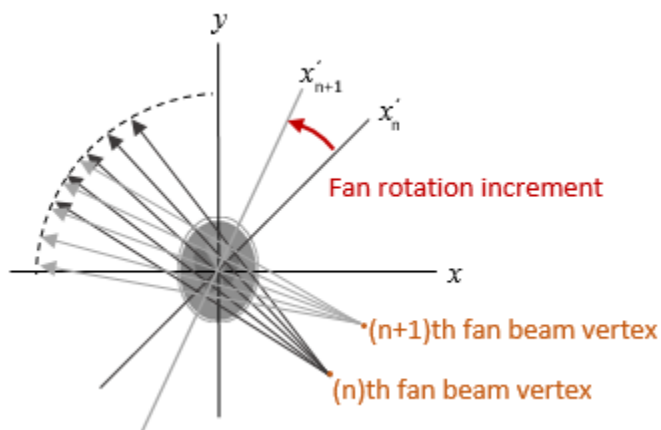
Range of fan-beam rotation, specified as the comma-separated pair consisting of 'FanCoverage' and one of these values.

- 'cycle' — Rotate through the full range [0, 360) degrees.
- 'minimal' — Rotate through the minimum range necessary to represent the object.

### FanRotationIncrement — Fan-beam rotation angle increment

1 (default) | positive scalar

Fan-beam rotation angle increment in degrees, specified as the comma-separated pair consisting of 'FanRotationIncrement' and a positive scalar.

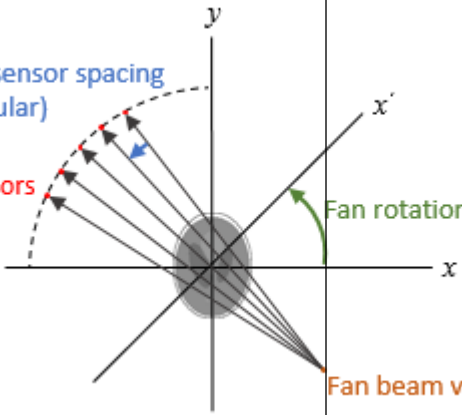


Data Types: double

**FanSensorGeometry — Fan-beam sensor positioning**

'arc' (default) | 'line'

Fan-beam sensor positioning, specified as the comma-separated pair consisting of 'FanSensorGeometry' and one of the following values.

Value	Meaning	Diagram
'arc'	<p>Sensors are spaced at equal angles along a circular arc at distance D from the center of rotation.</p> <p>FanSensorSpacing defines the angular spacing in degrees.</p>	

Value	Meaning	Diagram
'line'	<p>Sensors are spaced at equal distances along a line that is parallel to the <math>x'</math> axis. The closest sensor is distance <math>D</math> from the center of rotation.</p> <p>FanSensorSpacing defines the distance between fan-beams on the <math>x'</math> axis, in pixels.</p>	

### FanSensorSpacing — Fan-bean sensor spacing

1 (default) | positive scalar

Fan-bean sensor spacing, specified as the comma-separated pair consisting of 'FanSensorSpacing' and a positive scalar.

- If FanSensorGeometry is 'arc', then FanSensorSpacing defines the angular spacing in degrees.
- If FanSensorGeometry is 'line', then FanSensorSpacing defines the linear distance between fan-beams, in pixels. Linear spacing is measured on the  $x'$  axis.

Data Types: double

### Interpolation — Type of interpolation

'Linear' (default) | 'nearest' | 'spline' | 'pchip'

Type of interpolation used between the parallel-beam and fan-beam data, specified as the comma-separated pair consisting of 'Interpolation' and one of these values.

'nearest' — Nearest-neighbor

'linear' — Linear (the default)

'spline' — Piecewise cubic spline

'pchip' — Piecewise cubic Hermite (PCHIP)

## **ParallelCoverage — Range of parallel-beam rotation**

'halfcycle' (default) | 'cycle'

Range of parallel-beam rotation, specified as the comma-separated pair consisting of 'ParallelCoverage' and one of these values.

- 'cycle' — Parallel data covers the full range of [0, 360) degrees.
- 'halfcycle' — Parallel data covers [0, 180) degrees.

## **ParallelRotationIncrement — Parallel-beam rotation angle increment**

positive scalar

Parallel-beam rotation angle increment in degrees, specified as the comma-separated pair consisting of 'ParallelRotationIncrement' and a positive scalar  $k$  such that  $180/k$  is an integer. If you do not specify `ParallelRotationIncrement`, then the default value is equal to `FanRotationIncrement`.

Data Types: double

## **ParallelSensorSpacing — Parallel-beam sensor spacing**

positive scalar

Parallel-beam sensor spacing in pixels, specified as the comma-separated pair consisting of 'ParallelSensorSpacing' and a positive scalar. The range of parallel-beam sensor locations is computed from the range of fan angles, *fanangles*, according to:  $[D \cdot \sin(\min(\textit{fanangles})) \ D \cdot \sin(\max(\textit{fanangles}))]$ .

If you do not specify `ParallelSensorSpacing`, then 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.

Data Types: double

## Output Arguments

### **P — Parallel-beam projection data**

numeric matrix

Parallel-beam projection data, returned as a numeric matrix. Each column of **P** contains the parallel-beam data at one rotation angle. The number of columns indicates the total number of parallel-beam rotation angles and is equal to the length of `parallel_rotation_angles`. The number of rows indicates the total number of parallel-beam sensors and is equal to the length of `parallel_sensor_positions`.

Data Types: double

### **parallel\_sensor\_positions — Parallel-beam sensor locations**

numeric column vector

Parallel-beam sensor locations, returned as a numeric column vector.

Data Types: double

### **parallel\_rotation\_angles — Parallel-beam rotation angles**

numeric row vector

Parallel-beam rotation angles, returned as a numeric row vector.

Data Types: double

## See Also

fanbeam | ifanbeam | iradon | para2fan | phantom | radon

**Introduced before R2006a**

## fanbeam

Fan-beam transform

### Syntax

```
F = fanbeam(I,D)
F = fanbeam(I,D,Name,Value)
[F,fan_sensor_positions,fan_rotation_angles] = fanbeam( ___ )
```

### Description

`F = fanbeam(I,D)` computes the fan-beam projection data (sinogram) `F` from the image `I`. Each column of `F` contains fan-beam projection data at one rotation angle. `D` is the distance from the fan-beam vertex to the center of rotation.

`F = fanbeam(I,D,Name,Value)` uses name-value pairs to specify the rotation increment and sensor spacing. Argument names can be abbreviated, and case does not matter.

`[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`.

### Examples

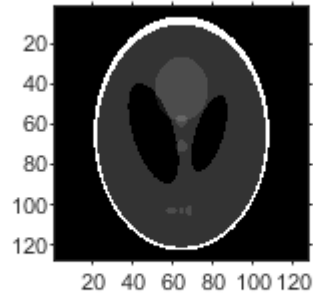
#### Compute Fan-beam Projections for Rotation Angles Over Entire Image

Set the IPT preference to make the axes visible.

```
iptsetpref('ImshowAxesVisible','on')
```

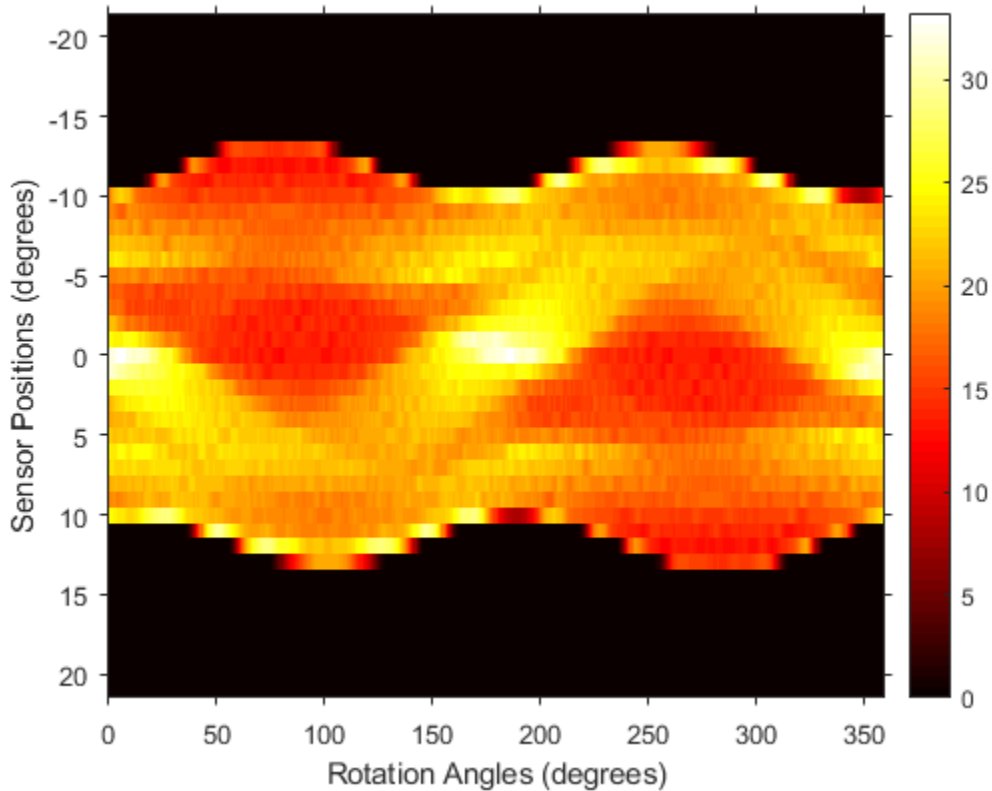
Create a sample image and display it.

```
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.

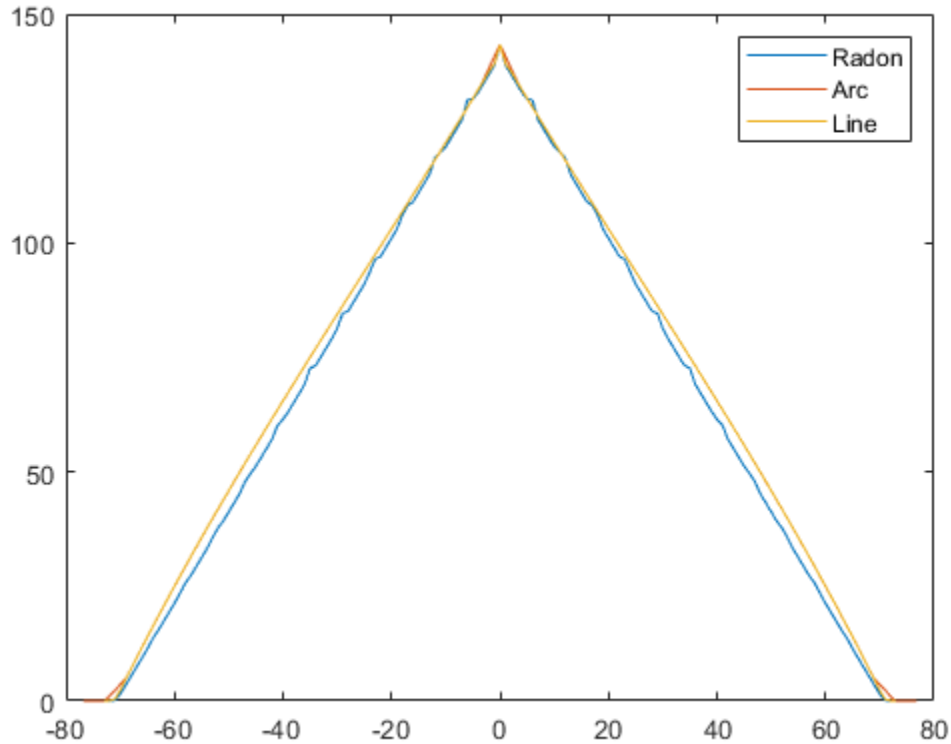
```
[Fline,FposLine] = fanbeam(I,D,...  
    'FanSensorGeometry','line',...  
    'FanRotationIncrement',dtheta);
```

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')
```



## Input Arguments

### **I** — Input image

2-D numeric matrix | 2-D logical matrix

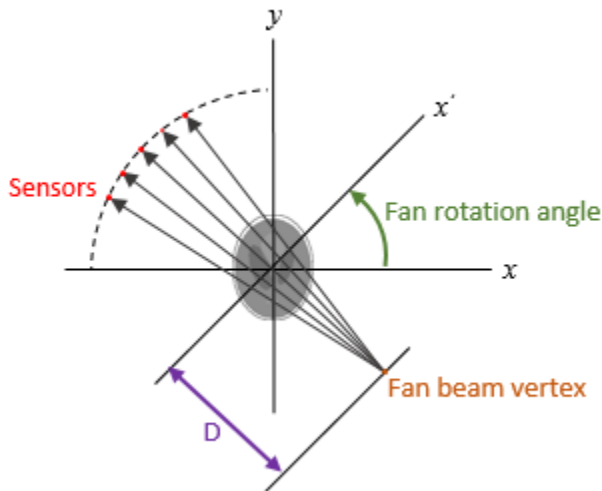
Input image, specified as a 2-D numeric or logical matrix.

### **D** — Distance from fan beam vertex to center of rotation

positive number

Distance in pixels from the fan beam vertex to the center of rotation, specified as a positive number. 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 1-664 for guidelines on specifying  $D$ . The figure illustrates  $D$  in relation to the fan-beam vertex for one fan-beam geometry.



## Name-Value Pair Arguments

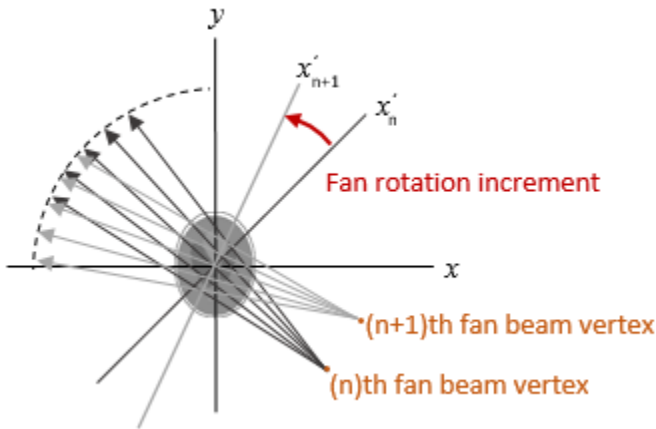
Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `F = fanbeam(I,D,'FanRotationIncrement',5)`

### **FanRotationIncrement** — Fan-beam rotation angle increment

1 (default) | positive scalar

Fan-beam rotation angle increment in degrees, specified as the comma-separated pair consisting of `'FanRotationIncrement'` and a positive scalar.



Data Types: double

**FanSensorGeometry — Fan-beam sensor positioning**

'arc' (default) | 'line'

Fan-beam sensor positioning, specified as the comma-separated pair consisting of 'FanSensorGeometry' and one of the following values.

Value	Meaning	Diagram
'arc'	<p>Sensors are spaced at equal angles along a circular arc at distance <math>D</math> from the center of rotation.</p> <p>FanSensorSpacing defines the angular spacing in degrees.</p>	<p>A diagram illustrating fan beam sensor positioning for the 'arc' value. A central sensor is shown as a grey circle. Sensors are positioned along a dashed circular arc. Blue arrows indicate the 'Fan sensor spacing (angular)'. Red dots represent the 'Sensors'. A green arrow indicates the 'Fan rotation'. An orange dot represents the 'Fan beam vertex'. The coordinate system has a horizontal <math>x</math>-axis and a vertical <math>y</math>-axis. A rotated axis is labeled <math>x'</math>.</p>

Value	Meaning	Diagram
'line'	<p>Sensors are spaced at equal distances along a line that is parallel to the <math>x'</math> axis. The closest sensor is distance <math>D</math> from the center of rotation.</p> <p>FanSensorSpacing defines the distance between fan-beams on the <math>x'</math> axis, in pixels.</p>	

### FanSensorSpacing — Fan-bean sensor spacing

1 (default) | positive scalar

Fan-bean sensor spacing, specified as the comma-separated pair consisting of 'FanSensorSpacing' and a positive scalar.

- If FanSensorGeometry is 'arc', then FanSensorSpacing defines the angular spacing in degrees.
- If FanSensorGeometry is 'line', then FanSensorSpacing defines the linear distance between fan-beams, in pixels. Linear spacing is measured on the  $x'$  axis.

Data Types: double

## Output Arguments

### F — Fan-beam projection data

*numsensors-by-numangles* numeric matrix

Fan-beam projection data, returned as a *numsensors-by-numangles* numeric matrix. *numsensors* is the number of fan-beam sensors and *numangles* is the number of fan-beam rotation angles. Each column of *F* contains the fan-beam sensor samples at one rotation angle.

Data Types: double

## **fan\_sensor\_positions — Location of fan-beam sensors**

*numsensors-by-1* numeric vector

Location of fan-beam sensors, returned as a *numsensors-by-1* numeric vector.

- If *FanSensorGeometry* is 'arc' (the default), then *fan\_sensor\_positions* contains the fan-beam spread angles.
- If *FanSensorGeometry* is 'line', then *fan\_sensor\_positions* contains the fan-beam sensor positions along the *x*' axis. See *FanSensorSpacing* for more information.

*fanbeam* determines the number of sensors by calculating how many beams are required to cover the entire image for any rotation angle. Fewer sensors are required to cover the image when the distance *D* between the fan-beam vertex and the center of rotation is large.

Data Types: double

## **fan\_rotation\_angles — Rotation angle of fan-beam sensors**

*1-by-numangles* numeric vector

Rotation angle of fan-beam sensors, returned as a *1-by-numangles* numeric vector. *numangles* is  $360/\text{FanRotationIncrement}$ .

Data Types: double

## **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 | ifanbeam | iradon | 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.

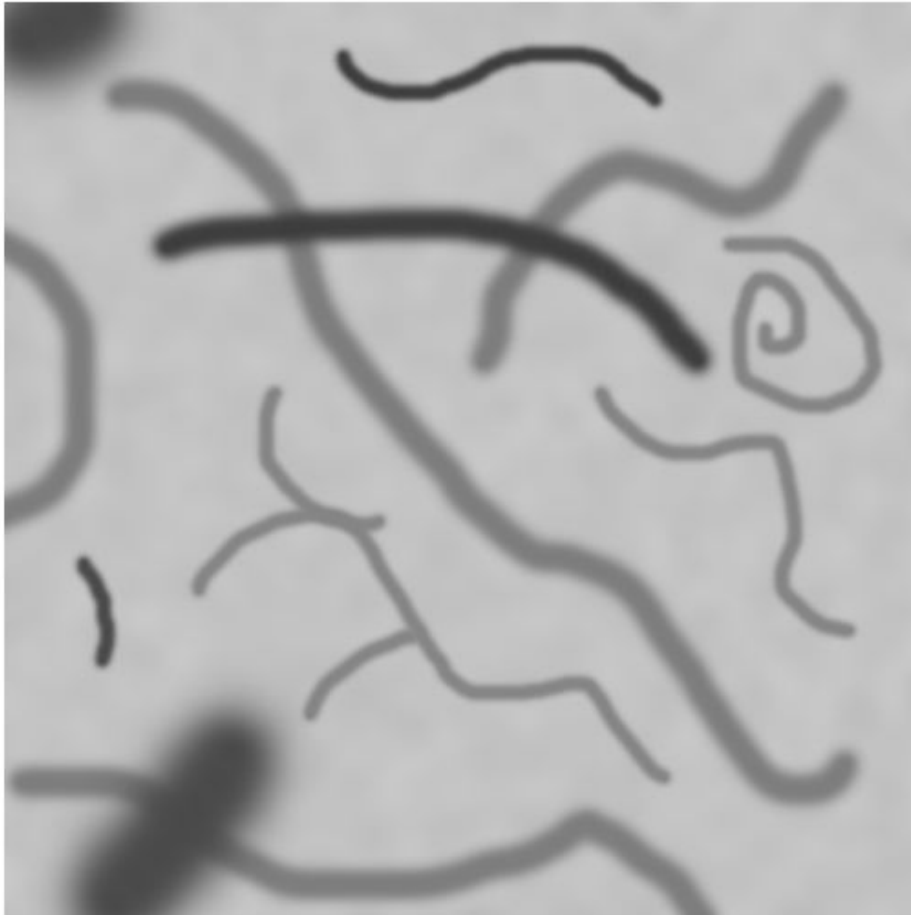
### Examples

#### 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')
```



## 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 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 values:

Value	Description
'bright'	Structure is brighter than the background.
'dark'	Structure is darker than the background.

Data Types: char | string

## 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

`edge` | `imgradient`

**Introduced in R2017a**

## findbounds

Find output bounds for spatial transformation

### Syntax

```
outbounds = findbounds(tform,inbounds)
```

### Description

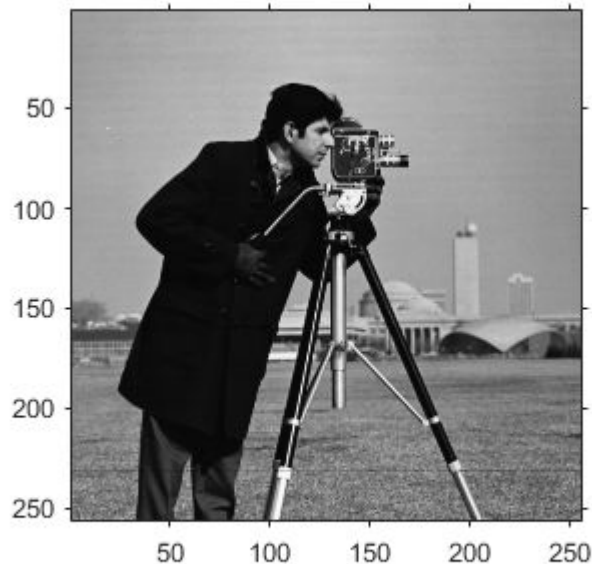
`outbounds = findbounds(tform,inbounds)` estimates the output bounds corresponding to a given spatial transformation and a set of input bounds. `tform` is a spatial transformation structure. `inbounds` is a 2-by-`num_dims` matrix that specifies the lower and upper bounds of the output image. `outbounds` 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 the image.

```
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])
```

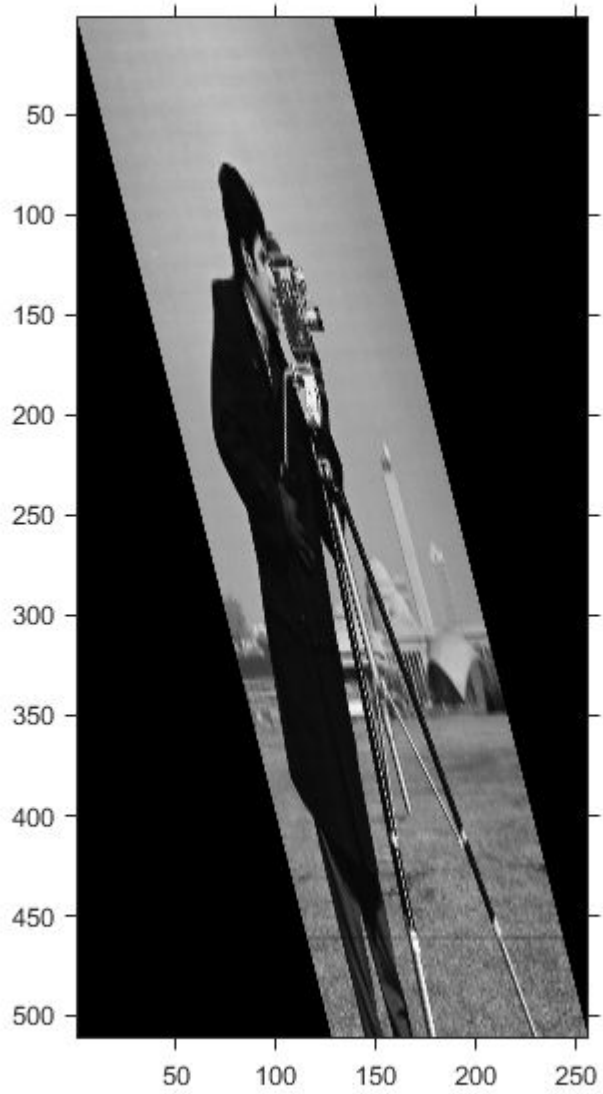
```
outb = 2x2
```

```
    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

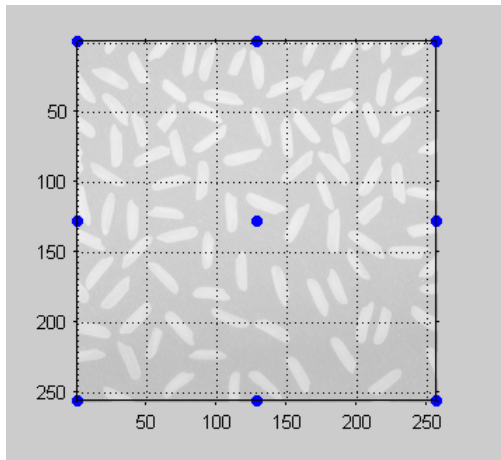
- 1 `findbounds` first creates a grid of input-space points. These points are at the center, corners, and middle of each edge in the image.

```
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

`tformarray` | `tformfwd` | `tforminv`

**Introduced before R2006a**

# fitbrisque

Fit custom model for BRISQUE image quality score

## Syntax

```
model = fitbrisque(imds,opinionScores)
```

## Description

`model = fitbrisque(imds,opinionScores)` creates a Blind/Referenceless Image Spatial Quality Evaluator (BRISQUE) model from a reference image datastore, `imds`, with corresponding human perceptual differential mean opinion score (DMOS) values, `opinionScore`.

---

**Note** To use the `fitbrisque` function, you must have Statistics and Machine Learning Toolbox™.

---

## Examples

### Calculate BRISQUE Score Using Custom Feature Model

Train a custom BRISQUE model from a set of quality-aware features and corresponding human opinion scores. Use the custom model to calculate a BRISQUE score for an image of a natural scene.

Save images from an image datastore. These images all have compression artifacts resulting from JPEG compression.

```
setDir = fullfile(toolboxdir('images'),'imdata');  
imds = imageDatastore(setDir,'FileExtensions',{' .jpg'});
```

Specify the opinion score for each image. The following differential mean opinion score (DMOS) values are for illustrative purposes only. They are not real DMOS values obtained through experimentation.

```
opinionScores = 100*rand(1,size(imds.Files,1));
```

Create the custom model of quality-aware features using the image datastore and the opinion scores. Because the scores are random, the property values will vary.

```
model = fitbrisque(imds,opinionScores')
```

```
Extracting features from 37 images.  
..  
Completed 4 of 37 images. Time: Calculating...  
...  
Completed 13 of 37 images. Time: 00:22 of 00:51  
...  
Completed 23 of 37 images. Time: 00:32 of 00:48  
..  
Completed 31 of 37 images. Time: 00:42 of 00:50  
..Training support vector regressor...
```

Done.

```
model =  
    brisqueModel with properties:  
  
        Alpha: [35x1 double]  
        Bias: 56.2632  
    SupportVectors: [35x36 double]  
        Kernel: 'gaussian'  
        Scale: 0.2717
```

Read an image of a natural scene that has the same type of distortion as the training images. Display the image.

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



Calculate the BRISQUE score for the image using the custom model. Display the score.

```
brisqueI = brisque(I,model);  
fprintf('BRISQUE score for the image is %0.4f.\n',brisqueI)
```

```
BRISQUE score for the image is 78.7367.
```

## Input Arguments

### **imds** — Reference image datastore

ImageDatastore object

Reference image datastore, specified as an ImageDatastore object. Images within the datastore must be real, nonsparse,  $m$ -by- $n$  or  $m$ -by- $n$ -by-3 arrays of data type single,

double, int16, uint8, or uint16. The images must have a known set of distortions such as compression artifacts, blurring, or noise.

## **opinionScores — Human opinion scores**

numeric vector

Human opinion scores, specified as a numeric vector with values in the range [0, 100]. Each element in `opinionScores` is the human perceptual DMOS value corresponding to an image in the datastore `imds`. The length of `opinionScores` is equal to the number of images in `imds`.

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

## **Output Arguments**

### **model — Custom model of image features**

`brisqueModel` object

Custom model of image features, returned as a `brisqueModel` object. `model` contains a support vector regressor (SVR) with a Gaussian kernel trained to predict the BRISQUE quality score.

## **References**

- [1] Mittal, A., A. K. Moorthy, and A. C. Bovik. "No-Reference Image Quality Assessment in the Spatial Domain." *IEEE Transactions on Image Processing*. Vol. 21, Number 12, December 2012, pp. 4695-4708.
- [2] Mittal, A., A. K. Moorthy, and A. C. Bovik. "Referenceless Image Spatial Quality Evaluation Engine." Presentation at the 45th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, November 2011.

## **See Also**

### **Functions**

`brisque` | `fitniqe` | `niqe`

### **Objects**

`brisqueModel`



## **Topics**

“Image Quality Metrics”

“Train and Use a No-Reference Quality Assessment Model”

**Introduced in R2017b**

## 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.

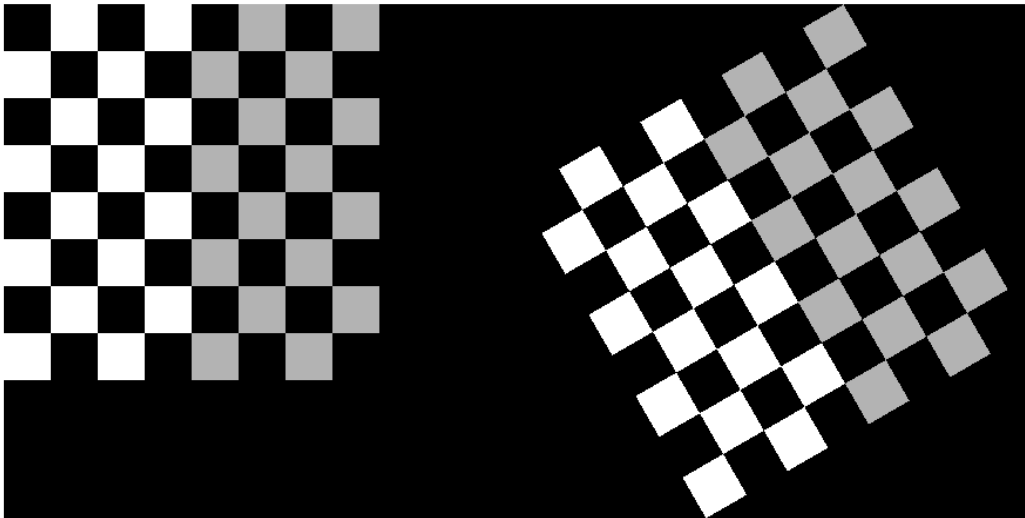
## Examples

### 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];
```

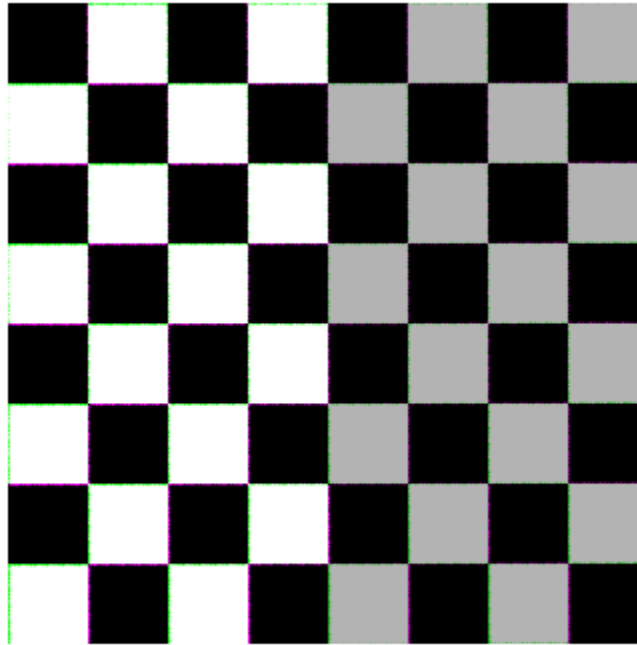
Create a geometric transformation that can be used to align the two images, returned as an `affine2d` geometric transformation object.

```
tform = fitgeotrans(movingPoints, fixedPoints, 'NonreflectiveSimilarity')
```

```
tform =  
  affine2d with properties:  
          T: [3x3 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 1-689.

Data Types: `char` | `string`

### **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

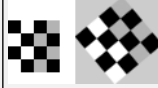

transformation object






Transformation, returned as a transformation object. The type of object depends on the transformation type. For example, if you specify the transformation type `'affine'`, then `tform` is an `affine2d` object. If you specify `'pwl'`, then `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
<code>'nonreflective similarity'</code>	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	
<code>'similarity'</code>	Same as <code>'nonreflective similarity'</code> with the addition of optional reflection.	3	

Transformation Type	Description	Minimum Number of Control Point Pairs	Example
'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)	
'pwl'	Use this transformation (piecewise linear) 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

#### Functions

`cpselect` | `imwarp`

#### Objects

`LocalWeightedMeanTransformation2D` | `PiecewiseLinearTransformation2D` | `PolynomialTransformation2D` | `affine2d` | `projective2d`

### Topics

“Matrix Representation of Geometric Transformations”

**Introduced in R2013b**

## fitniqe

Fit custom model for NIQE image quality score

### Syntax

```
model = fitniqe(imds)
model = fitniqe(imds,Name,Value)
```

### Description

`model = fitniqe(imds)` creates a Naturalness Image Quality Evaluator (NIQE) model from reference image datastore `imds`.

`model = fitniqe(imds,Name,Value)` creates a NIQE model using additional parameters to control the model calculation.

### Examples

#### Calculate NIQE Score Using Custom Feature Model

Train a custom NIQE model and calculate a NIQE score for a natural image using the trained model.

Train a custom model using natural images stored in an image datastore.

```
setDir = fullfile(toolboxdir('images'),'imdata');
imds = imageDatastore(setDir,'FileExtensions',{' .jpg'});
model = fitniqe(imds);
```

```
Extracting features from 37 images.
..
Completed 4 of 37 images. Time: Calculating...
...
Completed 13 of 37 images. Time: 00:20 of 00:49
...
```

```
Completed 24 of 37 images. Time: 00:32 of 00:47
...
Done.
```

Read an image of a natural scene. Display the image.

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



Calculate the NIQE score for the image using the custom model. Display the score.

```
nqeI = nqe(I,model);
fprintf('NIQE score for the image is %0.4f.\n',nqeI)
```

```
NIQE score for the image is 1.8601.
```

## Fit Custom NIQE Model Using Specified Block Size

Create a custom NIQE model from a set of natural images. Use the custom model to calculate a NIQE score for a new image.

Load images from an image datastore.

```
setDir = fullfile(toolboxdir('images'),'imdata');  
imds = imageDatastore(setDir,'FileExtensions',{' .jpg'});
```

Create the custom model of NSS features using the image datastore. Specify a block size and use the default sharpness threshold.

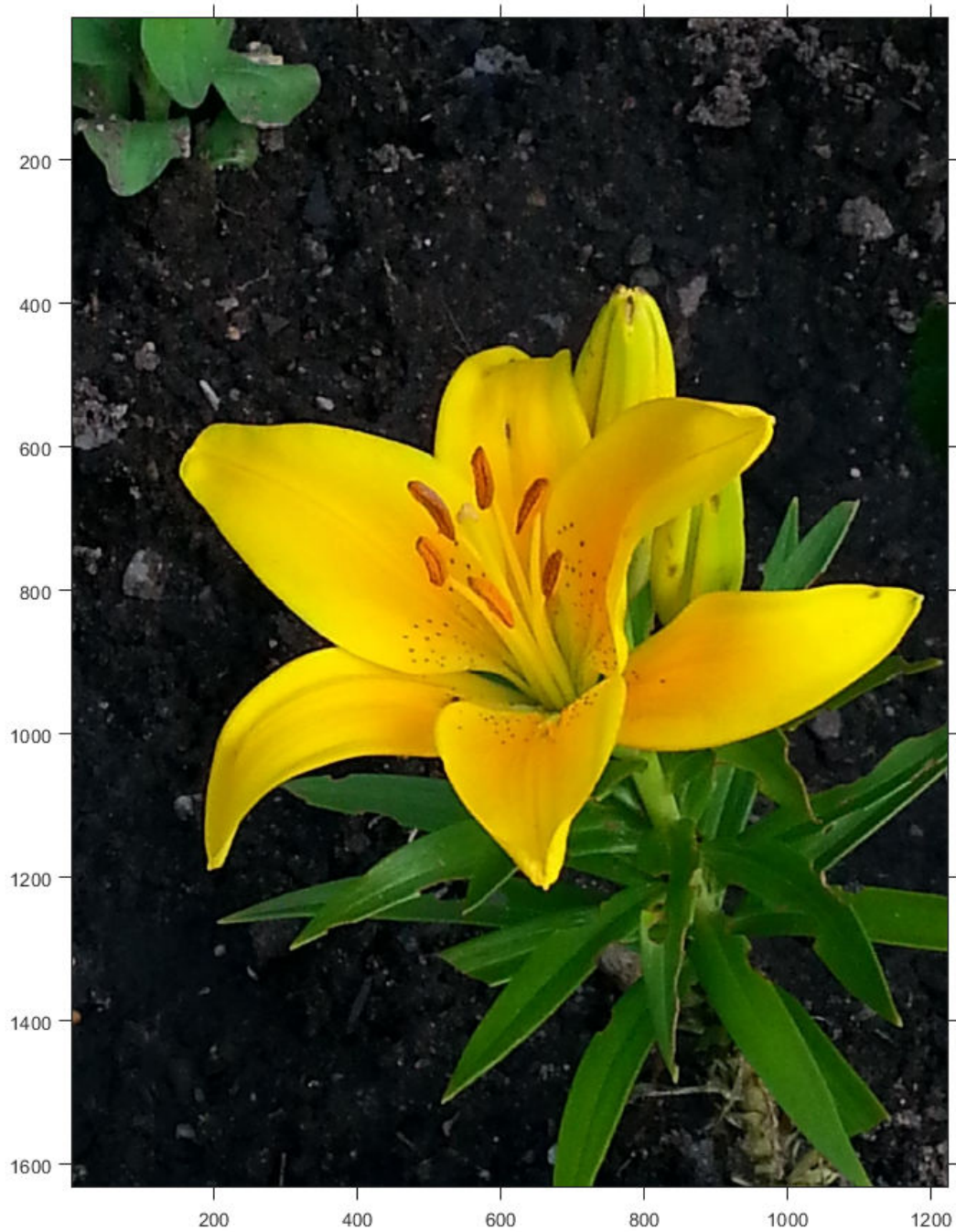
```
model = fitniqe(imds,'BlockSize',[48 96])
```

```
Extracting features from 37 images.  
..  
Completed 4 of 37 images. Time: Calculating...  
..  
Completed 12 of 37 images. Time: 00:25 of 01:00  
..  
Completed 17 of 37 images. Time: 00:39 of 01:18  
.....  
Completed 26 of 37 images. Time: 00:49 of 01:07  
.....  
Done.
```

```
model =  
    niqeModel with properties:  
  
        Mean: [1x36 double]  
    Covariance: [36x36 double]  
        BlockSize: [48 96]  
    SharpnessThreshold: 0
```

Read a natural image into the workspace. Display the image.

```
I = imread('yellowlily.jpg');  
imshow(I)
```



Calculate the NIQE score for the image using the custom model. Display the score.

```
niqeI = niqe(I,model);  
fprintf('NIQE score for the image is %0.4f.\n',niqeI)
```

NIQE score for the image is 2.9944.

## Input Arguments

### **imds** — Reference image datastore

ImageDatastore object

Reference image datastore, specified as an ImageDatastore object. Images within the datastore must be real, nonsparse,  $m$ -by- $n$  or  $m$ -by- $n$ -by-3 matrices of data type `single`, `double`, `int16`, `uint8`, or `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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

Example: `model = fitniqe(imds,'BlockSize',[48 36])` fits a NIQE model using 48-by-36 pixel blocks.

### **BlockSize** — Block size used to partition the images

[96 96] (default) | 2-element row vector of positive even integers

Block size used to partition the images, specified as the comma-separated pair consisting of `'BlockSize'` and a 2-element row vector of positive even integers. Blocks are nonoverlapping. Natural scene statistics, which are calculated from the blocks, define the output `model`.

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

### **SharpnessThreshold** — Sharpness threshold

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

Sharpness threshold, specified as the comma-separated pair consisting of `'SharpnessThreshold'` and a numeric scalar in the range [0, 1]. The sharpness

threshold, `s`, controls which image blocks are used to compute the model. `fitnqe` computes the model using all blocks that have sharpness more than `s` times the maximum sharpness among all blocks.

Data Types: `single` | `double`

## Output Arguments

**model** — Custom model of image features

`nqeModel` object

Custom model of image features, returned as a `nqeModel` object.

## Tips

- The custom dataset specified in the image datastore `imds` should consist of images that are perceptually pristine to human subjects. However, the definition of pristine depends on the application. For example, a pristine set of microscopy images has a different set of quality criteria than images of buildings or outdoor scenes. When training a custom NIQE model, use images with varied image content and with potentially different sets of quality criteria.

## References

- [1] Mittal, A., R. Soundararajan, and A. C. Bovik. "Making a Completely Blind Image Quality Analyzer." *IEEE Signal Processing Letters*. Vol. 22, Number 3, March 2013, pp. 209-212.

## See Also

### Functions

`brisque` | `fitbrisque` | `nqe`

### Objects

`nqeModel`

## **Topics**

“Image Quality Metrics”

“Train and Use a No-Reference Quality Assessment Model”

**Introduced in R2017b**



# fliptform

Flip input and output roles of spatial transformation structure

## Syntax

```
tflip = fliptform(T)
```

## Description

`tflip = fliptform(T)` creates a new TFORM spatial transformation structure by flipping the roles of the inputs and outputs in an existing TFORM structure.

## Examples

### Flip Spatial Transformation Structure

Create a spatial transformation structure.

```
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 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: [1x1 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

TFORM spatial transformation structure

Spatial transformation, specified as a TFORM spatial transformation structure.

Data Types: `struct`

## Output Arguments

### **tflip** — Flipped spatial transformation

TFORM spatial transformation structure

Flipped spatial transformation, returned as a TFORM spatial transformation structure.

## See Also

`maketform` | `tformfwd` | `tforminv`

**Introduced before R2006a**

## freqz2

2-D frequency response

### Syntax

```
[H, f1, f2] = freqz2(h)
[H, f1, f2] = freqz2(h, [n1 n2])
[H, f1, f2] = freqz2(h, f1, f2)
[ ___ ] = freqz2(h, ___ , [dx dy])
freqz2( ___ )
```

### Description

`[H, f1, f2] = freqz2(h)` returns `H`, the 64-by-64 frequency response of `h`, and the frequency vectors `f1` (of length 64) and `f2` (of length 64). `h` is a two-dimensional FIR filter, in the form of a computational molecule.

`freqz2` returns `f1` and `f2` as normalized frequencies in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians.

`[H, f1, f2] = freqz2(h, [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`). You can also specify `[n1 n2]` as two separate arguments, `n1`, `n2`.

`[H, f1, f2] = freqz2(h, f1, f2)` returns the frequency response for the FIR filter `h` at frequency values in `f1` and `f2`. These frequency values must be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians. You can also specify `[f1 f2]` as two separate arguments, `f1`, `f2`.

`[ ___ ] = freqz2(h, ___ , [dx dy])` uses `[dx dy]` to override the intersample spacing in `h`. You can also specify a scalar to specify the same spacing in both the `x` and `y` dimensions.

`freqz2( ___ )` produces a mesh plot of the two-dimensional magnitude frequency response when no output arguments are specified.

## 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.

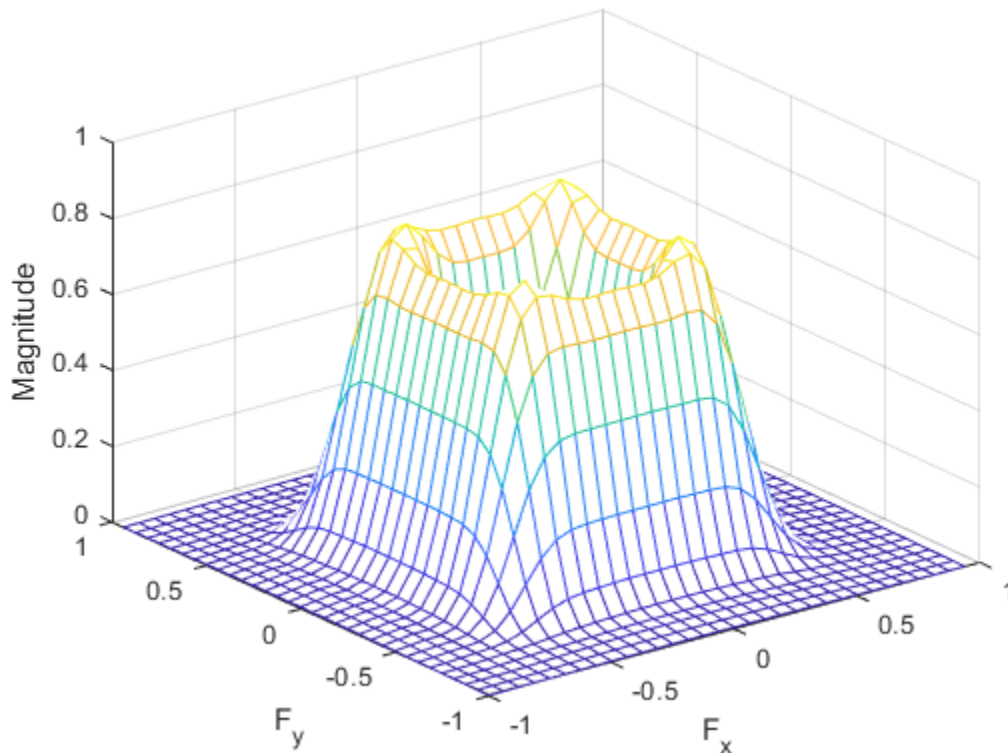
```
w = [0:2:16 16:-2:0]/16;
```

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])
```



## Input Arguments

### **h** — 2-D FIR filter

computational molecule

2-D FIR filter, specified in the form of a computational molecule.

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

### **[n1 n2]** — Number of points in the frequency response

`[64 64]` (default) | two-element vector

Number of points in the frequency response, specified as a two-element vector.

Data Types: double

**f1, f2 — Frequency vectors**

numeric vectors

Frequency vectors, specified as numeric vectors.

Data Types: double

**[dx dy] — Sample spacing**

0.5 (default) | two-element vector or scalar

Sample spacing, specified as a two-element vector of the form [dx dy]. The default spacing is 0.5, which corresponds to a sampling frequency of 2.0. dx determines the spacing for the x dimension and dy determines the spacing for the y dimension. If you specify a scalar, freqz2 uses the value to determine the intersample spacing in both dimensions.

Data Types: double

## Output Arguments

**H — Frequency response**

numeric array

Frequency response, returned as a numeric array.

**f1 — Frequency vector**

vector

Frequency vector, returned as a numeric vector.

Data Types: double

**f2 — Frequency vector**

vector

Frequency vector, returned as a numeric vector.

## **See Also**

freqz

## **Topics**

“Design Linear Filters in the Frequency Domain”

**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`. The filter `h` has a frequency response that passes through points in `Hd`. `fsamp2` designs two-dimensional FIR filters based on a desired two-dimensional frequency response sampled at points on the Cartesian plane.

`h = fsamp2(f1,f2,Hd,[m n])` produces an `m`-by-`n` FIR filter by matching the filter response at the points in the vectors `f1` and `f2`. The frequency vectors `f1` and `f2` are in normalized frequency, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians. The resulting filter fits the desired response as closely as possible in the least squares sense. For best results, there must be at least `m*n` desired frequency points. `fsamp2` issues a warning if you specify fewer than `m*n` points.

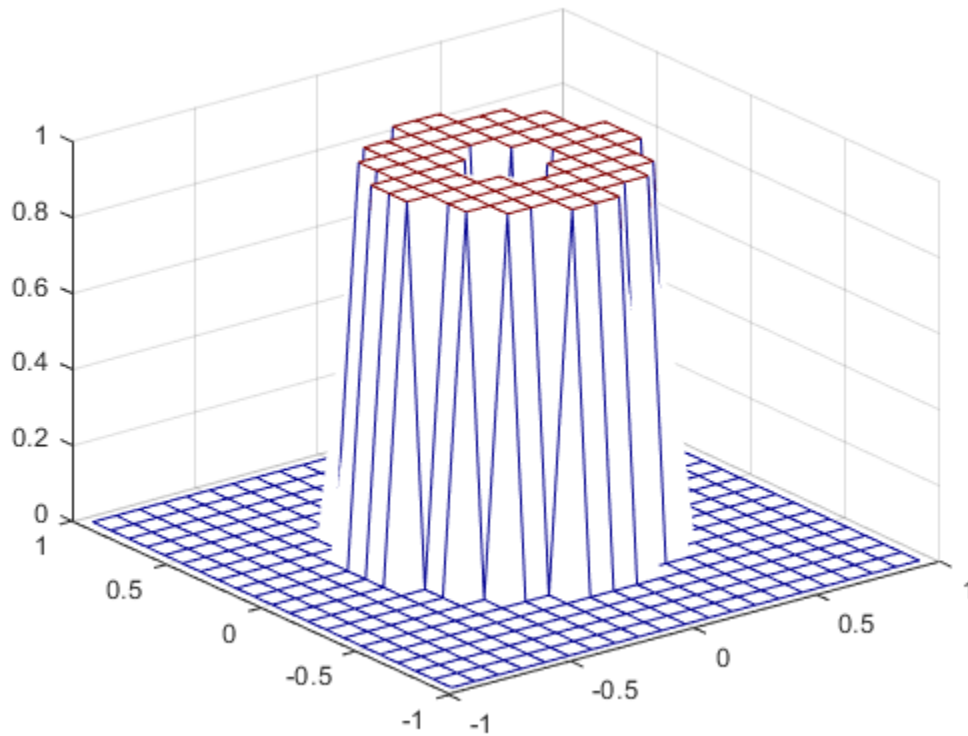
## Examples

### Create 2-D FIR Filter using Frequency Sampling

Use `fsamp2` to design an approximately symmetric, two-dimensional bandpass filter with passband between 0.1 and 0.5 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians).

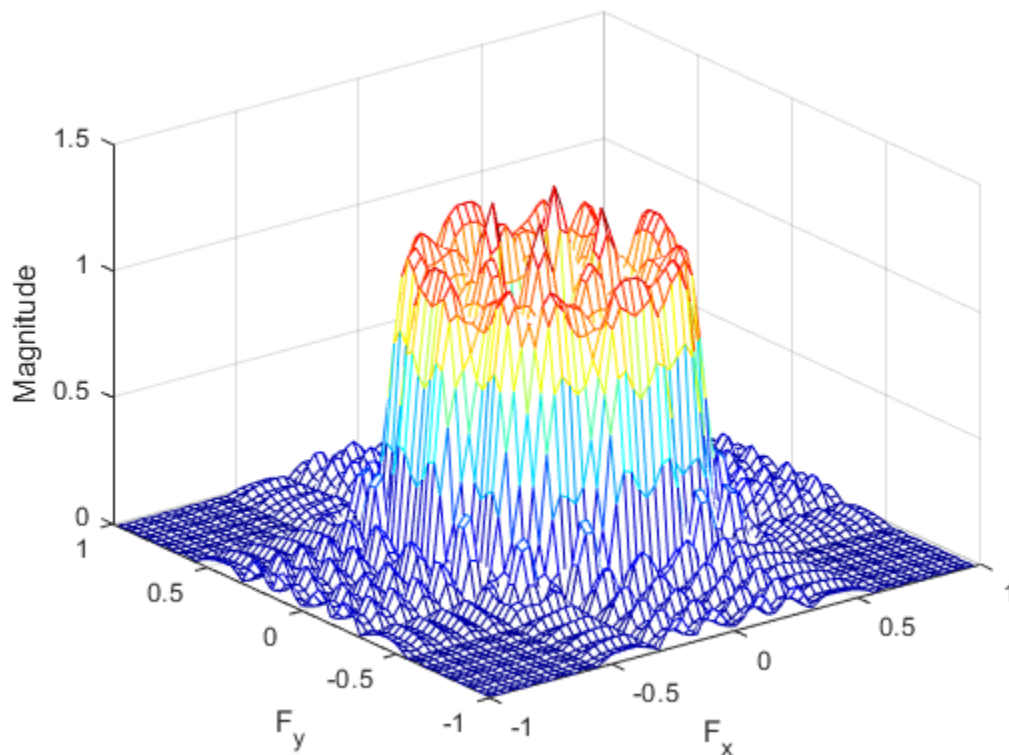
Create a matrix `Hd` that contains the desired bandpass response. Use `freqspace` to create the frequency vectors `f1` and `f2`.

```
[f1,f2] = freqspace(21,'meshgrid');  
Hd = ones(21);  
r = sqrt(f1.^2 + f2.^2);  
Hd((r<0.1)|(r>0.5)) = 0;  
colormap(jet(64))  
mesh(f1,f2,Hd)
```



Design the filter that passes through this response.

```
h = fsamp2(Hd);  
freqz2(h)
```



## Input Arguments

### **Hd — Frequency response**

numeric matrix

Frequency response, specified as a numeric matrix. 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. The value 1.0 corresponds to half the sampling frequency, or  $\pi$  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 best results, use frequency points returned by `freqspace` to create `Hd`.

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

### **f1 — Frequency vector**

numeric vector

Frequency vector, specified as a numeric vector.

Data Types: `double`

### **f2 — Frequency vector**

numeric vector

Frequency vector, specified as a numeric vector.

Data Types: `double`

## **Output Arguments**

### **h — 2-D FIR Filter**

numeric array

2-D FIR filter, returned as a numeric array of class `double`. `fsamp2` returns `h` as a computational molecule, which is the appropriate form to use with `filter2`. If `Hd` is of class `single`, `h` is also of class `single`. If `Hd` is `m-by-n`, then `h` is also `m-by-n`.

## **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`

## Topics

“Design Linear Filters in the Frequency Domain”

**Introduced before R2006a**

## **fspecial**

Create predefined 2-D filter

### **Syntax**

```
h = fspecial(type)
h = fspecial('average',hsize)
h = fspecial('disk',radius)
h = fspecial('gaussian',hsize,sigma)
h = fspecial('laplacian',alpha)
h = fspecial('log',hsize,sigma)
h = fspecial('motion',len,theta)
h = fspecial('prewitt')
h = fspecial('sobel')
```

### **Description**

`h = fspecial(type)` creates a two-dimensional filter `h` of the specified `type`. Some of the filter types have optional additional parameters, shown in the following syntaxes. `fspecial` returns `h` as a correlation kernel, which is the appropriate form to use with `imfilter`.

`h = fspecial('average',hsize)` returns an averaging filter `h` of size `hsize`.

`h = fspecial('disk',radius)` returns a circular averaging filter (pillbox) within the square matrix of size `2*radius+1`.

`h = fspecial('gaussian',hsize,sigma)` returns a rotationally symmetric Gaussian lowpass filter of size `hsize` with standard deviation `sigma`. 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, `alpha` controls the shape of the Laplacian.

`h = fspecial('log',hsize,sigma)` returns a rotationally symmetric Laplacian of Gaussian filter of size `hsize` with standard deviation `sigma`.

`h = fspecial('motion', len, theta)` returns a filter to approximate, once convolved with an image, the linear motion of a camera. `len` specifies the length of the motion and `theta` specifies the angle of motion in degrees in a counter-clockwise direction. The filter becomes a vector for horizontal and vertical motions. The default `len` is 9 and the default `theta` is 0, which corresponds to a horizontal motion of nine pixels.

`h = fspecial('prewitt')` returns a 3-by-3 filter that emphasizes horizontal edges by approximating a vertical gradient. To emphasize vertical edges, transpose the filter `h'`.

```
[ 1  1  1
   0  0  0
  -1 -1 -1 ]
```

`h = fspecial('sobel')` returns a 3-by-3 filter that emphasizes horizontal edges using the smoothing effect by approximating a vertical gradient. To emphasize vertical edges, transpose the filter `h'`.

```
[ 1  2  1
   0  0  0
  -1 -2 -1 ]
```

## Examples

### Create Various Filters and Filter an Image

Read image and display it.

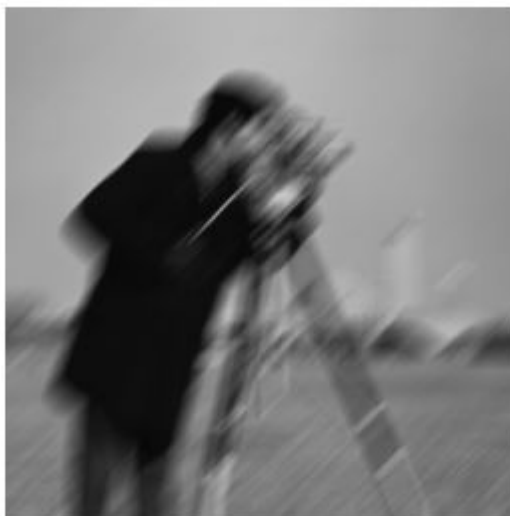
```
I = imread('cameraman.tif');
imshow(I);
```



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);
```



## Input Arguments

**type** — Type of filter

'average' | 'disk' | 'gaussian' | 'laplacian' | 'log' | 'motion' | 'prewitt' | 'sobel'

Type of filter, specified as one of the following values:

Value	Description
'average'	Averaging filter
'disk'	Circular averaging filter (pillbox)
'gaussian'	Gaussian lowpass filter. Not recommended. Use <code>imgaussfilt</code> or <code>imgaussfilt3</code> instead.

Value	Description
'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

Data Types: char | string

### **hsize — Size of the filter**

positive integer | 2-element vector of positive integers

Size of the filter, specified as a positive integer or 2-element vector of positive integers. Use a vector to specify the number of rows and columns in `h`. If you specify a scalar, then `h` is a square matrix.

When used with the 'average' filter type, the default filter size is [3 3]. When used with the Laplacian of Gaussian ('log') filter type, the default filter size is [5 5].

Data Types: double

### **radius — Radius of a disk-shaped filter**

5 (default) | positive number

Radius of a disk-shaped filter, specified as a positive number.

Data Types: double

### **sigma — Standard deviation**

0.5 (default) | positive number

Standard deviation, specified as a positive number.

Data Types: double

### **alpha — Shape of the Laplacian**

0.2 (default) | scalar in the range [0 1]

Shape of the Laplacian, specified as a scalar in the range [0 1].

Data Types: double

**len — Linear motion of camera**

9 (default) | numeric scalar

Linear motion of camera, specified as a numeric scalar, measured in pixels.

Data Types: double

**theta — Angle of camera motion**

0 (default) | numeric scalar

Angle of camera motion, specified as a numeric scalar, measured in degrees, in a counter-clockwise direction.

Data Types: double

## Output Arguments

**h — Correlation kernel**

matrix

Correlation kernel, returned as a matrix.

Data Types: double

## Algorithms

Averaging filters:

`ones(n(1),n(2))/(n(1)*n(2))`

Gaussian filters:

$$h_g(n_1, n_2) = e^{-\frac{(n_1^2 + n_2^2)}{2\sigma^2}}$$

$$h(n_1, n_2) = \frac{h_g(n_1, n_2)}{\sum_{n_1} \sum_{n_2} h_g}$$

Laplacian filters:

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

$$\nabla^2 = \frac{4}{(\alpha + 1)} \begin{bmatrix} \frac{\alpha}{4} & \frac{1-\alpha}{4} & \frac{\alpha}{4} \\ \frac{1-\alpha}{4} & -1 & \frac{1-\alpha}{4} \\ \frac{\alpha}{4} & \frac{1-\alpha}{4} & \frac{\alpha}{4} \end{bmatrix}$$

Laplacian of Gaussian (LoG) filters:

$$h_g(n_1, n_2) = e^{-\frac{(n_1^2 + n_2^2)}{2\sigma^2}}$$

$$h(n_1, n_2) = \frac{(n_1^2 + n_2^2 - 2\sigma^2)h_g(n_1, n_2)}{\sigma^4 \sum_{n_1} \sum_{n_2} h_g}$$

Note that `fspecial` shifts the equation to ensure that the sum of all elements of the kernel is zero (similar to the Laplace kernel) so that the convolution result of homogeneous regions is always zero.

Motion filters:

- 1 Construct an ideal line segment with the length and angle specified by the arguments `len` and `theta`, 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(:)))`

## 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` | `del2` | `edge` | `filter2` | `fsamp2` | `fspecial3` | `fwind1` | `fwind2` | `imfilter` | `imsharpen`

### Topics

“Filter Images Using Predefined Filters”

“What Is Image Filtering in the Spatial Domain?”

**Introduced before R2006a**

# fspecial3

Create predefined 3-D filter

## Syntax

```
h = fspecial3(type)
h = fspecial3('average',hsize)
h = fspecial3('ellipsoid',semiaxes)
h = fspecial3('gaussian',hsize,sigma)
h = fspecial3('laplacian',gamma1,gamma2)
h = fspecial3('log',hsize,sigma)
h = fspecial3('prewitt',direction)
h = fspecial3('sobel',direction)
```

## Description

`h = fspecial3(type)` creates a three-dimensional filter `h` of the specified `type`. Some of the filter types have optional additional parameters, shown in the following syntaxes. `fspecial3` returns `h` as a correlation kernel, which is the appropriate form to use with `imfilter`.

`h = fspecial3('average',hsize)` returns an averaging filter `h` of size `hsize`. Not recommended. Use `imboxfilt3` instead.

`h = fspecial3('ellipsoid',semiaxes)` returns an ellipsoidal averaging filter with the length of the principal semiaxes specified by `semiaxes`. The filter `h` is returned in an array of size `2*ceil(semiaxes)+1`.

`h = fspecial3('gaussian',hsize,sigma)` returns a Gaussian lowpass filter of size `hsize` with standard deviation `sigma`. Not recommended. Use `imgaussfilt3` instead.

`h = fspecial3('laplacian',gamma1,gamma2)` returns a 3-by-3-by-3 filter approximating the shape of the three-dimensional Laplacian operator. `gamma1` and `gamma2` control the shape of the Laplacian `[1][2]`.

`h = fspecial3('log',hsize,sigma)` returns a Laplacian of Gaussian filter of size `hsize` with standard deviation `sigma`.

`h = fspecial3('prewitt',direction)` returns a 3-by-3-by-3 filter that emphasizes gradients in the specified direction.

`h = fspecial3('sobel',direction)` returns a 3-by-3-by-3 filter that emphasizes gradients in the specified direction and smooths the other directions [3].

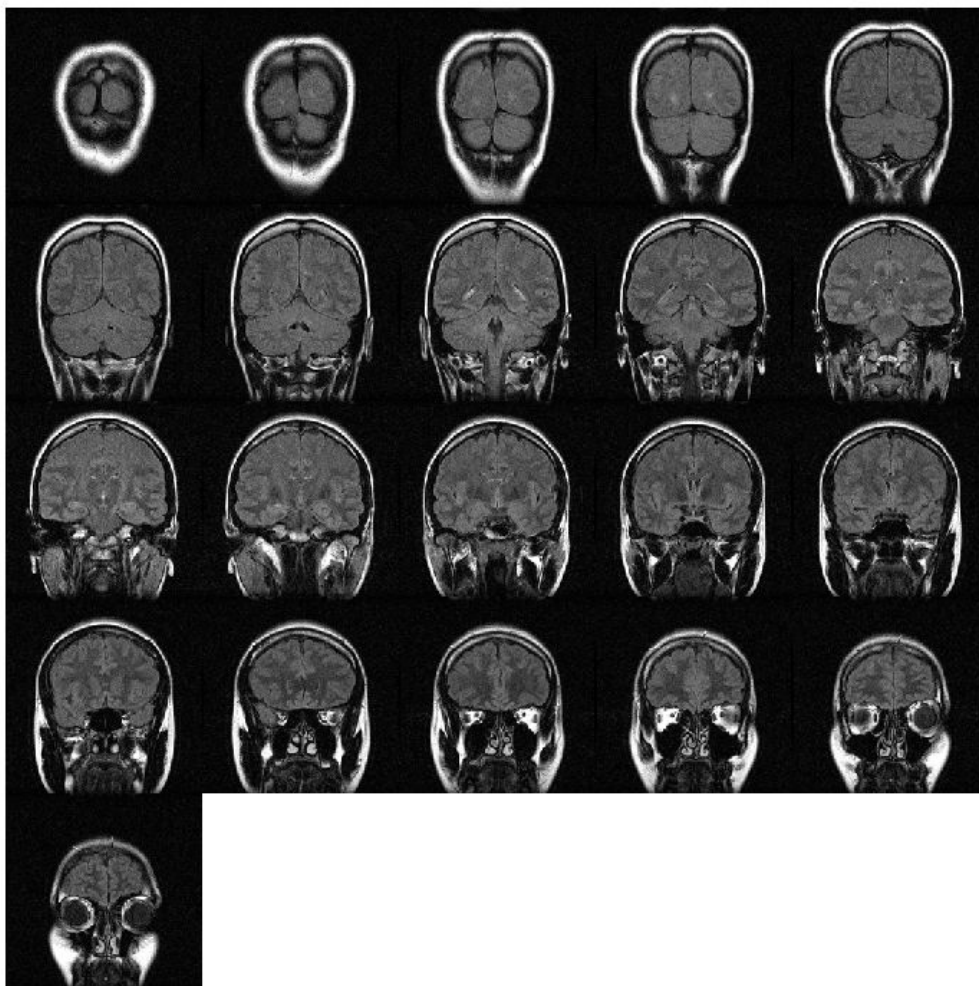
## Examples

### Smooth Volume Using 3-D Ellipsoidal Filter

Load a 3-D grayscale MRI volume. Display the planes of the volume.

```
load mrystack;  
montage(mrystack, 'BackgroundColor', 'w')
```





Create a 3-D ellipsoidal filter. Specify a semiaxis length of 7 pixels in the y (rows) and x (columns) directions, and a semiaxis length of 3 pixels in the z (planes) direction.

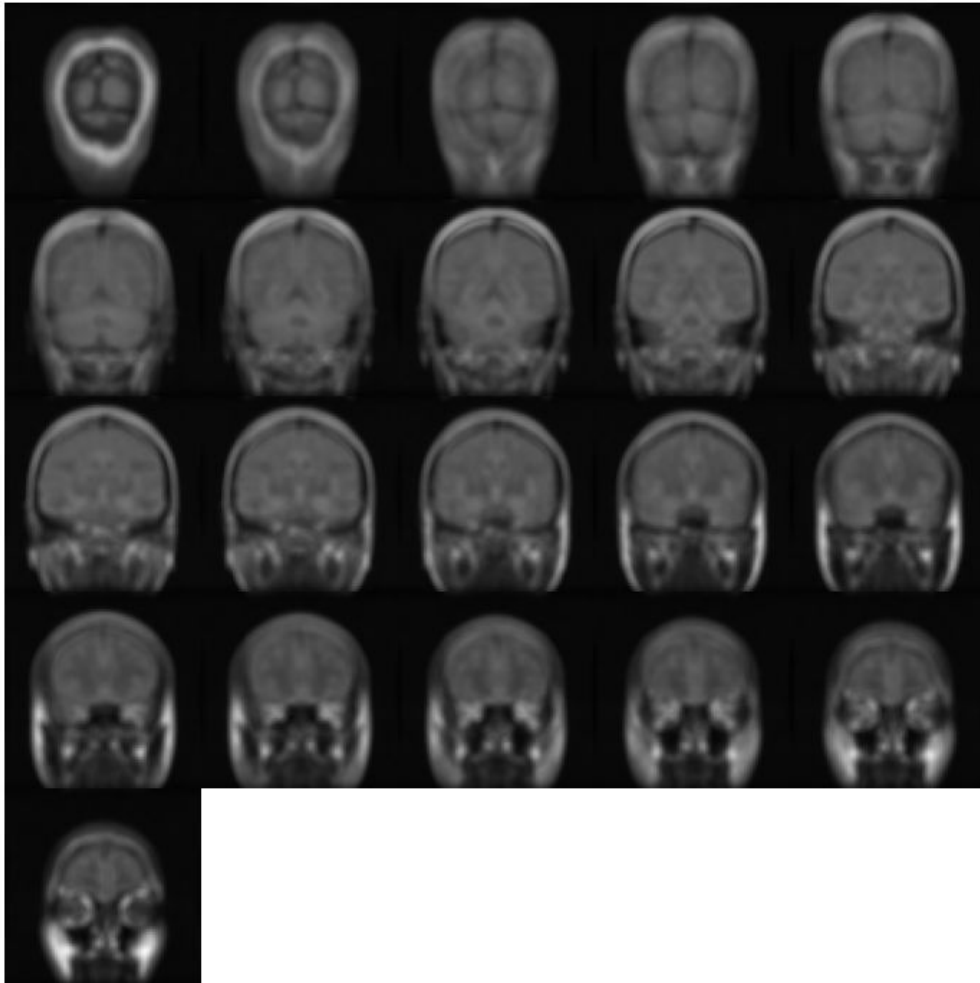
```
H = fspecial3('ellipsoid',[7 7 3]);
```

Smooth the volume with the filter.

```
volSmooth = imfilter(mristack,H,'replicate');
```

Display the planes of the smoothed volume.

```
montage(volSmooth,'BackgroundColor','w')
```



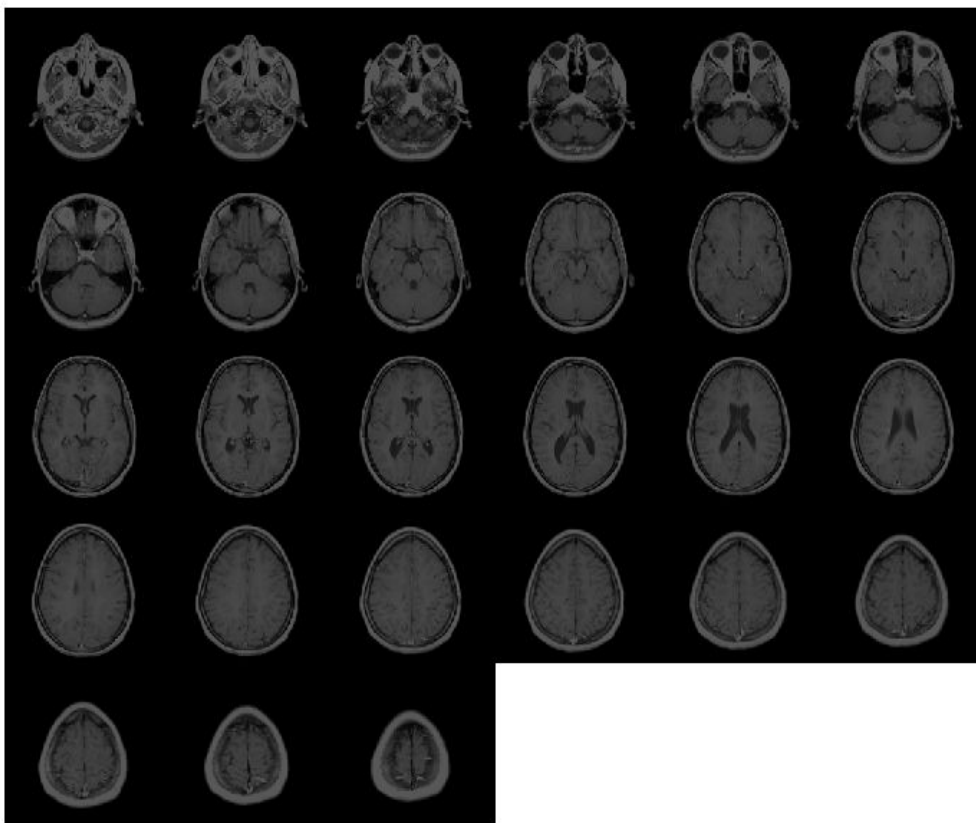
## Detect Horizontal Edges Using 3-D Sobel Filter

Load an MRI volume. This volume is stored as a 4-D array with a singleton dimension. Create a 3-D grayscale volume by using the `squeeze` function to remove the singleton dimension.

```
load mri;  
V = squeeze(D);
```

Display the planes of the volume.

```
montage(D, 'BackgroundColor', 'w')
```



Create a 3-D Sobel filter that detects horizontal edges in the volume. Horizontal edges appear where there is a large gradient magnitude in the y direction, so specify the direction of the Sobel filter as 'Y'. The Sobel filter smooths the gradient in the x and z directions.

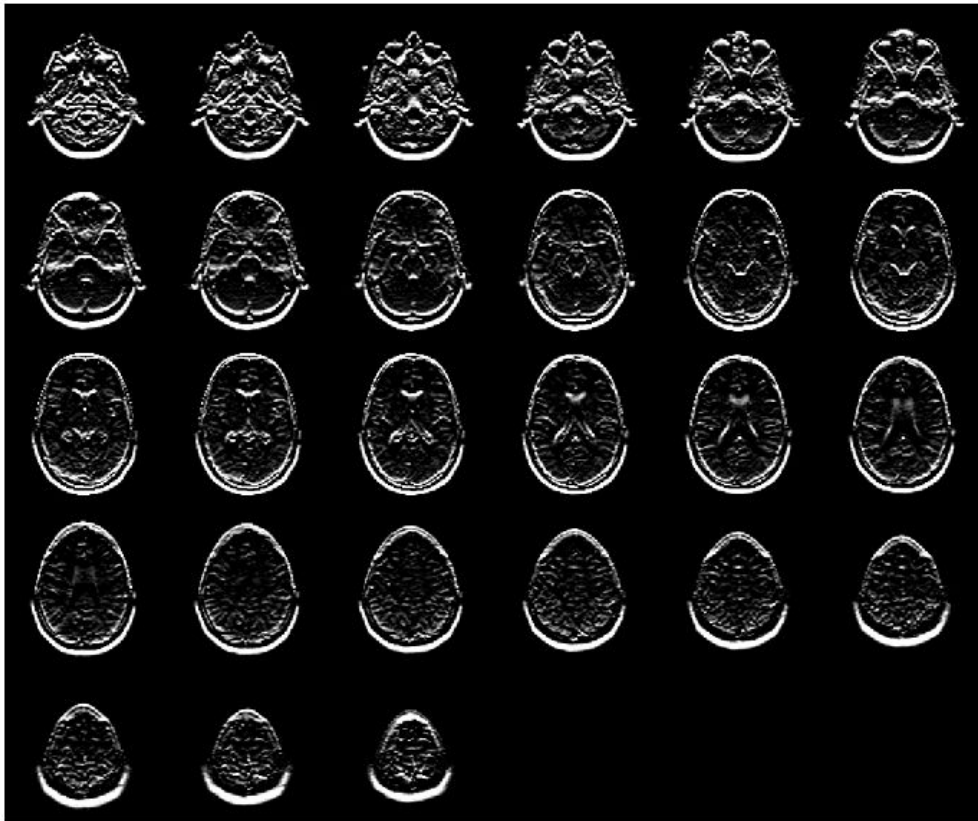
```
H = fspecial3('sobel','Y');
```

Filter the volume with the 3-D Sobel filter.

```
edgesHor = imfilter(V,H,'replicate');
```

Display the planes of the filtered volume.

```
montage(edgesHor)
```



## Input Arguments

### type — Type of filter

'average' | 'ellipsoid' | 'gaussian' | 'laplacian' | 'log' | 'prewitt' | 'sobel'

Type of filter, specified as one of the following values:

Value	Description
'average'	Averaging filter. Not recommended. Use <code>imboxfilt3</code> instead.
'ellipsoid'	Ellipsoidal averaging filter
'gaussian'	Gaussian lowpass filter. Not recommended. Use <code>imgaussfilt3</code> instead.
'laplacian'	Approximates the three-dimensional Laplacian operator
'log'	Laplacian of Gaussian filter
'prewitt'	Prewitt edge-emphasizing filter
'sobel'	Sobel edge-emphasizing filter

Data Types: `char` | `string`

### hsize — Size of the filter

[5 5 5] (default) | positive integer | 3-element vector of positive integers

Size of the filter, specified as a positive integer or 3-element vector of positive integers. Use a vector to specify the number of rows, columns, and planes in `h`. Use a scalar to specify the side length of a cube.

For the 'gaussian' and 'log' filter types, if you specify `hsize` as [], then `fspecial3` creates a filter with a default size of  $2 \times \text{ceil}(2 \times \text{sigma}) + 1$ .

Data Types: `double`

### semiaxes — Semiaxes length of ellipsoidal filter

5 (default) | positive number | 3-element vector of positive numbers

Semiaxes length of an ellipsoidal filter, specified as a positive number or 3-element vector of positive numbers. Use a vector to specify the length of the three principal semiaxes in

rows, columns, and planes. These values correspond to length in the Cartesian  $y$ ,  $x$ , and  $z$  directions, respectively. Use a scalar to specify the radius of a sphere.

Data Types: double

### **sigma — Standard deviation of Gaussian filter**

1 (default) | positive number | 3-element vector of positive numbers

Standard deviation of Gaussian filter, specified as a positive number or 3-element vector of positive numbers. If you specify a scalar, then `fspecial3` creates a cubic Gaussian kernel.

Data Types: double

### **gamma1, gamma2 — Shape of the Laplacian**

0 (default) | scalar in the range [0 1]

Shape of the Laplacian, specified as a scalar in the range [0 1]. The sum of `gamma1` and `gamma2` must not exceed 1.

Data Types: double

### **direction — Direction of gradients**

'X' (default) | 'Y' | 'Z'

Direction of gradients for Prewitt and Sobel filtering, specified as 'X', 'Y', or 'Z'.

Data Types: char | string

## Output Arguments

### **h — Correlation kernel**

numeric array

Correlation kernel, returned as a numeric array.

Data Types: double

## References

- [1] Lindeberg, T., *Scale-Space Theory in Computer Vision*. Boston, MA: Kluwer Academic Publishers, 1994.



[2] *Geometry-Driven Diffusion in Computer Vision*. Edited by B. M. ter Haar Romeny. Boston, MA: Kluwer Academic Publishers, 1994.

[3] Engel, K., M. Hadwiger, J. M. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-Time Volume Graphics*. Wellesley, MA: A K Peters, Ltd., 2006, pp. 112-114.

## See Also

`edge3` | `fspecial` | `imboxfilt3` | `imfilter` | `imgaussfilt3`

## Topics

“What Is Image Filtering in the Spatial Domain?”

**Introduced in R2018b**

## 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`. `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.

`h = ftrans2(b)` uses the McClellan transform matrix `t`.

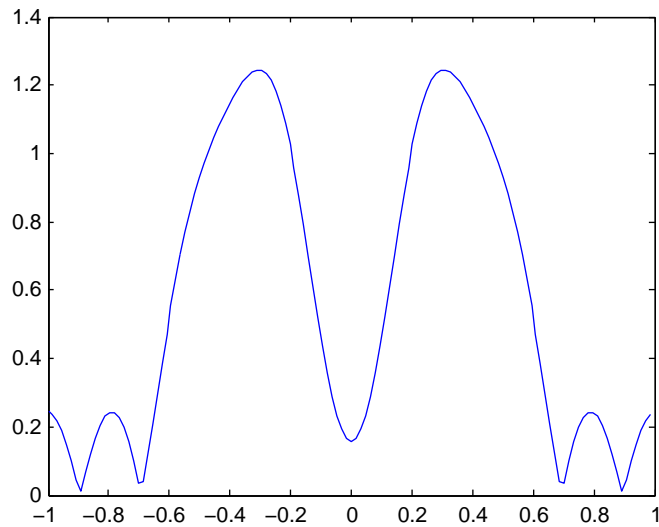
```
t = [1 2 1; 2 -4 2; 1 2 1]/8;
```

### Examples

#### Design Circularly Symmetric 2-D Bandpass Filter

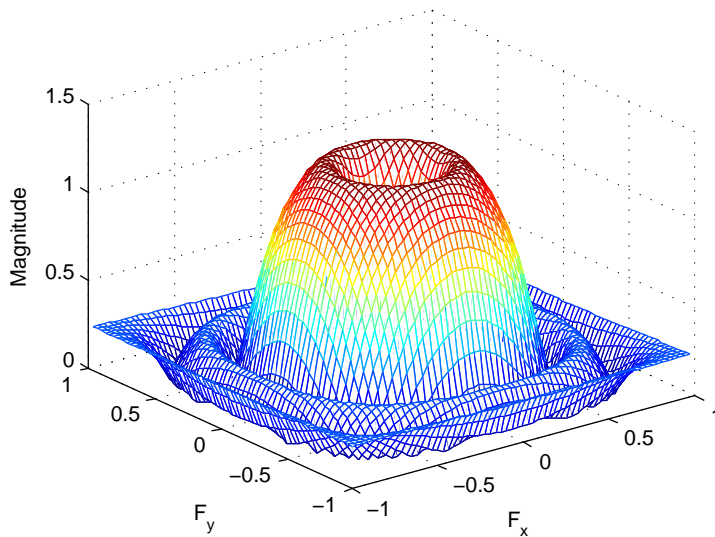
Use `ftrans2` to design an approximately circularly symmetric two-dimensional bandpass filter with passband between 0.1 and 0.6 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians). Since `ftrans2` transforms a one-dimensional FIR filter to create a two-dimensional 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)))
```



Use `ftans2` with the default McClellan transformation to create the desired approximately circularly symmetric filter.

```
h = ftans2(b);  
freqz(h)
```



## Input Arguments

### **b** — One-dimensional FIR filter

numeric matrix

1-D FIR filter, specified as a numeric matrix. **b** must be a 1-D Type I (even symmetric, odd-length) filter such as can be returned by `fir1`, `fir2`, or `firpm` in the Signal Processing Toolbox software,

Data Types: double

### **t** — Transform matrix

McClellan transform matrix (default) | numeric matrix

The transform matrix, specified as a numeric matrix. **t** contains coefficients that define the frequency transformation to use.

Data Types: double

## Output Arguments

### **h** — 2-D FIR filter

numeric matrix

2-D FIR filter, returned as a numeric matrix. `ftrans2` returns `h` as a computational molecule, which is the appropriate form to use with `filter2`. If `t` is `m`-by-`n` and `b` has length `Q`, then `h` is size  $((m-1)*(Q-1)/2+1)$ -by- $((n-1)*(Q-1)/2+1)$ .

## Algorithms

The transformation below defines the frequency response of the two-dimensional filter returned by `ftrans2`.

$$H(\omega_1, \omega_2) = B(\omega) \Big|_{\cos \omega = T(\omega_1, \omega_2)},$$

where  $B(\omega)$  is the Fourier transform of the one-dimensional filter `b`:

$$B(\omega) = \sum_{n=-N}^N b(n)e^{-j\omega n}$$

and  $T(\omega_1, \omega_2)$  is the Fourier transform of the transformation matrix `t`:

$$T(\omega_1, \omega_2) = \sum_{n_2} \sum_{n_1} t(n_1, n_2) e^{-j\omega_1 n_1} e^{-j\omega_2 n_2}.$$

The returned filter `h` is the inverse Fourier transform of  $H(\omega_1, \omega_2)$ :

$$h(n_1, n_2) = \frac{1}{(2\pi)^2} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} H(\omega_1, \omega_2) e^{j\omega_1 n_1} e^{j\omega_2 n_2} d\omega_1 d\omega_2.$$

## 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

## **Topics**

“Design Linear Filters in the Frequency Domain”

**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

`h = fwind1(Hd,win)` uses a 1-D window specification to design a two-dimensional FIR filter `h` based on the desired 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.

`fwind1` works with 1-D windows only; use `fwind2` to work with two-dimensional windows.

`h = fwind1(Hd,win1,win2)` uses the two 1-D windows, `win1` and `win2`, to create a separable 2-D window. If `length(win1)` is `n` and `length(win2)` is `m`, then `h` is `m-by-n`. The length of the windows controls the size of the resulting filter.

`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.

## 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);
```

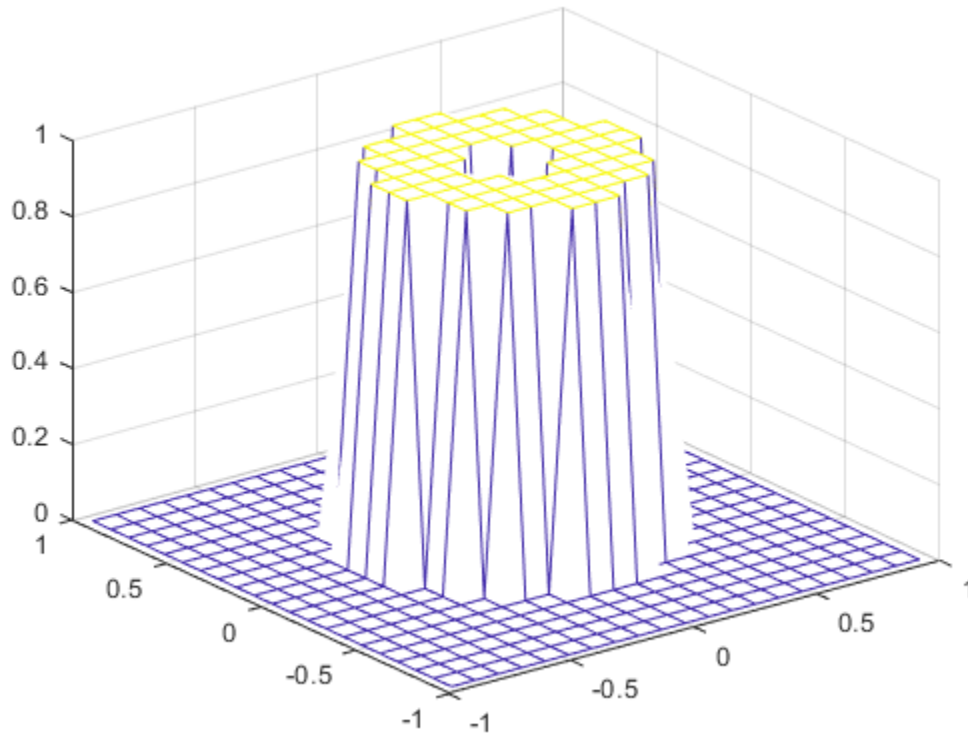
Create a matrix `Hd` that contains the desired bandpass response. In this example, the desired passband is between 0.1 and 0.5 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians).

```
Hd = ones(21);  
Hd((r<0.1)|(r>0.5)) = 0;
```

Display the ideal bandpass response.

```
colormap(parula(64))  
mesh(f1,f2,Hd)
```



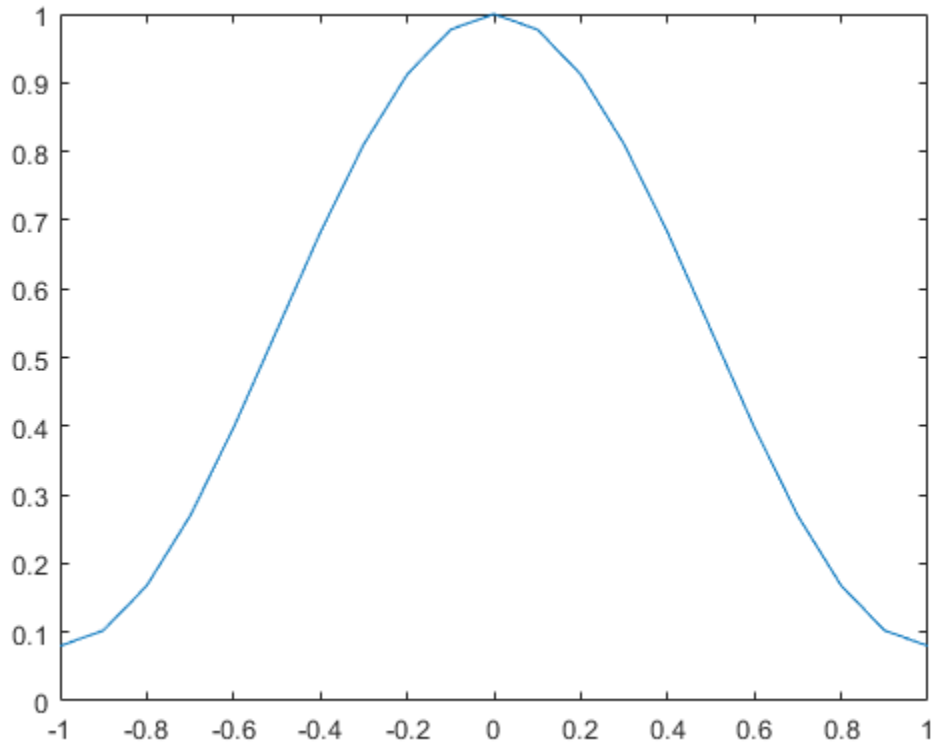


Design the 1-D window. This example uses a Hamming window of length 21.

```
win = 0.54 - 0.46*cos(2*pi*(0:20)/20);
```

Plot the 1-D window.

```
figure  
plot(linspace(-1,1,21),win);
```

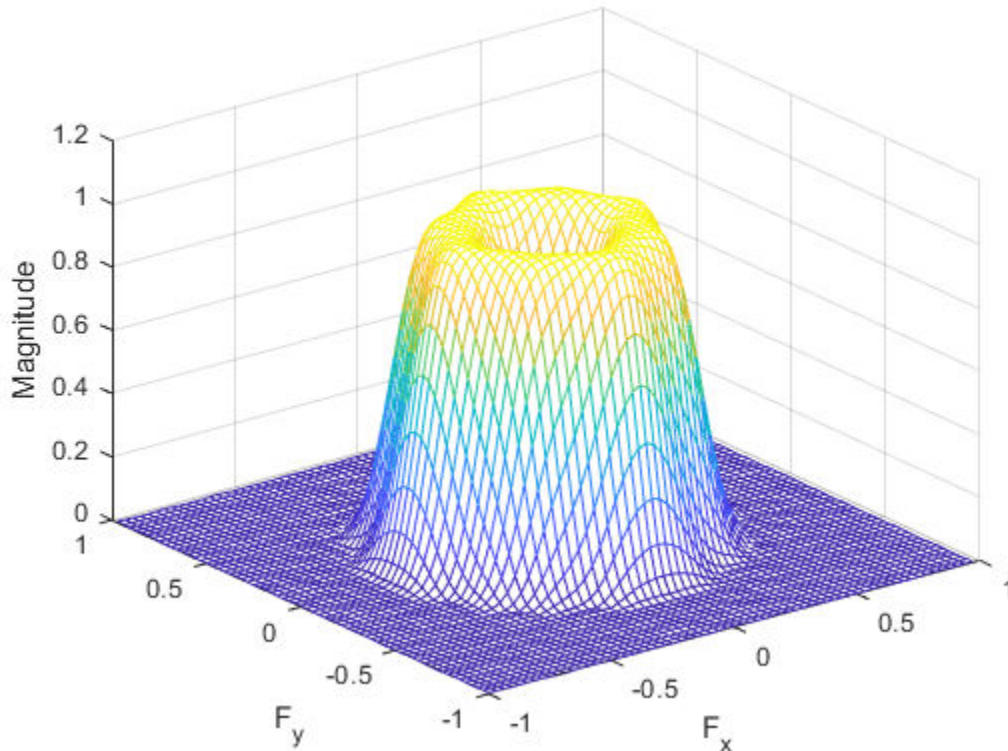


Using the 1-D window, design the filter that best produces this frequency response

```
h = fwind1(Hd,win);
```

Display the actual frequency response of this filter.

```
freqz2(h)
```



## Input Arguments

### **Hd** — Desired frequency response

numeric matrix

Desired frequency response, specified as a numeric matrix. Hd is sampled at equally spaced points between -1.0 and 1.0 (in normalized frequency, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians) along the x and y frequency axes. For accurate results, use frequency points returned by `freqspace` to create Hd.

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

## **win** — 1-D window

numeric matrix

1-D window, specified as a numeric matrix. 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`. The length of the window controls the size of the resulting filter.

Data Types: `single` | `double`

## **win1** — 1-D window

numeric matrix

1-D window, specified as a numeric matrix.

Data Types: `single` | `double`

## **win2** — 1-D window

numeric matrix

1-D window, specified as a numeric matrix.

Data Types: `single` | `double`

## **f1** — Desired frequency along the x-axis

vector

Desired frequency along the x-axis. The frequency vector should be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians.

Data Types: `single` | `double`

## **f2** — Desired frequency along the y-axis

vector

Desired frequency along the y-axis. The frequency vector should be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians.

Data Types: `single` | `double`

## Output Arguments

### **h** — 2-D FIR filter

2-D FIR filter, returned as a numeric matrix of class `double`, when the input `Hd` is of class `double` or any integer class. If `Hd` is of class `single`, the output matrix is of class `single`.

## 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 | freqspace | fsamp2 | ftrans2 | fwind2

## **Topics**

“Design Linear Filters in the Frequency Domain”

**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

`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`.

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.

For accurate results, use frequency points returned by `freqspace` to create `Hd`.

`h = fwind2(f1,f2,Hd,win)` lets you specify the desired frequency response `Hd` at arbitrary frequencies (`f1` and `f2`) along the *x*- and *y*-axes. The frequency vectors `f1` and `f2` should be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians. `h` is the same size as `win`.

## 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);
```

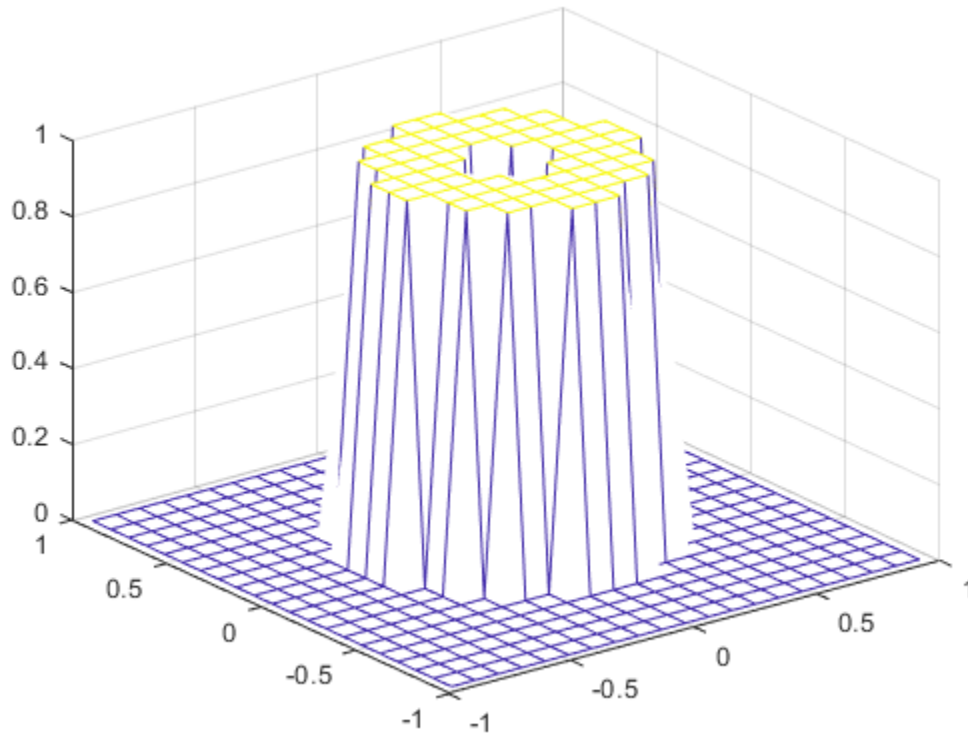
Create a matrix `Hd` that contains the desired bandpass response. In this example, the desired passband is between 0.1 and 0.5 (normalized frequency, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians).

```
Hd = ones(21);  
Hd((r<0.1)|(r>0.5)) = 0;
```

Display the ideal bandpass response.

```
colormap(parula(64))  
mesh(f1,f2,Hd)
```



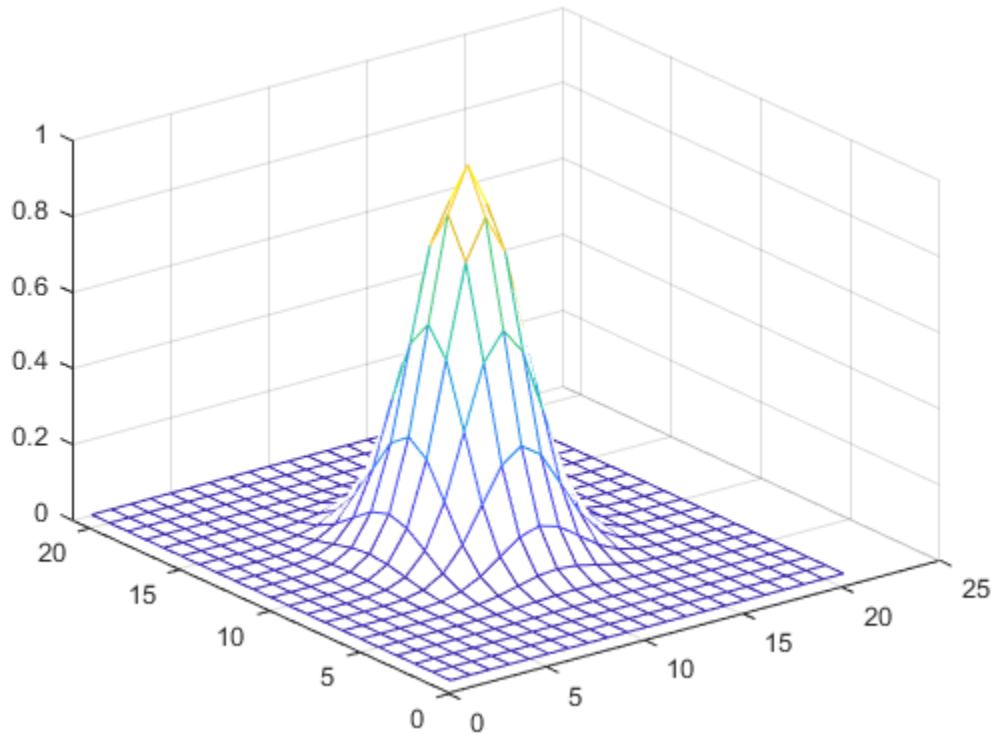


Create a 2-D Gaussian window using `fspecial`. Normalize the window.

```
win = fspecial('gaussian',21,2);  
win = win ./ max(win(:));
```

Display the window.

```
mesh(win)
```

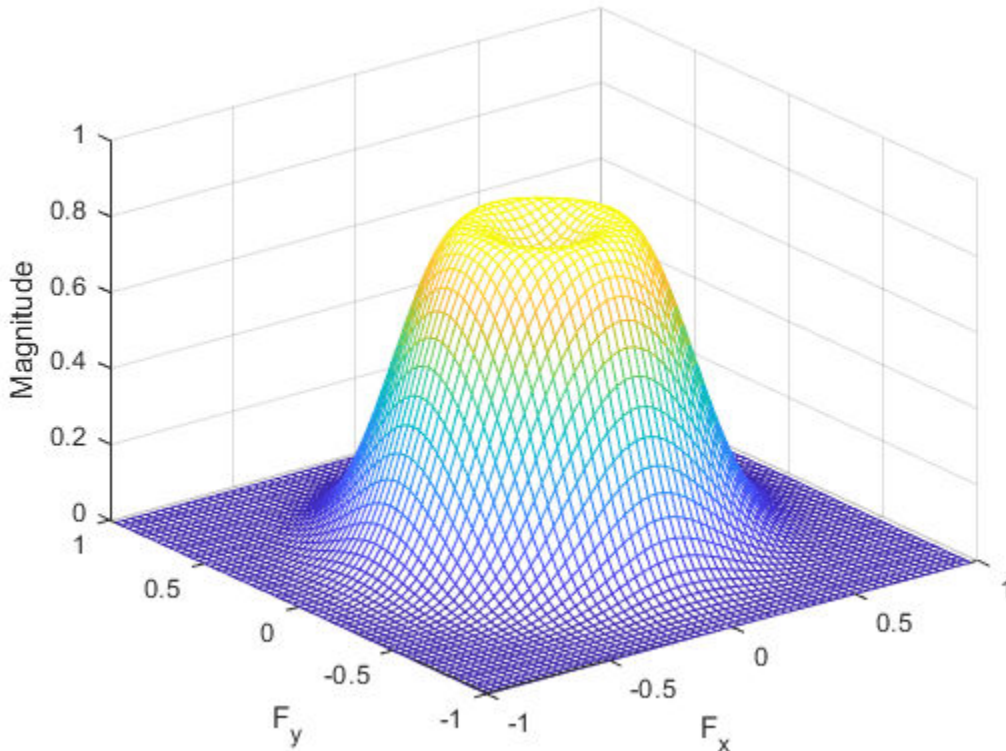


Using the 2-D window, design the filter that best produces the desired frequency response

```
h = fwind2(Hd,win);
```

Display the actual frequency response of this filter.

```
freqz2(h)
```



## Input Arguments

### **Hd** — Desired frequency response

numeric matrix

Desired frequency response, at equally spaced points in the Cartesian plane, specified as a numeric matrix. 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`.

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

**win — 2-D window**

numeric matrix

2-D window, specified as a numeric matrix.

Data Types: `single` | `double`**f1 — Desired frequency along the x-axis**

vector

Desired frequency along the x-axis. The frequency vector should be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians.Data Types: `single` | `double`**f2 — Desired frequency along the y-axis**

vector

Desired frequency along the y-axis. The frequency vector should be in the range -1.0 to 1.0, where 1.0 corresponds to half the sampling frequency, or  $\pi$  radians.Data Types: `single` | `double`

## Output Arguments

**h — 2-D FIR filter**

numeric matrix

2-D FIR filter, returned as a numeric matrix.

## 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 | freqspace | fsamp2 | 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.

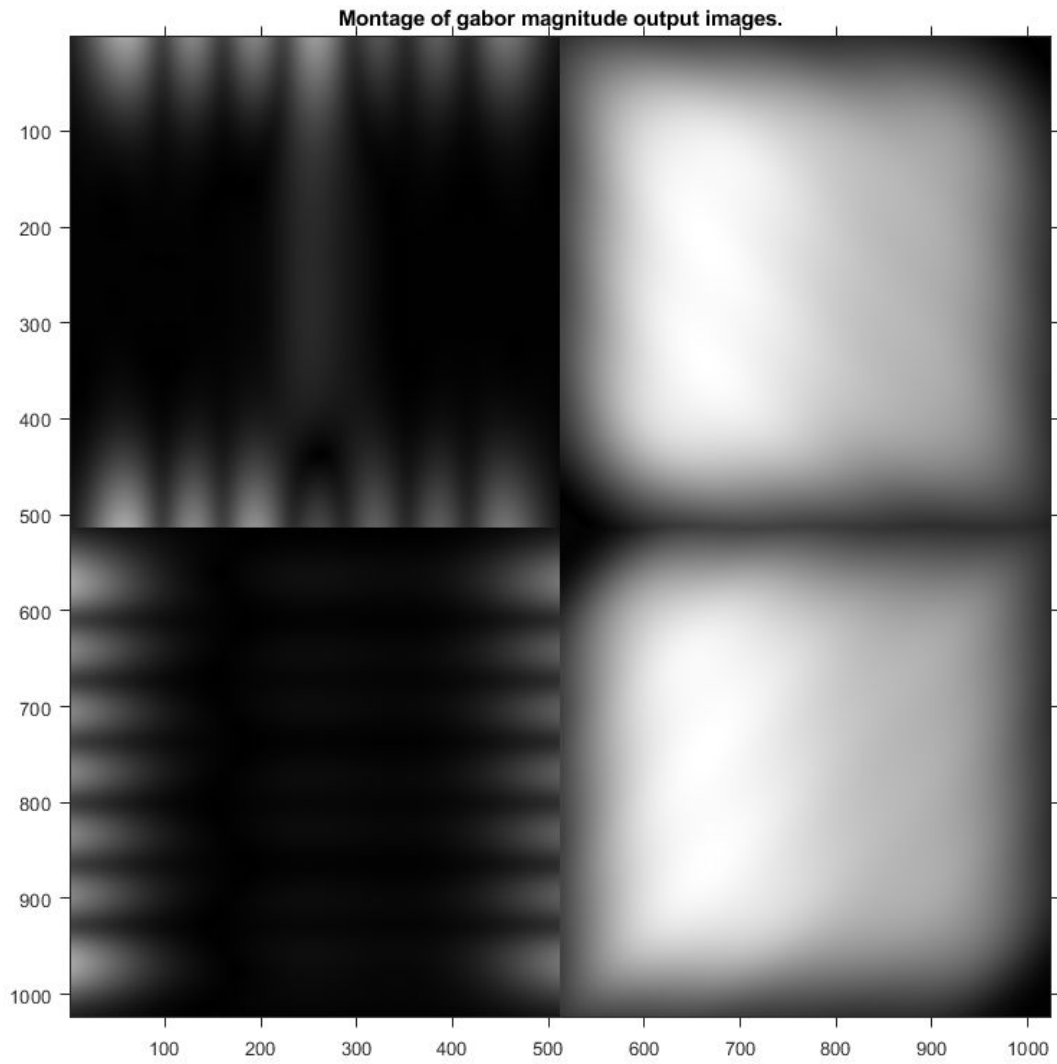
```
wavelength = 20;  
orientation = [0 45 90 135];  
g = gabor(wavelength,orientation);
```

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.');
```





## Construct Gabor Filter Array and Visualize Wavelength and Orientation

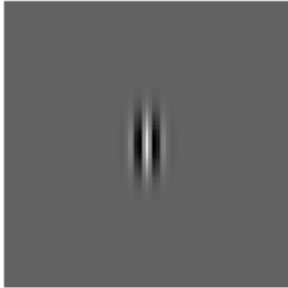
Create array of Gabor filters.

```
g = gabor([5 10],[0 90]);
```

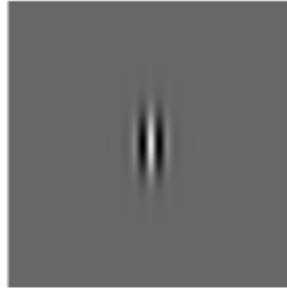
Visualize the real part of the spatial convolution kernel of each Gabor filter in the array.

```
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)],  $\lambda = 10, \theta = 90$**



## 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.

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.

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 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,  $1/\lambda$ . 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** — 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].

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**

`imgaborfilt`

## **Topics**

“Texture Segmentation Using Gabor Filters”

**Introduced in R2015b**

# geometricTransform2d

2-D geometric transformation object

## Description

A `geometricTransform2d` object defines a custom 2-D geometric transformation using point-wise mapping functions.

## Creation

## Syntax

```
tform = geometricTransform2d(inverseFcn)  
tform = geometricTransform2d(inverseFcn, forwardFcn)
```

## Description

`tform = geometricTransform2d(inverseFcn)` creates a `geometricTransform2d` object and sets the inverse mapping `InverseFcn` property.

`tform = geometricTransform2d(inverseFcn, forwardFcn)` also sets the forward mapping property, `ForwardFcn`.

## Properties

### **InverseFcn** — Inverse mapping function

function handle

Inverse mapping function, specified as a function handle. The function should accept and return coordinates as a  $n$ -by-2 numeric matrix representing the packed  $(x,y)$  coordinates of  $n$  points.

For more information about function handles, see “Create Function Handle” (MATLAB).

```
Example: ifcn = @(xy) [xy(:,1).^2, sqrt(xy(:,2))];
```

### **ForwardFcn — Forward mapping function**

function handle

Forward mapping function, specified as a function handle. The function should accept and return coordinates as a  $n$ -by-2 numeric matrix representing the packed (x,y) coordinates of  $n$  points.

For more information about function handles, see “Create Function Handle” (MATLAB).

```
Example: ffcn = @(xy) [sqrt(xy(:,1)), (xy(:,2).^2)];
```

## **Object Functions**

transformPointsForward Apply forward geometric transformation

transformPointsInverse Apply inverse geometric transformation

## **Examples**

### **Transform Packed Coordinates Using Custom 2-D Transformation**

Specify the packed (x,y) coordinates of five input points. The packed coordinates are stored in a 5-by-2 matrix, where the x-coordinate of each point is in the first column, and the y-coordinate of each point is in the second column.

```
XY = [10 15;11 32;15 34;2 7;2 10];
```

Define the inverse mapping function. The function accepts and returns points in packed (x,y) format.

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

```
inversefn = function handle with value:  
    @(c)[c(:,1)+c(:,2),c(:,1)-c(:,2)]
```

Create a 2-D geometric transform object, `tform`, that stores the inverse mapping function.

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

```
tform =
  geometricTransform2d with properties:
      InverseFcn: @(c)[c(:,1)+c(:,2),c(:,1)-c(:,2)]
      ForwardFcn: []
      Dimensionality: 2
```

Apply the inverse geometric transform to the input points.

```
UV = transformPointsInverse(tform,XY)
```

```
UV = 5×2
```

```
    25    -5
    43   -21
    49   -19
     9    -5
    12    -8
```

### Transform Coordinate Arrays Using Custom 2-D Transformation

Specify the x- and y-coordinates vectors of five points to transform.

```
x = [10 11 15 2 2];
y = [15 32 34 7 10];
```

Define the inverse and forward mapping functions. Both functions accept and return points in packed (x,y) format.

```
inversefn = @(c) [c(:,1).^2,sqrt(c(:,2))];
forwardfn = @(c) [sqrt(c(:,1)),c(:,2).^2];
```

Create a 2-D geometric transform object, `tform`, that stores the inverse mapping function and the optional forward mapping function.

```
tform = geometricTransform2d(inversefn,forwardfn)
```

```
tform =
  geometricTransform2d with properties:
```

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

Apply the inverse geometric transform to the input points.

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

```
u = 1×5
```

```
100 121 225 4 4
```

```
v = 1×5
```

```
3.8730 5.6569 5.8310 2.6458 3.1623
```

Apply the forward geometric transform to the transformed points u and v.

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

```
x = 1×5
```

```
10 11 15 2 2
```

```
y = 1×5
```

```
15.0000 32.0000 34.0000 7.0000 10.0000
```

## Transform Grayscale Image Using Custom 2-D Transformation

Define an inverse mapping function that applies anisotropic scaling. The function must accept and return packed (x,y) coordinates, where the x-coordinate of each point is in the first column, and the y-coordinate of each point is in the second column.

```
xscale = 0.3;  
yscale = 0.5;  
inversefn = @(xy) [xscale*xy(:,1), yscale*xy(:,2)];
```



Create a 2-D geometric transform object, `tform`, that stores the inverse mapping function.

```
tform = geometricTransform2d(inversefn)

tform =
  geometricTransform2d with properties:

    InverseFcn: @(xy)[xscale*xy(:,1),yscale*xy(:,2)]
    ForwardFcn: []
    Dimensionality: 2
```

Read an image to be transformed.

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



Use `imwarp` to apply the inverse geometric transform to the input image. The image is enlarged vertically by a factor of 2 (the inverse of `yscale`) and horizontally by a factor of  $10/3$  (the inverse of `xscale`).

```
Itransformed = imwarp(I,tform);  
imshow(Itransformed)
```



### **Transform Color Image Using Custom 2-D Transformation**

Define an inverse mapping function that accepts packed  $(x,y)$  coordinates, where the  $x$ -coordinate of each point is in the first column, and the  $y$ -coordinate of each point is in the second column. The inverse mapping function in this example takes the square of the polar radial component.

```
r = @(c) sqrt(c(:,1).^2 + c(:,2).^2);  
w = @(c) atan2(c(:,2), c(:,1));
```

```
f = @(c) [r(c).^2 .* cos(w(c)), r(c).^2 .* sin(w(c))];  
g = @(c) f(c);
```

Create a 2-D geometric transform object, `tform`, that stores the inverse mapping function.

```
tform = geometricTransform2d(g);
```

Read a color image to be transformed.

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



Create an `imref2d` object, specifying the size and world limits of the input and output images.

```
Rin = imref2d(size(I),[-1 1],[-1 1]);  
Rout = imref2d(size(I),[-1 1],[-1 1]);
```

Apply the inverse geometric transform to the input image.

```
Itransformed = imwarp(I,Rin,tform,'OutputView',Rout);  
imshow(Itransformed)
```



## See Also

[affine2d](#) | [geometricTransform3d](#) | [imwarp](#) | [projective2d](#)

## Topics

["2-D and 3-D Geometric Transformation Process Overview"](#)

**Introduced in R2018b**

## geometricTransform3d

3-D geometric transformation object

### Description

A `geometricTransform3d` object defines a custom 3-D geometric transformation using point-wise mapping functions.

### Creation

### Syntax

```
tform = geometricTransform3d(inverseFcn)
tform = geometricTransform3d(inverseFcn, forwardFcn)
```

### Description

`tform = geometricTransform3d(inverseFcn)` creates a `geometricTransform3d` object, and sets the value of inverse mapping function property, `InverseFcn` to `inverseFcn`.

`tform = geometricTransform3d(inverseFcn, forwardFcn)` also sets the value of forward mapping function property, `ForwardFcn` to `forwardFcn`.

### Properties

#### **InverseFcn — Inverse mapping function**

function handle

Inverse mapping function, specified as a function handle. The function must accept and return coordinates as an  $n$ -by-3 numeric matrix representing the packed  $(x,y,z)$  coordinates of  $n$  points.

For more information about function handles, see “Create Function Handle” (MATLAB).

Example: `ifcn = @(xyz) [xyz(:,1).^2,xyz(:,2).^2,xyz(:,3).^2];`

### **ForwardFcn – Forward mapping function**

function handle

Forward mapping function, specified as a function handle. The function must accept and return coordinates as an  $n$ -by-3 numeric matrix representing the packed  $(x,y,z)$  coordinates of  $n$  points.

For more information about function handles, see “Create Function Handle” (MATLAB).

Example: `ffcn = @(xyz) [sqrt(xyz(:,1)),sqrt(xyz(:,2)),sqrt(xyz(:,3))];`

## **Object Functions**

`transformPointsForward` Apply forward geometric transformation

`transformPointsInverse` Apply inverse geometric transformation

## **Examples**

### **Transform Packed Coordinates Using Custom 3-D Transformation**

Specify the packed  $(x,y,z)$  coordinates of five input points. The packed coordinates are stored as a 5-by-3 matrix, where the first, second, and third columns contain the  $x$ -,  $y$ -, and  $z$ - coordinates, respectively.

```
XYZ = [5 25 20;10 5 25;15 10 5;20 15 10;25 20 15];
```

Define an inverse mapping function that accepts and returns points in packed  $(x,y,z)$  format.

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

Create a 3-D geometric transformation object, `tform`, that stores this inverse mapping function.

```
tform = geometricTransform3d(inverseFcn)
```

```
tform =  
geometricTransform3d with properties:
```

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

Apply the inverse transformation of this 3-D geometric transformation to the input points.

```
UVW = transformPointsInverse(tform,XYZ)
```

```
UVW = 5×3
```

```
30   -20   400
15     5   625
25     5    25
35     5   100
45     5   225
```

### **Transform Coordinate Arrays Using Custom 3-D Transformation**

Specify the x-, y- and the z-coordinate vectors of five points to transform.

```
x = [3 5 7 9 11];
y = [2 4 6 8 10];
z = [5 9 13 17 21];
```

Define the inverse and forward mapping functions that accept and return points in packed (x,y,z) format.

```
inverseFcn = @(c)[c(:,1).^2,c(:,2).^2,c(:,3).^2];
forwardFcn = @(c)[sqrt(c(:,1)),sqrt(c(:,2)),sqrt(c(:,3))];
```

Create a 3-D geometric transformation object, `tform`, that stores these inverse and forward mapping functions.

```
tform = geometricTransform3d(inverseFcn,forwardFcn)
```

```
tform =
    geometricTransform3d with properties:
```

```
    InverseFcn: @(c)[c(:,1).^2,c(:,2).^2,c(:,3).^2]
```



```
ForwardFcn: @(c)[sqrt(c(:,1)),sqrt(c(:,2)),sqrt(c(:,3))]  
Dimensionality: 3
```

Apply the inverse transformation of this 3-D geometric transformation to the input points.

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

```
u = 1×5
```

```
    9    25    49    81   121
```

```
v = 1×5
```

```
    4    16    36    64   100
```

```
w = 1×5
```

```
   25    81   169   289   441
```

Apply the forward geometric transform to the transformed points u, v, and w.

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

```
x = 1×5
```

```
    3    5    7    9   11
```

```
y = 1×5
```

```
    2    4    6    8   10
```

```
z = 1×5
```

```
    5    9   13   17   21
```

### Transform 3-D Volumetric Image Using Custom 3-D Transformation

Define an inverse mapping function that performs reflection about horizontal axis. The function must accept and return packed  $(x,y,z)$  coordinates, where the first, second, and third columns contain the  $x$ -,  $y$ -, and  $z$ -coordinates, respectively.

```
inverseFcn = @(xyz)[xyz(:,1), -xyz(:,2), xyz(:,3)];
```

Create a 3-D geometric transformation object, `tform`, that stores this inverse mapping function.

```
tform = geometricTransform3d(inverseFcn)
```

```
tform =  
  geometricTransform3d with properties:  
    InverseFcn: @(xyz)[xyz(:,1), -xyz(:,2), xyz(:,3)]  
    ForwardFcn: []  
  Dimensionality: 3
```

Load and display an MRI volume to be transformed.

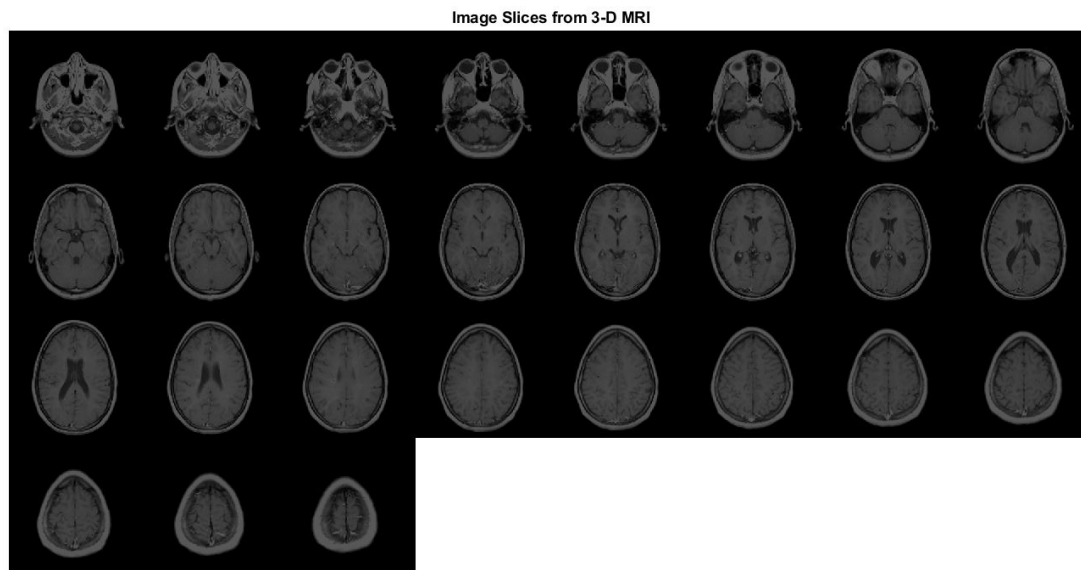
```
s = load('mri');  
mriVolume = squeeze(s.D);
```

Use `imwarp` to apply the inverse geometric transform to the input MRI volume.

```
[mriVolumeTransformed] = imwarp(mriVolume,tform,'nearest','SmoothEdges',true);
```

Display the image slices from the input MRI volume as montage.

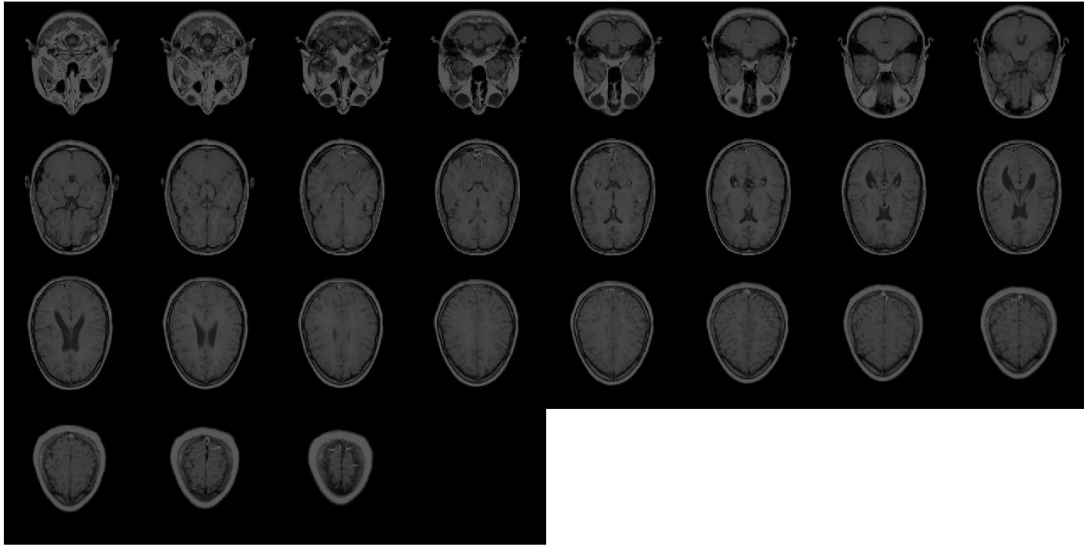
```
montage(mriVolume,'Size',[4 8],'BackgroundColor','w')  
title('Image Slices from 3-D MRI','FontSize',14)
```



Display the image slices from the transformed MRI volume as a montage. The transformed image slices are the reflection of the input image slices across the x-axis.

```
montage(mriVolumeTransformed, 'Size', [4 8], 'BackgroundColor', 'w')  
title('Image Slices from Inverse Geometric Transformation of 3-D MRI', 'FontSize', 14)
```

Image Slices from Inverse Geometric Transformation of 3-D MRI



## See Also

[affine2d](#) | [affine3d](#) | [geometricTransform2d](#) | [imwarp](#) | [projective2d](#)

## Topics

[“2-D and 3-D Geometric Transformation Process Overview”](#)

**Introduced in R2018b**

# 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

```
I = getimage(h)
[x,y,I] = getimage(h)
[ ____,flag] = getimage(h)
[ ____ ] = getimage
```

### Description

`I = getimage(h)` returns the first image data contained in the graphics object `h`.

`[x,y,I] = getimage(h)` also returns the image extent in the `x` and `y` direction.

`[ ____,flag] = getimage(h)` also returns a flag that indicates the type of image that `h` contains.

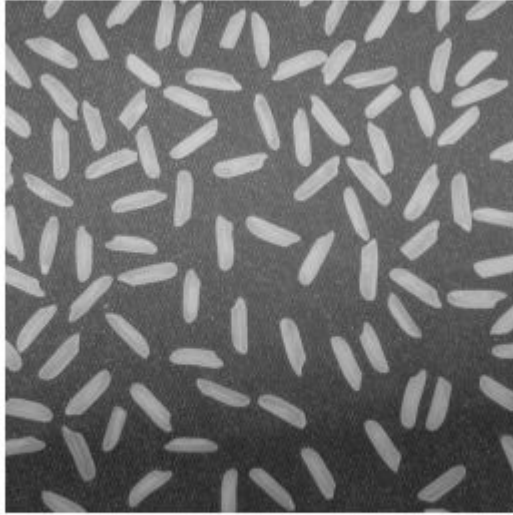
`[ ____ ] = getimage` returns information for the current axes object.

### 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);
```

## Input Arguments

### **h** — Handle to graphics object

handle

Handle to a figure, axes, uipanel, or image graphics object, specified as a handle. If **h** is an axes or figure handle containing multiple images, then `getimage` uses the first image returned by `findobj(h, 'Type', 'image')`.



## Output Arguments

### **I** — Image data

numeric array

Image data, returned as a numeric array. **I** is identical to the image **CData**; it contains the same values and is of the same class as the image **CData**. If **h** is not an image or does not contain an image, then **I** is empty.

### **x** — Image extent in x direction

2-element numeric vector

Image extent in the **x** direction, returned as a 2-element numeric vector of the form [**xmin** **xmax**]. **x** is identical to the image **XData**.

Data Types: double

### **y** — Image extent in y direction

2-element numeric vector

Image extent in the **y** direction, returned as a 2-element numeric vector of the form [**ymin** **ymax**]. **y** is identical to the image **YData**.

Data Types: double

### **flag** — Image type

integer

Image type, returned as an integer with one of these values:

Flag	Type of Image
0	Not an image; <b>I</b> is returned as an empty matrix
1	Indexed image
2	Intensity image with values in standard range. The standard range for <b>single</b> and <b>double</b> images is [0,1].
3	Intensity data, but not in standard range
4	RGB image
5	Binary image

Data Types: double

## **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.

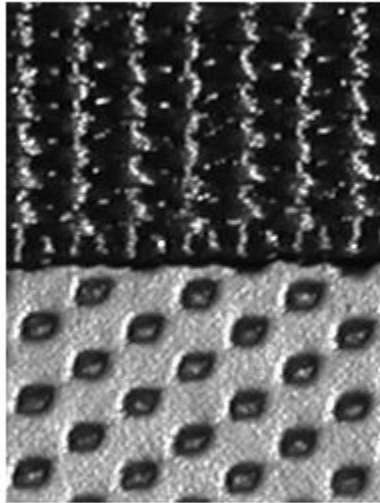
If `himage` does not have an associated image model object, `getimagemodel` creates one.

## 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

---

**Note** `getline` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

### Syntax

```
[xi,yi] = getline
[xi,yi] = getline(fig)
[xi,yi] = getline(ax)
[xi,yi] = getline( ____, 'closed')
```

### Description

`[xi,yi] = getline` lets you select a polyline in the current figure using the mouse. When you finish selecting the polyline, `getline` returns the coordinates of the polyline endpoints in `xi` and `yi`.

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.

`[xi,yi] = getline(fig)` lets you select a polyline in the current axes of figure `fig`, using the mouse.

`[xi,yi] = getline(ax)` lets you select a polyline in axes `ax`, using the mouse.

`[xi,yi] = getline( ____, 'closed')` animates and returns a closed polygon.

## Input Arguments

**fig — Figure handle**

handle

Figure handle, specified as a handle.

**ax — Axes handle**

handle

Axes handle, specified as a handle.

## Output Arguments

**xi — x-coordinates of selected polyline endpoints**

numeric vector

x-coordinates of selected polyline endpoints, returned as a numeric vector.

Data Types: double

**yi — y-coordinates of selected polyline endpoints**

numeric vector

y-coordinates of selected polyline endpoints, returned as a numeric vector.

Data Types: double

## See Also

getpts | getrect | imline

**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

Get 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`.

### Examples

#### Get Neighborhood of Flat Structuring Element

```
se = strel(eye(5));  
nhood = getnhood(se)
```

```
nhood =
```

```
5×5 logical array
```

```
 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
```

## Input Arguments

### SE — Structuring element

strel object | offsetstrel object

Structuring element, specified as a strel or offsetstrel object.

## Output Arguments

### nhood — Neighborhood of structuring element

logical array

Neighborhood of structuring element, returned as a logical array.

---

**Note** If SE is an offsetstrel object, then nhood indicates which pixels are in the neighborhood but does not return the offset of the pixels. You can get the offset from the property SE.Offset.

---

Data Types: logical

## See Also

### Topics

“Structuring Elements”

**Introduced before R2006a**

## getpts

Specify points with mouse

---

**Note** `getpts` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
[xi,yi] = getpts  
[xi,yi] = getpts(fig)  
[xi,yi] = getpts(ax)
```

## Description

`[xi,yi] = getpts` lets you choose points in the current figure using the mouse. When you finish selecting points, `getpts` returns the coordinates of the selected points in `xi` and `yi`.

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.

`[xi,yi] = getpts(fig)` lets you choose points in the current axes of figure `fig`, using the mouse.

`[xi,yi] = getpts(ax)` lets you choose points in axes `ax`, using the mouse.

## Examples

### Select Points in Image Interactively

Display an image using `imshow`.

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

Call `getpts` to choose points interactively in the displayed image using the mouse. Double-click to complete your selection. When you are done, `getpts` returns the coordinates of your points.

```
[x,y] = getpts
```

## Input Arguments

### **fig** — Figure handle

handle

Figure handle, specified as a handle.

### **ax** — Axes handle

handle

Axes handle, specified as a handle.

## Output Arguments

### **xi** — x-coordinates of sampled points

numeric vector

x-coordinates of sampled points, returned as a numeric vector.

Data Types: double

### **yi** — y-coordinates of sampled points

numeric vector

y-coordinates of sampled points, returned as a numeric vector.

Data Types: double

## See Also

`getline` | `getrect` | `impixel` | `impoint`

**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-MON02-16-ankle.dcm');
```

Get the display range from the image.

```
r = getrangefromclass(CT)
```

```
r = 1×2
```

```
    -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

`intmax` | `intmin`

**Introduced before R2006a**



# getrect

Specify rectangle with mouse

---

**Note** `getrect` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
rect = getrect
rect = getrect(fig)
rect = getrect(ax)
```

## Description

`rect = getrect` lets you select a rectangle in the current axes using the mouse. When you finish selecting the rectangle, `getrect` returns information about the position and size of the rectangle in `rect`.

Use the mouse to click and drag the desired rectangle. 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 axes `ax`, using the mouse.

## 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
```

## Input Arguments

### **fig** — Figure handle

handle

Figure handle, specified as a handle.

### **ax** — Axes handle

handle

Axes handle, specified as a handle.

## Output Arguments

### **rect** — Selected rectangle

4-element numeric vector

Selected rectangle, returned as a 4-element numeric vector with the form `[xmin ymin width height]`.

## See Also

`getline` | `getpts` | `imrect`

**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**

# grabcut

Segment image into foreground and background using iterative graph-based segmentation

## Syntax

```
BW = grabcut(A,L,ROI)
BW = grabcut(A,L,ROI,foremask,backmask)
BW = grabcut(A,L,ROI,foreind,backind)
BW = grabcut(V, ___)
BW = grabcut( ___,Name,Value)
```

## Description

`BW = grabcut(A,L,ROI)` segments the image `A` into foreground and background regions. The label matrix `L` specifies the subregions of the image. `ROI` is a logical mask designating the initial region of interest.

`BW = grabcut(A,L,ROI,foremask,backmask)` segments the image `A`, where `foremask` and `backmask` are masks designating pixels in the image as foreground and background, respectively.

`BW = grabcut(A,L,ROI,foreind,backind)` segments the image `A`, where `foreind` and `backind` specify the linear indices of the pixels in the image marked as foreground and background, respectively.

`BW = grabcut(V, ___)` segments the volume `V` into foreground and background regions.

`BW = grabcut( ___,Name,Value)` segments the image using name-value pairs to control aspects of the segmentation.

## Examples

## Segment Foreground from Background in Image Using Grabcut

Read an RGB image into the workspace.

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

Generate label matrix.

```
L = superpixels(RGB,500);
```

Specify a region of interest and create a mask image.

```
figure  
imshow(RGB)  
h1 = impoly(gca,[72,105; 1,231; 0,366; 104,359; ...  
394,307; 518,343; 510,39; 149,72]);
```



```
roiPoints = getPosition(h1);  
roi = poly2mask(roiParaments(:,1),roiPoints(:,2),size(L,1),size(L,2));
```

Perform the grab cut operation, specifying the original image, the label matrix and the ROI.

```
BW = grabcut(RGB,L,roi);  
figure  
imshow(BW)
```



Create masked image.

```
maskedImage = RGB;  
maskedImage(repmat(~BW,[1 1 3])) = 0;  
figure;  
imshow(maskedImage)
```





### Segment 3-D Volume Using Grabcut

Load 3-D volumetric data.

```
load mrystack  
V = mrystack;
```

Create a 2-D mask for initial foreground and background seed points.

```
seedLevel = 10;  
fseed = V(:,:,seedLevel) > 75;  
bseed = V(:,:,seedLevel) == 0;
```

Display foreground and background seed points.

```
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;
```

Create initial region of interest.

```
roi = false(size(V));  
roi(10:end-10,10:end-10,:) = true;
```

Generate label matrix.

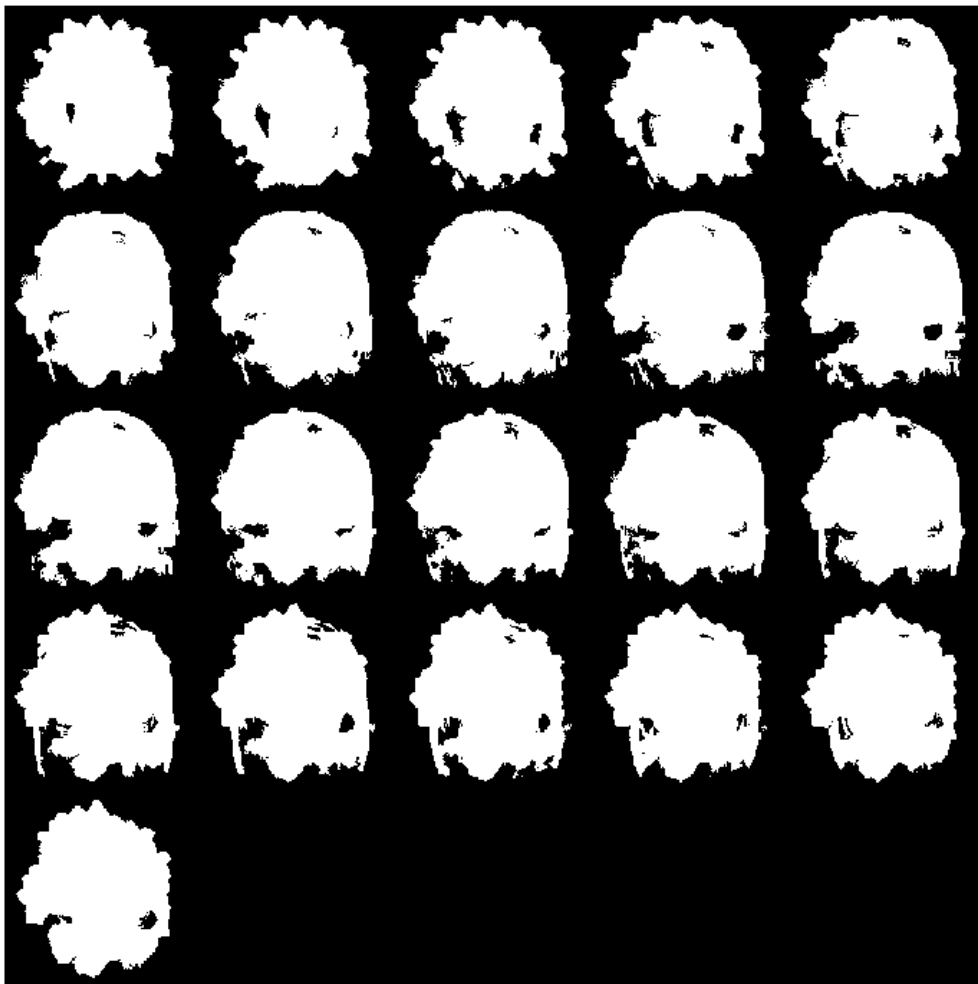
```
L = superpixels3(V,500);
```

Perform GrabCut.

```
bw = grabcut(V,L,roi,fmask,bmask);
```

Display 3D segmented image.

```
montage(reshape(bw,size(V)))
```



## Input Arguments

### **A — Input image**

real, finite, nonsparse numeric array

Input image, specified as a real, finite, nonsparse, numeric array. Only grayscale images can be `int16`.

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

### **V — Input volume**

real, finite, nonsparse, 3-D numeric array

Input volume, specified as a real, finite, nonsparse, 3-D numeric array.

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

### **L — Label matrix**

valid label matrix for input image

Label matrix, specified as a valid label matrix for input image.

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

### **ROI — Region of interest**

logical array

Region of interest, specified as a logical array, where all pixels that define the region of interest are equal to `true`.

Data Types: `logical`

### **foremask — Foreground mask**

logical array

Foreground mask, specified as a logical array.

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

### **backmask — Background mask**

logical array

Background mask, specified as a logical array.

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

### **foreind** — Indices of pixels in foreground

vector

Indices of pixels in foreground, specified as a vector of linear indices.

Data Types: `double`

### **backind** — Indices of pixels in background

vector

Indices of pixels in background, specified as a vector of linear indices.

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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.


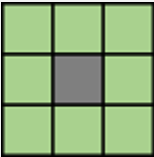
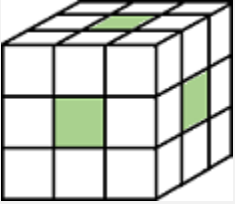
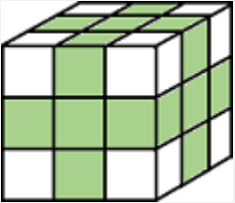
Example: `BW = grabcut(A,L,ROI,'Connectivity',4)`

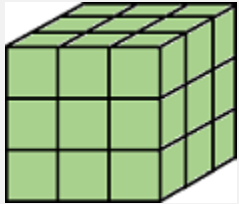
### **Connectivity** — Connectivity of connected components

`4` | `8` | `6` | `18` | `26`

Connectivity of connected components, specified as one of the following values. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

<b>Value</b>	<b>Meaning</b>
<b>Two-dimensional connectivities</b>	

Value	Meaning
4	4-connected neighborhood  Current pixel is shown in gray.
8	8-connected neighborhood  Current pixel is shown in gray.
<b>Three-dimensional connectivities</b>	
6	6-connected neighborhood  Current pixel is center of cube.
18	18-connected neighborhood  Current pixel is center of cube.

Value	Meaning
26	26-connected neighborhood  Current pixel is center of cube.

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

**MaximumIterations — Maximum number of iterations**

5 (default) | positive scalar

Maximum number of iterations performed by the algorithm. The algorithm can converge to a solution before reaching the maximum number of iterations.

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

## Output Arguments

**BW — Output image**

logical array

Output image, returned as binary image the same size as the label matrix L.

## Tips

- For `double` and `single` images, `grabcut` assumes the range of the image to be [0 1]. For `uint16`, `int16`, and `uint8` images, `grabcut` assumes the range to be the full range for the given data type.
- For grayscale images, the size of L, `foremask`, and `backmask` must match the size of the image A. For color and multi-channel images, L, `foremask`, and `backmask` must be 2-D arrays with the first two dimensions identical to the first two dimensions of the image A.



## Algorithms

- The algorithm treats all subregions fully or spatially outside the ROI mask as belonging to the background. To get an optimal segmentation, make sure the object to be segmented is fully contained within the ROI, surrounded by a small number of background pixels.
- Do not mark a 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, the algorithm effectively treats the region as unmarked.
- The algorithm assumes all subregions outside the region of interest belong to the background. Marking one of these subregions as belonging to foreground or background mask has no effect on the resulting segmentation.

## References

- [1] Rother, C., V. Kolmogorov, and A. Blake. "GrabCut - Interactive Foreground Extraction using Iterated Graph Cuts". *ACM Transactions on Graphics (SIGGRAPH)*. Vol. 23, Number 3, 2004, pp. 309-314.

## See Also

[Image Segmenter](#) | [lazysnapping](#) | [superpixels](#) | [watershed](#)

## Topics

"Label and Measure Objects in a Binary Image"

**Introduced in R2018a**

## 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')
```



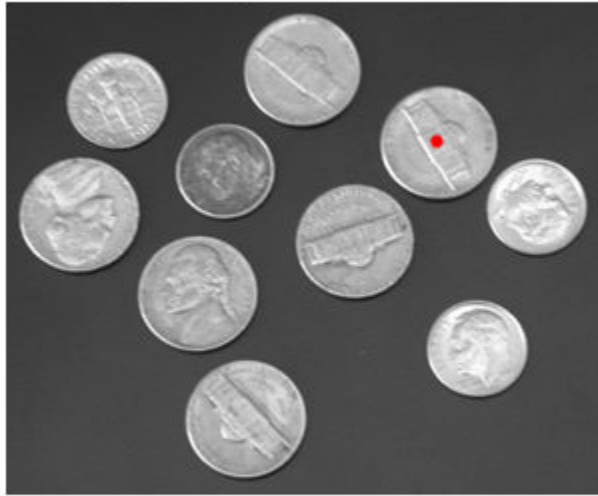
Compute weights based on image gradient.

```
sigma = 1.5;  
W = gradientweight(I, sigma, 'RolloffFactor', 3, 'WeightCutoff', 0.25);
```

Select a seed location.

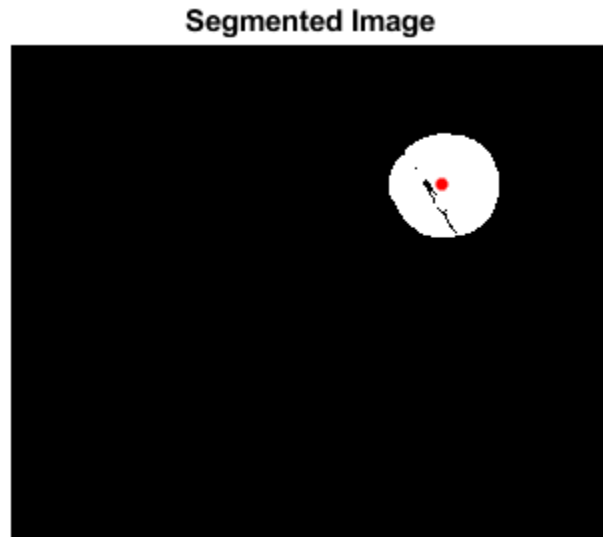
```
R = 70; C = 216;  
hold on;  
plot(C, R, 'r.', 'LineWidth', 1.5, 'MarkerSize', 15);  
title('Original Image with Seed Location')
```

**Original Image with Seed Location**



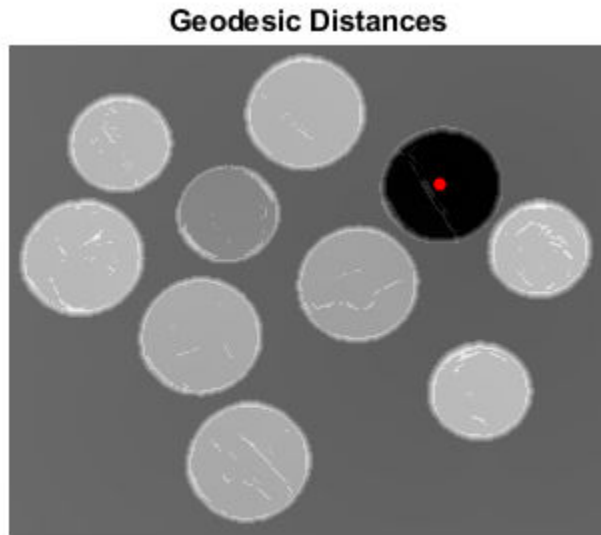
Segment the image using the weight array.

```
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);
```



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.

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 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, `imsegm`.

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

`graydiffweight` | `imsegfmm`

**Introduced in R2014b**



# gray2ind

Convert grayscale or binary image to indexed image

## Syntax

```
[X,cmap] = gray2ind(I,c)  
[X,cmap] = gray2ind(BW,c)
```

## Description

`[X,cmap] = gray2ind(I,c)` converts the grayscale image `I` to an indexed image `X` with colormap `cmap` with `c` colors.

`[X,cmap] = gray2ind(BW,c)` converts the binary image `BW` to an indexed image.

## 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);
```



## Input Arguments

### **I** — Grayscale image

numeric array

Grayscale image, specified as a numeric array of any dimension.

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

### **BW** — Binary image

numeric array

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

Data Types: `logical`

**c — Number of colormap colors**

positive integer

Number of colormap colors, specified as a positive integer between 1 and 65536.

- If the input image is grayscale, then the default value of `c` is 64.
- If the input image is binary, then the default value of `c` is 2.

## Output Arguments

**X — Indexed image**

numeric array

Indexed image, returned as a numeric array of the same dimensionality as the input grayscale or binary image. If the colormap length is less than or equal to 256, then the class of the output image is `uint8`; otherwise it is `uint16`.

Data Types: `uint8` | `uint16`**cmap — Color map**

c-by-3 numeric array

Color map associated with indexed image `X`, returned as a c-by-3 numeric array. The color map produced is equivalent to `gray(c)`.

Data Types: `double`

## 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.

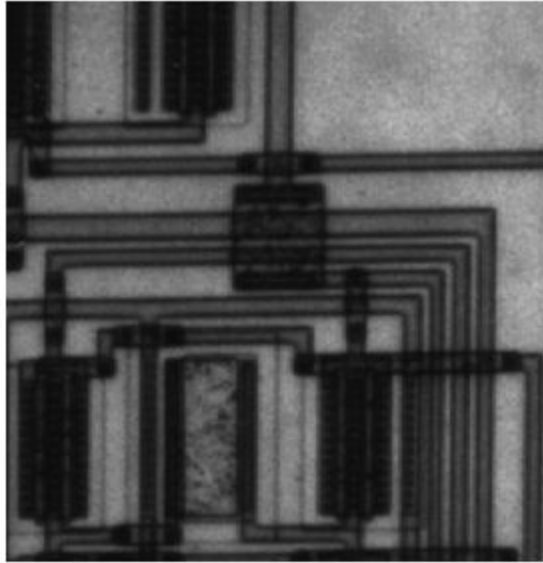
`[glcms,SI] = graycomatrix( ___ )` returns the scaled image, `SI`, used to calculate the gray-level co-occurrence matrix. The values in `SI` are between 1 and `NumLevels`.

### Examples

#### Create Gray-Level Co-occurrence Matrix for Grayscale Image

Read a grayscale image into the workspace.

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



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 = 8×8
```

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 = 3×6

```

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 values in the input image as limits.

```
[glcm,SI] = graycomatrix(I,'NumLevels',9,'GrayLimits',[])
```

glcm = 9×9

```

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 = 3×6

```

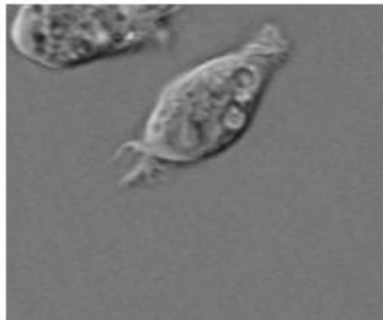
2     2     6     7     9     9
3     4     6     8     1     3
1     3     4     6     7     8

```

### Calculate GLCMs using Four Different Offsets

Read a grayscale image into the workspace.

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

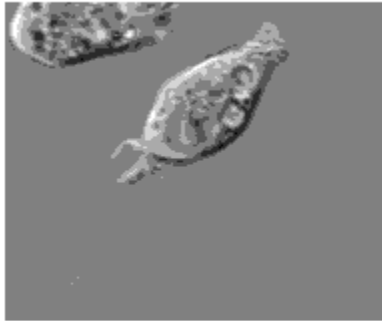


Define four offsets.

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

Calculate the GLCMs, returning the scaled image as well. Display the scaled image, performing an additional rescaling of data values to the range [0, 1].

```
[glcms,SI] = graycomatrix(I, 'Offset', offsets);  
imshow(rescale(SI))
```



Note how the function returns an array of four GLCMs.

```
whos
```

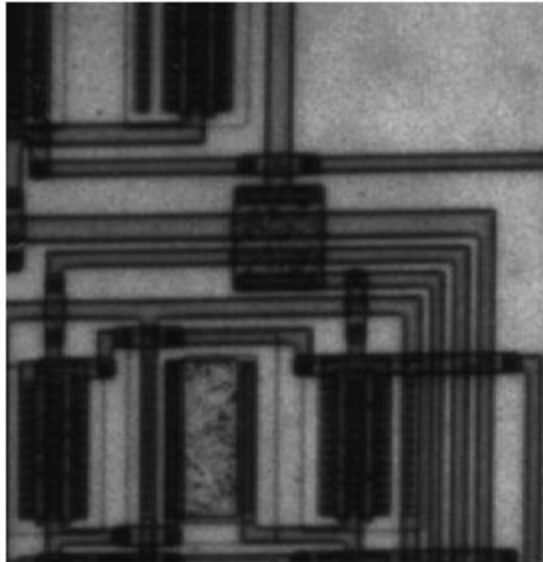
Name	Size	Bytes	Class	Attributes
I	159x191	30369	uint8	
SI	159x191	242952	double	
glcms	8x8x4	2048	double	
offsets	4x2	64	double	

## Calculate Symmetric GLCM for Grayscale Image

Read a grayscale image into the workspace.

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





Calculate the GLCM using the `Symmetric` option, returning the scaled image as well. 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,SI] = graycomatrix(I, 'Offset',[2 0], 'Symmetric',true);
glcm
```

```
glcm = 8x8
```

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

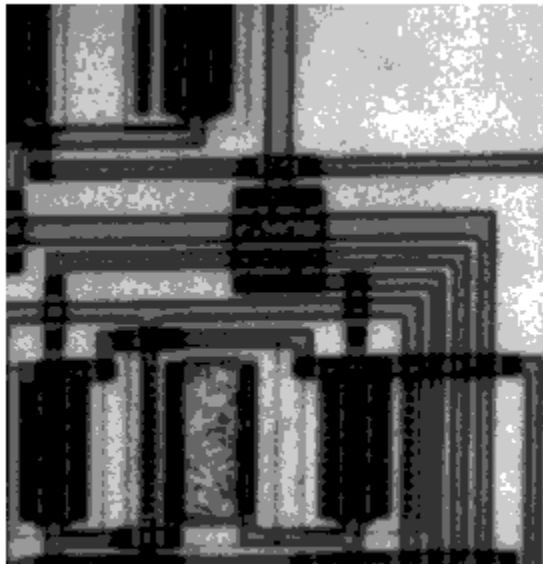
0

0

0

Display the scaled image, performing an additional rescaling of data values to the range [0, 1].

```
imshow(rescale(SI))
```



## 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 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`

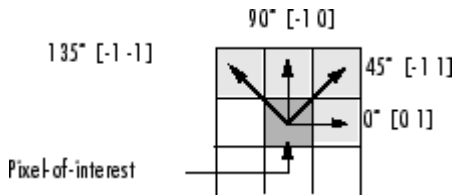
**Offset — Distance between the pixel of interest and its neighbor**

[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]
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

### **gIcms** — 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

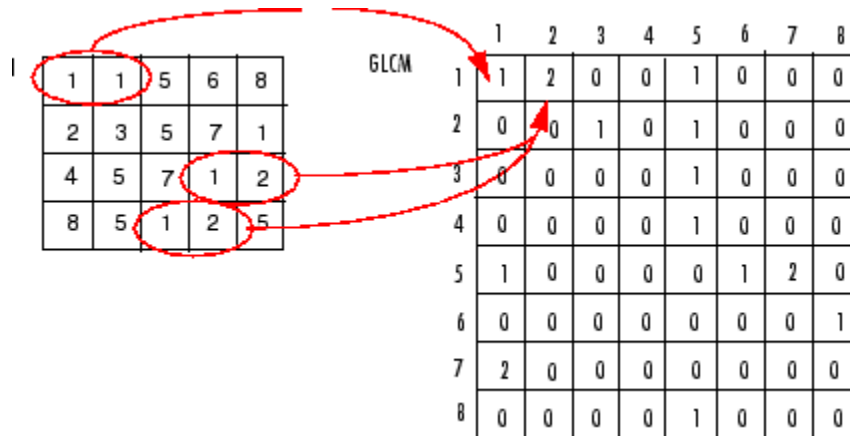
double matrix

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 Infs with the value NumLevels, and replaces negative Infs with the value 1. graycomatrix ignores border pixels, if the corresponding neighbor pixel falls outside the image boundaries.

The GLCM 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**

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  $[(seedvalue - tolerance), (seedvalue + tolerance)]$ . 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.

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 “Understand Code Generation with Image Processing Toolbox”.

## See Also

**Image Segmenter** | `bwselect` | `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 = 4×4
```

```
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 =
glcm(:,:,1) =
```

Columns 1 through 6

14205	2107	126	0	0	0
2242	14052	3555	400	0	0
191	3579	7341	1505	37	0
0	683	1446	7184	1368	0
0	7	116	1502	10256	1124
0	0	0	2	1153	1435
0	0	0	0	0	0
0	0	0	0	0	0

Columns 7 through 8

0	0
0	0
0	0
0	0

```
0      0
0      0
0      0
0      0
```

```
glcm(:,:,2) =
```

```
Columns 1 through 6
```

```
13938    2615    204      4      0      0
2406    14062    3311    630    23      0
145     3184    7371    1650   133      0
2        371    1621    6905   1706      0
0         0    116    1477   9974    1173
0         0      0      1    1161    1417
0         0      0      0      0      0
0         0      0      0      0      0
```

```
Columns 7 through 8
```

```
0      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(glcm,{'contrast','homogeneity'})
```

```
stats = struct with fields:
```

```
Contrast: [0.3420 0.3567]
Homogeneity: [0.8567 0.8513]
```

## Input Arguments

### glcm — 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 | cell array | space-separated string scalar or character vector

Statistical properties of the image derived from GLCM, specified as a comma-separated list string scalars or character vectors, space-separated string scalar or character vector, cell array of string scalars or character vectors, 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 = $[0 \ (\text{size}(\text{GLCM}, 1) - 1)^2]$  Contrast is 0 for a constant image.  The property Contrast is also known as variance and inertia.	$\sum_{i,j}  i - j ^2 p(i, j)$
'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.	$\sum_{i,j} \frac{(i - \mu_i)(j - \mu_j) p(i, j)}{\sigma_i \sigma_j}$

Property	Description	Formula
'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 uniformity, uniformity of energy, and angular second moment.	$\sum_{i,j} p(i,j)^2$
'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 | string | 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

`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 = imsegfmm(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 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 `imsegm`), 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**

`gradientweight` | `graydist` | `imsegfmm`

**Introduced in R2014b**



# graydist

Gray-weighted distance transform of grayscale image

## Syntax

```
T = graydist(I,mask)
T = graydist(I,C,R)
T = graydist(I,ind)
T = graydist( ____,method)
```

## Description

`T = graydist(I,mask)` computes the gray-weighted distance transform of the grayscale image `I`. Locations where `mask` is `true` are seed locations.

`T = graydist(I,C,R)` specifies the column and row coordinates of seed locations in vectors `C` and `R`.

`T = graydist(I,ind)` specifies the linear indices of seed locations, `ind`.

`T = graydist( ____,method)` specifies an alternate distance metric, `method`.

## 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 = 3×3
```

```
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 = 3x3
```

```
10   11   17
13   10   13
17   17   10
```

## Input Arguments

### **I** — Grayscale image

numeric array | logical array

Grayscale image, specified as a numeric or logical array.

### **mask** — Binary mask

logical array

Binary mask that specifies seed locations, specified as a logical array the same size as I.

### **C, R** — Column and row coordinates

vector of positive integers

Column and row coordinates of seed locations, specified as a vector of positive integers. Coordinate values are valid C,R subscripts in I.

**ind — Indices**

vector of positive integers

Indices of seed locations, specified as a vector of positive integers.

**method — Distance metric**

'chessboard' (default) | 'cityblock' | 'quasi-euclidean'

Distance metric, specified as one of these values.

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

For more information, see “Distance Transform of a Binary Image”.

## Output Arguments

**T — Gray-weighted distance transform**

numeric array

Gray-weighted distance transform, returned as a numeric array of the same size as I. If the input numeric type of I is double, then the output numeric type of T is double. If the input is any other numeric type, then the output T is single.

Data Types: single | double

## Algorithms

`graydist` uses the geodesic time algorithm [1]. The basic equation for geodesic time along a path is:

$$\tau_f(P) = \frac{f(p_o)}{2} + \frac{f(p_l)}{2} + \sum_{i=1}^{l-1} f(p_i)$$

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.

## References

- [1] Soille, P. "Generalized geodesy via geodesic time." *Pattern Recognition Letters*. Vol.15, December 1994, pp. 1235-1240.

## See Also

`bwdist` | `bwdistgeodesic` | `watershed`

## Topics

"Distance Transform of a Binary Image"

**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.

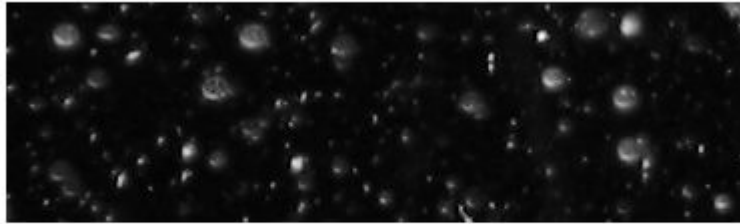
```
I = imread('snowflakes.png');
```

Threshold the intensity image, returning an indexed image.

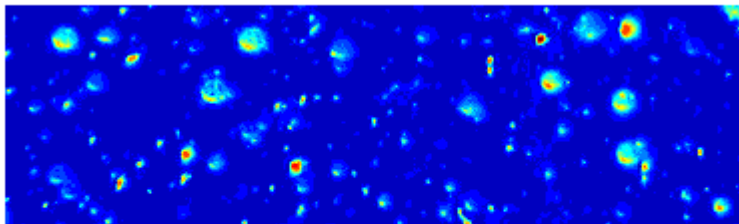
```
X = grayslice(I,16);
```

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].

`[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
```

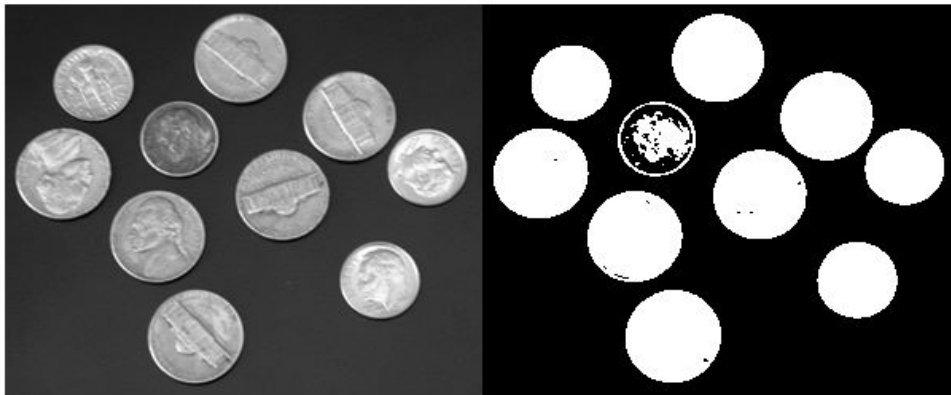


Convert the image into a binary image using the threshold.

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

Display the original image next to the binary image.

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



## Input Arguments

### **I** — Intensity image

nonsparse N-D array

Intensity image, specified as a nonsparse N-D array. The `graythresh` function converts multidimensional arrays to 2-D arrays, using `reshape`, and ignores any nonzero imaginary part of `I`.

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

`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, `hdr`, from the file specified by `filename`. For scene-referred data sets, pixel values usually are scene illumination in radiance units.

## 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);
```



## Input Arguments

### **filename** — File name

character vector | string scalar

File name of HDR image, specified as a character vector or string scalar.

Example: 'office.hdr' or "office.hdr"

Data Types: char | string

## Output Arguments

### **hdr** — HDR image

*m*-by-*n*-by-3 numeric array

HDR image, returned as an *m*-by-*n*-by-3 numeric array with values in the range  $[0, \text{Inf})$ .

Data Types: `single`

## Tips

- To display HDR images, use an appropriate tone-mapping function, such as `tonemap`.

## 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 high dynamic range (HDR) image file

### Syntax

```
hdrwrite(hdr, filename)
```

### Description

`hdrwrite(hdr, filename)` writes high dynamic range (HDR) image `hdr` to a file with name `filename`. The function 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');
```

### Input Arguments

#### **hdr** — HDR image

*m*-by-*n*-by-3 numeric array

HDR image, specified as an *m*-by-*n*-by-3 numeric array of positive numbers.

Data Types: `single` | `double`

**filename — File name**

character vector | string scalar

File name of HDR image, specified as a character vector or string scalar ending with extension 'hdr'.

Example: 'office.hdr' or "office.hdr"

Data Types: char | string

**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)

newmap = histeq(X,map)
newmap = histeq(X,map,hgram)
[newmap,T] = histeq(X, ___)
```

### Description

`J = histeq(I,hgram)` transforms the grayscale image `I` so that the histogram of the output grayscale image `J` with `length(hgram)` bins approximately matches the target histogram `hgram`.

You optionally can perform histogram equalization of grayscale images using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`J = histeq(I,n)` transforms the grayscale image `I`, returning in `J` an grayscale 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`.

`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`.

This syntax is not supported on a GPU.



`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)`.

This syntax is not supported on a GPU.

`[newmap,T] = histeq(X, ___)` returns the grayscale transformation `T` that maps the gray component of `map` to the gray component of `newmap`.

This syntax is not supported on a GPU.

## 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);
```

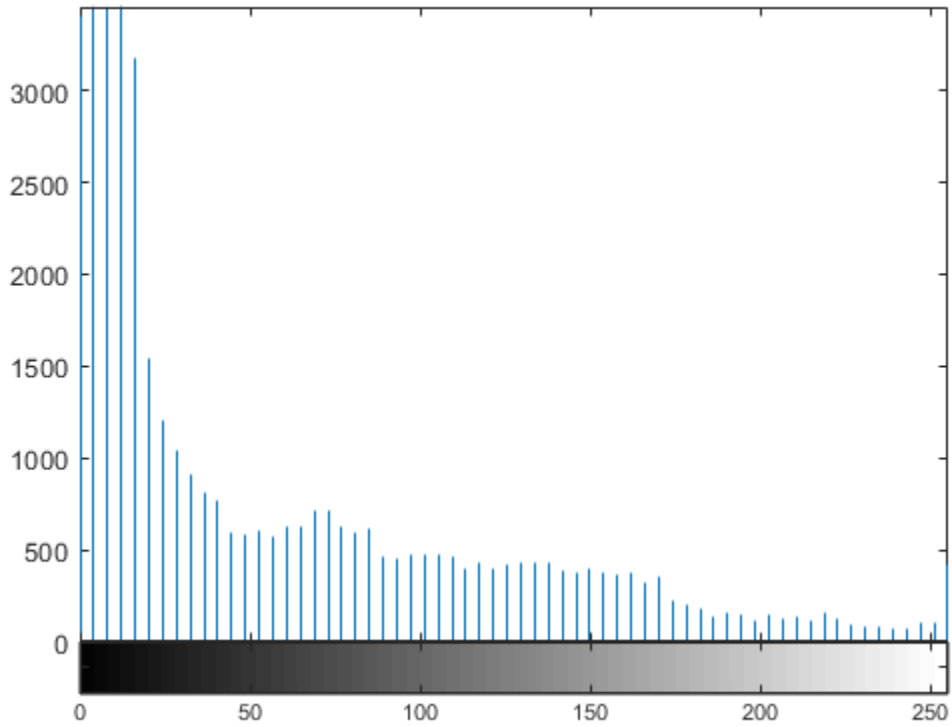
Display the original image and the adjusted image.

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



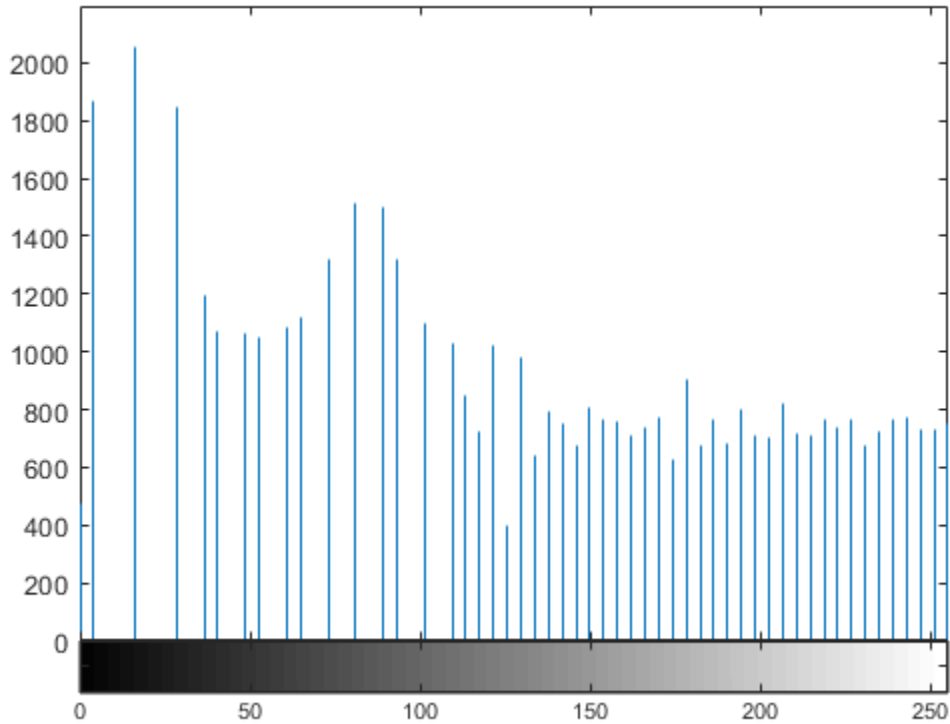
Display a histogram of the original image.

```
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.

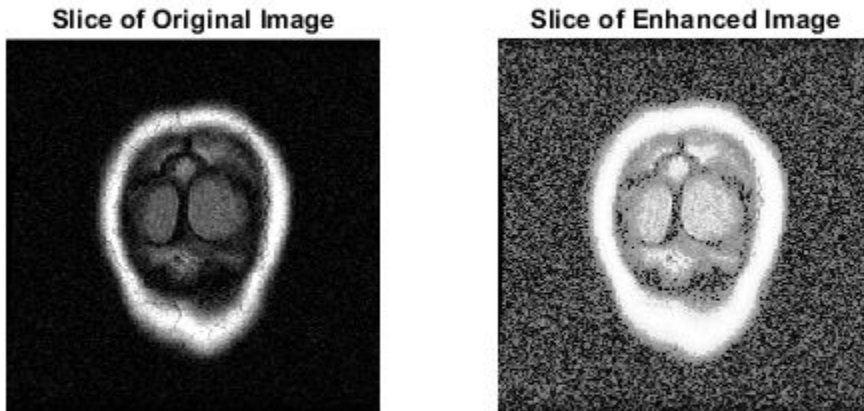
```
load mrstack
```

Perform histogram equalization.

```
enhanced = histeq(mrstack);
```

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')
```



### 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 grayscale image

numeric array | gpuArray

Input grayscale image, specified as a numeric array of any dimension.

To perform histogram equalization using a GPU, specify **I** as a `gpuArray` that contains a numeric array of any dimension.

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

### **hgram** — Target histogram

numeric vector | gpuArray

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`
- [-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**.

If you perform histogram equalization using a GPU, then you optionally can specify `hgram` as a `gpuArray` that contains a numeric vector.

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`

### **X — Indexed image**

numeric array

Indexed image, specified as a numeric array of any dimension. The values in *X* are an index into the colormap *map*.

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

### **map — Colormap**

*c*-by-3 array

Colormap, specified as an *c*-by-3 array. Each row specifies an RGB color value.

Data Types: `double`

## **Output Arguments**

### **J — Output grayscale image**

numeric array | `gpuArray`

Output grayscale image, returned as a numeric array of the same size and class as the input image *I*.

If histogram equalization is performed using a GPU, then *J* is returned as a `gpuArray` containing a numeric array.

### **T — Grayscale transformation**

numeric vector | `gpuArray`

Grayscale transformation, returned as a numeric vector. The transformation *T* maps gray levels in the image *I* to gray levels in *J*.

If histogram equalization is performed using a GPU, then *T* is returned as a `gpuArray` containing a numeric vector.

Data Types: `double`

### **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`, then `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 “Understand Code Generation with Image Processing Toolbox”.



- When generating code, `histeq` does not support indexed images.

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

## See Also

`brighten` | `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 1-881.

`[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 an image, and convert it to a grayscale image.

```
RGB = imread('gantrycrane.png');
I = rgb2gray(RGB);
```

Extract edges.

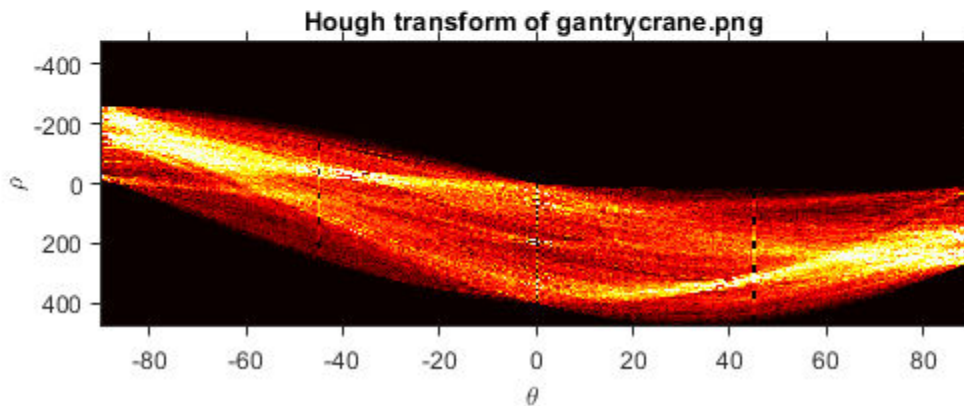
```
BW = edge(I, 'canny');
```

Calculate Hough transform.

```
[H,T,R] = hough(BW, 'RhoResolution',0.5, 'Theta', -90:0.5:89);
```

Display the original image and the Hough matrix.

```
subplot(2,1,1);  
imshow(IMG);  
title('gantrycrane.png');  
subplot(2,1,2);  
imshow(imadjust(rescale(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);
```



### Compute Hough Transform Over Limited Theta Range

Read an image, and convert it to grayscale.

```
RGB = imread('gantrycrane.png');  
I = rgb2gray(RGB);
```

Extract edges.

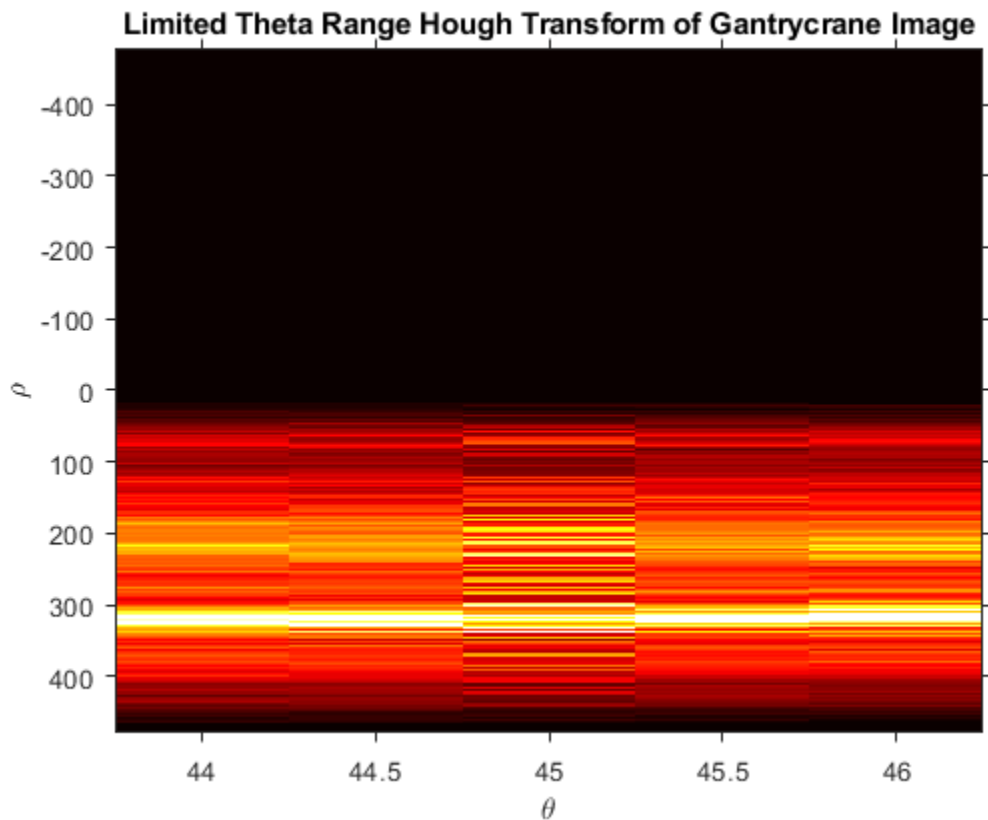
```
BW = edge(I, 'canny');
```

Calculate the Hough transform over a limited range of angles.

```
[H,T,R] = hough(BW, 'Theta', 44:0.5:46);
```

Display the Hough transform.

```
figure  
imshow(imadjust(rescale(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)
```



## 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 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, *n**rho*-by-*n**theta* in size. The rows and columns correspond to *rho* and *theta* values. For more information, see “Algorithms” on page 1-881.

### **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 1-881.

### **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 1-881.

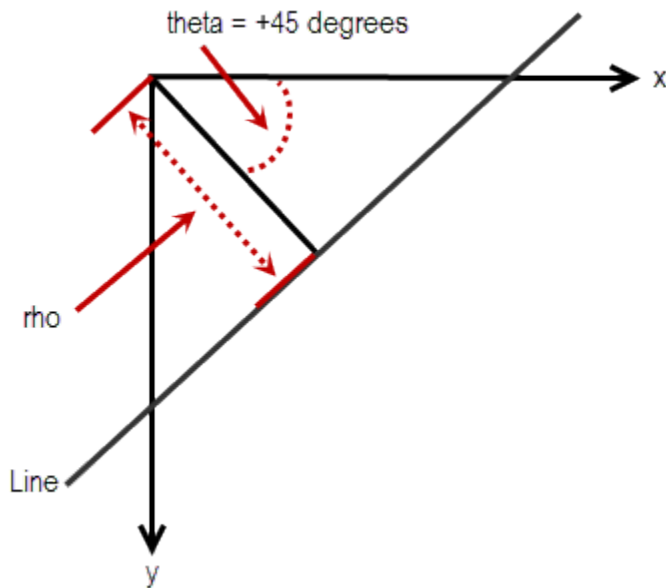
## Algorithms

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:

$nrho = 2 * (\text{ceil}(D/RhoResolution)) + 1$ , and  
 $D = \text{sqrt}((\text{numRowsInBW} - 1)^2 + (\text{numColsInBW} - 1)^2)$ .  
*rho* values range from -*diagonal* to *diagonal*, where  
 $\text{diagonal} = RhoResolution * \text{ceil}(D/RhoResolution)$ .

$ntheta = \text{length}(\text{theta})$



## 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

[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.

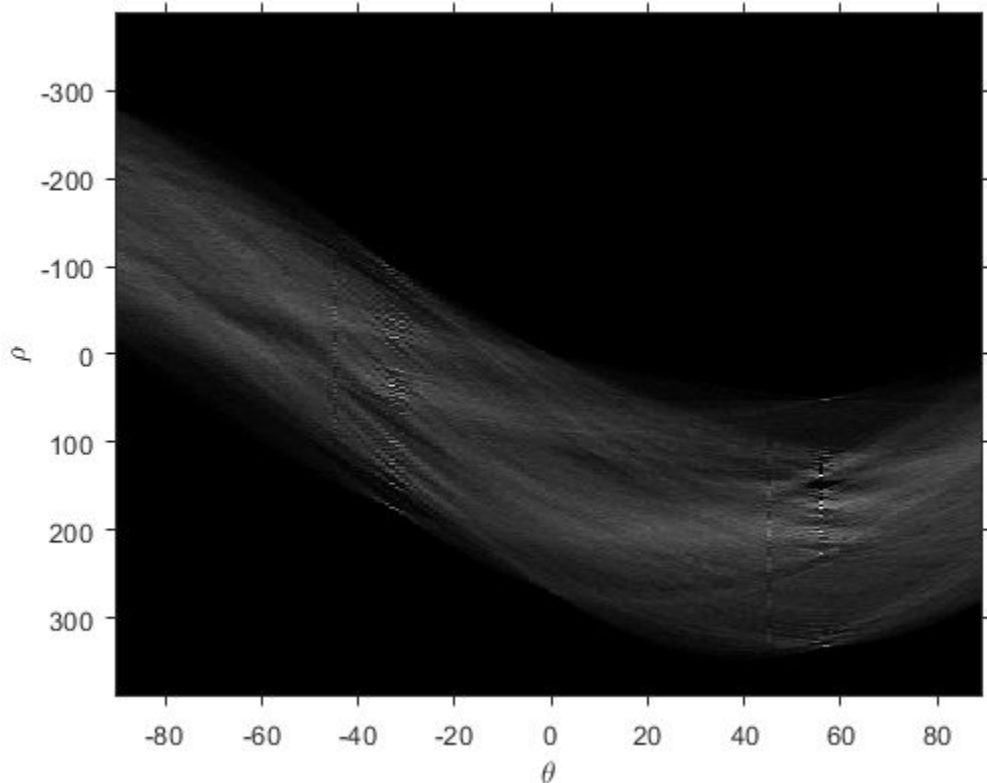
```
rotI = imrotate(I,33,'crop');
```

Create a binary image.

```
BW = edge(rotI,'canny');
```

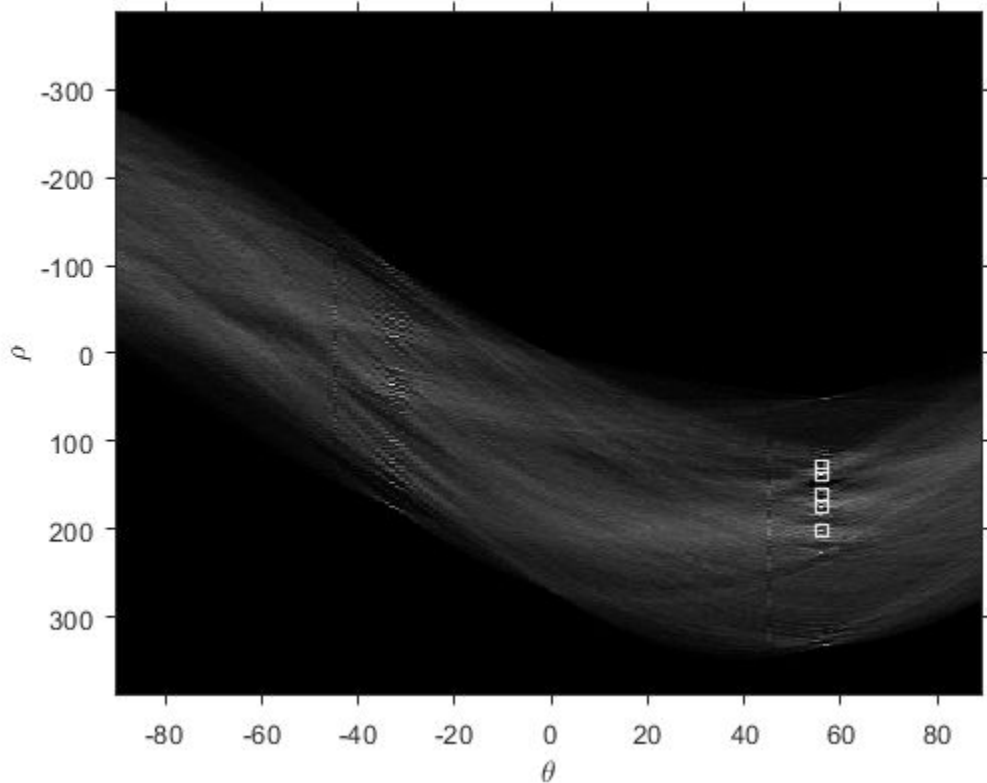
Create the Hough transform using the binary image.

```
[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');
```



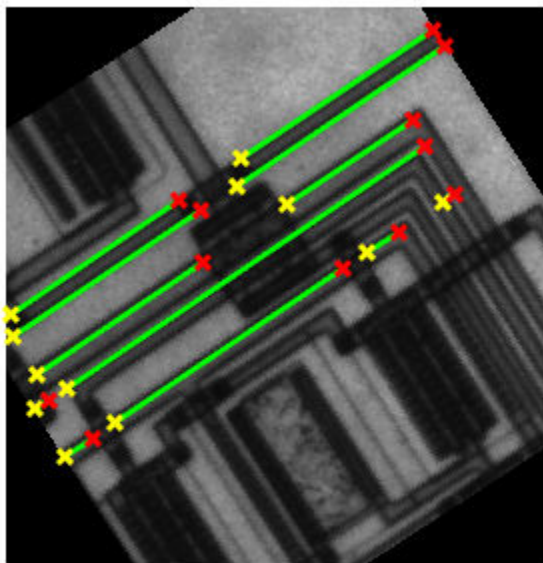
Find lines and plot them.

```
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');

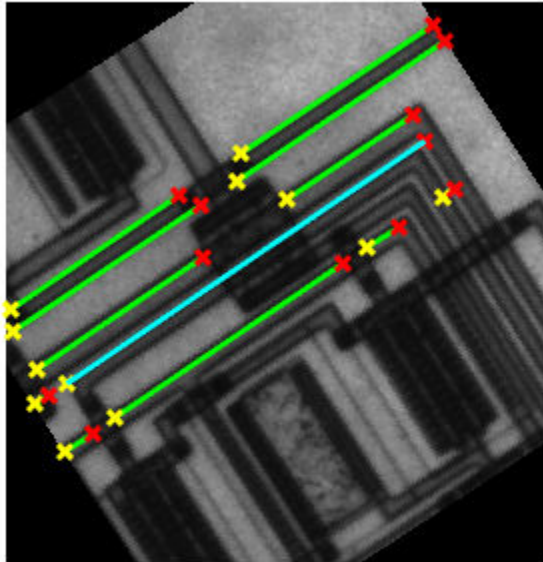
    % Determine the endpoints of the longest line segment
```

```
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 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

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.

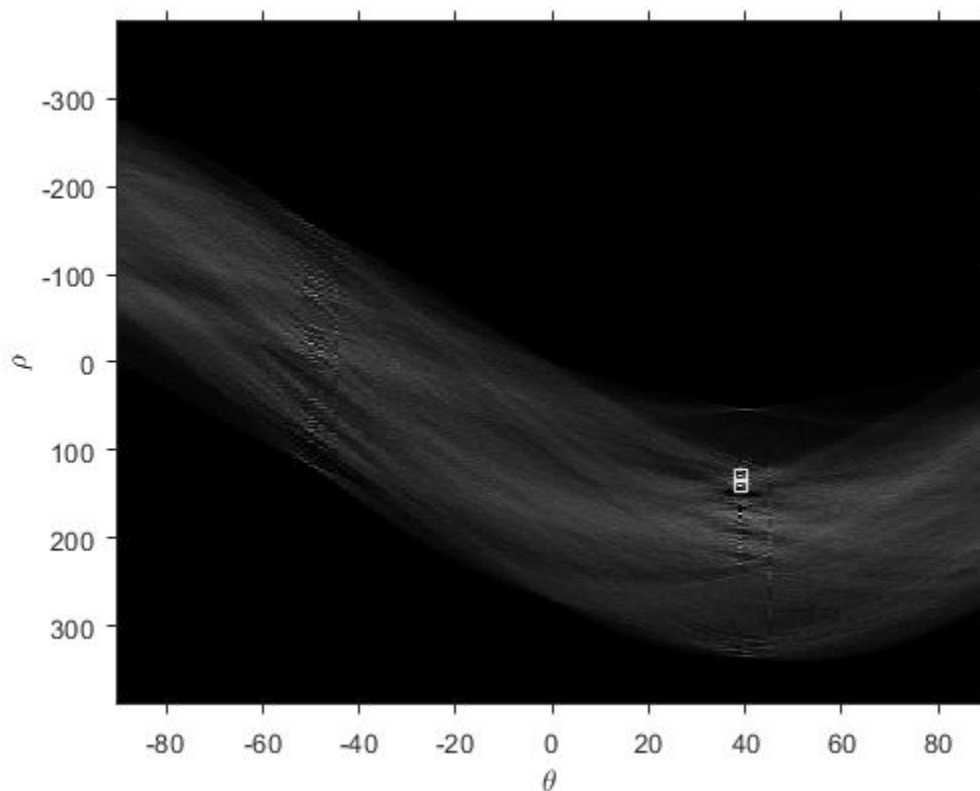
```
BW = edge(imrotate(I,50,'crop'),'canny');
```

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 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`

**Theta — Hough transform theta values**

`-90:89` (default) | vector

Hough transform theta values, specified as a vector returned by the `hough` function. Each element of the vector specifies the *theta* value for the corresponding column of the output matrix `H`. `houghpeaks` uses the *theta* values specified for peak suppression.

---

**Note** If you specify the 'Theta' parameter as input to the `hough` function, you must specify the `theta` parameter with the `houghpeaks` function. Use the `theta` output value from the `hough` function as the `theta` input value for `houghpeaks`. Otherwise, peak suppression can result in unexpected results.

---

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

`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](http://www.color.org).

ICC profiles provide color management systems with the information necessary to convert color data between native device color spaces and device independent color spaces, called the Profile Connection Space (PCS). You can use the profile as the source or destination profile with the `makecform` function to compute color space transformations.

The number of fields in `P` depends on the profile class and the choices made by the profile creator. `iccread` returns all the tags for a given profile, both public and private. Private tags and certain public tags are left as encoded `uint8` data. The following table lists fields that are found in any profile structure generated by `iccread`, in the order they appear in the structure.

Field	Data Type	Description
Header	1-by-1 struct array	Profile header fields
TagTable	n-by-3 cell array	Profile tag table
Copyright	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`:

<code>AToB0</code>	<code>BToA0</code>	<code>Preview0</code>
<code>AToB1</code>	<code>BToA1</code>	<code>Preview1</code>
<code>AToB2</code>	<code>BToA2</code>	<code>Preview2</code>
<code>AToB3</code>	<code>BToA3</code>	<code>Gamut</code>

## 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: [1x1 struct]
    TagTable: {17x3 cell}
    Copyright: 'Copyright (c) 1999 Hewlett-Packard Company'
    Description: [1x1 struct]
    MediaWhitePoint: [0.9505 1 1.0891]
    MediaBlackPoint: [0 0 0]
    DeviceMfgDesc: [1x1 struct]
    DeviceModelDesc: [1x1 struct]
    ViewingCondDesc: [1x1 struct]
    ViewingConditions: [1x1 struct]
    Luminance: [76.0365 80 87.1246]
    Measurement: [1x1 struct]
    Technology: 'Cathode Ray Tube Display'
    MatTRC: [1x1 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
```

```
[1x1 struct]  
[1x1 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](http://www.color.org).

---

**Note** `iccwrite` does not perform automatic conversions from one version of the ICC specification to another. Such conversions have to be done manually, by adding fields or modifying fields. Use `isicc` to validate a profile.

---



## 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](#) | [iccread](#) | [isicc](#) | [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)` and

`B = idct2(A,[m n])` pads `A` with 0s to size `m`-by-`n` before applying the inverse transformation. If `m` or `n` is smaller than the corresponding dimension of `A`, then `idct2` crops `A` before the transformation.

### 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.

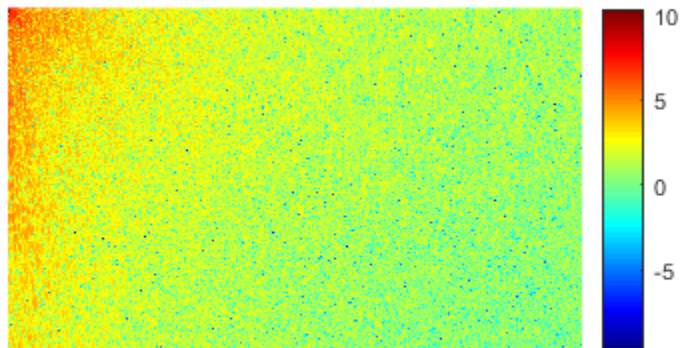
```
RGB = imread('autumn.tif');
I = rgb2gray(RGB);
```

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)



## Input Arguments

### **A** — Input matrix

2-D numeric matrix

Input matrix, specified as a 2-D numeric matrix.

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

### **m** — Number of image rows

`size(A,1)` (default) | positive integer

Number of image rows, specified as a positive integer. `dct2` pads image A with 0s or truncates image A so that it has m rows. By default, m is equal to `size(A,1)`.

### **n** — Number of image columns

`size(A,2)` (default) | positive integer

Number of image columns, specified as a positive integer. `dct2` pads image A with 0s or truncates image A so that it has n columns. By default, n is equal to `size(A,2)`

## Output Arguments

### B — Transformed matrix

m-by-n numeric matrix

Transformed matrix using a two-dimensional discrete cosine transform, returned as an m-by-n numeric matrix.

Data Types: double

## Tips

- For any matrix A, `idct2(dct2(A))` equals A to within roundoff error.

## Algorithms

`idct2` computes the two-dimensional inverse DCT using:

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

where

$$\alpha_p = \begin{cases} \frac{1}{\sqrt{M}}, & p=0 \\ \sqrt{\frac{2}{M}}, & 1 \leq p \leq M-1 \end{cases}$$

and

$$\alpha_q = \begin{cases} \frac{1}{\sqrt{N}}, & q=0 \\ \sqrt{\frac{2}{N}}, & 1 \leq q \leq N-1 \end{cases}.$$

## 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(F,D,Name,Value)
[I,H] = ifanbeam( ___ )
```

## Description

`I = ifanbeam(F,D)` reconstructs the image `I` from fan-beam projection data in `F`. Each column of `F` contains fan-beam projection data at one rotation angle. The angle between sensors is assumed to be uniform and equal to the increment between fan-beam rotation angles. `D` is the distance from the fan-beam vertex to the center of rotation.

`I = ifanbeam(F,D,Name,Value)` uses name-value pairs to control various aspects of the reconstruction. Argument names can be abbreviated, and case does not matter.

`[I,H] = ifanbeam( ___ )` also returns the frequency response of the filter, `H`.

## 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.

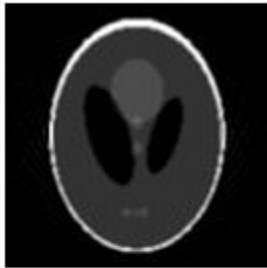
```
d = 100;
F = fanbeam(ph,d);
```

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.

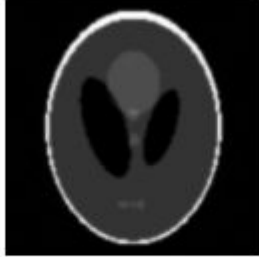
```
phReconstructed = ifanbeam(F,100,...  
                           'FanSensorSpacing',0.5,...  
                           'Filter','Shepp-Logan',...  
                           'OutputSize',128,...  
                           'FanCoverage','minimal',...  
                           'FanRotationIncrement',1);
```

Display the original and the transformed image.

```
imshow(ph)
```



```
figure  
imshow(phReconstructed)
```



## Input Arguments

### **F — Fan-beam projection data**

*numsensors-by-numangles* numeric matrix

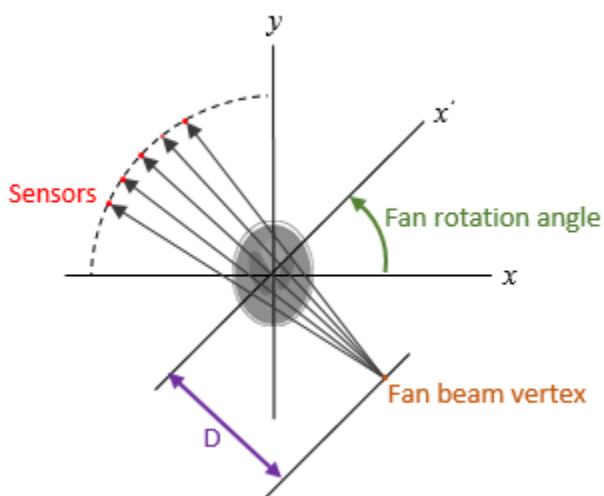
Fan-beam projection data, specified as a *numsensors-by-numangles* numeric matrix. *numsensors* is the number of fan-beam sensors and *numangles* is the number of fan-beam rotation angles. Each column of *F* contains the fan-beam sensor samples at one rotation angle.

Data Types: `double` | `single`

### **D — Distance from fan-beam vertex to center of rotation**

positive number

Distance in pixels from the fan-beam vertex to the center of rotation, specified as a positive number. `ifanbeam` assumes that the center of rotation is the center point of the projections, which is defined as `ceil(size(F,1)/2)`. The figure illustrates *D* in relation to the fan-beam vertex for one fan-beam projection.



Data Types: double | 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `I = ifanbeam(F,D,'FanRotationIncrement',5)`

### FanCoverage — Range of fan-beam rotation

'cycle' (default) | 'minimal'

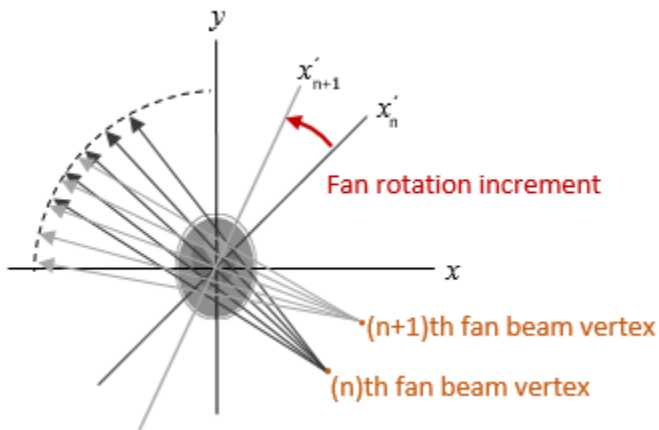
Range of fan-beam rotation, specified as the comma-separated pair consisting of 'FanCoverage' and one of these values.

- 'cycle' — Rotate through the full range [0, 360) degrees.
- 'minimal' — Rotate through the minimum range necessary to represent the object.

### FanRotationIncrement — Fan-beam rotation angle increment

1 (default) | positive scalar

Fan-beam rotation angle increment in degrees, specified as the comma-separated pair consisting of 'FanRotationIncrement' and a positive scalar.

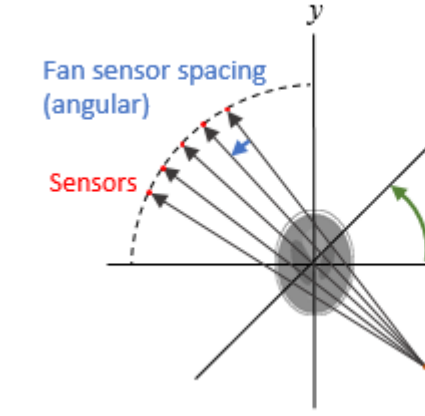
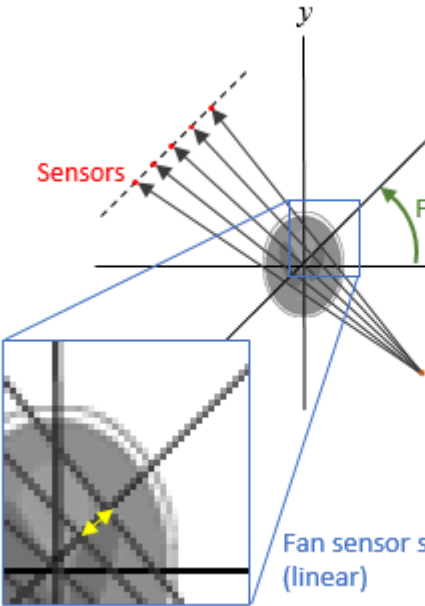


Data Types: double

### FanSensorGeometry — Fan-beam sensor positioning

'arc' (default) | 'line'

Fan-beam sensor positioning, specified as the comma-separated pair consisting of 'FanSensorGeometry' and one of the following values.

Value	Meaning	Diagram
'arc'	<p>Sensors are spaced at equal angles along a circular arc at distance <math>D</math> from the center of rotation.</p> <p>FanSensorSpacing defines the angular spacing in degrees.</p>	 <p>Fan sensor spacing (angular)</p> <p>Sensors</p> <p>Fan rotation</p> <p>Fan beam vectors</p>
'line'	<p>Sensors are spaced at equal distances along a line that is parallel to the <math>x'</math> axis. The closest sensor is distance <math>D</math> from the center of rotation.</p> <p>FanSensorSpacing defines the distance between fan-beams on the <math>x'</math> axis, in pixels.</p>	 <p>Sensors</p> <p>Fan rotation</p> <p>Fan beam vectors</p> <p>Fan sensor spacing (linear)</p>

### FanSensorSpacing – Fan-bean sensor spacing

1 (default) | positive scalar

Fan-beam sensor spacing, specified as the comma-separated pair consisting of 'FanSensorSpacing' and a positive scalar.

- If FanSensorGeometry is 'arc', then FanSensorSpacing defines the angular spacing in degrees.
- If FanSensorGeometry is 'line', then FanSensorSpacing defines the linear distance between fan-beams, in pixels. Linear spacing is measured on the x' axis.

Data Types: double

**Filter — Filter**

'Ram-Lak' (default) | 'Shepp-Logan' | 'Cosine' | 'Hamming' | 'Hann' | 'None'

Filter to use for frequency domain filtering, specified as the comma-separated pair consisting of 'Filter' and one of the values in the table. For more information, see `iradon`.

Value	Description
'Ram-Lak'	Cropped Ram-Lak or ramp filter. The frequency response of this filter is $ f $ . Because this filter is sensitive to noise in the projections, one of the filters listed below might be preferable. These filters multiply the Ram-Lak filter by a window that deemphasizes high frequencies.
'Shepp-Logan'	Multiplies the Ram-Lak filter by a <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. <code>ifanbeam</code> returns unfiltered data.

Data Types: char | string

**FrequencyScaling — Scale factor**

1 (default) | positive number in the range (0, 1]

Scale factor for rescaling the frequency axis, specified as the comma-separated pair consisting of 'FrequencyScaling' and a positive number in the range (0, 1]. If 'FrequencyScaling' is less than 1, then the filter is compressed to fit into the frequency range  $[0, \text{FrequencyScaling}]$ , in normalized frequencies; all frequencies above `FrequencyScaling` are set to 0. For more information, see `iradon`.

Data Types: double

### Interpolation — Type of interpolation

'Linear' (default) | 'nearest' | 'spline' | 'pchip'

Type of interpolation used between the parallel-beam and fan-beam data, specified as the comma-separated pair consisting of 'Interpolation' and one of the following values.

'nearest' — Nearest-neighbor

'linear' — Linear (the default)

'spline' — Piecewise cubic spline

'pchip' — Piecewise cubic Hermite (PCHIP)

Data Types: char | string

### OutputSize — Size of reconstructed image

positive integer

Size of the reconstructed image, specified as the comma-separated pair consisting of 'OutputSize' and a positive integer. The image has an equal number of rows and columns.

If you specify `OutputSize`, then `ifanbeam` reconstructs a smaller or larger portion of the image but does not change the scaling of the data.

---

**Note** If the projections were calculated with the `fanbeam` function, then the reconstructed image might not be the same size as the original image.

---

If you do not specify `OutputSize`, then the size is calculated automatically by:

$$\text{OutputSize} = 2 * \text{floor}(\text{size}(R, 1) / (2 * \text{sqrt}(2)))$$

where `R` is the length of parallel-beam projection data used by `iradon`. For more information, see “Algorithms” on page 1-922.

Data Types: double

## Output Arguments

### **I — Reconstructed image**

2-D numeric matrix

Reconstructed image, specified as a 2-D numeric matrix.

### **H — Frequency response**

numeric vector

Frequency response of the filter, returned as a numeric vector.

Data Types: `double`

## Tips

- To perform an inverse fan-beam reconstruction, you must give `ifanbeam` the same parameters that were used to calculate the projection data, `F`. If you use `fanbeam` to calculate the projection, then make sure the parameters are consistent when calling `ifanbeam`.

## Algorithms

`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.

## References

- [1] Kak, A. C., and M. Slaney, *Principles of Computerized Tomographic Imaging*, New York, NY, IEEE Press, 1988.

## See Also

`fan2para` | `fanbeam` | `iradon` | `para2fan` | `phantom` | `radon`



**Introduced before R2006a**

## illumgray

Estimate illuminant using gray world algorithm

### Syntax

```
illuminant = illumgray(A)
illuminant = illumgray(A,percentile)
illuminant = illumgray( ____,Name,Value)
```

### Description

`illuminant = illumgray(A)` estimates the illumination of the scene in RGB image `A` by assuming that the average color of the scene is gray.

`illuminant = illumgray(A,percentile)` estimates the illumination, excluding the specified bottom and top percentiles of pixel values.

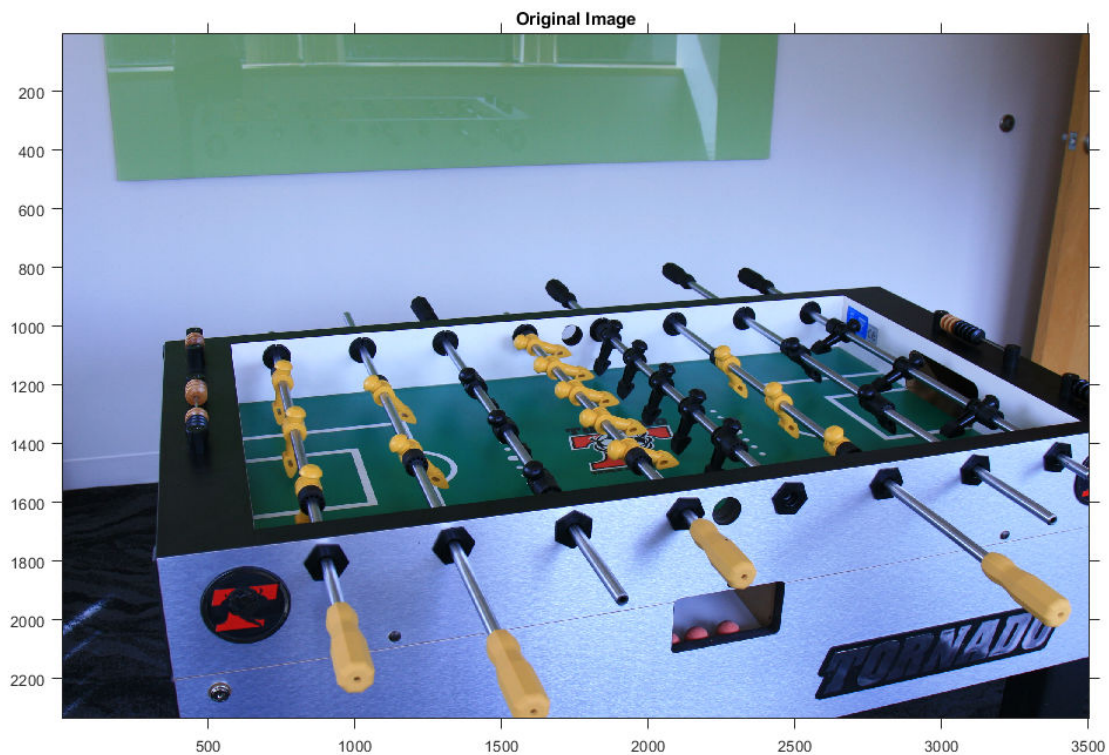
`illuminant = illumgray( ____,Name,Value)` estimates the illumination using name-value pairs to control additional options.

### Examples

#### Correct White Balance Using Gray World Algorithm

Open an image and display it. Specify an optional magnification to shrink the size of the displayed image.

```
A = imread('foosball.jpg');
figure
imshow(A,'InitialMagnification',25)
title('Original Image')
```



The gray world algorithm assumes that the RGB values are linear. However, the JPEG file format saves images in the gamma-corrected sRGB color space. Undo the gamma correction by using the `rgb2lin` function.

```
A_lin = rgb2lin(A);
```

Estimate the scene illumination, excluding the top and bottom 10% of pixels. Because the input image has been linearized, `illumgray` returns the illuminant in the linear RGB color space.

```
percentiles = 10;
illuminant = illumgray(A_lin,percentiles)
```

```
illuminant = 1×3
```

```
0.2206    0.2985    0.5219
```

The third coefficient of `illuminant` is the largest, which is consistent with the blue tint of the image.

Correct colors by providing the estimated illuminant to the `chromadapt` function.

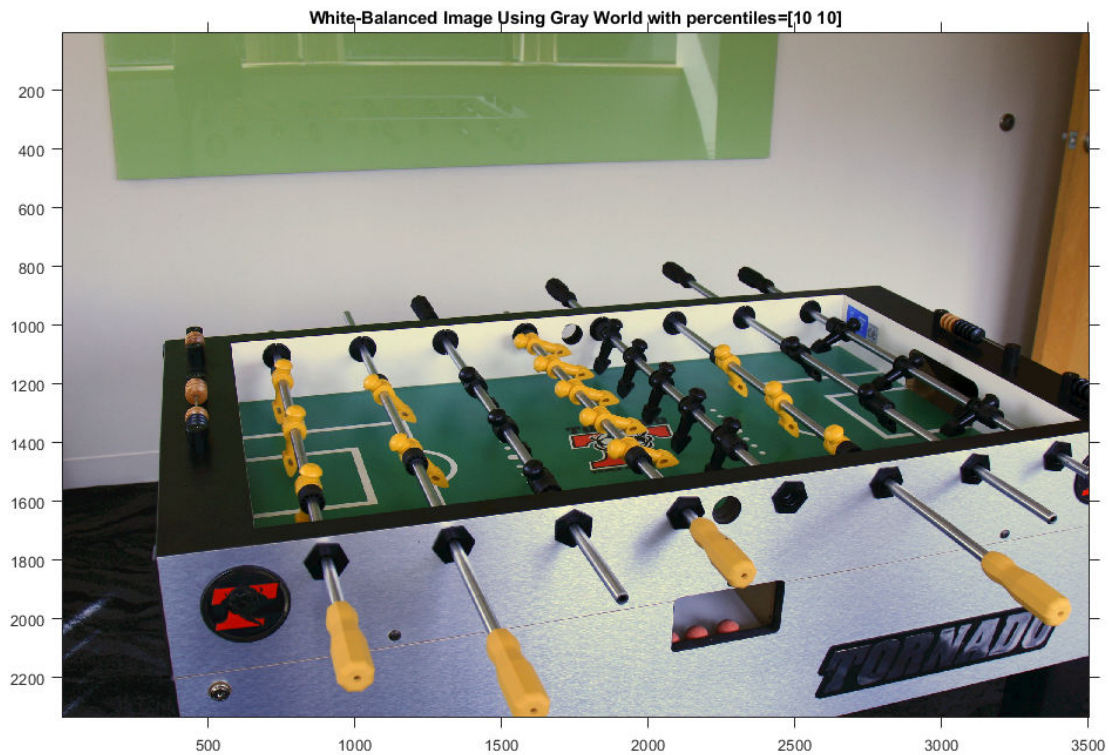
```
B_lin = chromadapt(A_lin,illuminant,'ColorSpace','linear-rgb');
```

To display the white-balanced image correctly on the screen, apply gamma correction by using the `lin2rgb` function.

```
B = lin2rgb(B_lin);
```

Display the corrected image, setting the optional magnification.

```
figure
imshow(B,'InitialMagnification',25)
title(['White-Balanced Image Using Gray World with percentiles=[' ...
      num2str(percentiles) ' ' num2str(percentiles) '']])
```



## Input Arguments

### **A** — Input RGB image

real, nonsparse,  $m$ -by- $n$ -by-3 array

Input RGB image, specified as a real, nonsparse,  $m$ -by- $n$ -by-3 array.

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

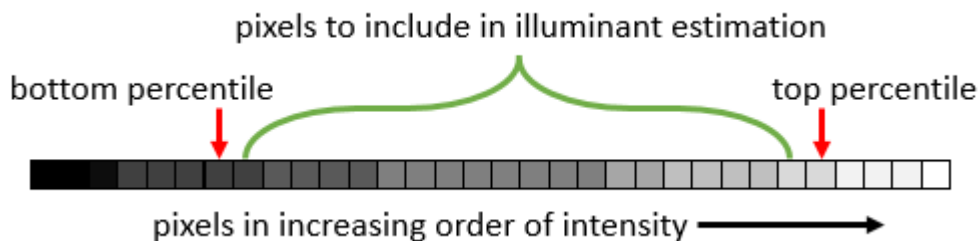
### **percentile** — Percentile of pixels to exclude

1 (default) | numeric scalar | 2-element numeric vector

Percentile of pixels to exclude from the illuminant estimation, specified as a numeric scalar or 2-element numeric vector. Excluding pixels helps prevent overexposed and underexposed pixels from skewing the estimation.

- If `percentile` is a scalar, the same value is used for both the bottom percentile and the top percentile. In this case, `percentile` must be in the range `[0, 50]` so that the sum of the bottom and top percentiles does not exceed 100.
- If `percentile` is a 2-element vector, the first element is the bottom percentile and the second element is the top percentile. Both percentiles must be in the range `[0, 100]` and their sum cannot exceed 100.

The following image indicates the range of pixels that are included in the illuminant estimation. The selection is separate for each color channel.



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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `illuminant = illumgray(I, 'Mask', m)` estimates the scene illuminant using a subset of pixels in image `I`, selected according to a binary mask, `m`.

### Mask — Image mask

*m*-by-*n* logical or numeric array

Image mask, specified as the comma-separated pair consisting of `'Mask'` and an *m*-by-*n* logical or numeric array. The mask indicates which pixels of the input image `A` to use

when estimating the illuminant. The computation excludes pixels in **A** that correspond to a mask value of 0. By default, the mask has all 1s, and all pixels in **A** are included in the estimation.

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

### **Norm — Type of vector norm (p-norm)**

1 (default) | positive numeric scalar

Type of vector norm (p-norm), specified as the comma-separated pair consisting of 'Norm' and a positive numeric scalar. The p-norm affects the calculation of the average RGB value in the input image **A**. The p-norm is defined as  $\text{sum}(\text{abs}(x)^p)^{1/p}$ .

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

## **Output Arguments**

### **illuminant — Estimate of scene illumination**

3-element numeric row vector

Estimate of scene illumination, returned as a 3-element numeric row vector. The three elements correspond to the red, green, and blue values of the illuminant.

Data Types: `double`

## **Tips**

- The gray world algorithm assumes uniform illumination and linear RGB values. If you are working with nonlinear sRGB or Adobe RGB images, use the `rgb2lin` function to undo the gamma correction before using `illumgray`. Also, make sure to convert the chromatically adapted image back to sRGB by using the `lin2rgb` function.
- When you specify `Mask` on page 1-0, the bottom percentile and top percentile apply to the masked image.
- You can adjust the color balance of the image to remove the scene illumination by using the `chromadapt` function.

## References

[1] Ebner, Marc. "The Gray World Assumption." *Color Constancy*. Chichester, West Sussex: John Wiley & Sons, 2007.

## See Also

`chromadapt` | `illumcpa` | `illumwhite` | `lin2rgb` | `rgb2lin`

**Introduced in R2017b**



# illumpca

Estimate illuminant using principal component analysis (PCA)

## Syntax

```
illuminant = illumpca(A)  
illuminant = illumpca(A,percentage)  
illuminant = illumpca( ____,Name,Value)
```

## Description

`illuminant = illumpca(A)` estimates the illumination of the scene in RGB image `A` from large color differences using principal component analysis (PCA).

`illuminant = illumpca(A,percentage)` estimates the illumination using the specified percentage of darkest and brightest pixels.

`illuminant = illumpca( ____,Name,Value)` estimates the illumination using name-value pairs to control additional options.

## Examples

### Correct White Balance Using Principal Component Analysis

Open an image and display it. Specify an optional magnification to shrink the size of the displayed image.

```
A = imread('foosball.jpg');  
figure  
imshow(A,'InitialMagnification',25)  
title('Original Image')
```

Original Image



Principal component analysis assumes that the RGB values are linear. However, the JPEG file format saves images in the gamma-corrected sRGB color space. Undo the gamma correction by using the `rgb2lin` function.

```
A_lin = rgb2lin(A);
```

Estimate the scene illumination from the darkest and brightest 3.5% of pixels (the default percentage). Because the input image is linear, the `illumpca` function returns the illuminant in the linear RGB color space,

```
illuminant = illumpca(A_lin)
```

```
illuminant = 1×3
```

```
    0.4075    0.5547    0.7254
```

The third coefficient of `illuminant` is the largest, which is consistent with the blue tint of the image.

Correct colors by providing the estimated illuminant to the `chromadapt` function.

```
B_lin = chromadapt(A_lin,illuminant,'ColorSpace','linear-rgb');
```

To display the white-balanced image correctly on the screen, apply gamma correction by using the `lin2rgb` function.

```
B = lin2rgb(B_lin);
```

Display the corrected image, setting the optional magnification.

```
figure  
imshow(B,'InitialMagnification',25)  
title('White-Balanced Image using Principal Component Analysis')
```

White-Balanced Image using Principal Component Analysis



## Input Arguments

### **A — Input RGB image**

real, nonsparse,  $m$ -by- $n$ -by-3 array

Input RGB image, specified as a real, nonsparse,  $m$ -by- $n$ -by-3 array.

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

### **percentage — Percentage of pixels to retain**

3.5 (default) | numeric scalar

Percentage of pixels to retain for the illuminant estimation, specified as a numeric scalar in the range (0, 50].

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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `illuminant = illumpca(I, 'Mask', m)` estimates the scene illuminant using a subset of pixels in image `I`, selected according to a binary mask, `m`.

### **Mask — Image mask**

$m$ -by- $n$  logical or numeric array

Image mask, specified as the comma-separated pair consisting of `'Mask'` and an  $m$ -by- $n$  logical or numeric array. The mask indicates which pixels of the input image `A` to use when estimating the illuminant. The computation excludes pixels in `A` that correspond to a mask value of 0. By default, the mask has all 1s, and all pixels in `A` are included in the estimation.

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

## Output Arguments

### **illuminant** — Estimate of scene illumination

3-element numeric row vector

Estimate of scene illumination, returned as a 3-element numeric row vector. The three elements correspond to the red, green, and blue values of the illuminant.

Data Types: `double`

## Tips

- The algorithm assumes uniform illumination and linear RGB values. If you are working with nonlinear sRGB or Adobe RGB images, use the `rgb2lin` function to undo the gamma correction before using `illumpca`. Also, make sure to convert the chromatically adapted image back to sRGB or Adobe RGB by using the `lin2rgb` function.

## Algorithms

Pixel colors are represented as vectors in the RGB color space. The algorithm orders colors according to the brightness, or norm, of their projection on the average color in the image. The algorithm retains only the darkest and brightest colors, according to this ordering. Principal component analysis (PCA) is then performed on the subset of colors. The first component of PCA indicates the illuminant estimate.

## References

- [1] Cheng, Dongliang, Dilip K. Prasad, and Michael S. Brown. "Illuminant Estimation for Color Constancy: Why spatial-domain methods work and the role of the color distribution." *Journal of the Optical Society of America A*. Vol. 31, Number 5, 2014, pp. 1049-1058.

## See Also

`chromadapt` | `illumgray` | `illumwhite` | `lin2rgb` | `rgb2lin`

**Introduced in R2017b**

# illumwhite

Estimate illuminant using White Patch Retinex algorithm

## Syntax

```
illuminant = illumwhite(A)  
illuminant = illumwhite(A,topPercentile)  
illuminant = illumwhite( ____,Name,Value)
```

## Description

`illuminant = illumwhite(A)` estimates the scene illumination in RGB image `A` by assuming that the top 1% brightest red, green, and blue values represent the color white.

`illuminant = illumwhite(A,topPercentile)` estimates the illumination using the `topPercentile` percentage brightest red, green, and blue values.

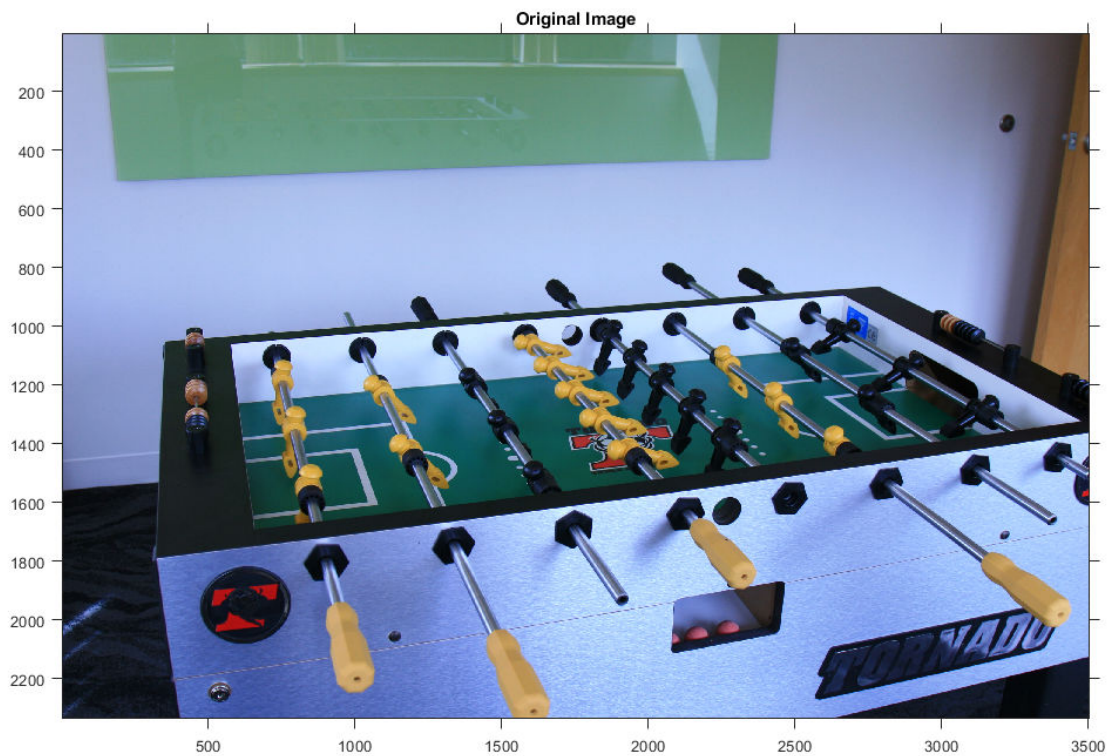
`illuminant = illumwhite( ____,Name,Value)` estimates the illumination using name-value pairs to control additional options.

## Examples

### Correct White Balance Using White Patch Retinex Algorithm

Open an image and display it. Specify an optional magnification to shrink the size of the displayed image.

```
A = imread('foosball.jpg');  
figure  
imshow(A,'InitialMagnification',25)  
title('Original Image')
```



The JPEG file format saves images in the gamma-corrected sRGB color space. Undo the gamma correction by using the `rgb2lin` function.

```
A_lin = rgb2lin(A);
```

Estimate the scene illumination from the top 5% brightest pixels. Because the input image has been linearized, the `illumwhite` function returns the illuminant in the linear RGB color space.

```
topPercentile = 5;  
illuminant = illumwhite(A,topPercentile)
```

```
illuminant = 1×3  
    0.7333    0.8314    1.0000
```



The third coefficient of `illuminant` is the largest, which is consistent with the blue tint of the image.

Correct colors by providing the estimated illuminant to the `chromadapt` function.

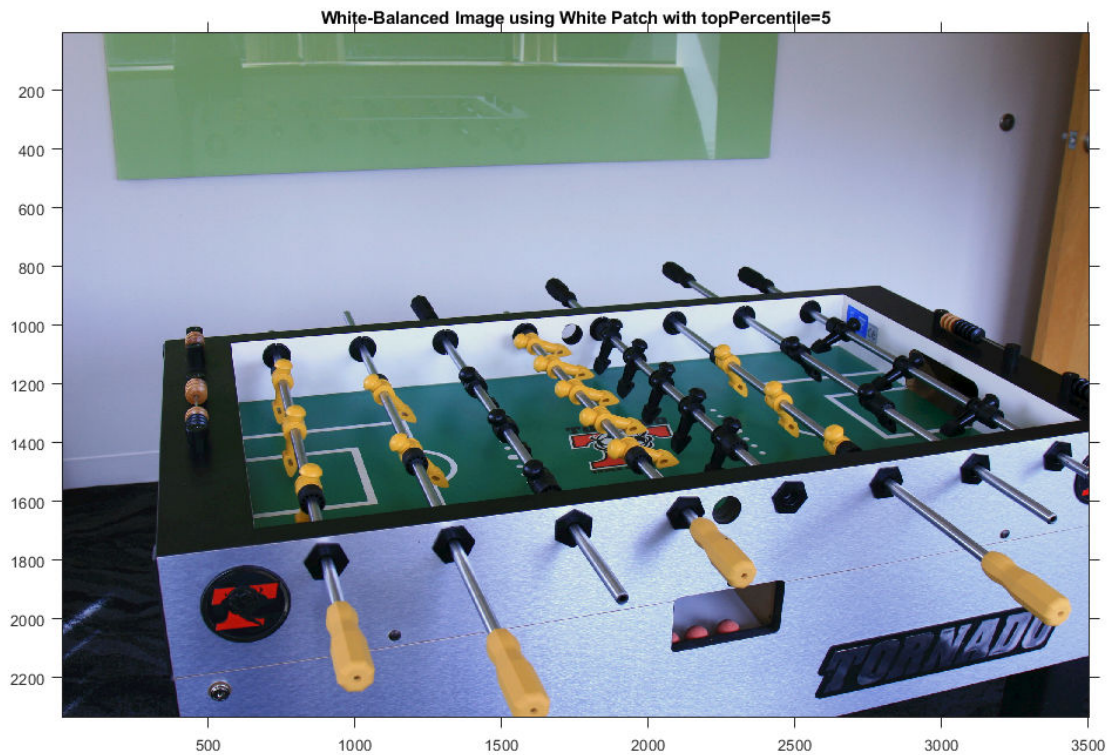
```
B_lin = chromadapt(A_lin,illuminant,'ColorSpace','linear-rgb');
```

To display the white-balanced image correctly on the screen, apply gamma correction by using the `lin2rgb` function.

```
B = lin2rgb(B_lin);
```

Display the corrected image, setting the optional magnification.

```
figure
imshow(B,'InitialMagnification',25)
title(['White-Balanced Image using White Patch with topPercentile=' ...
      num2str(topPercentile)])
```



## Input Arguments

### **A** — Input RGB image

real, nonsparse,  $m$ -by- $n$ -by-3 array

Input RGB image, specified as a real, nonsparse,  $m$ -by- $n$ -by-3 array.

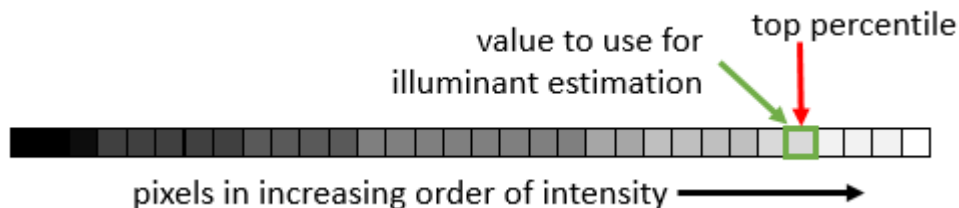
Data Types: `single` | `double` | `uint8` | `uint16`

### **topPercentile** — Percentile of brightest colors

1 (default) | numeric scalar

Percentile of brightest colors to use for illuminant estimation, specified as a numeric scalar in the range [0, 100). To return the maximum red, green, and blue values, set `topPercentile` to 0.

The image indicates the red, green, and blue value that is selected to estimate the illuminant. The selection is separate for each color channel.



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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `illuminant = illumwhite(I, 'Mask', m)` estimates the scene illuminant using a subset of pixels in image `I`, selected according to a binary mask, `m`.

### Mask — Image mask

*m*-by-*n* logical or numeric array

Image mask, specified as the comma-separated pair consisting of `'Mask'` and an *m*-by-*n* logical or numeric array. The mask indicates which pixels of the input image `A` to use when estimating the illuminant. The computation excludes pixels in `A` that correspond to a mask value of 0. By default, the mask has all 1s, and all pixels in `A` are included in the estimation.

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

## Output Arguments

### **illuminant** — Estimate of scene illumination

3-element numeric row vector

Estimate of scene illumination, returned as a 3-element numeric row vector. The three elements correspond to the red, green, and blue values of the illuminant.

Data Types: `double`

## References

[1] Ebner, Marc. "White Patch Retinex." *Color Constancy*. Chichester, West Sussex: John Wiley & Sons, 2007.

## See Also

`chromadapt` | `illumgray` | `illumpca` | `lin2rgb` | `rgb2lin` | `whitepoint`

**Introduced in R2017b**

# im2bw

Convert image to binary image, based on threshold

---

**Note** `im2bw` is not recommended. Use `imbinarize` instead.

---

## Syntax

```
BW = im2bw(I,level)
BW = im2bw(X,cmap,level)
BW = im2bw(RGB,level)
```

## Description

`BW = im2bw(I,level)` converts the grayscale image `I` to binary image `BW`, by replacing all pixels in the input image with luminance greater than `level` with the value 1 (white) and replacing all other pixels with the value 0 (black).

This range is relative to the signal levels possible for the image's class. Therefore, a `level` value of 0.5 corresponds to an intensity value halfway between the minimum and maximum value of the class.

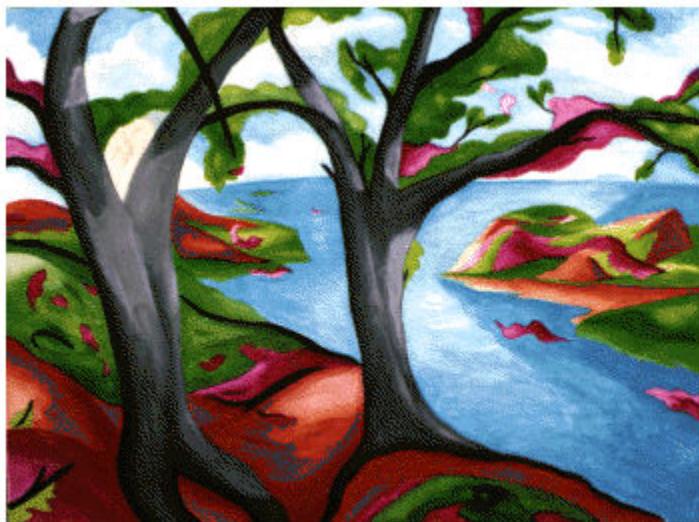
`BW = im2bw(X,cmap,level)` converts the indexed image `X` with colormap `cmap` to a binary image.

`BW = im2bw(RGB,level)` converts the truecolor image `RGB` to a binary image.

## Examples

### Convert an Indexed Image To a Binary Image

```
load trees
BW = im2bw(X,map,0.4);
imshow(X,map), figure, imshow(BW)
```





## Input Arguments

### **I** — 2-D grayscale image

*m*-by-*n* numeric matrix

2-D grayscale image, specified as an *m*-by-*n* numeric matrix.

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

### **X** — 2-D indexed image

*m*-by-*n* numeric matrix

2-D indexed image, specified as an *m*-by-*n* numeric matrix.

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

## **cmap — Color map**

*c*-by-3 numeric matrix

Color map associated with indexed image *X*, specified as a *c*-by-3 numeric matrix containing the RGB values of *c* colors.

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

## **RGB — 2-D RGB image**

*m*-by-*n*-by-3 numeric matrix

2-D RGB image, specified as an *m*-by-*n*-by-3 numeric matrix.

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

## **level — Luminance threshold**

0.5 (default) | number in the range [0, 1]

Luminance threshold, specified as a number in the range [0, 1]. To compute `level`, you can use the `graythresh` function.

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

## **Output Arguments**

### **BW — Binary image**

*m*-by-*n* logical matrix

Binary image, returned as an *m*-by-*n* logical matrix.

Data Types: `logical`

## **Algorithms**

If the input image is not a grayscale image, `im2bw` converts the input image to grayscale using `ind2gray` or `rgb2gray`, and then converts this grayscale image to binary by thresholding.

## **See Also**

`graythresh` | `ind2gray` | `rgb2gray`



**Introduced before R2006a**

## im2col

Rearrange image blocks into columns

### Syntax

```
B = im2col(A,[m n],'distinct')
B = im2col(A,[m n],'sliding')
B = im2col(A,[m n])
B = im2col(A,'indexed', ___)
```

### Description

`B = im2col(A,[m n],'distinct')` rearranges discrete image blocks of size `m`-by-`n` into columns, and returns the concatenated columns in matrix `B`. The `im2col` function pads image `A`, if necessary. For more information about the padding value, see “Tips” on page 1-950.

The order of the columns in matrix `B` is determined by traversing the image `A` in a column-wise manner. For example, if `A` consists of distinct blocks  $A_{ij}$  arranged as `A = [A11 A12; A21 A22]`, then `B = [A11(:) A21(:) A12(:) A22(:)]`.

`B = im2col(A,[m n],'sliding')` or

`B = im2col(A,[m n])` rearranges sliding image neighborhoods of size `m`-by-`n` into columns with no zero-padding, and returns the concatenated columns in matrix `B`.

`B = im2col(A,'indexed', ___)` interprets `A` as an indexed image.

### Examples

#### Calculate Local Mean Using [2 2] Neighborhood

Create a matrix.

```
A = reshape(linspace(0,1,16),[4 4])'
```

```
A = 4x4
```

```

    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 = 4x9
```

```

    0    0.2667    0.5333    0.0667    0.3333    0.6000    0.1333    0.4000    0.6667
    0.2667    0.5333    0.8000    0.3333    0.6000    0.8667    0.4000    0.6667    0.9333
    0.0667    0.3333    0.6000    0.1333    0.4000    0.6667    0.2000    0.4667    0.7333
    0.3333    0.6000    0.8667    0.4000    0.6667    0.9333    0.4667    0.7333    1.0000

```

Calculate the mean.

```
M = mean(B)
```

```
M = 1x9
```

```

    0.1667    0.4333    0.7000    0.2333    0.5000    0.7667    0.3000    0.5667    0.8333

```

Rearrange the values back into their original, row-wise orientation.

```
newA = col2im(M,[1 1],[3 3])
```

```
newA = 3x3
```

```

    0.1667    0.2333    0.3000
    0.4333    0.5000    0.5667
    0.7000    0.7667    0.8333

```

## Input Arguments

### **A — Image**

2-D grayscale image | 2-D binary image | 2-D indexed image

Image, specified as a 2-D grayscale image, 2-D binary image, or 2-D indexed image.

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

### **[m n] — Block size**

2-element vector

Block size, specified as a 2-element vector. `m` is the number of rows and `n` is the number of columns in the block.

## Output Arguments

### **B — Image blocks**

numeric matrix | logical matrix

Image blocks, returned as a numeric matrix or logical matrix with `m*n` rows. The number of columns depends on whether the image blocks are discrete blocks or sliding neighborhoods. Each column of `B` contains a block or neighborhood of `A` reshaped as a column vector.

- For distinct block processing, `B` has as many columns as there are `m`-by-`n` blocks in `A`. For example, if the size of `A` is `[mm nn]`, then `B` has  $(mm/m) * (nn/n)$  columns.
- For sliding neighborhood processing, `B` has as many columns as there are `m`-by-`n` neighborhoods of `A`. For example, if the size of `A` is `[mm nn]`, then `B` has  $((mm - m + 1) * (nn - n + 1))$  columns.

## Tips

- For distinct block processing, `im2col` zero-pads `A`, if necessary, so its size is an integer multiple of `m`-by-`n`. The padding value is `0` when `A` is data type `uint8`, `uint16`, or `logical`. For other data types, the value of padding depends on whether `A` is interpreted as an indexed image.

- The padding value is 1 when A is interpreted as an indexed image.
- The padding value is 0 when A is not interpreted as an indexed image.
- `im2col` orders the columns of B so that they can be reshaped to form a matrix according to `reshape`.

For example, 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);
```

## See Also

`blockproc` | `col2im` | `colfilt` | `nlfilter` | `reshape`

**Introduced before R2006a**

## im2int16

Convert image to 16-bit signed integers

### Syntax

```
J = im2int16(I)
```

### Description

`J = im2int16(I)` converts the grayscale, RGB, or binary image `I` to `int16`, rescaling the data if necessary.

If the input image is of class `int16`, then the output image is identical to it. If the input image is of class `logical`, then `im2int16` changes false-valued elements to -32768 and true-valued elements to 32767.

You optionally can perform the conversion using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

### Examples

#### Convert Array from double to int16

Create an array of class `double`.

```
I = reshape(linspace(0,1,20),[5 4])
```

```
I = 5×4
```

```
      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 = 5x4 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 int16.

```
I2 = im2int16(I1)
```

## Input Arguments

### I — Image

numeric array | logical array | gpuArray

Intensity, RGB, or binary image, specified as a numeric array of any size and dimensionality.

To perform the conversion using a GPU, specify I as a `gpuArray` that contains a numeric array of any size and dimensionality.

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

## Output Arguments

### **J** — Image with class `int16`

numeric array | `gpuArray`

Image with class `int16`, returned as a numeric array with the same size and dimensionality as `I`.

If the conversion is performed using a GPU, then `J` is returned as a `gpuArray` containing a numeric array.

Data Types: `int16`

## 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 “Understand Code Generation with Image Processing Toolbox”.

### **GPU Arrays**

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.



## See Also

`gpuArray` | `im2double` | `im2single` | `im2uint16` | `im2uint8` | `int16`

**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

```
J = im2single(I)
J = im2single(I,'indexed')
```

### Description

`J = im2single(I)` converts the grayscale, RGB, or binary image `I` to `single`, rescaling or offsetting the data as necessary.

If the input image is of class `single`, then the output image is identical. If the input image is of class `logical`, then `im2single` changes true-valued elements to 65535.

You optionally can perform the conversion using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`J = im2single(I, 'indexed')` converts the indexed image `I` to `single`, offsetting the data if necessary.

### 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 = 5x5 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 = 5x5 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

## Input Arguments

### I — Input image

numeric array | logical array | gpuArray

Input image, specified as a numeric or logical array of any size and dimension.

- If I is a grayscale or RGB image, then it can be `uint8`, `uint16`, `double`, `logical`, `single`, or `int16`.
- If I is an indexed image, then it can be `uint8`, `uint16`, `double` or `logical`.
- If I is a binary image, then it must be `logical`.

To perform the conversion using a GPU, specify I as a `gpuArray` that contains a numeric or logical array of any size and dimensionality.

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

## Output Arguments

### J — Image with class `single`

numeric array | gpuArray

Image with class `single`, returned as a numeric array with the same size and dimensionality as I.

If the conversion is performed using a GPU, then J is returned as a `gpuArray` containing a numeric array.

Data Types: `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”.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

### See Also

`im2double` | `im2int16` | `im2uint16` | `im2uint8` | `single`

**Introduced before R2006a**

## im2uint16

Convert image to 16-bit unsigned integers

### Syntax

```
J = im2uint16(I)
J = im2uint16(I,'indexed')
```

### Description

`J = im2uint16(I)` converts the grayscale, RGB, or binary image `I` to `uint16`, rescaling or offsetting the data as necessary.

If the input image is of class `uint16`, then the output image is identical. If the input image is of class `logical`, then `im2uint16` changes true-valued elements to 65535.

You optionally can perform the conversion using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`J = im2uint16(I,'indexed')` converts the indexed image `I` to `uint16`, offsetting the data if necessary.

### Examples

#### Convert Array from double to uint16

Create an array of class `double`.

```
I = reshape(linspace(0,1,20),[5 4])
```

```
I = 5×4
```

```
         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 = 5x4 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)
```

## Input Arguments

### I — Input image

numeric array | logical array | gpuArray

Input image, specified as a numeric or logical array of any size and dimension.

- If I is a grayscale or RGB image, then it can be uint8, uint16, double, logical, single, or int16.
- If I is an indexed image, then it can be uint8, uint16, double or logical.

**Note** It is not always possible to convert an indexed image to `uint8`. If the indexed image is of class `double`, then the maximum value must be 65536 or less.

- If `I` is a binary image, then it must be `logical`.

To perform the conversion using a GPU, specify `I` as a `gpuArray` that contains a numeric or logical array of any size and dimensionality.

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

## Output Arguments

### J — Image with class `uint16`

numeric array | `gpuArray`

Image with class `uint16`, returned as a numeric array with the same size and dimensionality as `I`.

If the conversion is performed using a GPU, then `J` is returned as a `gpuArray` containing a numeric array.

Data Types: `uint16`

## 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 “Understand Code Generation with Image Processing Toolbox”.

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

## See Also

[double](#) | [im2double](#) | [im2uint8](#) | [imapprox](#) | [uint16](#) | [uint8](#)

**Introduced before R2006a**

## im2uint8

Convert image to 8-bit unsigned integers

### Syntax

```
J = im2uint8(I)
J = im2uint8(I,'indexed')
```

### Description

`J = im2uint8(I)` converts the grayscale, RGB, or binary image `I` to `uint8`, rescaling or offsetting the data as necessary.

If the input image is of class `uint8`, then the output image is identical. If the input image is of class `logical`, then `im2uint8` changes true-valued elements to 255.

You optionally can perform the conversion using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`J = im2uint8(I, 'indexed')` converts the indexed image `I` to `uint8`, offsetting the data if necessary.

### Examples

#### Convert uint16 Array to uint8 Array

Create an array of class `uint16`.

```
I = reshape(uint16(linspace(0,65535,25)),[5 5])
```

*I = 5x5 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 = 5x5 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 image

numeric array | logical array | `gpuArray`

Input image, specified as a numeric or logical array of any size and dimension.

- If `I` is a grayscale or RGB image, then it can be `uint8`, `uint16`, `double`, `logical`, `single`, or `int16`.
- If `I` is an indexed image, then it can be `uint8`, `uint16`, `double` or `logical`.

**Note** It is not always possible to convert an indexed image to `uint8`. If the indexed image is of class `double`, then the maximum value must be 256 or less. If the indexed image is of class `uint16`, then the maximum value must be 255 or less.

- If `I` is a binary image, then it must be `logical`.

To perform the conversion using a GPU, specify `I` as a `gpuArray` that contains a numeric or logical array of any size and dimensionality.

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

## Output Arguments

### J — Image with class `uint8`

numeric array | `gpuArray`

Image with class `uint8`, returned as a numeric array with the same size and dimensionality as `I`.

If the conversion is performed using a GPU, then `J` is returned as a `gpuArray` containing a numeric array.

Data Types: `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. 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 “Understand Code Generation with Image Processing Toolbox”.

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

## See Also

[im2double](#) | [im2int16](#) | [im2single](#) | [im2uint16](#) | [uint8](#)

**Introduced before R2006a**

## imabsdiff

Absolute difference of two images

### Syntax

```
Z = imabsdiff(X,Y)
```

### Description

`Z = imabsdiff(X,Y)` subtracts each element in array `Y` from the corresponding element in array `X` and returns the absolute difference in the corresponding element of the output array `Z`.

You optionally can compute the absolute difference of `X` and `Y` using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

### 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 Images 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 = imgaussfilt(I);
```

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 | gpuArray

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

To compute the absolute difference using a GPU, specify X as a `gpuArray` that contains a numeric array.

```
Example: x = imread('cameraman.tif');
```

```
Example: X = gpuArray(imread('cameraman.tif'));
```

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

### **Y — Input image**

numeric array | gpuArray

Input image, specified as a numeric array. Y must be the same size and class as X.

To compute the absolute difference using a GPU, specify Y as a `gpuArray` that contains a numeric array of the same size and class as the array in X.

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

## Output Arguments

### **Z — Difference image**

numeric array | gpuArray

Difference image, returned as a numeric array. Z has the same class and size as X and Y. If X and Y are integer arrays, then `imabsdiff` truncates elements in the output that exceed the range of the integer type.

If the difference image is computed using a GPU, then  $Z$  is returned as a `gpuArray` containing a numeric array of the same size and class as in  $X$  and  $Y$ .

## Tips

- If  $X$  is of class `double`, then use the expression `abs(X-Y)` instead of this function.
- If  $X$  is of class `logical`, then use the expression `XOR(X,Y)` instead of this function.
- When  $X$  and  $Y$  are of class `uint8`, `int16`, or `single`, then `imabsdiff` can use hardware optimization to run faster.
- `imabsdiff` runs on a GPU if at least one of  $X$  and  $Y$  are `gpuArrays`. It is not necessary that both inputs are `gpuArrays`.

## 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”.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

### See Also

`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`.

If `X` is an integer array, 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 = 2x3 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.

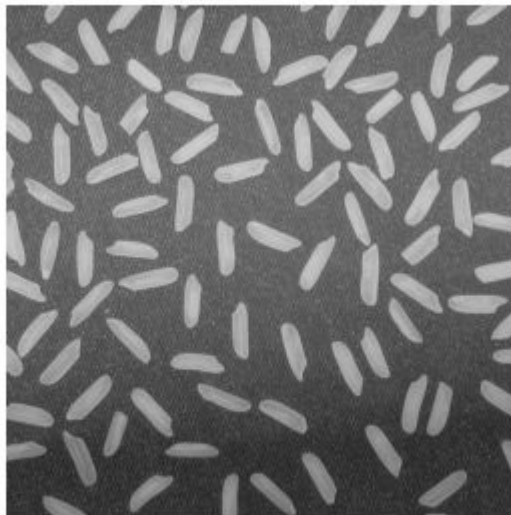
```
I = imread('rice.png');
```

Add a constant to the image.

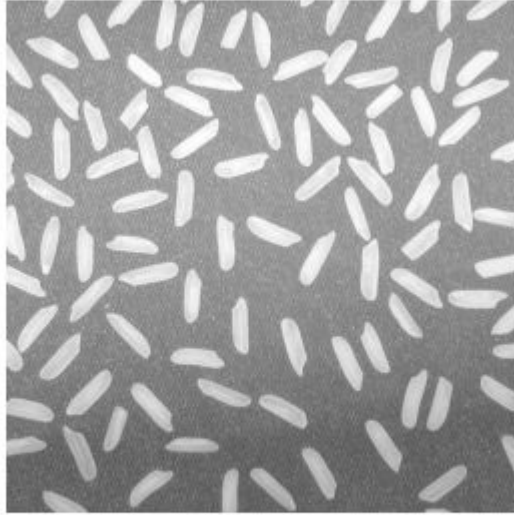
```
J = imadd(I,50);
```

Display the original image and the result.

```
imshow(I)
```



```
figure  
imshow(J)
```



## Input Arguments

### **X — First array**

real, nonsparse, numeric or logical array

First array, specified as a real, nonsparse, numeric or logical array of any dimension.

### **Y — Second array**

real, nonsparse, numeric or logical array

Second array to be added to *X*, specified as a real, nonsparse, numeric or logical array. *Y* either has the same size and class as *X*, or *Y* is a scalar of type double.



## Output Arguments

### **Z – Sum**

numeric array

Sum, returned as a numeric array of the same size as X. Z is the same class as X unless X is logical, in which case Z is data type **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.

## 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])
J = imadjust(I,[low_in high_in],[low_out high_out])
J = imadjust(I,[low_in high_in],[low_out high_out],gamma)

J = imadjust(RGB,[low_in high_in],___)
newmap = imadjust(cmap,[low_in high_in],___)
```

### Description

`J = imadjust(I)` maps the intensity values in grayscale image `I` to new values in `J`. By default, `imadjust` saturates the bottom 1% and the top 1% of all pixel values. This operation increases the contrast of the output image `J`.

You optionally can perform contrast adjustment using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

This syntax is equivalent to `imadjust(I,stretchlim(I))`.

`J = imadjust(I,[low_in high_in])` maps intensity values in `I` to new values in `J` such that values between `low_in` and `high_in` map to values between 0 and 1.

`J = imadjust(I,[low_in high_in],[low_out high_out])` maps intensity 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`.

`J = imadjust(I,[low_in high_in],[low_out high_out],gamma)` maps intensity 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`.

`J = imadjust(RGB,[low_in high_in], ___)` maps the values in truecolor image RGB to new values in J. You can apply the same mapping or unique mappings for each color channel.

`newmap = imadjust(cmap,[low_in high_in], ___)` maps the values in colormap cmap to new values in newmap. You can apply the same mapping or unique mappings for each color channel.

## 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`.

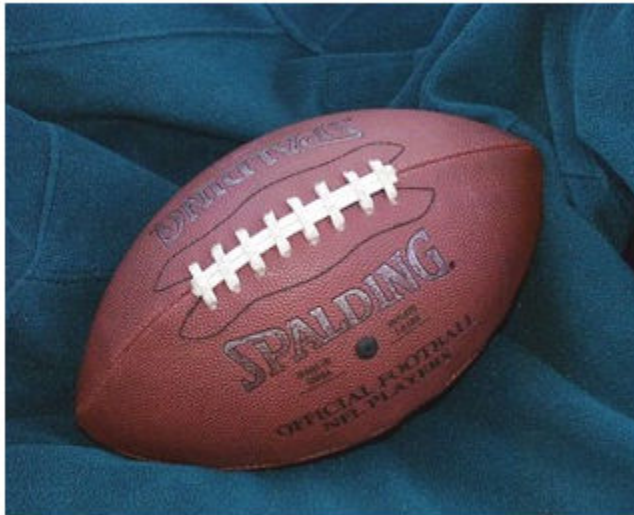
```
I = gpuArray(imread('pout.tif'));  
figure  
imshow(I)
```

```
K = imadjust(I,[0.3 0.7],[0 1]);  
figure  
imshow(K)
```

### **Adjust Contrast of Color 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],[,]);  
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. Each color channel has different contrast limits.

```
RGB = gpuArray(imread('football.jpg'));  
RGB2 = imadjust(RGB,[.2 .3 0; .6 .7 1],[[]]);  
figure  
imshow(RGB)  
figure  
imshow(RGB2)
```

## Standard Deviation Based Image Stretching

Read an image into the workspace, and display it.

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



Calculate the standard deviation and the image mean for stretching.

```
n = 2;  
Idouble = im2double(I);  
avg = mean2(Idouble);  
sigma = std2(Idouble);
```

Adjust the contrast based on the standard deviation.

```
J = imadjust(I, [avg-n*sigma avg+n*sigma], []);
```

Display the adjusted image.

```
imshow(J)
```



## Input Arguments

### **I** — Grayscale image

*m*-by-*n* numeric matrix | gpuArray

Grayscale image, specified as an *m*-by-*n* numeric matrix.

To perform contrast adjustment using a GPU, specify **I** as a `gpuArray` that contains an *m*-by-*n* numeric matrix.

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

## **RGB — Truecolor image**

*m*-by-*n*-by-3 numeric array | `gpuArray`

Truecolor image, specified as an *m*-by-*n*-by-3 numeric array.

To perform contrast adjustment using a GPU, specify `RGB` as a `gpuArray` that contains an *m*-by-*n*-by-3 numeric array.

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

## **cmap — Colormap**

*c*-by-3 numeric matrix | `gpuArray`

Colormap, specified as a *c*-by-3 numeric matrix representing *c* colors.

To perform contrast adjustment using a GPU, specify `cmap` as a `gpuArray` that contains a *c*-by-3 numeric matrix.

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

## **[low\_in high\_in] — Contrast limits for input image**

`[0 1]` (default) | 2-element numeric vector | 2-by-3 numeric matrix

Contrast limits for input image, specified in one of the following forms:

## Contrast Limits

Input Type	Value	Description
grayscale	1-by-2 vector of the form [low_in high_in]	Specifies the contrast limits in the input grayscale image that you want to map to values in the output image. Values must be in the range [0 1.0]. The value low_in must be less than the value high_in.
RGB or colormap	2-by-3 matrix of the form [low_RGB_triplet ; high_RGB_triplet]	Specifies the contrast limits in the input RGB image or colormap that you want to map to values in the output image or colormap. Each row in the array is an RGB color triplet. Values must be in the range [0 1]. The value low_RGB_triplet must be less than the value high_RGB_triplet.
RGB or colormap	1-by-2 vector of the form [low_in high_in]	Specifies the contrast limits in the input RGB image that you want to map to values in the output image. Each value must be in the range [0 1.0]. The value low_in must be less than the value high_in. If you specify a 1-by-2 vector with an RGB image or colormap, then imadjust applies the same adjustment to each color plane or channel.
all types	[]	If you specify an empty matrix ([]), then imadjust uses the default limits [0 1].

imadjust clips value below low\_in and above high\_in: Values below low\_in map to low\_out and values above high\_in map to high\_out.

Data Types: single | double

### [low\_out high\_out] — Contrast limits for the output image

[0 1] (default) | 2-element numeric vector | 2-by-3 numeric matrix

Contrast limits for output image, specified in one of the following forms:

### Contrast Limits

Input Type	Value	Description
grayscale	1-by-2 vector of the form <code>[low_out high_out]</code>	Specifies the contrast limits of the output grayscale image. Each value must be in the range <code>[0 1]</code> .
RGB or colormap	2-by-3 matrix of the form <code>[low_RGB_triplet ; high_RGB_triplet]</code>	Specifies the contrast limits of the output RGB image or colormap. Each row in the array is an RGB color triplet. Values must be in the range <code>[0 1]</code> .
RGB or colormap	1-by-2 vector of the form <code>[low_out high_out]</code>	Specifies the contrast limits in the output image. Each value must be in the range <code>[0 1]</code> . If you specify a 1-by-2 vector with an RGB image or colormap, then <code>imadjust</code> applies the same adjustment to each plane or channel.
all types	<code>[]</code>	If you specify an empty matrix ( <code>[]</code> ), then <code>imadjust</code> uses the default limits <code>[0 1]</code> .

If `high_out` is less than `low_out`, then `imadjust` reverses the output image, as in a photographic negative.

Data Types: `single` | `double`

### **gamma** — Shape of the curve describing relationship of input and output values

1 (default) | nonnegative scalar | 1-by-3 numeric vector

Shape of curve describing relationship of input and output values, specified as a nonnegative scalar or a 1-by-3 numeric vector.

- If `gamma` is less than 1, then `imadjust` weights the mapping toward higher (brighter) output values.
- If `gamma` is greater than 1, then `imadjust` weights the mapping toward lower (darker) output values.
- If `gamma` is a 1-by-3 vector, then `imadjust` applies a unique gamma to each color component or channel.
- If you omit the argument, then `gamma` defaults to 1 (linear mapping).

Data Types: double

## Output Arguments

### **J** — Adjusted image

grayscale image | RGB image | gpuArray

Adjusted image, returned as a grayscale or RGB image. J has the same size and class as the input grayscale image I or truecolor image RGB.

If contrast adjustment is performed using a GPU, then J is returned as a gpuArray containing a grayscale or RGB image.

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

### **newmap** — Adjusted colormap

c-by-3 numeric matrix | gpuArray

Adjusted colormap, returned as an c-by-3 numeric matrix of the same class as the input colormap, map.

If contrast adjustment is performed using a GPU, then newmap is returned as a gpuArray containing a c-by-3 numeric matrix.

Data Types: single | 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. 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 “Understand Code Generation with Image Processing Toolbox”.

- When generating code, `imadjust` does not support indexed images.

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

## See Also

`brighten` | `histeq` | `stretchlim`

**Introduced before R2006a**



# imadjustn

Adjust intensity values in *N*-D volumetric image

## Syntax

```
J = imadjustn(V)
J = imadjustn(V,[low_in high_in],[low_out high_out])
J = imadjustn(V,[low_in high_in],[low_out high_out],gamma)
```

## Description

`J = imadjustn(V)` maps the values in the *N*-D volumetric intensity image `V` to new values in `J`. `imadjustn` increases the contrast of the output volumetric image `J`.

By default, `imadjustn` saturates the bottom 1% and the top 1% of all pixel values. This syntax is equivalent to `imadjustn(V,stretchlim(V(:)))`.

`J = imadjustn(V,[low_in high_in],[low_out high_out])` maps the values in `V` to new values in `J` such that values between `low_in` and `high_in` map to values between `low_out` and `high_out`. `imadjustn` clips values below `low_in` and above `high_in`. Values below `low_in` map to `low_out` and values above `high_in` map to `high_out`. If you omit the `[low_out high_out]` argument, in which case, `imadjustn` uses the default `[0 1]`.

`J = imadjustn(V,[low_in high_in],[low_out high_out],gamma)` maps the values of `V` to new values in `J`, where `gamma` specifies the shape of the curve describing the relationship between the values in `V` and `J`. If `gamma` is less than 1, `imadjustn` weights the mapping toward higher (brighter) output values. If `gamma` is greater than 1, `imadjustn` weights the mapping toward lower (darker) output values.

If `high_out` is less than `low_out`, `imadjustn` reverses the output image volume, as in a photographic negative.

## Examples

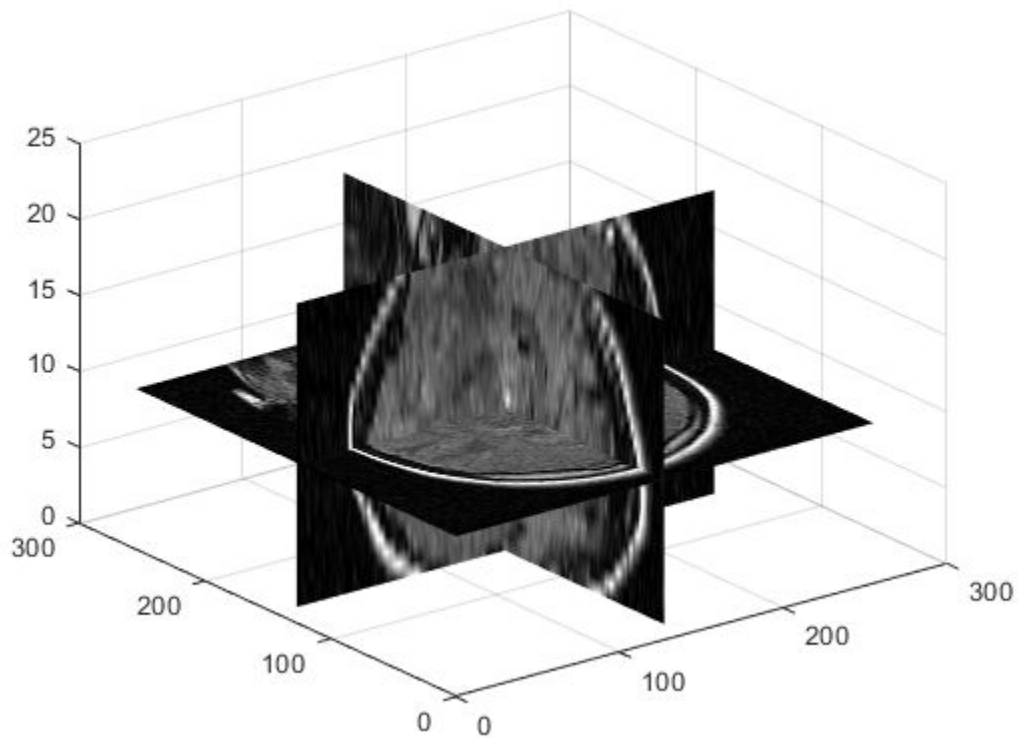
### Scale Intensity of 3-D Volume of MRI Data

Load a 3-D image into the workspace, then save the image as data type double.

```
load mrystack;  
V1 = im2double(mrystack);
```

Display cross-sections of the image.

```
figure  
slice(V1,size(V1,2)/2,size(V1,1)/2,size(V1,3)/2)  
colormap gray  
shading interp
```

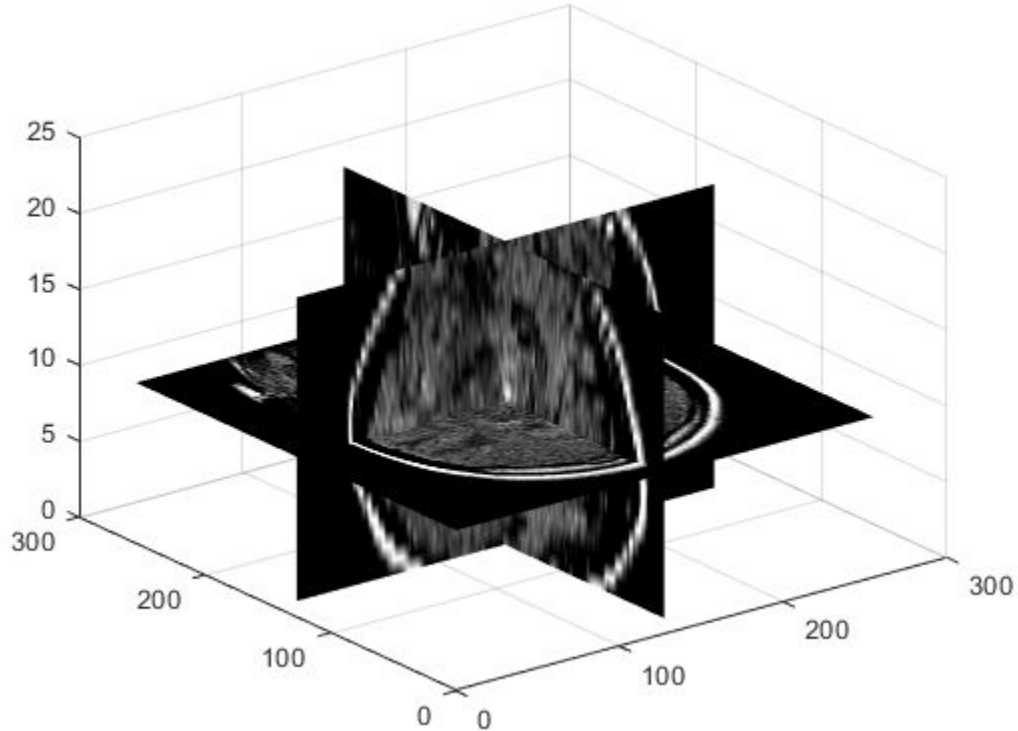


Adjust the image intensity values. `imadjustn` maps input values between 0.2 and 0.8 to the default output range of [0, 1]. `imadjustn` clips input values below 0.2 and above 0.8.

```
V2 = imadjustn(V1,[0.2 0.8],[1]);
```

Display cross-sections of the contrast-adjusted image.

```
figure  
slice(V2,size(V2,2)/2,size(V2,1)/2,size(V2,3)/2)  
colormap gray  
shading interp
```



## Input Arguments

### **V — Volumetric intensity image**

real, nonsparse, *N*-D, numeric array

Volumetric intensity image, specified as a real, nonsparse, *N*-D, numeric array.

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

### **[low\_in high\_in] — Range of values in input image**

[0 1] (default) | 2-element vector

Range of values in the input image, specified as a 2-element vector of the form [low\_in high\_in], with values in the range [0, 1]. Before adjusting intensity values, `imadjustn` converts the input image to class `double` (using `im2double`), rescaling values to the range [0,1]. `low_in` and `high_in` correspond to the specified input range after conversion to `double`.

You can use an empty matrix ([]) for [low\_in high\_in] to specify the default of [0 1].

Data Types: `double`

### **[low\_out high\_out] — Range of values in output image**

[0 1] (default) | 2-element vector

Range of values in output image, specified as a 2-element vector of the form [low\_out high\_out], with values in the range [0, 1]. Before adjusting intensity values, `imadjustn` converts the input image to class `double` (using `im2double`), rescaling values to the range [0,1]. `low_out` and `high_out` correspond to the specified output range after conversion to `double`. After adjusting intensity values, `imadjustn` converts the image to the data type of the input image.

You can omit the argument or use an empty matrix ([]) for [low\_out high\_out] to specify the default of [0 1].

Data Types: `double`

### **gamma — Shape of curve describing relationship between values in V and J**

1 (default) | numeric scalar

Shape of curve describing relationship between values in *V* and *J*, specified as a numeric scalar. If the value is less than 1, `imadjustn` weights the mapping toward higher

(brighter) output values. If the value is greater than 1, `imadjustn` weights the mapping toward lower (darker) output values. If you omit the argument, `gamma` defaults to 1 (linear mapping).

Data Types: `double`

## Output Arguments

### **J** – Volume with adjusted intensity values

*N*-D volumetric intensity image

Volume with adjusted intensity values, returned as an *N*-D volumetric intensity image. The output volume has the same class as the input image.

## See Also

`decorrstretch` | `histeq` | `imhistmatchn` | `stretchlim`

**Introduced in R2017b**

## 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)

“Perform Block Processing on Image Files in Unsupported Formats”

## **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)
```

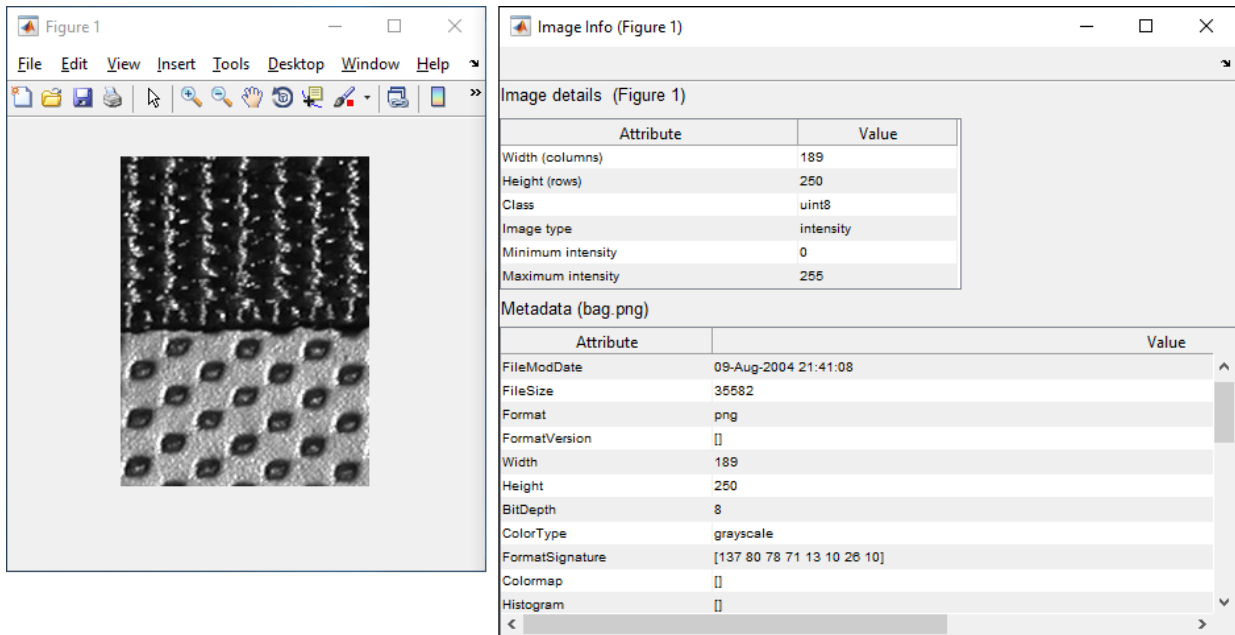
### Description

`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

Use the `imageinfo` function to create an Image Information tool. The tool displays information about the basic attributes and metadata of the target image in a separate figure.



## Syntax

```
imageinfo
imageinfo(h)
imageinfo(filename)
imageinfo(info)
imageinfo(himage, filename)
imageinfo(himage, info)
htool = 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(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.

`imageinfo(info)` creates an Image Information tool containing the image metadata in the structure `info`.

`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`.

`htool = imageinfo( ___ )` returns a handle to the Image Information tool figure.

## Examples

### Open Image Information Tool

```
imageinfo('peppers.png')  
  
h = imshow('bag.png');  
info = imfinfo('bag.png');  
imageinfo(h,info);
```

```
imshow('canoe.tif');  
imageinfo;
```

## Input Arguments

### **h** — Handle to graphics object

handle

Handle to a figure, axes, uipanel, or image graphics object, specified as a handle. If `h` is an axes or figure handle, `imageinfo` uses the first image returned by `findobj(h, 'Type', 'image')`.

### **filename** — File name

character vector | string scalar

File name, specified as a character vector. `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.

### **info** — Image metadata

structure

Image metadata, specified as a structure returned by the functions `imfinfo`, `dicominfo`, `nitfinfo`, `interfileinfo`, or `analyze75info`. `info` can also be a user-created structure.

### **himage** — Handle to image object

handle

Handle to an image graphics object, specified as a handle.

## Output Arguments

### **htool** — Handle to Image Information tool figure

handle

Handle to Image Information tool figure, returned as a handle.

## Tips

- The table lists the basic image attribute 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 'uint8'.  <b>Note</b> For single or int16 images, <code>imageinfo</code> returns a 'Class' value of 'double', because the image object converts the <code>CData</code> of these images to double.
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.  This field is 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.  This field is not included for 'binary' or 'truecolor' images.

- `imageinfo` gets information about image attributes by querying the image object's `CData`. The image object converts the `CData` for `single` or `int16` images to class `double`. In these cases, `imageinfo(H)` displays a 'Class' attribute of 'double', even though the image is of class `single` or `int16`. For example,

```
h = imshow(ones(10,'int16'));  
class(get(h,'CData'))
```

## 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.

`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.

`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.

`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.

`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'.

`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.

`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.

`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.

`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.

`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]'`.

`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)
```

`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.

`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]'`.

`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'`.

`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)
```

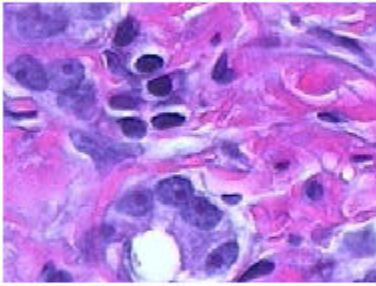
```
im =
```

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**

# images.dicom.decodeUID

Get information about DICOM unique identifier

## Syntax

```
details = images.dicom.decodeUID(UID)
```

## Description

`details = images.dicom.decodeUID(UID)` returns information about the DICOM unique identifier contained in `UID`. `details` contains the name of the UID and its type (SOP class, transfer syntax, etc.). If `UID` corresponds to a transfer syntax, `details` also contains the endianness, the DICOM value representation necessary for reading the image pixels, and information about compression.

The `images.dicom.decodeUID` function can interpret IDs defined in the PS 3.6-1999 specification, with some additions from PS 3.6-2009.

## Examples

### Decode DICOM UID

Read metadata from a DICOM file and extract a UID field.

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

Decode the UID.

```
uid_info = images.dicom.decodeUID(uid)  
  
uid_info = struct with fields:  
  Value: '1.2.840.10008.5.1.4.1.1.7'  
  Name: 'Secondary Capture Image Storage'
```

Type: 'SOP Class'

## Input Arguments

### **UID — DICOM unique identifier**

character vector | string | cell array

DICOM unique identifier, specified as a string or character vector.

Example: `uid_info =  
images.dicom.decodeUID('1.2.840.10008.5.1.4.1.1.7')`

Data Types: char | string | cell

## Output Arguments

### **details — Information from UID**

struct

Information from UID, returned as a struct.

## See Also

`dicominfo` | `dicomuid`

**Introduced in R2017b**



# images.dicom.parseDICOMDIR

Extract metadata from DICOMDIR file

## Syntax

```
DICOMDIR = images.dicom.parseDICOMDIR(filename)
```

## Description

`DICOMDIR = images.dicom.parseDICOMDIR(filename)` extracts the metadata from the DICOMDIR file named in `filename`, returning the information in the structure `DICOMDIR`. If `filename` is not a DICOMDIR file, the function returns an error.

A DICOM directory file (DICOMDIR) is a special DICOM file that serves as a directory to a collection of DICOM files stored on removable media, such as CD/DVD ROMs. When devices write DICOM files to removable media, they typically write a DICOMDIR file on the disk to serve as a list of the disk contents.

## Examples

### Extract Metadata from DICOMDIR File

Read information about the contents of a DICOM folder into the workspace.

```
detailsStruct = images.dicom.parseDICOMDIR('DICOMDIR');
```

## Input Arguments

**filename** — DICOMDIR file  
string scalar | character vector

DICOMDIR file, specified as a string scalar or character vector. `filename` can contain a full path name or a relative path name to the file. The name of this file is DICOMDIR, with no file extension.

Data Types: `char` | `string`

## Output Arguments

### **DICOMDIR — Metadata from DICOMDIR file**

`struct`

Metadata from DICOMDIR file, returned as a struct.

## See Also

`dicominfo`

**Introduced in R2017b**

# images.roi.AssistedFreehand class

**Package:** images.roi

Assisted freehand region of interest

## Description

The `images.roi.AssistedFreehand` class creates an `AssistedFreehand` region-of-interest (ROI) object. You call the `draw` method of the object to begin drawing the ROI interactively by using your mouse. With assisted freehand, the line you are drawing follows the edges in the underlying image automatically. You can also adjust the size and position of the ROI by using your mouse. The ROI also has a context menu that controls aspects of its appearance and behavior.

The `images.roi.AssistedFreehand` class is a `handle` class.

## Creation

### Description

`h = images.roi.AssistedFreehand` creates an instance of the `images.roi.AssistedFreehand` class with default properties.

`h = images.roi.AssistedFreehand(ax)` creates the ROI on the axes specified by `ax`.

`h = images.roi.AssistedFreehand( ____, Name, Value)` modifies the appearance or behavior of the ROI by specifying values for one or more ROI properties. You can set properties using name-value pairs with or without specifying an axes.

## Input Arguments

### **ax** — Parent of ROI

`gca` (default) | Axes object

Parent of the ROI, specified as an `Axes` object.

## Properties

### Public Properties

#### **Closed** — Close freehand ROI

`true` (default) | `false`

Close the freehand ROI, specified as a logical value `true` or `false`. If `true` (default), `images.roi.Freehand` closes the ROI by connecting the last point drawn to the first point drawn.

#### **Color** — ROI color

`blue` (default) | `ColorSpec`

ROI color, specified as a MATLAB `ColorSpec` (Color Specification).

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

#### **Deletable** — ROI can be interactively deleted via context menu

`true` (default) | `false`

ROI can be interactively deleted via a context menu, specified as `true` or `false`. When the value is `true` (default), you can delete the ROI via the context menu. To disable this context menu item, set this property to `false`. When the value is set to `false`, you can still delete the ROI by calling the `delete` function, specifying the handle to the ROI as the input.

#### **DrawingArea** — Area of axes in which you can interactively place ROI

`'auto'` (default) | `'unlimited'` | `[x,y,w,h]`

Area of the axes in which you can interactively place the ROI, specified as one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangular region beginning at (x,y), and extending to width w and height h.

### FaceAlpha — Transparency of ROI face

0.2 (default) | numeric scalar

Transparency of the ROI face, specified as a numeric scalar value in the range [0 1]. When the value is 1, the ROI is fully opaque. When the value is 0, the ROI is completely transparent.

### FaceSelectable — ROI face can capture clicks

true (default) | false

ROI face can capture clicks, specified as true or false. When true (default), the ROI face captures mouse clicks. When false, the ROI face does not capture mouse clicks.

### HandleVisibility — Visibility of ROI handle in Children property of parent

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**Image — Image on which to draw ROI**

handle to Image object

Image on which to draw ROI, specified as a handle to an Image object.

**InteractionsAllowed — Interactivity of ROI**

'all' (default) | 'none' | 'reshape' | 'translate'

Interactivity of the ROI, specified as one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.
'translate'	The ROI can be translated (moved) within the drawing area but not reshaped. This value is not available for use with <code>images.roi.Point</code> objects.
'reshape'	The ROI can be reshaped but not translated. This value is not available for use with <code>images.roi.Point</code> objects or <code>images.roi.Rectangle</code> objects.

**Label — ROI label**

' ' (default) | character vector | string

ROI label, specified as a character vector or string. By default, the ROI has no label (' ').

**LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Parent — ROI parent**

Axes object

ROI parent, specified as an Axes object. A UIAxes cannot be the parent of the ROI.

**Position — Position of ROI**

*n*-by-2 numeric array

Position of the ROI, specified as an  $n$ -by-2 numeric array of the form  $[x_1 \ y_1; \dots; x_n \ y_n]$ , where  $n$  is the total number of vertices. Each row specifies the position of a point on the ROI edge.

### **Selected** — Selection state of ROI

false (default) | true

Selection state of the ROI, specified as the logical value `true` or `false`. You can also set this property interactively. For example, clicking on the ROI selects the ROI and sets this property to `true`. Similarly, if you press **Ctrl** and click the ROI, the ROI is deselected and the value of this property is set to `false`.

### **SelectedColor** — Color of ROI when selected

'none' (default) | ColorSpec

Color of the ROI when selected, specified as a MATLAB `ColorSpec` (Color Specification). The intensities must be in the range  $[0, 1]$ . If 'SelectedColor' is set to 'none', then the value of `Color` defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example:  $[0 \ 1 \ 0]$

### **Smoothing** — Smooth edge of freehand ROI

1 (default) | nonnegative numeric scalar

Smooth the edge of the freehand ROI using the standard deviation of the Gaussian smoothing kernel, specified as a nonnegative numeric scalar. The 'Smoothing' value filters the  $x$  and  $y$  coordinates of the freehand ROI. The filter size is  $2 * \text{ceil}(2 * \text{Smoothing}) + 1$ .

### **StripeColor** — Color of ROI stripe

'none' (default) | ColorSpec value

Color of the ROI stripe, specified as a MATLAB `ColorSpec` (Color Specification) value. By default, the edge of an ROI is solid colored. If you specify `StripeColor`, the ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

**Tag — Tag to associate with the ROI**

' ' (default) | character vector | string scalar

Tag to associate with the ROI, specified as a character vector or string. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

**UIContextMenu — Context menu**

ContextMenu object

Context menu, specified as a ContextMenu object. Use this property to display a custom context menu when you right-click the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

**UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

**Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as one of the values in this table.

Value	Description
'on'	Display the ROI (default).
'off'	Hide the ROI without deleting it. You still can access the properties of an invisible ROI.

**Waypoints — Control points to reshape freehand ROI**

[] (default) |  $n$ -by-1 logical vector

Control points to reshape the freehand ROI, specified as an  $n$ -by-1 logical vector, the same length as `Position`. Each element of the `Waypoints` vector indicates if the corresponding point in the `Position` array is a waypoint. Positions that are waypoints



are set to `true` in `Waypoints`. Dragging a waypoint modifies the ROI between the specified waypoint and its immediate neighboring waypoints. When you draw the freehand ROI, `images.roi.AssistedFreehand` automatically generates `Waypoints` at locations of increased curvature.

## Methods

### Public Methods

```
<infotypegroup type="method"> beginDrawingFromPoint createMask draw inROI
</infotypegroup>
```

## Events

Event Name	Trigger	Event Data	Event Attributes
DeletingROI	ROI is about to be interactively deleted.	<code>event.EventData</code>	NotifyAccess: private  ListenAccess: public
DrawingStarted	ROI is about to be interactively drawn.	<code>event.EventData</code>	NotifyAccess: private  ListenAccess: public
DrawingFinished	ROI has been interactively drawn.	<code>event.EventData</code>	NotifyAccess: private  ListenAccess: public
MovingROI	ROI shape or location is being interactively changed.	<code>images.roi.ROIMovingEventData</code>	NotifyAccess: private  ListenAccess: public

<b>Event Name</b>	<b>Trigger</b>	<b>Event Data</b>	<b>Event Attributes</b>
ROIMoved	ROI shape or location has been interactively changed.	images.roi.ROIMovingEventData	NotifyAccess: private ListenAccess: public
ROIClicked	ROI has been clicked.	images.roi.ROIClickedEventData	NotifyAccess: private ListenAccess: public
AddingWaypoint	A waypoint is about to be interactively added to the ROI.	event.EventData	NotifyAccess: private ListenAccess: public
WaypointAdded	A waypoint has been interactively added to the ROI.	event.EventData	NotifyAccess: private ListenAccess: public
RemovingWaypoint	A waypoint is about to be interactively removed from the ROI.	event.EventData	NotifyAccess: private ListenAccess: public
WaypointRemoved	A waypoint has been interactively removed from the ROI.	event.EventData	NotifyAccess: private ListenAccess: public

## Examples

### Create Freehand ROI Using Assisted Freehand

Read an image into the workspace and display it.

```
figure;  
imshow(imread('baby.jpg'));
```

Create an instance of an `images.roi.AssistedFreehand` class. By default, the class creates the ROI on the current axes. Note that the axes must contain an image.

```
h = images.roi.AssistedFreehand;
```

Call the `draw` method, specifying the `AssistedFreehand` object as an argument. The cursor changes to a cross-hair shape when you move it over the image. You can begin drawing the ROI. Note how, as you move the cursor, the line you draw follows the edges in the underlying image.

```
draw(h);
```

## See Also

`drawfreehand`

## Topics

[“ROI Creation Overview”](#)

**Introduced in R2018b**

## images.roi.Circle class

**Package:** images.roi

Circular region-of-interest

### Description

The `images.roi.Circle` class creates a circular region-of-interest (ROI) object. You call the `draw` method of the object to begin drawing the ROI interactively by using your mouse. You can also adjust the size and position of the ROI by using the mouse. The ROI also has a context menu that controls aspects of its appearance and behavior.

The `images.roi.Circle` class is a handle class.

### Creation

#### Description

`h = images.roi.Circle` creates an instance of the `images.roi.Circle` class with default properties.

`h = images.roi.Circle(ax)` creates the ROI on the axes specified by `ax`.

`h = images.roi.Circle(___,Name,Value)` modifies the appearance or behavior of the ROI by specifying values for one or more ROI properties. You can set properties using name-value pairs with or without specifying an axes.

### Input Arguments

#### **ax — Parent of ROI**

`gca` (default) | Axes object

Parent of the ROI, specified as an Axes object.

## Properties

### Public Properties

#### Center — Center of ROI

[] (default) | 1-by-2 numeric array

Center of the ROI, specified as a 1-by-2 numeric array of the form [x y]. The values x and y are the coordinates of the center point of the ROI. The value of this property changes automatically when you draw or move the ROI.

#### Color — ROI color

blue (default) | ColorSpec

ROI color, specified as a MATLAB ColorSpec (Color Specification).

Example: 'green'

Example: 'g'

Example: [0 1 0]

#### Deletable — ROI can be interactively deleted via context menu

true (default) | false

ROI can be interactively deleted via a context menu, specified as true or false. When the value is true (default), you can delete the ROI via the context menu. To disable this context menu item, set this property to false. When the value is set to false, you can still delete the ROI by calling the delete function, specifying the handle to the ROI as the input.

#### DrawingArea — Area of axes in which you can interactively place ROI

'auto' (default) | 'unlimited' | [x,y,w,h]

Area of the axes in which you can interactively place the ROI, specified as one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).

Value	Description
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangular region beginning at (x,y), and extending to width w and height h.

**FaceAlpha — Transparency of ROI face**

0.2 (default) | numeric scalar

Transparency of the ROI face, specified as a numeric scalar value in the range [0 1]. When the value is 1, the ROI is fully opaque. When the value is 0, the ROI is completely transparent.

**FaceSelectable — ROI face can capture clicks**

true (default) | false

ROI face can capture clicks, specified as true or false. When true (default), the ROI face captures mouse clicks. When false, the ROI face does not capture mouse clicks.

**HandleVisibility — Visibility of ROI handle in Children property of parent**

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**InteractionsAllowed — Interactivity of ROI**

'all' (default) | 'none' | 'reshape' | 'translate'

Interactivity of the ROI, specified as one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.
'translate'	The ROI can be translated (moved) within the drawing area but not reshaped. This value is not available for use with <code>images.roi.Point</code> objects.
'reshape'	The ROI can be reshaped but not translated. This value is not available for use with <code>images.roi.Point</code> objects or <code>images.roi.Rectangle</code> objects.

**Label — ROI label**

' ' (default) | character vector | string

ROI label, specified as a character vector or string. By default, the ROI has no label (' ').

**LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Parent — ROI parent**

Axes object

ROI parent, specified as an Axes object. A UIAxes cannot be the parent of the ROI.

**Radius — Radius of circle**

nonnegative numeric scalar

Radius of the circle, specified as a nonnegative numeric scalar. You can also set this property by drawing or resizing the circle.

**Selected — Selection state of ROI**

false (default) | true

Selection state of the ROI, specified as the logical value `true` or `false`. You can also set this property interactively. For example, clicking on the ROI selects the ROI and sets this

property to `true`. Similarly, if you press **Ctrl** and click the ROI, the ROI is deselected and the value of this property is set to `false`.

**SelectedColor — Color of ROI when selected**

'none' (default) | `ColorSpec`

Color of the ROI when selected, specified as a MATLAB `ColorSpec` (`Color Specification`). The intensities must be in the range  $[0, 1]$ . If 'SelectedColor' is set to 'none', then the value of `Color` defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example:  $[0 \ 1 \ 0]$

**StripeColor — Color of ROI stripe**

'none' (default) | `ColorSpec` value

Color of the ROI stripe, specified as a MATLAB `ColorSpec` (`Color Specification`) value. By default, the edge of an ROI is solid colored. If you specify `StripeColor`, the ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example:  $[0 \ 1 \ 0]$

**Tag — Tag to associate with the ROI**

' ' (default) | character vector | string scalar

Tag to associate with the ROI, specified as a character vector or string. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

**UIContextMenu — Context menu**

`ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a custom context menu when you right-click the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.



**UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

**Vertices — Locations of points on edge of Circle** $n$ -by-2 array

Locations of points on the edge of the ROI, returned as an  $n$ -by-2 array, where  $n$  is the total number of vertices.

This property is read-only.

**Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as one of the values in this table.

Value	Description
'on'	Display the ROI (default).
'off'	Hide the ROI without deleting it. You still can access the properties of an invisible ROI.

## Methods

**Public Methods**

`<infotYPEgroup type="method"> beginDrawingFromPoint createMask draw inROI </infotYPEgroup>`

## Events

Event Name	Trigger	Event Data	Event Attributes
DeletingROI	ROI is about to be interactively deleted.	event.EventData	NotifyAccess: private  ListenAccess: public
DrawingStarted	ROI is about to be interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public
DrawingFinished	ROI has been interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public
MovingROI	ROI shape or location is being interactively changed.	images.roi.CircleMovingEventData	NotifyAccess: private  ListenAccess: public
ROIMoved	ROI shape or location has been interactively changed.	images.roi.CircleMovingEventData	NotifyAccess: private  ListenAccess: public
ROIClicked	ROI has been clicked.	images.roi.ROIClickedEventData	NotifyAccess: private  ListenAccess: public

## Examples

### Create Circular ROI Non-interactively

Read an image into the workspace and display it.

```
I = imread('baby.jpg');  
figure  
imshow(I)
```



Create a circular ROI on the image. Use the 'Center' property to specify the location and the 'Radius' property to specify the size. For programmatically created ROIs, if you want the ROI drawn in a specific axes, you must specify that axes as an input argument. Otherwise, an instance of the `images.roi.Circle` class is created but not displayed. In this example, specify the current axes (`gca`) to draw the ROI on the image in that axes.

```
h = images.roi.Circle(gca, 'Center', [1000 1000], 'Radius', 500);
```



1-1040

**See Also**

## **Topics**

“ROI Creation Overview”

**Introduced in R2018b**

## images.roi.Ellipse class

**Package:** images.roi

Elliptical region-of-interest

### Description

The `images.roi.Ellipse` class creates an elliptical region-of-interest (ROI) object. You call the `draw` method of the object to begin drawing the ROI interactively by using your mouse. You can also adjust the size and position of the ROI by using the mouse. The ROI also has a context menu that controls aspects of its appearance and behavior.

The `images.roi.Ellipse` class is a handle class.

### Creation

#### Description

`h = images.roi.Ellipse` creates an instance of the `images.roi.Ellipse` class with default properties.

`h = images.roi.Ellipse(ax)` creates the ROI on the axes specified by `ax`.

`h = images.roi.Ellipse( ____, Name, Value)` modifies the appearance or behavior of the ROI by specifying values for one or more ROI properties. You can set properties using name-value pairs with or without specifying an axes.

#### Input Arguments

**ax — Parent of ROI**

`gca` (default) | Axes object

Parent of the ROI, specified as an Axes object.



# Properties

## Public Properties

### AspectRatio — Aspect ratio of ellipse

(1+sqrt(5))/2 (default) | positive numeric scalar

Aspect ratio of the ellipse, specified as a numeric scalar. The value of this property changes automatically when you draw or resize the ellipse, or by setting the `SemiAxes` property. The `images.roi.Ellipse` object calculates this value as height/width.

### Center — Center of ROI

[] (default) | 1-by-2 numeric array

Center of the ROI, specified as a 1-by-2 numeric array of the form [x y]. The values x and y are the coordinates of the center point of the ROI. The value of this property changes automatically when you draw or move the ROI.

### Color — ROI color

blue (default) | ColorSpec

ROI color, specified as a MATLAB `ColorSpec` (Color Specification).

Example: 'green'

Example: 'g'

Example: [0 1 0]

### Deletable — ROI can be interactively deleted via context menu

true (default) | false

ROI can be interactively deleted via a context menu, specified as `true` or `false`. When the value is `true` (default), you can delete the ROI via the context menu. To disable this context menu item, set this property to `false`. When the value is set to `false`, you can still delete the ROI by calling the `delete` function, specifying the handle to the ROI as the input.

### DrawingArea — Area of axes in which you can interactively place ROI

'auto' (default) | 'unlimited' | [x,y,w,h]

Area of the axes in which you can interactively place the ROI, specified as one of the values in this table.

<b>Value</b>	<b>Description</b>
'auto'	The drawing area is the current axes limits (default).
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangular region beginning at (x,y), and extending to width w and height h.

**FaceAlpha — Transparency of ROI face**

0.2 (default) | numeric scalar

Transparency of the ROI face, specified as a numeric scalar value in the range [0 1]. When the value is 1, the ROI is fully opaque. When the value is 0, the ROI is completely transparent.

**FaceSelectable — ROI face can capture clicks**

true (default) | false

ROI face can capture clicks, specified as true or false. When true (default), the ROI face captures mouse clicks. When false, the ROI face does not capture mouse clicks.

**FixedAspectRatio — Aspect ratio remains constant during interaction**

false (default) | true

Aspect ratio remains constant during interaction, specified as true or false. When the value is true, the ROI maintains the aspect ratio when you draw or resize the ellipse. When the value is false (default), you can change the aspect ratio when drawing or resizing the ellipse. You can change the state of this property using the default context menu.

**HandleVisibility — Visibility of ROI handle in Children property of parent**

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

### InteractionsAllowed – Interactivity of ROI

'all' (default) | 'none' | 'reshape' | 'translate'

Interactivity of the ROI, specified as one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.
'translate'	The ROI can be translated (moved) within the drawing area but not reshaped. This value is not available for use with <code>images.roi.Point</code> objects.
'reshape'	The ROI can be reshaped but not translated. This value is not available for use with <code>images.roi.Point</code> objects or <code>images.roi.Rectangle</code> objects.

### Label – ROI label

' ' (default) | character vector | string

ROI label, specified as a character vector or string. By default, the ROI has no label ('').

### LineWidth – Width of ROI edge

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

## **Parent — ROI parent**

Axes object

ROI parent, specified as an Axes object. A UIAxes cannot be the parent of the ROI.

## **RotationAngle — Angle around center of ROI**

0 (default) | nonnegative numeric scalar

Angle around the center of the ROI, specified as a nonnegative numeric scalar. The angle is measured in degrees in a clockwise direction. The value of this property changes automatically when you draw or move the ROI.

The value of `RotationAngle` does not impact the value of `Position`. The `Position` property represents the initial position of the ROI, before rotation. To determine the location of a rotated ROI, use the `Vertices` property.

## **Selected — Selection state of ROI**

false (default) | true

Selection state of the ROI, specified as the logical value `true` or `false`. You can also set this property interactively. For example, clicking on the ROI selects the ROI and sets this property to `true`. Similarly, if you press **Ctrl** and click the ROI, the ROI is deselected and the value of this property is set to `false`.

## **SelectedColor — Color of ROI when selected**

'none' (default) | ColorSpec

Color of the ROI when selected, specified as a MATLAB `ColorSpec` (`Color Specification`). The intensities must be in the range `[0,1]`. If `'SelectedColor'` is set to `'none'`, then the value of `Color` defines the color of the ROI for all states, selected or not.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

## **SemiAxes — Lengths of semiaxes of ellipse**

`[semiaxis1 semiaxis2]`

Lengths of the semiaxis of the ellipse, specified as a 1-by-2 array of the form `[semiaxis1 semiaxis2]`. The `images.roi.Ellipse` object assigns the length of the semiaxis that

is closest to the  $x$  direction to `semiaxis1`. Note however that the shape and orientation of the ellipse can change through interaction. The value of this property changes automatically when you draw or reshape the ROI.

### **StripeColor — Color of ROI stripe**

'none' (default) | `ColorSpec` value

Color of the ROI stripe, specified as a MATLAB `ColorSpec` (Color Specification) value. By default, the edge of an ROI is solid colored. If you specify `StripeColor`, the ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

### **Tag — Tag to associate with the ROI**

' ' (default) | character vector | string scalar

Tag to associate with the ROI, specified as a character vector or string. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

### **UIContextMenu — Context menu**

`ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a custom context menu when you right-click the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

### **UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

### **Vertices — Locations of points that lie along perimeter of ellipse**

$n$ -by-2 array

Locations of points that lie along the perimeter of the ellipse, specified as an  $n$ -by-2 array. This property is read-only.

**Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as one of the values in this table.

Value	Description
'on'	Display the ROI (default).
'off'	Hide the ROI without deleting it. You still can access the properties of an invisible ROI.

**Methods****Public Methods**

```
<infotYPEgroup type="method"> beginDrawingFromPoint createMask draw inROI  
</infotYPEgroup>
```

**Events**

Event Name	Trigger	Event Data	Event Attributes
DeletingROI	ROI is about to be interactively deleted.	event.EventData	NotifyAccess: private  ListenAccess: public
DrawingStarted	ROI is about to be interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public
DrawingFinished	ROI has been interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public

Event Name	Trigger	Event Data	Event Attributes
MovingROI	ROI shape or location is being interactively changed.	images.roi.EllipseMovingEventData	NotifyAccess: private ListenAccess: public
ROIMoved	ROI shape or location has been interactively changed.	images.roi.EllipseMovingEventData	NotifyAccess: private ListenAccess: public
ROIClicked	ROI has been clicked.	images.roi.ROIClickedEventData	NotifyAccess: private ListenAccess: public

## Examples

### Create Elliptical ROI Non-interactively

Read an image into the workspace and display it.

```
I = imread('baby.jpg');
figure
imshow(I)
```





Create an elliptical ROI on the image, using the `Center` property to specify the location and the `SemiAxes` property to specify its shape. For programmatically created ROIs, if you want the ROI drawn in a specific axes, you must specify that axes as an input argument. Otherwise, an instance of the `images.roi.Ellipse` class is created but not displayed. In this example, specify the current axes (`gca`) to draw the ROI on the image in that axes.

```
h = images.roi.Ellipse(gca, 'Center', [1000 1000], 'Semiaxes', [350 150]);
```



1-1052

**See Also**

## **Topics**

“ROI Creation Overview”

**Introduced in R2018b**

## images.roi.Freehand class

**Package:** images.roi

Freehand region-of-interest

### Description

The `images.roi.Freehand` class creates a freehand region-of-interest (ROI) object. You call the `draw` method of the object to begin drawing the ROI interactively by using your mouse. You can also adjust the size and position of the ROI by using the mouse. The ROI also has a context menu that controls aspects of its appearance and behavior.

The `images.roi.Freehand` class is a `handle` class.

### Creation

#### Description

`h = images.roi.Freehand` creates an instance of the `images.roi.Freehand` class with default properties.

`h = images.roi.Freehand(ax)` creates the ROI in the axes specified by `ax`.

`h = images.roi.Freehand( ____, Name, Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs to specify properties. You can set properties using name-value pairs with or without specifying `ax`.

#### Input Arguments

**ax — Parent of ROI**

`gca` (default) | Axes object

Parent of the ROI, specified as an Axes object.

# Properties

## Public Properties

### Closed — Close freehand ROI

true (default) | false

Close the freehand ROI, specified as a logical value `true` or `false`. If `true` (default), `images.roi.Freehand` closes the ROI by connecting the last point drawn to the first point drawn.

### Color — ROI color

blue (default) | `ColorSpec`

ROI color, specified as a MATLAB `ColorSpec` (Color Specification).

Example: 'green'

Example: 'g'

Example: [0 1 0]

### Deletable — ROI can be interactively deleted via context menu

true (default) | false

ROI can be interactively deleted via a context menu, specified as `true` or `false`. When the value is `true` (default), you can delete the ROI via the context menu. To disable this context menu item, set this property to `false`. When the value is set to `false`, you can still delete the ROI by calling the `delete` function, specifying the handle to the ROI as the input.

### DrawingArea — Area of axes in which you can interactively place ROI

'auto' (default) | 'unlimited' | [x,y,w,h]

Area of the axes in which you can interactively place the ROI, specified as one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).

Value	Description
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangular region beginning at (x,y), and extending to width w and height h.

**FaceAlpha — Transparency of ROI face**

0.2 (default) | numeric scalar

Transparency of the ROI face, specified as a numeric scalar value in the range [0 1]. When the value is 1, the ROI is fully opaque. When the value is 0, the ROI is completely transparent.

**FaceSelectable — ROI face can capture clicks**

true (default) | false

ROI face can capture clicks, specified as true or false. When true (default), the ROI face captures mouse clicks. When false, the ROI face does not capture mouse clicks.

**HandleVisibility — Visibility of ROI handle in Children property of parent**

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**InteractionsAllowed — Interactivity of ROI**

'all' (default) | 'none' | 'reshape' | 'translate'

Interactivity of the ROI, specified as one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.
'translate'	The ROI can be translated (moved) within the drawing area but not reshaped. This value is not available for use with <code>images.roi.Point</code> objects.
'reshape'	The ROI can be reshaped but not translated. This value is not available for use with <code>images.roi.Point</code> objects or <code>images.roi.Rectangle</code> objects.

**Label — ROI label**

' ' (default) | character vector | string

ROI label, specified as a character vector or string. By default, the ROI has no label ('').

**LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Multiclick — Control freehand drawing style during interactive placement**

false (default) | true

Control the freehand drawing style during interactive placement, specified as the logical value `true` or `false`. When the value is `false` (the default), a single click and drag gesture completes the freehand ROI. When the value is `true`, multiple click and drag gestures can be combined with straight edges to make a more complex freehand ROI shape.

**Parent — ROI parent**

Axes object

ROI parent, specified as an Axes object. A UIAxes cannot be the parent of the ROI.

**Position — Position of ROI**

*n*-by-2 numeric array

Position of the ROI, specified as an  $n$ -by-2 numeric array of the form `[x1 y1; ... ;xn yn]`, where  $n$  is the number of points. Each row specifies the position of a point defining the ROI.

**Selected** — Selection state of ROI`false` (default) | `true`

Selection state of the ROI, specified as the logical value `true` or `false`. You can also set this property interactively. For example, clicking on the ROI selects the ROI and sets this property to `true`. Similarly, if you press **Ctrl** and click the ROI, the ROI is deselected and the value of this property is set to `false`.

**SelectedColor** — Color of ROI when selected`'none'` (default) | `ColorSpec`

Color of the ROI when selected, specified as a MATLAB `ColorSpec` (Color Specification). The intensities must be in the range `[0,1]`. If `'SelectedColor'` is set to `'none'`, then the value of `Color` defines the color of the ROI for all states, selected or not.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

**Smoothing** — Smooth ROI during interactive placement`1` (default) | positive numeric scalar

Smooth ROI during interactive placement, specified as a positive numeric scalar. Filter the  $x$  and  $y$  coordinates of the ROI after interactive placement with a Gaussian smoothing kernel with standard deviation specified by `Smoothing`. The filter size is `2*ceil(2*Smoothing) + 1`.

**StripeColor** — Color of ROI stripe`'none'` (default) | `ColorSpec` value

Color of the ROI stripe, specified as a MATLAB `ColorSpec` (Color Specification) value. By default, the edge of an ROI is solid colored. If you specify `StripeColor`, the ROI edge is striped. The striping consists of a combination of the value specified by `'Color'` and this value.

Example: `'green'`



Example: 'g'

Example: [0 1 0]

### Tag — Tag to associate with the ROI

' ' (default) | character vector | string scalar

Tag to associate with the ROI, specified as a character vector or string. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

### UIContextMenu — Context menu

ContextMenu object

Context menu, specified as a ContextMenu object. Use this property to display a custom context menu when you right-click the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

### UserData — Data to associate with ROI

any MATLAB data

Data to associate with the ROI, specified as any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

### Visible — ROI visibility

'on' (default) | 'off'

ROI visibility, specified as one of the values in this table.

Value	Description
'on'	Display the ROI (default).
'off'	Hide the ROI without deleting it. You still can access the properties of an invisible ROI.

### Waypoints — Control points used to reshape ROI

[] (default) |  $n$ -by-1 array

Control points used to reshape ROI, specified as an  $n$ -by-1 array. In the array, a row with the value `true` indicates the position on the ROI edge is a waypoint. The length of the Waypoints array must match the number of rows in the `Position` property. Dragging a

waypoint modifies the freehand-drawn region between the specified waypoint and its immediate neighboring waypoints. If empty (the default), `images.roi.Freehand` generates waypoints automatically at locations of increased curvature.

## Methods

### Public Methods

`<infotypegroup type="method">` `beginDrawingFromPoint` `createMask` `draw` `inROI`  
`</infotypegroup>`

## Events

Event Name	Trigger	Event Data	Event Attributes
DeletingROI	ROI is about to be interactively deleted.	<code>event.EventData</code>	NotifyAccess: private  ListenAccess: public
DrawingStarted	ROI is about to be interactively drawn.	<code>event.EventData</code>	NotifyAccess: private  ListenAccess: public
DrawingFinished	ROI has been interactively drawn.	<code>event.EventData</code>	NotifyAccess: private  ListenAccess: public
MovingROI	ROI shape or location is being interactively changed.	<code>images.roi.ROIMovingEventData</code>	NotifyAccess: private  ListenAccess: public

Event Name	Trigger	Event Data	Event Attributes
ROIMoved	ROI shape or location has been interactively changed.	images.roi.ROIMovingEventData	NotifyAccess: private ListenAccess: public
ROIClicked	ROI has been clicked.	images.roi.ROIClickedEventData	NotifyAccess: private ListenAccess: public
AddingWaypoint	A waypoint is about to be interactively added to the ROI.	event.EventData	NotifyAccess: private ListenAccess: public
WaypointAdded	A waypoint has been interactively added to the ROI.	event.EventData	NotifyAccess: private ListenAccess: public
RemovingWaypoint	A waypoint is about to be interactively removed from the ROI.	event.EventData	NotifyAccess: private ListenAccess: public
WaypointRemoved	A waypoint has been interactively removed from the ROI.	event.EventData	NotifyAccess: private ListenAccess: public

## Examples

### Create Freehand ROI Non-interactively

Read an image into the workspace and display it.

```
I = imread('baby.jpg');  
figure  
imshow(I)
```



Create a freehand ROI on the image, using the `Position` property to specify the vertices of the ROI. For programmatically created ROIs, if you want the ROI drawn in a specific axes, you must specify that axes as an input argument. Otherwise, an instance of the `images.roi.Freehand` class is created but not displayed. In this example, specify the current axes (`gca`) to draw the ROI on the image in that axes.

```
h = images.roi.Freehand(gca, 'Position', [100 150; 200 250; 300 350; 150 450]);
```



**Topics**

“ROI Creation Overview”

**Introduced in R2018b**



# images.roi.Line class

**Package:** images.roi

Line region-of-interest

## Description

The `images.roi.Line` class creates a linear region-of-interest (ROI) object. You call the `draw` method of the object to begin drawing the ROI interactively by using your mouse. You can also adjust the size and position of the ROI by using the mouse. The ROI also has a context menu that controls aspects of its appearance and behavior.

The `images.roi.Line` class is a `handle` class.

## Creation

### Description

`h = images.roi.Line` creates an instance of the `images.roi.Line` class with default properties.

`h = images.roi.Line(ax)` creates the ROI on the axes specified by `ax`.

`h = images.roi.Line( ____, Name, Value)` modifies the appearance or behavior of the ROI by specifying values for one or more ROI properties. You can set properties using name-value pairs with or without specifying an axes.

## Input Arguments

### **ax — Parent of ROI**

`gca` (default) | Axes object

Parent of the ROI, specified as an Axes object.

## Properties

### Public Properties

#### Color — ROI color

blue (default) | ColorSpec

ROI color, specified as a MATLAB ColorSpec (Color Specification).

Example: 'green'

Example: 'g'

Example: [0 1 0]

#### Deletable — ROI can be interactively deleted via context menu

true (default) | false

ROI can be interactively deleted via a context menu, specified as true or false. When the value is true (default), you can delete the ROI via the context menu. To disable this context menu item, set this property to false. When the value is set to false, you can still delete the ROI by calling the delete function, specifying the handle to the ROI as the input.

#### DrawingArea — Area of axes in which you can interactively place ROI

'auto' (default) | 'unlimited' | [x,y,w,h]

Area of the axes in which you can interactively place the ROI, specified as one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangular region beginning at (x,y), and extending to width w and height h.

**HandleVisibility – Visibility of ROI handle in Children property of parent**

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**InteractionsAllowed – Interactivity of ROI**

'all' (default) | 'none' | 'reshape' | 'translate'

Interactivity of the ROI, specified as one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.
'translate'	The ROI can be translated (moved) within the drawing area but not reshaped. This value is not available for use with <code>images.roi.Point</code> objects.
'reshape'	The ROI can be reshaped but not translated. This value is not available for use with <code>images.roi.Point</code> objects or <code>images.roi.Rectangle</code> objects.

**Label – ROI label**

' ' (default) | character vector | string

ROI label, specified as a character vector or string. By default, the ROI has no label (' ').

## **LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

## **Parent — ROI parent**

Axes object

ROI parent, specified as an Axes object. A UIAxes cannot be the parent of the ROI.

## **Position — Position of ROI**

[] (default) | 2-by-2 numeric array

Position of the ROI, specified as a 2-by-2 array of the form [x1 y1; x2 y2], where each row specifies the respective end-point of the line segment. You can also set this property by drawing or moving the line.

## **Selected — Selection state of ROI**

false (default) | true

Selection state of the ROI, specified as the logical value `true` or `false`. You can also set this property interactively. For example, clicking on the ROI selects the ROI and sets this property to `true`. Similarly, if you press **Ctrl** and click the ROI, the ROI is deselected and the value of this property is set to `false`.

## **SelectedColor — Color of ROI when selected**

'none' (default) | ColorSpec

Color of the ROI when selected, specified as a MATLAB `ColorSpec` (`Color Specification`). The intensities must be in the range `[0,1]`. If `'SelectedColor'` is set to `'none'`, then the value of `Color` defines the color of the ROI for all states, selected or not.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

## **StripeColor — Color of ROI stripe**

'none' (default) | ColorSpec value

Color of the ROI stripe, specified as a MATLAB `ColorSpec` (`Color Specification`) value. By default, the edge of an ROI is solid colored. If you specify `StripeColor`, the

ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

### Tag — Tag to associate with the ROI

' ' (default) | character vector | string scalar

Tag to associate with the ROI, specified as a character vector or string. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

### ContextMenu — Context menu

ContextMenu object

Context menu, specified as a `ContextMenu` object. Use this property to display a custom context menu when you right-click the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

### UserData — Data to associate with ROI

any MATLAB data

Data to associate with the ROI, specified as any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

### Visible — ROI visibility

'on' (default) | 'off'

ROI visibility, specified as one of the values in this table.

Value	Description
'on'	Display the ROI (default).
'off'	Hide the ROI without deleting it. You still can access the properties of an invisible ROI.

## Methods

### Public Methods

```
<infotypegroup type="method"> beginDrawingFromPoint createMask draw </infotypegroup>
```

## Events

Event Name	Trigger	Event Data	Event Attributes
DeletingROI	ROI is about to be interactively deleted.	event.EventData	NotifyAccess: private  ListenAccess: public
DrawingStarted	ROI is about to be interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public
DrawingFinished	ROI has been interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public
MovingROI	ROI shape or location is being interactively changed.	images.roi.ROIMovingEventData	NotifyAccess: private  ListenAccess: public
ROIMoved	ROI shape or location has been interactively changed.	images.roi.ROIMovingEventData	NotifyAccess: private  ListenAccess: public

Event Name	Trigger	Event Data	Event Attributes
ROIClicked	ROI has been clicked.	images.roi.ROIClickedEventData	NotifyAccess: private  ListenAccess: public

## Examples

### Create Linear ROI Non-interactively

Read an image into the workspace and display it.

```
I = imread('baby.jpg');  
figure  
imshow(I)
```





Create a linear ROI on the image, using the `Position` property to specify the ROI location. For programmatically created ROIs, if you want the ROI drawn in a specific axes, you must specify that axes as an input argument. Otherwise, an instance of the `images.roi.Line` class is created but not displayed. In this example, specify the current axes (`gca`) to draw the ROI on the image in that axes.

```
h = images.roi.Line(gca,'Position',[100 150;400 650]);
```



1-1076

**See Also**

## **Topics**

“ROI Creation Overview”

**Introduced in R2018b**

## images.roi.Point class

**Package:** images.roi

Point region of interest

### Description

The `images.roi.Point` class creates a point region-of-interest (ROI) object. You call the `draw` method of the object to begin drawing the ROI interactively by using your mouse. You can also adjust the size and position of the ROI by using the mouse. The ROI also has a context menu that controls aspects of its appearance and behavior.

The `images.roi.Point` class is a handle class.

### Creation

#### Description

`h = images.roi.Point` creates an instance of the `images.roi.Point` class with default properties.

`h = images.roi.Point(ax)` creates the ROI in the axes specified by `ax`.

`h = images.roi.Point(___, Name, Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs to specify properties. You can set properties using name-value pairs with or without specifying `ax`.

#### Input Arguments

**ax — Parent of ROI**

`gca` (default) | Axes object

Parent of ROI, specified as an Axes object.

## Properties

### Public Properties

#### Color — ROI color

blue (default) | ColorSpec

ROI color, specified as a MATLAB ColorSpec (Color Specification).

Example: 'green'

Example: 'g'

Example: [0 1 0]

#### Deletable — ROI can be interactively deleted via context menu

true (default) | false

ROI can be interactively deleted via a context menu, specified as true or false. When the value is true (default), you can delete the ROI via the context menu. To disable this context menu item, set this property to false. When the value is set to false, you can still delete the ROI by calling the delete function, specifying the handle to the ROI as the input.

#### DrawingArea — Area of axes in which you can interactively place ROI

'auto' (default) | 'unlimited' | [x,y,w,h]

Area of the axes in which you can interactively place the ROI, specified as one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangular region beginning at (x,y), and extending to width w and height h.

**HandleVisibility — Visibility of ROI handle in Children property of parent**

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**InteractionsAllowed — Interactivity of ROI**

'all' (default) | 'none' | 'reshape' | 'translate'

Interactivity of the ROI, specified as one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.
'translate'	The ROI can be translated (moved) within the drawing area but not reshaped. This value is not available for use with <code>images.roi.Point</code> objects.
'reshape'	The ROI can be reshaped but not translated. This value is not available for use with <code>images.roi.Point</code> objects or <code>images.roi.Rectangle</code> objects.

**Label — ROI label**

' ' (default) | character vector | string

ROI label, specified as a character vector or string. By default, the ROI has no label (' ').

**LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Parent — ROI parent**

Axes object

ROI parent, specified as an Axes object. A UIAxes cannot be the parent of the ROI.

**Position — Position of ROI**

1-by-2 numeric array

Position of the ROI, specified as a 1-by-2 numeric array of the form  $[x \ y]$ , where  $x$  and  $y$  specify the location of the point. You can modify this property by drawing or moving the point.

**Selected — Selection state of ROI**

false (default) | true

Selection state of the ROI, specified as the logical value `true` or `false`. You can also set this property interactively. For example, clicking on the ROI selects the ROI and sets this property to `true`. Similarly, if you press **Ctrl** and click the ROI, the ROI is deselected and the value of this property is set to `false`.

**SelectedColor — Color of ROI when selected**

'none' (default) | ColorSpec

Color of the ROI when selected, specified as a MATLAB `ColorSpec` (`Color Specification`). The intensities must be in the range  $[0, 1]$ . If 'SelectedColor' is set to 'none', then the value of `Color` defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example:  $[0 \ 1 \ 0]$

**StripeColor — Color of ROI stripe**

'none' (default) | ColorSpec value

Color of the ROI stripe, specified as a MATLAB `ColorSpec` (`Color Specification`) value. By default, the edge of an ROI is solid colored. If you specify `StripeColor`, the

ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

### **Tag — Tag to associate with the ROI**

' ' (default) | character vector | string scalar

Tag to associate with the ROI, specified as a character vector or string. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

### **UIContextMenu — Context menu**

ContextMenu object

Context menu, specified as a `ContextMenu` object. Use this property to display a custom context menu when you right-click the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

### **UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

### **Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as one of the values in this table.

<b>Value</b>	<b>Description</b>
'on'	Display the ROI (default).
'off'	Hide the ROI without deleting it. You still can access the properties of an invisible ROI.



## Methods

### Public Methods

`<infotYPEgroup type="method"> beginDrawingFromPoint draw </infotYPEgroup>`

## Events

Event Name	Trigger	Event Data	Event Attributes
DeletingROI	ROI is about to be interactively deleted.	event.EventData	NotifyAccess: private  ListenAccess: public
DrawingStarted	ROI is about to be interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public
DrawingFinished	ROI has been interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public
MovingROI	ROI shape or location is being interactively changed.	images.roi.ROIMovingEventData	NotifyAccess: private  ListenAccess: public
ROIMoved	ROI shape or location has been interactively changed.	images.roi.ROIMovingEventData	NotifyAccess: private  ListenAccess: public

<b>Event Name</b>	<b>Trigger</b>	<b>Event Data</b>	<b>Event Attributes</b>
ROIClicked	ROI has been clicked.	images.roi.ROIClickedEventData	NotifyAccess: private  ListenAccess: public

## Examples

### Create Point ROI Non-interactively

Read an image into the workspace and display it.

```
I = imread('baby.jpg');  
figure;  
imshow(I)
```



Create a point ROI on the image, using the 'Position' property to specify the location. For programmatically created ROIs, if you want the ROI drawn in a specific axes, you must specify that axes as an input argument. Otherwise, an instance of the `images.roi.Point` class is created but not displayed. In this example, specify the current axes (`gca`) to draw the ROI on the image in that axes.

```
h = images.roi.Point(gca,'Position',[400 650]);
```



## **Topics**

“ROI Creation Overview”

**Introduced in R2018b**

# images.roi.Polygon class

**Package:** images.roi

Polygonal region of interest

## Description

The `images.roi.Polygon` class defines a polygonal region-of-interest (ROI) over an image. You call the `draw` method of the object to begin drawing the ROI interactively by using your mouse. You can also adjust the size and position of the ROI by using the mouse. The ROI also has a context menu that controls aspects of its appearance and behavior.

The `images.roi.Polygon` class is a `handle` class.

## Creation

### Description

`h = images.roi.Polygon` creates an instance of the `images.roi.Polygon` object with default properties.

`h = images.roi.Polygon(ax)` creates the ROI in the axes specified by `ax`.

`h = images.roi.Polygon( ____, Name, Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs to specify properties. You can set properties using name-value pairs with or without specifying `ax`.

## Input Arguments

### **ax — Parent of ROI**

`gca` (default) | Axes object

Parent of the ROI, specified as an Axes object.

## Properties

### Public Properties

#### **Color — ROI color**

blue (default) | `ColorSpec`

ROI color, specified as a MATLAB `ColorSpec` (Color Specification).

Example: 'green'

Example: 'g'

Example: [0 1 0]

#### **Deletable — ROI can be interactively deleted via context menu**

true (default) | false

ROI can be interactively deleted via a context menu, specified as `true` or `false`. When the value is `true` (default), you can delete the ROI via the context menu. To disable this context menu item, set this property to `false`. When the value is set to `false`, you can still delete the ROI by calling the `delete` function, specifying the handle to the ROI as the input.

#### **DrawingArea — Area of axes in which you can interactively place ROI**

'auto' (default) | 'unlimited' | [x,y,w,h]

Area of the axes in which you can interactively place the ROI, specified as one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangular region beginning at (x,y), and extending to width <i>w</i> and height <i>h</i> .



**FaceAlpha — Transparency of ROI face**

0.2 (default) | numeric scalar

Transparency of the ROI face, specified as a numeric scalar value in the range [0 1]. When the value is 1, the ROI is fully opaque. When the value is 0, the ROI is completely transparent.

**FaceSelectable — ROI face can capture clicks**

true (default) | false

ROI face can capture clicks, specified as true or false. When true (default), the ROI face captures mouse clicks. When false, the ROI face does not capture mouse clicks.

**HandleVisibility — Visibility of ROI handle in Children property of parent**

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**InteractionsAllowed — Interactivity of ROI**

'all' (default) | 'none' | 'reshape' | 'translate'

Interactivity of the ROI, specified as one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.

Value	Description
'translate'	The ROI can be translated (moved) within the drawing area but not reshaped. This value is not available for use with <code>images.roi.Point</code> objects.
'reshape'	The ROI can be reshaped but not translated. This value is not available for use with <code>images.roi.Point</code> objects or <code>images.roi.Rectangle</code> objects.

**Label — ROI label**`' '` (default) | character vector | string

ROI label, specified as a character vector or string. By default, the ROI has no label (' ').

**LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

**Parent — ROI parent**

Axes object

ROI parent, specified as an Axes object. A UIAxes cannot be the parent of the ROI.

**Position — Position of ROI***n*-by-2 numeric array

Position of the ROI, specified as an *n*-by-2 numeric array of the form `[x1 y1; ...; xn yn]`, where each row specifies the position of a vertex of the polygon.

**Selected — Selection state of ROI**`false` (default) | `true`

Selection state of the ROI, specified as the logical value `true` or `false`. You can also set this property interactively. For example, clicking on the ROI selects the ROI and sets this property to `true`. Similarly, if you press **Ctrl** and click the ROI, the ROI is deselected and the value of this property is set to `false`.

**SelectedColor — Color of ROI when selected**`'none'` (default) | `ColorSpec`

Color of the ROI when selected, specified as a MATLAB `ColorSpec` (`Color Specification`). The intensities must be in the range  $[0, 1]$ . If `'SelectedColor'` is set to `'none'`, then the value of `Color` defines the color of the ROI for all states, selected or not.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

**StripeColor — Color of ROI stripe**`'none'` (default) | `ColorSpec` value

Color of the ROI stripe, specified as a MATLAB `ColorSpec` (`Color Specification`) value. By default, the edge of an ROI is solid colored. If you specify `StripeColor`, the ROI edge is striped. The striping consists of a combination of the value specified by `'Color'` and this value.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

**Tag — Tag to associate with the ROI**`''` (default) | character vector | string scalar

Tag to associate with the ROI, specified as a character vector or string. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

**UIContextMenu — Context menu**`ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a custom context menu when you right-click the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

**UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

**Vertices — Locations of points on the edge of ROI**

*n*-by-2 array

Locations of points on the edge of the ROI, specified as an *n*-by-2 array. This property is read-only.

**Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as one of the values in this table.

Value	Description
'on'	Display the ROI (default).
'off'	Hide the ROI without deleting it. You still can access the properties of an invisible ROI.

## Methods

**Public Methods**

`<infotypegroup type="method"> beginDrawingFromPoint createMask draw inROI  
</infotypegroup>`

## Events

Event Name	Trigger	Event Data	Event Attributes
DeletingROI	ROI is about to be interactively deleted.	event.EventData	NotifyAccess: private  ListenAccess: public

<b>Event Name</b>	<b>Trigger</b>	<b>Event Data</b>	<b>Event Attributes</b>
DrawingStarted	ROI is about to be interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public
DrawingFinished	ROI has been interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public
MovingROI	ROI shape or location is being interactively changed.	images.roi.ROIMovingEventData	NotifyAccess: private  ListenAccess: public
ROIMoved	ROI shape or location has been interactively changed.	images.roi.ROIMovingEventData	NotifyAccess: private  ListenAccess: public
ROIClicked	ROI has been clicked.	images.roi.ROIClickedEventData	NotifyAccess: private  ListenAccess: public
AddingVertex	A vertex is about to be interactively added to the ROI.	event.EventData	NotifyAccess: private  ListenAccess: public
VertexAdded	A vertex has been interactively added to the ROI.	event.EventData	NotifyAccess: private  ListenAccess: public

<b>Event Name</b>	<b>Trigger</b>	<b>Event Data</b>	<b>Event Attributes</b>
DeletingVertex	A vertex is about to be interactively removed from the ROI.	event.EventData	NotifyAccess: private  ListenAccess: public
VertexDeleted	A vertex has been interactively removed from the ROI.	event.EventData	NotifyAccess: private  ListenAccess: public

## Examples

### Create Polygonal ROI Non-interactively

Read an image into the workspace and display it.

```
I = imread('baby.jpg');  
figure;  
imshow(I)
```



Create a polygonal ROI on the image, using the `Position` property to specify the vertices of the ROI. Note that you must specify the axes where you want to draw the ROI as the first argument.

```
h = images.roi.Polygon(gca, 'Position', [100 150; 200 250; 300 350; 150 450]);
```





**Topics**

“ROI Creation Overview”

**Introduced in R2018b**

# images.roi.Polyline class

**Package:** images.roi

Polyline region-of-interest

## Description

The `images.roi.Polyline` class creates a polyline region-of-interest (ROI) object. You call the `draw` method of the object to begin drawing the ROI interactively by using your mouse. You can also adjust the size and position of the ROI by using the mouse. The ROI also has a context menu that controls aspects of its appearance and behavior.

The `images.roi.Polyline` class is a handle class.

## Creation

### Description

`h = images.roi.Polyline` creates an instance of the `images.roi.Polyline` class with default properties.

`h = images.roi.Polyline(ax, ___)` creates the ROI in the axes specified by `ax`.

`h = images.roi.Polyline(___ ,Name,Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs to specify properties. You can set properties using name-value pairs with or without specifying `ax`.

## Input Arguments

### **ax — Parent of ROI**

`gca` (default) | Axes object

Parent of the ROI, specified as an Axes object.

## Properties

### Public Properties

#### Color — ROI color

blue (default) | ColorSpec

ROI color, specified as a MATLAB ColorSpec (Color Specification).

Example: 'green'

Example: 'g'

Example: [0 1 0]

#### Deletable — ROI can be interactively deleted via context menu

true (default) | false

ROI can be interactively deleted via a context menu, specified as `true` or `false`. When the value is `true` (default), you can delete the ROI via the context menu. To disable this context menu item, set this property to `false`. When the value is set to `false`, you can still delete the ROI by calling the `delete` function, specifying the handle to the ROI as the input.

#### DrawingArea — Area of axes in which you can interactively place ROI

'auto' (default) | 'unlimited' | [x,y,w,h]

Area of the axes in which you can interactively place the ROI, specified as one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangular region beginning at (x,y), and extending to width <i>w</i> and height <i>h</i> .

**HandleVisibility – Visibility of ROI handle in Children property of parent**

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

**InteractionsAllowed – Interactivity of ROI**

'all' (default) | 'none' | 'reshape' | 'translate'

Interactivity of the ROI, specified as one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.
'translate'	The ROI can be translated (moved) within the drawing area but not reshaped. This value is not available for use with <code>images.roi.Point</code> objects.
'reshape'	The ROI can be reshaped but not translated. This value is not available for use with <code>images.roi.Point</code> objects or <code>images.roi.Rectangle</code> objects.

**Label – ROI label**

' ' (default) | character vector | string

ROI label, specified as a character vector or string. By default, the ROI has no label (' ').

## **LineWidth — Width of ROI edge**

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

## **Parent — ROI parent**

Axes object

ROI parent, specified as an Axes object. A UIAxes cannot be the parent of the ROI.

## **Position — Position of ROI**

[] (default) | *n*-by-2 numeric array

Position of ROI, specified as a *n*-by-2 array of the form [x1 y1; x2 y2], where *n* is the total number of vertices. Each row specifies the respective end-point of a line segment. You can also set this property by drawing or moving the line.

## **Selected — Selection state of ROI**

false (default) | true

Selection state of the ROI, specified as the logical value `true` or `false`. You can also set this property interactively. For example, clicking on the ROI selects the ROI and sets this property to `true`. Similarly, if you press **Ctrl** and click the ROI, the ROI is deselected and the value of this property is set to `false`.

## **SelectedColor — Color of ROI when selected**

'none' (default) | ColorSpec

Color of the ROI when selected, specified as a MATLAB ColorSpec (Color Specification). The intensities must be in the range [0,1]. If 'SelectedColor' is set to 'none', then the value of Color defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example: [0 1 0]

## **StripeColor — Color of ROI stripe**

'none' (default) | ColorSpec value

Color of the ROI stripe, specified as a MATLAB ColorSpec (Color Specification) value. By default, the edge of an ROI is solid colored. If you specify StripeColor, the

ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

### Tag — Tag to associate with the ROI

' ' (default) | character vector | string scalar

Tag to associate with the ROI, specified as a character vector or string. Use the tag value to find the ROI object in a hierarchy of objects using the `findobj` function.

### UIContextMenu — Context menu

ContextMenu object

Context menu, specified as a `ContextMenu` object. Use this property to display a custom context menu when you right-click the ROI. You can create a custom context menu by using the `uicontextmenu` function and then configuring context menu properties.

### UserData — Data to associate with ROI

any MATLAB data

Data to associate with the ROI, specified as any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

### Visible — ROI visibility

'on' (default) | 'off'

ROI visibility, specified as one of the values in this table.

Value	Description
'on'	Display the ROI (default).
'off'	Hide the ROI without deleting it. You still can access the properties of an invisible ROI.

## Methods

### Public Methods

```
<infotypegroup type="method"> beginDrawingFromPoint createMask draw </infotypegroup>
```

## Events

Event Name	Trigger	Event Data	Event Attributes
DeletingROI	ROI is about to be interactively deleted.	event.EventData	NotifyAccess: private  ListenAccess: public
DrawingStarted	ROI is about to be interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public
DrawingFinished	ROI has been interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public
MovingROI	ROI shape or location is being interactively changed.	images.roi.ROIMovingEventData	NotifyAccess: private  ListenAccess: public
ROIMoved	ROI shape or location has been interactively changed.	images.roi.ROIMovingEventData	NotifyAccess: private  ListenAccess: public



Event Name	Trigger	Event Data	Event Attributes
ROIClicked	ROI has been clicked.	images.roi.ROIClickedEventData	NotifyAccess: private  ListenAccess: public
AddingVertex	A vertex is about to be interactively added to the ROI.	event.EventData	NotifyAccess: private  ListenAccess: public
VertexAdded	A vertex has been interactively added to the ROI.	event.EventData	NotifyAccess: private  ListenAccess: public
DeletingVertex	A vertex is about to be interactively deleted from the ROI.	event.EventData	NotifyAccess: private  ListenAccess: public
VertexDeleted	A vertex has been interactively deleted from the ROI.	event.EventData	NotifyAccess: private  ListenAccess: public

## Examples

### Create Polyline ROI Non-interactively

Read an image into the workspace and display it.

```
I = imread('baby.jpg');
figure;
imshow(I)
```



Create a polyline ROI on the image, using properties to specify the location of vertices. For programmatically created ROIs, if you want the ROI drawn in a specific axes, you must specify that axes as an input argument. Otherwise, an instance of the `images.roi.Polyline` class is created but not displayed. In this example, specify the current axes (`gca`) to draw the ROI on the image in that axes.

```
h = images.roi.Polyline(gca, 'Position', [100 150; 200 250; 300 350; 150 450]);
```



1-1110

**See Also**

## **Topics**

“ROI Creation Overview”

**Introduced in R2018b**

## images.roi.Rectangle class

**Package:** images.roi

Rectangular region-of-interest

### Description

The `images.roi.Rectangle` class creates a rectangular region-of-interest (ROI) object. You call the `draw` method of the object to begin drawing the ROI interactively by using your mouse. You can also adjust the size and position of the ROI by using the mouse. The ROI also has a context menu that controls aspects of its appearance and behavior.

The `images.roi.Rectangle` class is a handle class.

### Creation

#### Description

`h = images.roi.Rectangle` creates an instance of the `images.roi.Rectangle` class with default properties.

`h = images.roi.Rectangle(ax)` creates an ROI in the axes specified by `ax`.

`h = images.roi.Rectangle( ____, Name, Value)` modifies the appearance and behavior of the ROI using one or more name-value pairs to specify properties. You can set properties using name-value pairs with or without specifying `ax`.

#### Input Arguments

**ax — Parent of ROI**

`gca` (default) | Axes object

Parent of the ROI, specified as an Axes object.

# Properties

## Public Properties

### AspectRatio — Aspect ratio of rectangle

1 (default) | positive numeric scalar

Aspect ratio of the rectangle, specified as a positive numeric scalar. The value of this property changes automatically when you draw or resize the rectangle. The `images.roi.Rectangle` object calculates this value as `height/width`

### Color — ROI color

blue (default) | `ColorSpec`

ROI color, specified as a MATLAB `ColorSpec` (Color Specification).

Example: 'green'

Example: 'g'

Example: [0 1 0]

### Deletable — ROI can be interactively deleted via context menu

true (default) | false

ROI can be interactively deleted via a context menu, specified as `true` or `false`. When the value is `true` (default), you can delete the ROI via the context menu. To disable this context menu item, set this property to `false`. When the value is set to `false`, you can still delete the ROI by calling the `delete` function, specifying the handle to the ROI as the input.

### DrawingArea — Area of axes in which you can interactively place ROI

'auto' (default) | 'unlimited' | [x,y,w,h]

Area of the axes in which you can interactively place the ROI, specified as one of the values in this table.

Value	Description
'auto'	The drawing area is the current axes limits (default).

Value	Description
'unlimited'	The drawing area has no boundary and ROIs can be drawn or dragged to extend beyond the axes limits.
[x,y,w,h]	The drawing area is restricted to a rectangular region beginning at (x,y), and extending to width w and height h.

**FaceAlpha — Transparency of ROI face**

0.2 (default) | numeric scalar

Transparency of the ROI face, specified as a numeric scalar value in the range [0 1]. When the value is 1, the ROI is fully opaque. When the value is 0, the ROI is completely transparent.

**FaceSelectable — ROI face can capture clicks**

true (default) | false

ROI face can capture clicks, specified as true or false. When true (default), the ROI face captures mouse clicks. When false, the ROI face does not capture mouse clicks.

**FixedAspectRatio — Aspect ratio remains constant during interaction**

false (default) | true

Aspect ratio remains constant during interaction, specified as true or false. When the value is true, the ROI maintains the aspect ratio when you draw or resize the rectangle. When the value is false (default), you can change the aspect ratio when drawing or resizing the rectangle. You can change the state of this property using the default context menu.

**HandleVisibility — Visibility of ROI handle in Children property of parent**

'on' (default) | 'off' | 'callback'

Visibility of the ROI handle in the Children property of the parent, specified as one of the values in this table.

Value	Description
'on'	The object handle is always visible (default).



Value	Description
'off'	The object handle is hidden at all times.
'callback'	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line.

### InteractionsAllowed – Interactivity of ROI

'all' (default) | 'none' | 'reshape' | 'translate'

Interactivity of the ROI, specified as one of the values in this table.

Value	Description
'all'	The ROI is fully interactable (default).
'none'	The ROI is not interactable, and no drag points are visible.
'translate'	The ROI can be translated (moved) within the drawing area but not reshaped. This value is not available for use with <code>images.roi.Point</code> objects.
'reshape'	The ROI can be reshaped but not translated. This value is not available for use with <code>images.roi.Point</code> objects or <code>images.roi.Rectangle</code> objects.

### Label – ROI label

'' (default) | character vector | string

ROI label, specified as a character vector or string. By default, the ROI has no label ('').

### LineWidth – Width of ROI edge

three times the number of points per screen pixel (default) | positive numeric scalar

Width of the ROI edge, specified as a positive numeric scalar in points. The default value is three times the number of points per screen pixel.

### Parent – ROI parent

Axes object

ROI parent, specified as an Axes object. A UIAxes cannot be the parent of the ROI.

### **Position — Position of ROI**

1-by-4 numeric array

Position of the ROI, specified as a 1-by-4 numeric array of the form `[xmin, ymin, width, height]`. `xmin` and `ymin` specify the location of the upper left corner of the rectangle. `width` and `height` specify the extent of the rectangle in two dimensions.

### **Rotatable — Ability of the rectangle to be rotated**

`false` (default) | `true`

Ability of the rectangle to be rotated, specified as `true` or `false`. When the value is `false` (default), the rectangle cannot be rotated. When the value is `true`, you can rotate the rectangle by clicking near the markers at the corners.

### **RotationAngle — Angle around center of rectangle**

0 (default) | numeric scalar

Angle around the center of the rectangle, specified as a numeric scalar. The angle is measured in degrees in a clockwise direction. The value of this property changes automatically when you draw or move the ROI.

The value of `RotationAngle` does not impact the values in `Position`. The `Position` property represents the initial position of the ROI, before rotation. To determine the location of a rotated ROI, use the `Vertices` property.

### **Selected — Selection state of ROI**

`false` (default) | `true`

Selection state of the ROI, specified as the logical value `true` or `false`. You can also set this property interactively. For example, clicking on the ROI selects the ROI and sets this property to `true`. Similarly, if you press **Ctrl** and click the ROI, the ROI is deselected and the value of this property is set to `false`.

### **SelectedColor — Color of ROI when selected**

'none' (default) | `ColorSpec`

Color of the ROI when selected, specified as a MATLAB `ColorSpec` (`Color Specification`). The intensities must be in the range `[0, 1]`. If 'SelectedColor' is set to 'none', then the value of `Color` defines the color of the ROI for all states, selected or not.

Example: 'green'

Example: 'g'

Example: [0 1 0]

### **StripeColor — Color of ROI stripe**

'none' (default) | ColorSpec value

Color of the ROI stripe, specified as a MATLAB ColorSpec (Color Specification) value. By default, the edge of an ROI is solid colored. If you specify StripeColor, the ROI edge is striped. The striping consists of a combination of the value specified by 'Color' and this value.

Example: 'green'

Example: 'g'

Example: [0 1 0]

### **Tag — Tag to associate with the ROI**

' ' (default) | character vector | string scalar

Tag to associate with the ROI, specified as a character vector or string. Use the tag value to find the ROI object in a hierarchy of objects using the findobj function.

### **UIContextMenu — Context menu**

ContextMenu object

Context menu, specified as a ContextMenu object. Use this property to display a custom context menu when you right-click the ROI. You can create a custom context menu by using the uicontextmenu function and then configuring context menu properties.

### **UserData — Data to associate with ROI**

any MATLAB data

Data to associate with the ROI, specified as any MATLAB data. For example, you can specify a scalar, vector, matrix, cell array, string, character array, table, or structure. MATLAB does not use this data.

### **Vertices — Locations of points on edge of rectangle**

*n*-by-2 array

Locations of points on the edge of the ROI, specified as an *n*-by-2 array.

This property is read-only.

**Visible — ROI visibility**

'on' (default) | 'off'

ROI visibility, specified as one of the values in this table.

Value	Description
'on'	Display the ROI (default).
'off'	Hide the ROI without deleting it. You still can access the properties of an invisible ROI.

## Methods

### Public Methods

`<infotypegroup type="method"> beginDrawingFromPoint createMask draw inROI  
</infotypegroup>`

## Events

Event Name	Trigger	Event Data	Event Attributes
DeletingROI	ROI is about to be interactively deleted.	event.EventData	NotifyAccess: private  ListenAccess: public
DrawingStarted	ROI is about to be interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public

Event Name	Trigger	Event Data	Event Attributes
DrawingFinished	ROI has been interactively drawn.	event.EventData	NotifyAccess: private  ListenAccess: public
MovingROI	ROI shape or location is being interactively changed.	images.roi.RectangleMovingEventData	NotifyAccess: private  ListenAccess: public
ROIMoved	ROI shape or location has been interactively changed.	images.roi.RectangleMovingEventData	NotifyAccess: private  ListenAccess: public
ROIClicked	ROI has been clicked.	images.roi.ROIClickedEventData	NotifyAccess: private  ListenAccess: public

## Examples

### Create Rectangular ROI Non-interactively

Read image into the workspace and display it.

```
I = imread('baby.jpg');
figure
imshow(I)
```



Create a rectangular ROI on the image, using the `Position` parameter to specify its location and size. The example also specifies that the edge of the rectangle is a striped line. For programmatically created ROIs, if you want the ROI drawn in a specific axes, you must specify that axes as an input argument. Otherwise, an instance of the `images.roi.Rectangle` class is created but not displayed. In this example, specify the current axes (`gca`) to draw the ROI on the image in that axes.

```
h = images.roi.Rectangle(gca, 'Position', [500,500,1000,1000], 'StripeColor', 'r');
```



1-1122

**See Also**



## **Topics**

“ROI Creation Overview”

**Introduced in R2018b**

## **images.roi.CircleMovingEventData class**

**Package:** images.roi

Event data passed when the circle ROI is moving

### **Description**

The `images.roi.CircleMovingEventData` class is the class passed to listeners when a Circle ROI is moving. When the ROI class triggers an event using the `notify handle` class method, MATLAB assigns values to the properties of an `images.roi.CircleMovingEventData` object and passes that object to the listener callback function (the event handler).

The `images.roi.CircleMovingEventData` class is a handle class.

### **Class Attributes**

`ConstructOnLoad`

    true

`HandleCompatible`

    true

For information on class attributes, see “Class Attributes” (MATLAB).

### **Creation**

The `notify handle` class method creates an `images.roi.CircleMovingEventData` object when called to trigger an event. `images.roi.CircleMovingEventData` accepts no input arguments.

## Properties

### Public Properties

#### Source — Event source

object

Event source, specified as a handle to the object that triggered the event.

#### Attributes:

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

#### EventName — Name of event

character vector

Name of the event, specified as a character vector.

#### Attributes:

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

Data Types: char

**PreviousCenter — Position before ROI moved**

two-element numeric vector

Position before ROI moved, specified as a two-element numeric vector of the form [x y].

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

**CurrentCenter — Position after ROI moved**

two-element numeric vector

Position after ROI moved, specified as a two-element numeric vector of the form [x y].

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

**PreviousRadius — Radius before change in size**

numeric scalar

Radius before change in size, specified as a numeric scalar.

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

**CurrentRadius — Radius after change in size**

numeric scalar

Radius after change in size, specified as a numeric scalar.

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

## Examples

**Access Event Data**

This callback function gets the event source object handle and the event name and other properties from the `images.roi.CircleMovingEventData` object passed to it when the event is triggered.

```
function myCallbk(s,evtData)
    eventSource = evtData.Source;
    eventName = evtData.EventName;
    previousCenter = evtData.PreviousCenter;
    currentCenter = evtData.CurrentCenter;
    previousRadius = evtData.PreviousRadius;
    currentRadius = evtData.CurrentRadius;
end
```

## See Also

`images.roi.Circle`

**Introduced in R2018b**

# images.roi.EllipseMovingEventData class

**Package:** images.roi

Event data passed when the ellipse ROI is moving

## Description

The `images.roi.EllipseMovingEventData` class is the class passed to listeners when an Ellipse ROI is moving. When the ROI class triggers an event using the `notify handle` class method, MATLAB assigns values to the properties of an `images.roi.EllipseMovingEventData` object and passes that object to the listener callback function (the event handler).

The `images.roi.EllipseMovingEventData` class is a `handle` class.

## Class Attributes

`ConstructOnLoad`

    true

`HandleCompatible`

    true

For information on class attributes, see “Class Attributes” (MATLAB).

## Creation

The `notify handle` class method creates an `images.roi.EllipseMovingEventData` object when called to trigger an event. `images.roi.EllipseMovingEventData` accepts no input arguments.

## Properties

### Public Properties

#### Source — Event source

object

Event source, specified as a handle to the object that triggered the event.

#### Attributes:

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

#### EventName — Name of event

character vector

Name of event, specified as a character vector.

#### Attributes:

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

Data Types: char



**PreviousCenter — Position before ROI moved**

two-element numeric vector

Position before ROI moved, specified as a two-element numeric vector of the form [m n].

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

**CurrentCenter — Position after ROI moved**

two-element numeric vector

Position after ROI moved, specified as a two-element numeric vector of the form [m n].

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

**PreviousSemiAxes — Lengths of semiaxes before ROI was reshaped**

two-element numeric vector

Lengths of semiaxes before ROI was reshaped, specified as a two-element numeric vector.

**Attributes:**

GetAccess  
    public  
SetAccess  
    private  
GetObservable  
    true  
SetObservable  
    true

**CurrentSemiAxes — Lengths of semiaxes after ROI was reshaped**

two-element numeric vector

Lengths of semiaxes after ROI was reshaped, specified as a two-element numeric vector.

**Attributes:**

GetAccess  
    public  
SetAccess  
    private  
GetObservable  
    true  
SetObservable  
    true

**PreviousRotationAngle — Orientation of ROI before rotation**

numeric scalar

Orientation of ROI before rotation, specified as a numeric scalar, measured in degrees.

**Attributes:**

GetAccess  
    public  
SetAccess  
    private

GetObservable

true

SetObservable

true

### **CurrentRotationAngle – Orientation of ROI after rotation**

numeric scalar

Position after ROI moved, specified as a numeric scalar, measured in degrees.

#### **Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

## **Examples**

### **Access Event Data**

This callback function gets the event source object handle and the event name and other properties from the `images.roi.EllipseMovingEventData` object passed to it when the event is triggered.

```
function myCallbk(s,evtData)
    eventSource = evtData.Source;
    eventName = evtData.EventName;
    previousCenter = evtData.PreviousCenter;
    currentCenter = evtData.CurrentCenter;
    previousSemiAxes = evtData.PreviousRadius;
    currentSemiAxes = evtData.CurrentRadius;
    previousRotationAngle = evtData.PreviousRotationAngle;
```

```
        currentRotationAngle = evtData.CurrentRotationAngle;  
end
```

## **See Also**

`images.roi.Ellipse`

**Introduced in R2018b**

# images.roi.RectangleMovingEventData class

**Package:** images.roi

Event data passed when the rectangle ROI is moving

## Description

The `images.roi.RectangleMovingEventData` class is the class passed to listeners when a Rectangle ROI is moving. When the ROI class triggers an event using the `notify` handle class method, MATLAB assigns values to the properties of an `images.roi.RectangleMovingEventData` object and passes that object to the listener callback function (the event handler).

The `images.roi.RectangleMovingEventData` class is a handle class.

## Class Attributes

`ConstructOnLoad`

true

`HandleCompatible`

true

For information on class attributes, see “Class Attributes” (MATLAB).

## Creation

The `notify` handle class method creates an `images.roi.RectangleMovingEventData` object when called to trigger an event. `images.roi.RectangleMovingEventData` accepts no input arguments.

## Properties

### Public Properties

#### **Source — Event source**

object

Event source, specified as a handle to the object that triggered the event.

#### **Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

#### **EventName — Name of event**

character vector

Name of the event, specified as a character vector.

#### **Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

**PreviousPosition — Position before ROI moved**

two-element numeric vector

Position before ROI moved, specified as a two-element numeric vector of the form [m n].

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

**CurrentPosition — Position after ROI moved**

two-element numeric vector

Position after ROI moved, specified as a two-element numeric vector of the form [m n].

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

**PreviousRotationAngle — Orientation before ROI moved**

numeric scalar

Orientation before ROI moved, specified as a numeric scalar, measured in degrees.

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

**CurrentRotationAngle — Orientation after ROI moved**

numeric scalar

Orientation after ROI moved, specified as a numeric scalar, measured in degrees.

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

## Examples

**Access Event Data**

This callback function gets the event source object handle, the event name, and other properties from the `images.roi.RectangleMovingEventData` object passed to it when the event is triggered.



```
function myCallbk(s,evtData)
  eventSource = evtData.Source;
  eventName = evtData.EventName;
  previousCenter = evtData.PreviousCenter;
  currentCenter = evtData.CurrentCenter;
  previousRotationAngle = evtData.PreviousRotationAngle;
  currentRotationAngle = evtData.CurrentRotationAngle;
end
```

## See Also

`images.roi.Rectangle`

**Introduced in R2018b**

## **images.roi.ROIClickedEventData class**

**Package:** images.roi

Event data passed when ROI is clicked

### **Description**

The `images.roi.ROIClickedEventData` class is the class passed to listeners when a region-of-interest (ROI) is clicked. When the ROI class triggers an event using the `notify` handle class method, MATLAB assigns values to the properties of an `images.roi.ROIClickedEventData` object and passes that object to the listener callback function (the event handler).

The `images.roi.ROIClickedEventData` class is a handle class.

### **Class Attributes**

`ConstructOnLoad`

    true

`HandleCompatible`

    true

For information on class attributes, see “Class Attributes” (MATLAB).

### **Creation**

The `notify` handle class method creates an `images.roi.ROIClickedEventData` object when called to trigger an event. `images.roi.ROIClickedEventData` does not accept input arguments.

# Properties

## Public Properties

### Source — Event source

object

Event source object, specified as a handle to the object that triggered the event.

#### Attributes:

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

### EventName — Name of event

character vector

Name of the event, specified as a character vector.

#### Attributes:

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

**SelectionMode — Type of selection**

character vector

Type of selection, specified as one of the following character vectors.

<b>SelectionMode Value</b>	<b>Description</b>
'left'	Left mouse-click
'right'	Right mouse-click
'double'	Double-click
'shift'	Shift-left mouse-click
'ctrl'	Control-left mouse-click

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

**SelectedPart — Part of ROI that was clicked**

character vector

Part of the ROI that was clicked, specified as one of the character vectors in this table.

<b>SelectedPart Value</b>	<b>Description</b>
'edge'	Clicked edge of ROI.
'face'	Clicked face of ROI.
'label'	Clicked ROI label.
'marker'	Clicked marker used to reshape the ROI.

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

## Examples

**Access Event Data**

This callback function gets the event source object handle, the event name, and other properties from the `images.roi.ROIClickedEventData` object passed to it when the event is triggered.

```
function myCallbk(s,evtData)
    eventSource = evtData.Source;
    eventName = evtData.EventName;
    selectionType = evtData.SelectionType;
    selectedPart = evtData.SelectedPart;
end
```

## See Also

`images.roi.AssistedFreehand` | `images.roi.Circle` | `images.roi.Ellipse` |  
`images.roi.Freehand` | `images.roi.Line` | `images.roi.Point` |  
`images.roi.Polygon` | `images.roi.Polyline` | `images.roi.Rectangle`

**Introduced in R2018b**

## **images.roi.ROIMovingEventData class**

**Package:** images.roi

Event data passed when ROI is moving

### **Description**

The `images.roi.ROIMovingEventData` class is the class passed to listeners when a region-of-interest (ROI) is moving. When the ROI class triggers an event using the `notify handle` class method, MATLAB assigns values to the properties of an `images.roi.ROIMovingEventData` object and passes that object to the listener callback function (the event handler).

The `images.roi.ROIMovingEventData` class is a `handle` class.

### **Class Attributes**

`ConstructOnLoad`

    true

`HandleCompatible`

    true

For information on class attributes, see “Class Attributes” (MATLAB).

### **Creation**

The `notify handle` class method creates an `images.roi.ROIMovingEventData` object when called to trigger an event. `images.roi.ROIMovingEventData` does not accept input arguments. Subclasses of `event.EventData` cannot pass arguments to the superclass constructor.

# Properties

## Public Properties

### Source — Event source

object

Event source, specified as a handle to the object that triggered the event.

#### Attributes:

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

### EventName — Name of event

character vector

Name of the event, specified as a character vector.

#### Attributes:

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

Data Types: char

**PreviousPosition — Position before ROI moved**

two-element numeric vector

Position before the ROI moved, specified as a two-element numeric vector of the form [x y].

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

Data Types: char

**CurrentPosition — Position after ROI moved**

two-element numeric vector

Position after the ROI moved, specified as a two-element numeric vector of the form [x y].

**Attributes:**

GetAccess

public

SetAccess

private

GetObservable

true

SetObservable

true

Data Types: char



## Examples

### Access Event Data

This callback function gets the event source object handle, the event name, and other properties from the `images.roi.ROIMovingEventData` object passed to it when the event is triggered.

```
function myCallbk(s,evtData)
    eventSource = evtData.Source;
    eventName = evtData.EventName;
    previousPosition = evtData.PreviousPosition;
    currentPosition = evtData.CurrentPosition;
end
```

### See Also

`images.roi.AssistedFreehand` | `images.roi.Circle` | `images.roi.Ellipse` |  
`images.roi.Freehand` | `images.roi.Line` | `images.roi.Point` |  
`images.roi.Polygon` | `images.roi.Polyline` | `images.roi.Rectangle`

**Introduced in R2018b**

## beginDrawingFromPoint

**Package:** images.roi

Begin drawing ROI from specified point

### Syntax

```
beginDrawingFromPoint(ROI,[x y])
```

### Description

`beginDrawingFromPoint(ROI,[x y])` enters interactive mode to draw the shape for object ROI starting at location (x,y) in the axes. This method is intended to be used within the `ButtonDownFcn` callback of an `Image` or `Axes` object.

### Examples

#### Draw Line ROI When Button Pressed

Create a new script called `sampleDrawLine.m`. Inside the script, copy and paste this code, then save the file.

```
hIm = imshow(imread('coins.png'));  
hIm.ButtonDownFcn = @(~,~) buttonPressedCallback(hIm.Parent);  
  
function buttonPressedCallback(hAx)  
    cp = hAx.CurrentPoint;  
    cp = [cp(1,1) cp(1,2)];  
    obj = images.roi.Line('Parent',hAx,'Color',rand([1,3]));  
    beginDrawingFromPoint(obj,cp);  
end
```

Return to the MATLAB command window. Run the script by entering the command:

```
sampleDrawLine
```

An image of coins is displayed in a figure window. Each time you click the mouse over the figure, the script executes the callback function, `buttonPressedCallback`. The script begins drawing a new ROI starting from the pixel you clicked.

## Input Arguments

### ROI — Region of interest

ROI object

Region of interest, specified as an ROI object of one of the following types:

<code>images.roi.AssistedFreehand</code>	<code>images.roi.Freehand</code>	<code>images.roi.Polygon</code>
<code>images.roi.Circle</code>	<code>images.roi.Line</code>	<code>images.roi.Polyline</code>
<code>images.roi.Ellipse</code>	<code>images.roi.Point</code>	<code>images.roi.Rectangle</code>

### [x y] — Starting point in axes

numeric array

Starting point in the axes, specified as a numeric array.

## See Also

`draw` | `drawassisted` | `drawcircle` | `drawellipse` | `drawfreehand` | `drawline` | `drawpoint` | `drawpolygon` | `drawpolyline` | `drawrectangle`

**Introduced in R2018b**

## createMask

**Package:** images.roi

Create binary mask image from ROI

### Syntax

```
bw = createMask(ROI)
bw = createMask(ROI,m,n)
bw = createMask(ROI,I)
```

### Description

`bw = createMask(ROI)` returns a binary mask image with pixels inside the ROI set to `true` and pixels outside the ROI set to `false`.

`bw = createMask(ROI,m,n)` returns a binary mask image that is size `[m,n]`.

`bw = createMask(ROI,I)` returns a binary mask image that is the size of the image `I`.

### Examples

#### Create Mask From Ellipse ROI

Read image into the workspace and display it.

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



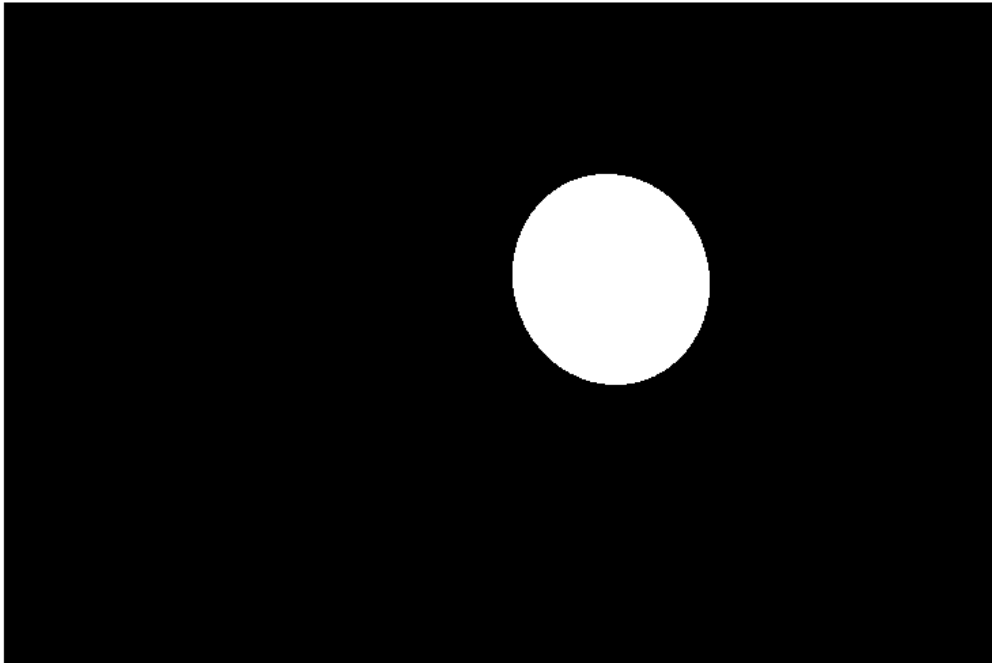
Draw an elliptical ROI on the image, using the `Center` parameter to specify the location of the ellipse and the `SemiAxes` parameter to specify the shape of the ellipse. The example also specifies that the edge of the ellipse is a striped line.

```
h = drawellipse('Center',[447 204],'SemiAxes',[78 72], ...  
               'RotationAngle',287,'StripeColor','m');
```



Get a binary mask from the ROI. Pixels inside the ROI are `true` and pixels outside the ROI are `false`. Display the mask.

```
mask = createMask(h);  
imshow(mask)
```



## Input Arguments

### **ROI — Region of interest**

ROI object

Region of interest, specified as an ROI object of one of the following types:

<code>images.roi.AssistedFreehand</code>	<code>images.roi.Freehand</code>	<code>images.roi.Polyline</code>
<code>images.roi.Circle</code>	<code>images.roi.Line</code>	<code>images.roi.Rectangle</code>
<code>images.roi.Ellipse</code>	<code>images.roi.Polygon</code>	

**m — Row dimension of mask image**

positive integer

Row dimension of the mask image, specified as a positive integer.

**n — Column dimension of mask image**

positive integer

Column dimension of the mask image, specified as a positive integer.

**I — Input image**

numeric array

Input image, specified as a numeric array.

## Output Arguments

**bw — Binary mask image**

logical array

Binary mask image, returned as a logical array.

Data Types: `logical`

## Tips

- If you specify a mask size that does not match the size of the image associated with the ROI object, then `createMask` crops or zero-pads the mask to the specified size. The image associated with the ROI object is `ROI.Parent.Children(2,1).CData`.

## See Also

`drawassisted` | `drawcircle` | `drawellipse` | `drawfreehand` | `drawline` | `drawpolygon` | `drawpolyline` | `drawrectangle` | `inROI`

**Introduced in R2018b**



# draw

**Package:** images.roi

Begin drawing ROI interactively

## Syntax

```
draw(ROI)
```

## Description

`draw(ROI)` enters interactive mode to draw the shape for object `ROI`. If no parent has been specified for `ROI`, then `draw` uses the current axes (`gca`).

## Examples

### Draw ROI Interactively

Read an image into the workspace and display it.

```
I = imread('wagon.jpg');  
figure  
imshow(I);
```



Draw a triangular ROI on the image interactively. To improve the visibility of the ROI edge, specify a thick line width and bright cyan color of the ROI edge.

```
p = drawpolygon('LineWidth',7,'Color','cyan');
```



Get the coordinates of the vertices.

`p.Position`

`ans =`

```
284.7500 725.5000
331.2500 871.0000
359.7500 707.5000
```

The spokes of the wheels define many other triangles. Suppose you want to get the vertices of a second triangle. You can use the `draw` function to start over and begin drawing a new polygonal ROI interactively. The line width and color parameters of the ROI are preserved.

`draw(p)`



```
p.Position
```

```
ans =
```

```
398.7500  710.5000
377.7500  865.0000
461.7500  734.5000
```

## Input Arguments

### ROI — Region of interest

ROI object

Region of interest, specified as an ROI object of one of the following types:

<code>images.roi.AssistedFreehand</code>	<code>images.roi.Freehand</code>	<code>images.roi.Polygon</code>
<code>images.roi.Circle</code>	<code>images.roi.Line</code>	<code>images.roi.Polyline</code>
<code>images.roi.Ellipse</code>	<code>images.roi.Point</code>	<code>images.roi.Rectangle</code>

## See Also

`beginDrawingFromPoint` | `drawassisted` | `drawcircle` | `drawellipse` | `drawfreehand` | `drawline` | `drawpoint` | `drawpolygon` | `drawpolyline` | `drawrectangle`

**Introduced in R2018b**

## inROI

**Package:** images.roi

Query if points are located in ROI

## Syntax

```
tf = inROI(ROI,x,y)
```

## Description

`tf = inROI(ROI,x,y)` returns a logical array, `tf`, that indicates whether points with coordinates `(x,y)` are inside the ROI.

## Examples

### Query Points Inside Rectangular ROI

Read an image into the workspace and display it.

```
I = imread('trailer.jpg');  
figure  
imshow(I)
```

Draw a rectangular ROI on the image, using the `Position` argument to specify the position of the rectangle as `[xmin,ymin,width,height]`.

```
h = drawrectangle('Position',[190 308 682 276],'StripeColor','r');
```





Specify the  $x$ - and  $y$ -coordinates of three points. The last point is the upper left corner of the rectangular ROI.

```
xcoords = [335 335 190];  
ycoords = [200 400 308];
```

Query if the three points are inside the ROI.

```
tf = inROI(h,xcoords,ycoords)
```

*tf = 3x1 logical array*

```
0  
1  
1
```

## Input Arguments

### **ROI — Region of interest**

ROI object

Region of interest, specified as an ROI object of one of the following types:

<code>images.roi.AssistedFreehand</code>	<code>images.roi.Ellipse</code>	<code>images.roi.Polygon</code>
<code>images.roi.Circle</code>	<code>images.roi.Freehand</code>	<code>images.roi.Rectangle</code>

### **x — X-coordinates of query points**

numeric vector

X-coordinates of the query points, specified as a numeric vector.

### **y — Y-coordinates of query points**

numeric vector

Y-coordinates of the query points, specified as a numeric vector.

## Output Arguments

### **tf — Points are inside ROI**

logical array

Points are inside the ROI, returned as a logical array. The array is the same length as the input arrays `x` and `y`. Elements of the logical array set to `true` indicate that the corresponding query point is inside the ROI.

## See Also

`createMask` | `drawassisted` | `drawcircle` | `drawellipse` | `drawfreehand` | `drawpolygon` | `drawrectangle`

**Introduced in R2018b**

# 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 `C` to each combination.

`Y = imapplymatrix( ____,output_type)` returns the result of the linear combination in an array of type `output_type`.

## 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

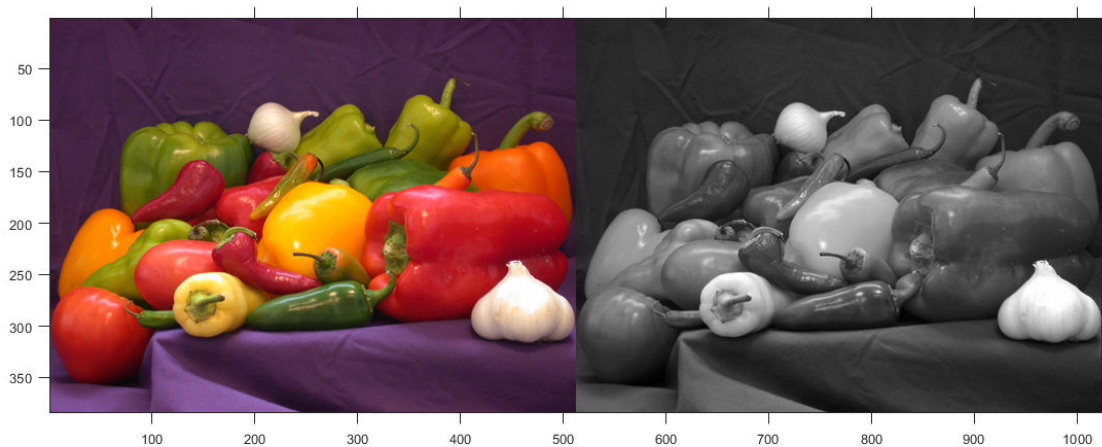
```
M = [0.30, 0.59, 0.11];
```

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')
```



## Input Arguments

### **M** — Weighting coefficients for each color channel

*q*-by-*p* numeric array

Weighting coefficients for each color channel, specified as a *q*-by-*p* numeric array. *p* is the length of the third dimension of *X*. In other words,  $p = \text{size}(X, 3)$ . *q* is in the range  $[1, p]$ .

Data Types: double

### **X** — Input image

*m*-by-*n*-by-*p* numeric array

Input image, specified as an *m*-by-*n*-by-*p* numeric array.

**C — Constant to add to each channel**

*q*-element numeric vector

Constant to add to each channel during the linear combination, specified as *q*-element numeric vector, where *q* is the 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: 'double', 'single', 'uint8', 'uint16', 'uint32', 'int8', 'int16', or 'int32'.

Data Types: char | string

## Output Arguments

**Y — Output image**

numeric array

Output image comprised of the linear combination of the rows of *M* with the color channels of *X*, returned as a numeric array. If `output_type` is not specified, the data type of *Y* is the same as the data type of *X*.

## 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> .
	<b>Note</b> 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.

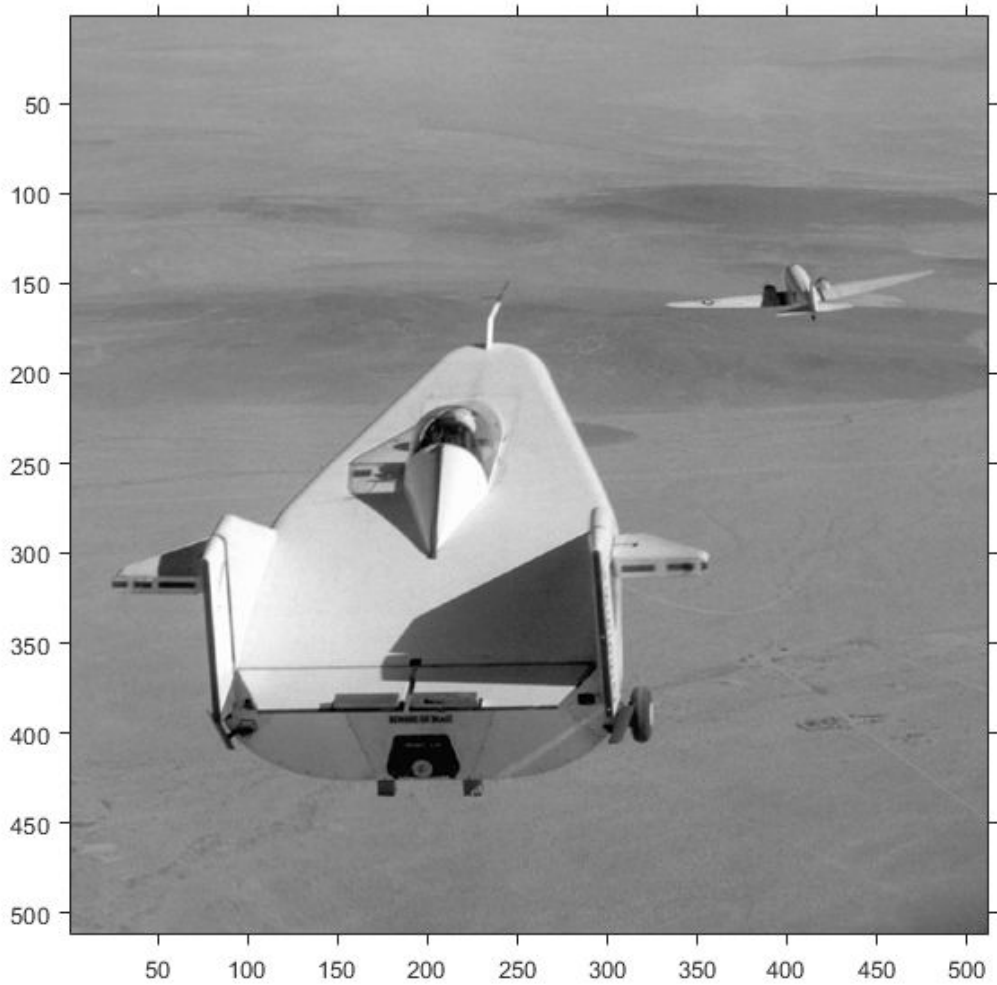
`attrs = imattributes(imgmodel)` returns information about the image represented by the image model object, `imgmodel`.

## 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**

# imblatfilt

Bilateral filtering of images with Gaussian kernels

## Syntax

```
J = imblatfilt(I)
J = imblatfilt(I,degreeOfSmoothing)
J = imblatfilt(I,degreeOfSmoothing,spatialSigma)
J = imblatfilt( __ ,Name,Value)
```

## Description

`J = imblatfilt(I)` applies an edge-preserving Gaussian bilateral filter to the grayscale or RGB image, `I`.

`J = imblatfilt(I,degreeOfSmoothing)` specifies the amount of smoothing. When `degreeOfSmoothing` is a small value, `imblatfilt` smooths neighborhoods with small variance (uniform areas) but does not smooth neighborhoods with large variance, such as strong edges. When the value of `degreeOfSmoothing` increases, `imblatfilt` smooths both uniform areas and neighborhoods with larger variance.

`J = imblatfilt(I,degreeOfSmoothing,spatialSigma)` also specifies the standard deviation, `spatialSigma`, of the spatial Gaussian smoothing kernel. Larger values of `spatialSigma` increase the contribution of more distant neighboring pixels, effectively increasing the neighborhood size.

`J = imblatfilt( __ ,Name,Value)` uses name-value pairs to change the behavior of the bilateral filter.

## Examples

## Smooth Grayscale Image Using Bilateral Filtering

Read and display a grayscale image. Observe the horizontal striation artifact in the sky region.

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



Inspect a patch of the image from the sky region. Compute the variance of the patch, which approximates the variance of the noise.

```
patch = imcrop(I,[170, 35, 50 50]);  
imshow(patch)
```



```
patchVar = std2(patch)^2;
```

Filter the image using bilateral filtering. Set the degree of smoothing to be larger than the variance of the noise.

```
DoS = 2*patchVar;  
J = imblatfilt(I,DoS);  
imshow(J)  
title(['Degree of Smoothing: ',num2str(DoS)])
```

**Degree of Smoothing: 51.9395**



The striation artifact is reduced, but not eliminated. To improve the smoothing, increase the value of `spatialSigma` to 2 so that distant neighboring pixels contribute more to the Gaussian smoothing kernel. This effectively increases the spatial extent of the bilateral filter.

```
K = imbilatfilt(I,DoS,2);  
imshow(K)  
title(['Degree of Smoothing: ',num2str(DoS),', Spatial Sigma: 2'])
```

**Degree of Smoothing: 51.9395, Spatial Sigma: 2**



The stration artifact in the sky is successfully removed. The sharpness of strong edges such as the silhouette of the man, and textured regions such as the grass in the foreground of the image, have been preserved.

### **Smooth Color Image Using Bilateral Filtering**

Read an RGB image.

```
imRGB = imread('coloredChips.png');  
imshow(imRGB)
```



Convert the image to the L\*a\*b colorspace, so that the bilateral filter smooths perceptually similar colors.

```
imLAB = rgb2lab(imRGB);
```

Extract an L\*a\*b patch that contains no sharp edges. Compute the variance in the Euclidean distance from the origin, in the L\*a\*b space.

```
patch = imcrop(imLAB,[34,71,60,55]);  
patchSq = patch.^2;  
edist = sqrt(sum(patchSq,3));  
patchVar = std2(edist).^2;
```

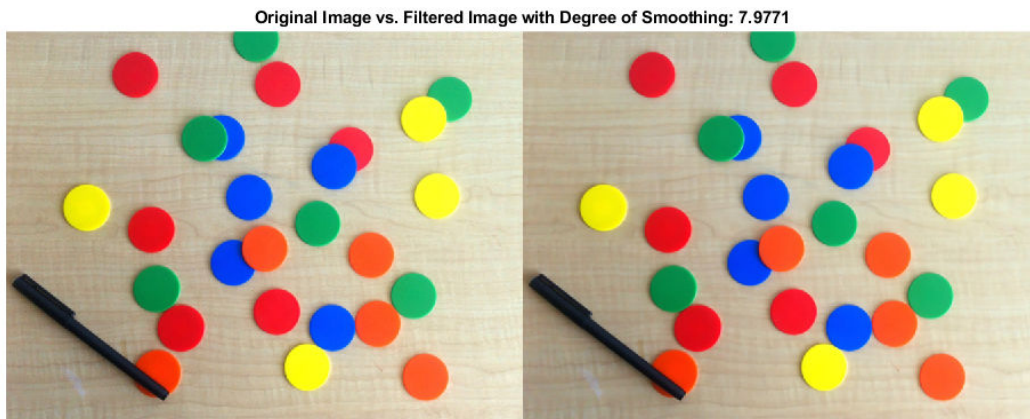


Filter the image in the L\*a\*b\* color space using bilateral filtering. Set the DegreeOfSmoothing value to be higher than the variance of the patch.

```
DoS = 2*patchVar;
smoothedLAB = imbilatfilt(imLAB,DoS);
```

Convert the image back to the RGB color space, and display the smoothed image.

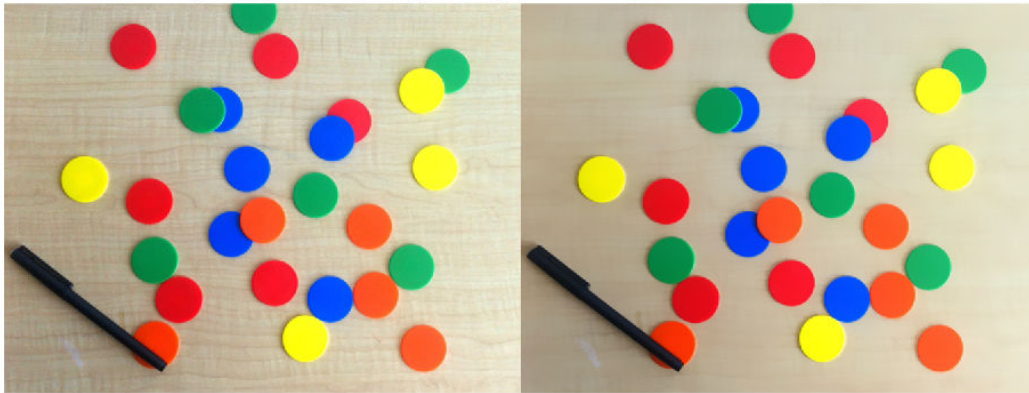
```
smoothedRGB = lab2rgb(smoothedLAB,'Out','uint8');
montage({imRGB,smoothedRGB})
title(['Original Image vs. Filtered Image with Degree of Smoothing: ',num2str(DoS)])
```



The colors of the chips and black pen appear more uniform, but the horizontal grains in the table are still visible. Increase the spatial extent of the filter so that the effective neighborhood of the filter spans the space between the horizontal grains (this distance is approximately seven pixels). Also increase the DegreeOfSmoothing to smooth these regions more aggressively.

```
DoS2 = 4*patchVar;
smoothedLAB2 = imbilatfilt(imLAB,DoS2,7);
smoothedRGB2 = lab2rgb(smoothedLAB2,'Out','uint8');
montage({imRGB,smoothedRGB2})
title(['Original Image vs. Filtered Image with Degree of Smoothing: ',num2str(DoS),' and ',num2str(7)])
```

Original Image vs. Filtered Image with Degree of Smoothing: 7.9771 and Spatial Sigma: 7



The color of the wooden table is more uniform with the larger neighborhood and larger degree of smoothing. The edge sharpness of the chips and pen is preserved.

## Input Arguments

### **I** — Image to filter

2-D grayscale image | 2-D color image

Image to filter, specified as a 2-D grayscale image of size  $m$ -by- $n$  or a 2-D color image of size  $m$ -by- $n$ -by-3.

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

### **degreeOfSmoothing** — Degree of smoothing

positive number

Degree of smoothing, specified as a positive number. The default value of `degreeOfSmoothing` depends on the data type of image `I`, and is calculated as  $0.01 * \text{diff}(\text{getrangefromclass}(I)).^2$ . For example, the default degree of smoothing is 650.25 for images of data type `uint8`, and the default is 0.01 for images of data type `double` with pixel values in the range [0, 1].

**spatialSigma — Standard deviation of spatial Gaussian smoothing kernel**

1 (default) | positive number

Standard deviation of spatial Gaussian smoothing kernel, specified as a positive number.

**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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `imbatfilt(I, 'NeighborhoodSize', 7)` performs bilateral filtering on image `I` using a 7-by-7 pixel neighborhood.

**NeighborhoodSize — Neighborhood size**

odd-valued positive integer

Neighborhood size, specified as the comma-separated pair consisting of `'NeighborhoodSize'` and an odd-valued positive integer. By default, the neighborhood size is  $2 * \text{ceil}(2 * \text{SpatialSigma}) + 1$  pixels

Example: `'NeighborhoodSize', 7`

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

**Padding — Padding**

`'replicate'` (default) | `'symmetric'` | numeric scalar

Padding, specified as the comma-separated pair consisting of `'Padding'` and one of these values.

Value	Description
<code>'replicate'</code>	Input array values outside the bounds of the array are assumed to equal the nearest array border value.
<code>'symmetric'</code>	Input array values outside the bounds of the array are computed by mirror-reflecting the array across the array border.

Value	Description
numeric scalar, $x$	Input image values outside the bounds of the image are assigned the value $x$ .

Example: 'Padding', 'symmetric'

Example: 'Padding', 128

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

## Output Arguments

### J — Filtered image

numeric array

Filtered image, returned as a numeric array of the same size and data type as the input image, I.

## Tips

- The value of `degreeOfSmoothing` corresponds to the variance of the Range Gaussian kernel of the bilateral filter [1]. The Range Gaussian is applied on the Euclidean distance of a pixel value from the values of its neighbors.
- To smooth perceptually close colors of an RGB image, convert the image to the CIE L\*a\*b space using `rgb2lab` before applying the bilateral filter. To view the results, convert the filtered image to RGB using `lab2rgb`.
- Increasing `spatialSigma` increases `NeighborhoodSize`, which increases the filter execution time. You can specify a smaller `NeighborhoodSize` to trade accuracy for faster execution time.

## References

- [1] Tomasi, C., and R. Manduchi. "Bilateral Filtering for Gray and Color Images". *Proceedings of the 1998 IEEE® International Conference on Computer Vision*. Bombay, India. Jan 1998, pp. 836-846.

## See Also

`imdiffusefilt` | `imfilter` | `imgaussfilt` | `imguidedfilter` | `imnlmfilt` |  
`locallapfilt` | `nlfilter`

**Introduced in R2018a**

## imbinarize

Binarize 2-D grayscale image or 3-D volume 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 2-D or 3-D grayscale 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 [1]. `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`: 'global' or 'adaptive'.

`BW = imbinarize(I,T)` creates a binary image from image `I` using the threshold value `T`. `T` can be a global image threshold, specified as a scalar luminance value, or a locally adaptive threshold, specified as a matrix of luminance values.

`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.

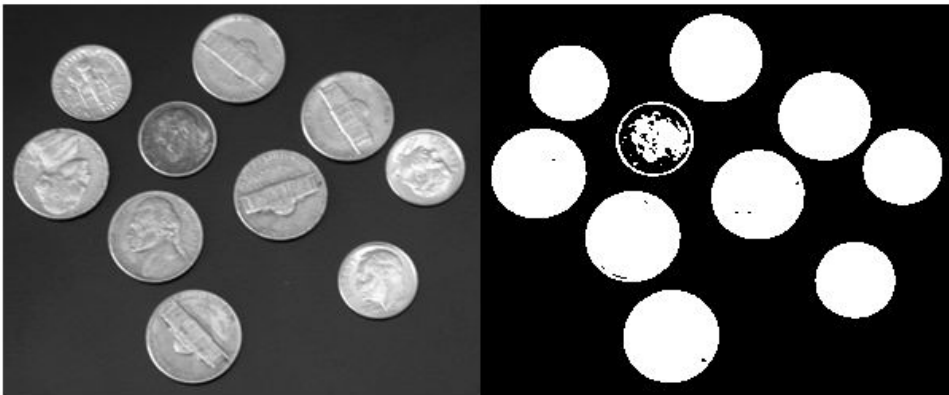
```
I = imread('coins.png');
```

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.

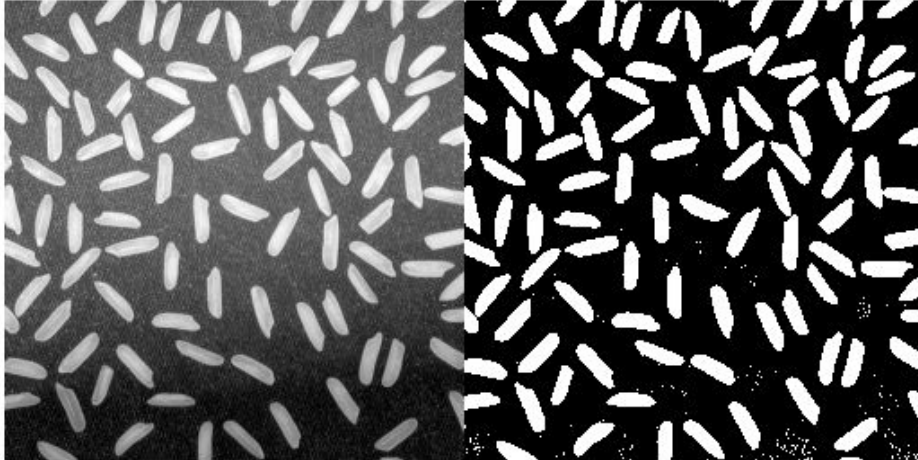
```
I = imread('rice.png');
```

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

## 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]
```

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 22 1 1]
```

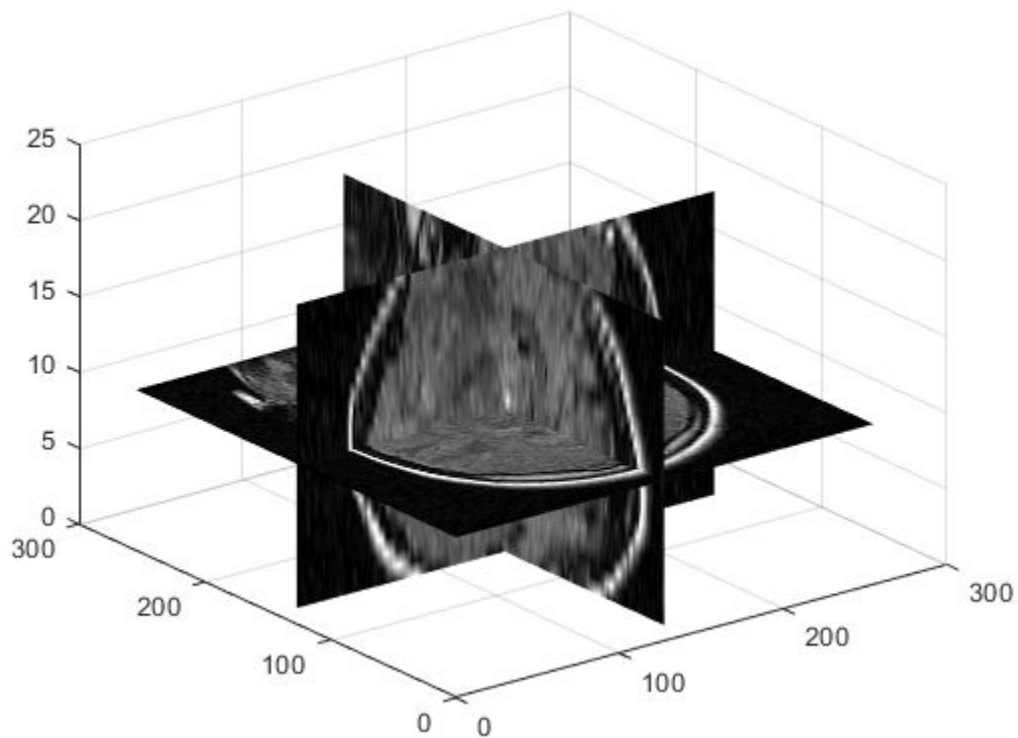
## Binarize 3-D Volume Using Global Thresholding

Load 3-D grayscale intensity data into the workspace.

```
load mrystack;
V = mrystack;
```

View the 3-D volume.

```
figure
slice(double(V), size(V,2)/2, size(V,1)/2, size(V,3)/2)
colormap gray
shading interp
```

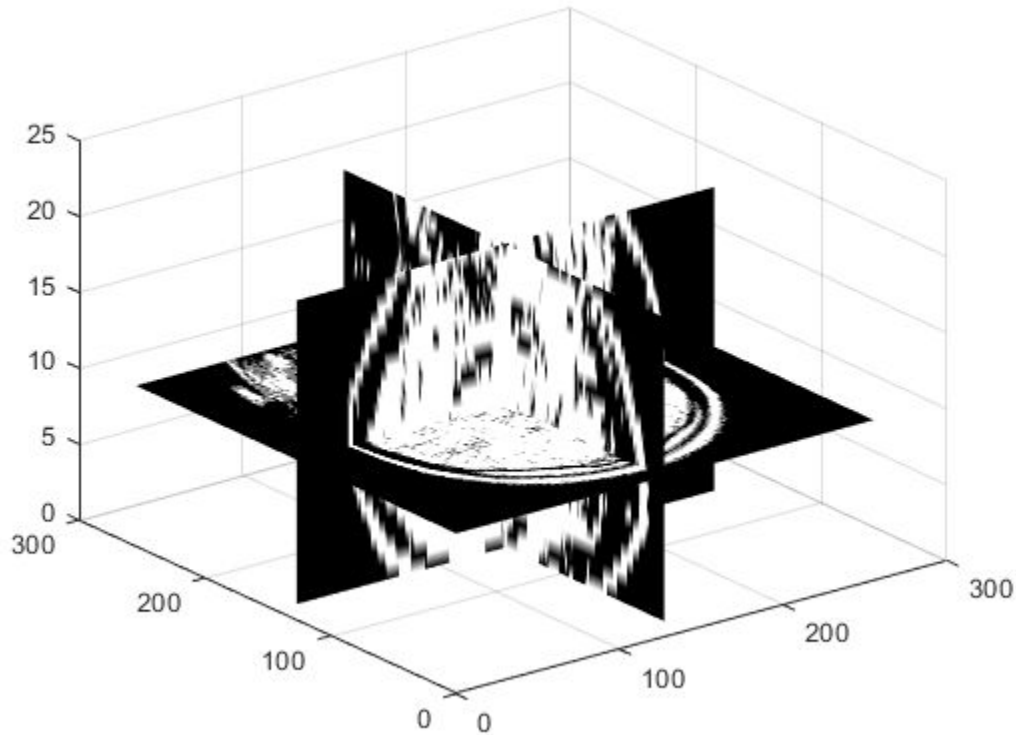


Convert the intensity volume into a 3-D binary volume.

```
J = imbinarize(V);
```

View the 3-D binary volume.

```
figure  
slice(double(J),size(J,2)/2,size(J,1)/2,size(J,3)/2)  
colormap gray  
shading interp
```



## Input Arguments

### **I** — Input image

2-D grayscale image | 3-D grayscale volume

Input image, specified as a 2-D grayscale image or a 3-D grayscale volume.

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

### **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> might not be localized to the neighborhood around <code>Inf</code> and <code>NaN</code> pixels.

Data Types: `char` | `string`

### T — Threshold

numeric scalar | numeric array

Threshold luminance value, specified as a numeric scalar or numeric array of the same size as image `I`. If `T` is a scalar luminance value, then `imbinarize` interprets it as a global image threshold. If `T` is an array of luminance values, then `imbinarize` interprets it as a locally adaptive threshold. `T` must have a value between 0 and 1. Use the functions `graythresh`, `otsuthresh`, or `adaptthresh` to compute `T`.

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 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) | number in the range [0, 1]

Sensitivity factor for adaptive thresholding, specified as the comma-separated pair consisting of 'Sensitivity' and a number 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 the comma-separated pair consisting of `'ForegroundPolarity'` and one of the following values.

Value	Meaning
<code>'bright'</code>	The foreground is brighter than the background.
<code>'dark'</code>	The foreground is darker than the background

Data Types: `char` | `string`

## **Output Arguments**

### **BW — Output binary image**

logical matrix | logical array

Output binary image, returned as a logical matrix or logical array of 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`.
- `imbinarize` expects floating point images to be normalized in the range `[0, 1]`.

## **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 [2]. 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`.

## 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.
- [2] Bradley, D., G. Roth, "Adapting Thresholding Using the Integral Image," *Journal of Graphics Tools*. Vol. 12, No. 2, 2007, pp.13-21.

## 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 "Understand Code Generation with Image Processing Toolbox".
- When generating code, all character vector input arguments must be compile-time constants.

### See Also

**Image Segmenter** | `adaptthresh` | `graythresh` | `otsuthresh`

**Introduced in R2016a**

## imbothat

Bottom-hat filtering

### Syntax

```
J = imbothat(I,SE)
J = imbothat(I,nhood)
```

### Description

`J = imbothat(I,SE)` performs morphological bottom-hat filtering on the grayscale or binary image `I`, returning the filtered image, `J`. Bottom-hat filtering computes the morphological closing of the image (using `imclose`) and then subtracts the original image from the result. `SE` is a single structuring element object returned by the `strel` or `offsetstrel` functions.

You optionally can perform the bottom-hat filtering using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`J = imbothat(I,nhood)` bottom-hat filters the image `I`, where `nhood` is a matrix of 0s and 1s that specifies the structuring element neighborhood. This is equivalent to the syntax `imbothat(I,strel(nhood))`. The `imbothat` function determines the center element of the neighborhood by `floor((size(nhood)+1)/2)`.

### 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);
```

## Input Arguments

### I — Input image

grayscale image | binary image | gpuArray

Input image, specified as a grayscale image or binary image. I can have any dimension, and must be real and nonsparse.

To perform the bottom-hat filtering using a GPU, specify I as a `gpuArray` that contains an image of type `uint8` or `logical`.

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

### SE — Structuring element

`strel` object | `offsetstrel` object

Structuring element, specified as a single `strel` object or `offsetstrel` object. If the image I is data type `logical`, the structuring element must be flat.

If you perform the bottom-hat filtering using a GPU, then SE must be flat and two-dimensional.

### nhood — Structuring element neighborhood

matrix of 0s and 1s

Structuring element neighborhood, specified as a matrix of 0s and 1s.

Example: `[0 1 0; 1 1 1; 0 1 0]`

## Output Arguments

### J — Bottom-hat filtered image

grayscale image | binary image | gpuArray

Bottom-hat filtered image, returned as a grayscale image or binary image. J has the same class as input image I.

If the filtering is performed using a GPU, then `J` is returned as a `gpuArray` that contains a grayscale or binary image of the same class 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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, the image input argument, `I`, must be 2-D or 3-D and the structuring element input argument, `SE`, must be a compile-time constant.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `gpuArray` input must be of type `uint8` or `logical` and the structuring element must be flat and two-dimensional.

For more information, see “Image Processing on a GPU”.

## See Also

### Functions

`imclose` | `imdilate` | `imerode` | `imopen` | `imtophat`

**Objects**

offsetstrel | strel

**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 `Infs` or `NaNs`, the behavior of `imboxfilt` 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.

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 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
<code>'circular'</code>	Input image values outside the bounds of the image are computed by implicitly assuming the input image is periodic.
<code>'replicate'</code>	Input image values outside the bounds of the image are assumed equal to the nearest image border value.
<code>'symmetric'</code>	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` | `string`

### 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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, all character vector input arguments must be compile-time constants.

## 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 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.
'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` | `char` | `string`

### **NormalizationFactor** — Normalization factor applied to box filter

$1/\text{filterSize}.^3$ , 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. 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

`imboxfilt` | `imfilter` | `integralBoxFilter3`

**Introduced in R2015b**

## imclearborder

Suppress light structures connected to image border

### Syntax

```
J = imclearborder(I)
J = imclearborder(I,conn)
```

### Description

`J = imclearborder(I)` suppresses structures in image `I` that are lighter than their surroundings and that are connected to the image border. Use this function to clear the image border. For grayscale images, `imclearborder` tends to reduce the overall intensity level in addition to suppressing border structures. The output image, `J`, is grayscale or binary, depending on the input.

`J = imclearborder(I,conn)` specifies the pixel connectivity, `conn`.

### 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 = 9×9
```

```

0     0     0     0     0     0     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 = 9×9
```

```

0     0     0     0     0     0     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

**I** — Grayscale or binary image

numeric array | logical array

Grayscale or binary image, specified as a numeric or logical array.

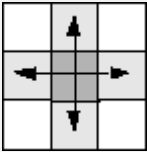
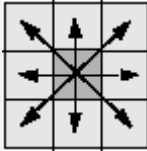
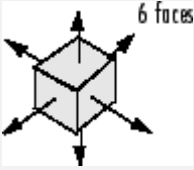
Example: `I = imread('pout.tif');`

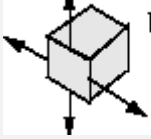
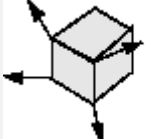
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

**conn — Pixel connectivity**

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	

Value	Meaning	
18-connected	<p>Pixels are connected if their faces or edges touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	 <p>6 faces + 12 edges</p>
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	 <p>6 faces + 12 edges + 8 corners</p>

For higher dimensions, `imclearborder` uses the default value `conndef(ndims(I), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

**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.

Data Types: `double` | `logical`

## Output Arguments

### J — Processed image

numeric array | logical array

Processed grayscale or binary image, returned as 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 “Understand 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**

`conndef`

**Introduced before R2006a**

## imclose

Morphologically close image

### Syntax

```
J = imclose(I,SE)
J = imclose(I,nhood)
```

### Description

`J = imclose(I,SE)` performs morphological closing on the grayscale or binary image `I`, returning the closed image, `J`. `SE` is a single structuring element object returned by the `strel` or `offsetstrel` functions. The morphological close operation is a dilation followed by an erosion, using the same structuring element for both operations.

You optionally can perform the morphological closing using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

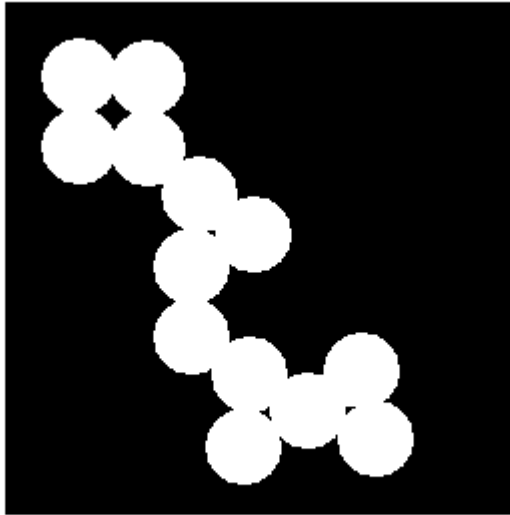
`J = imclose(I,nhood)` closes the image `I`, where `nhood` is a matrix of 0s and 1s that specifies the structuring element neighborhood. This is equivalent to the syntax `imclose(I,strel(nhood))`. The `imclose` function determines the center element of the neighborhood by `floor((size(nhood)+1)/2)`.

### 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);
```

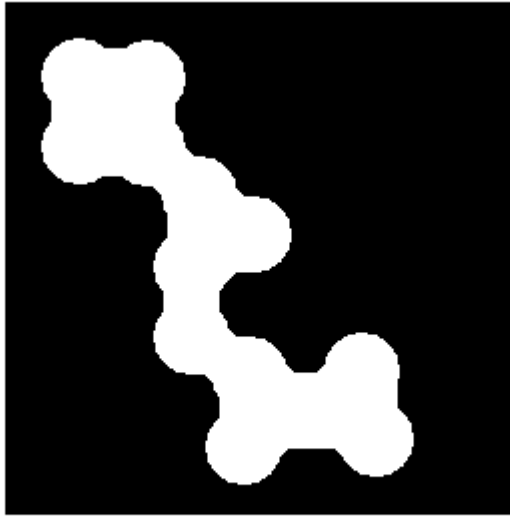


Create a disk-shaped structuring element. Use a disk structuring element to preserve the circular nature of the object. Specify a radius of 10 pixels so that the largest gap gets filled.

```
se = strel('disk',10);
```

Perform a morphological close operation on the image.

```
closeBW = imclose(originalBW,se);  
figure, imshow(closeBW)
```



## Perform Morphological Closing Using 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 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)
```

## Input Arguments

### **I** — Input image

grayscale image | binary image | gpuArray

Input image, specified as a grayscale image or binary image. I can have any dimension, and must be real and nonsparse.

To perform the closing using a GPU, specify I as a `gpuArray` that contains an image of type `uint8` or `logical`.

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

### **SE** — Structuring element

`strel` object | `offsetstrel` object

Structuring element, specified as a single `strel` object or `offsetstrel` object. If the image I is data type `logical`, then the structuring element must be flat.

If you perform the closing using a GPU, then SE must be flat and two-dimensional.

### **nhood** — Structuring element neighborhood

matrix of 0s and 1s

Structuring element neighborhood, specified as a matrix of 0s and 1s.

Example: `[0 1 0; 1 1 1; 0 1 0]`

## Output Arguments

### **J** — Closed image

grayscale image | binary image | gpuArray

Closed image, returned as a grayscale image or binary image. J has the same class as input image I.

If the closing is performed using a GPU, then `J` is returned as a `gpuArray` that contains a grayscale or binary image of the same class 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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, the input image `I` must be 2-D or 3-D.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `gpuArray` input must be of type `uint8` or `logical` and the structuring element must be flat and two-dimensional.

For more information, see “Image Processing on a GPU”.

## See Also

### Functions

`imdilate` | `imerode` | `imopen`

**Objects**

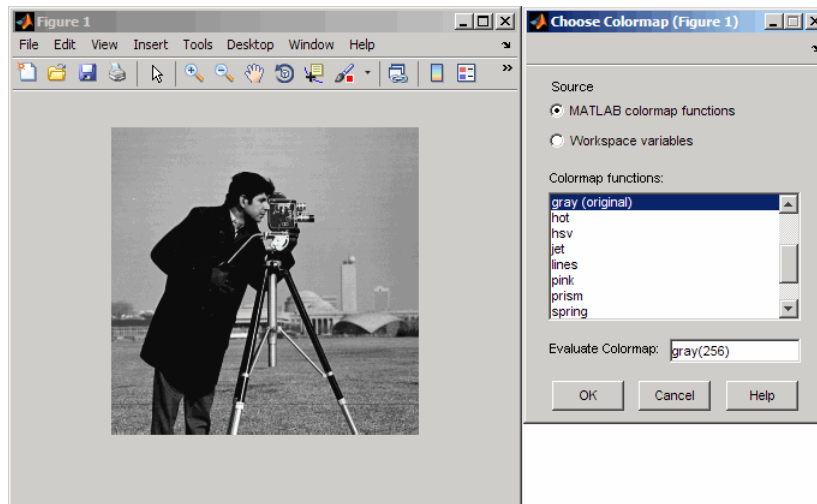
offsetstrel | strel

**Introduced before R2006a**

# imcolormaptool

Choose Colormap tool

Use the `imcolormaptool` function to create a Choose Colormap tool. The Choose Colormap tool is an interactive colormap selection tool that allows you to change the colormap of a figure by selecting a colormap from a list of MATLAB colormap functions or workspace variables, or by entering a custom MATLAB expression.



**Choose Colormap Tool**

## Syntax

```
imcolormaptool  
imcolormaptool(h)  
htool = imcolormaptool(____)
```

## Description

`imcolormaptool` launches the Choose Colormap tool in a separate figure, which is associated with the current figure.

`imcolormaptool(h)` launches the Choose Colormap tool using `h` as the target figure. `h` must contain either a grayscale or an indexed image.

`htool = imcolormaptool( ___ )` returns a handle to the Choose Colormap tool figure, `htool`.

## Examples

### Open Choose Colormap Tool

```
h = figure;  
imshow('cameraman.tif');  
imcolormaptool(h);
```

## Input Arguments

**h — Handle to graphics object**

handle

Handle to a figure or axes, specified as a handle.

## Output Arguments

**htool — Handle to Choose Colormap tool figure**

handle

Handle to Choose Colormap tool figure, returned as a handle.

## See Also

`colormap` | `imshow` | `imtool`

**Introduced in R2009a**

# imcomplement

Complement image

## Syntax

```
J = imcomplement(I)
```

## Description

`J = imcomplement(I)` computes the complement on page 1-1232 of the image `I` and returns the result in `J`.

You optionally can compute the image complement using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

## Examples

### Create the Complement of a uint8 Array

```
X = uint8([ 255 10 75; 44 225 100]);  
X2 = imcomplement(X)
```

*X2 = 2x3 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')
```

**The term watershed  
refers to a ridge that ...**

**... divides areas  
drained by different  
river systems.**

**The term watershed  
refers to a ridge that ...**

**... divides areas  
drained by different  
river systems.**

### **Create the Complement of an Intensity Image**

```
I = imread('cameraman.tif');  
J = imcomplement(I);  
imshowpair(I,J,'montage')
```

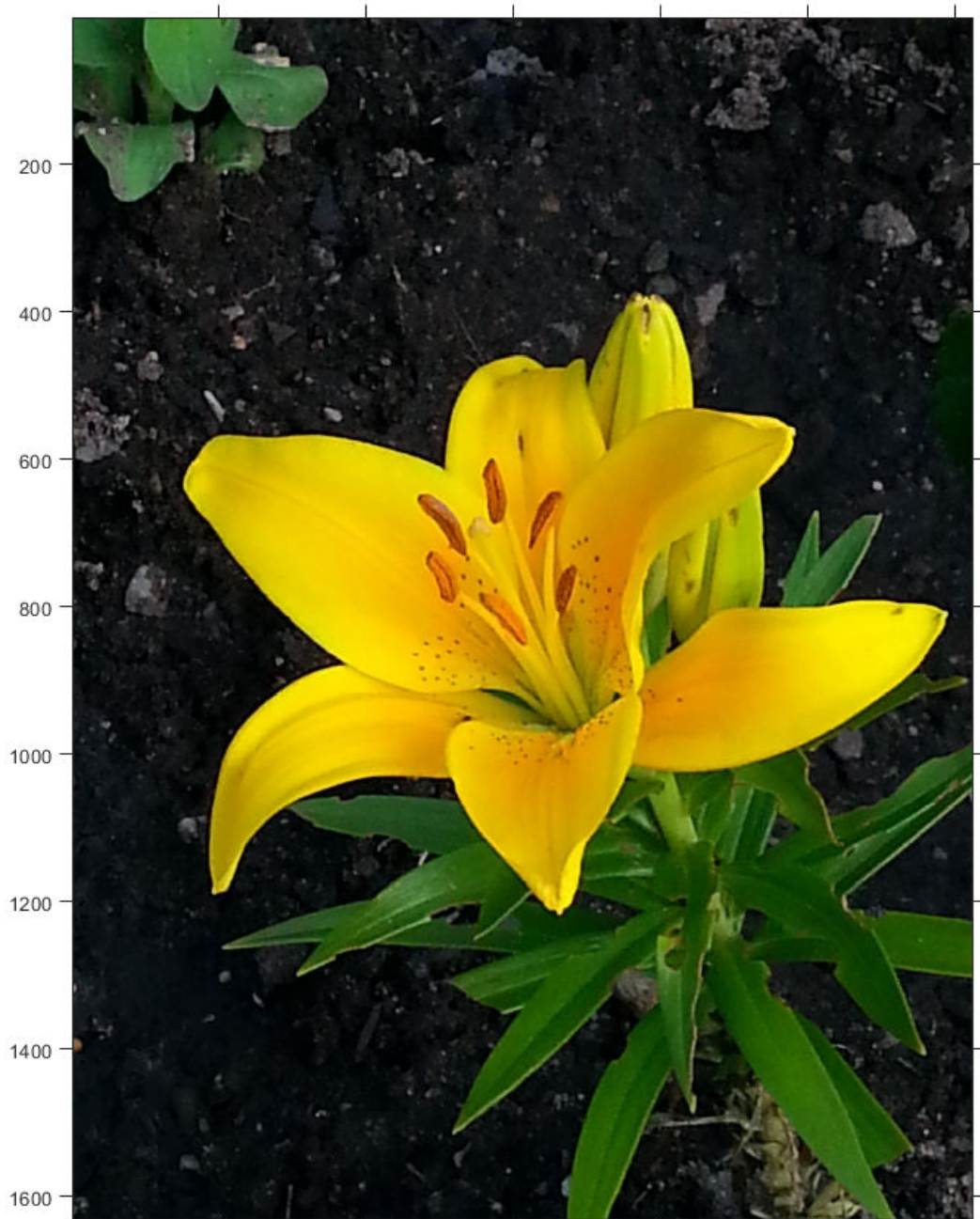




### Create the Complement of a Color Image

Read a color image into the workspace.

```
rgb = imread('yellowlily.jpg');  
imshow(rgb)
```



**1-1228**

200

400

600

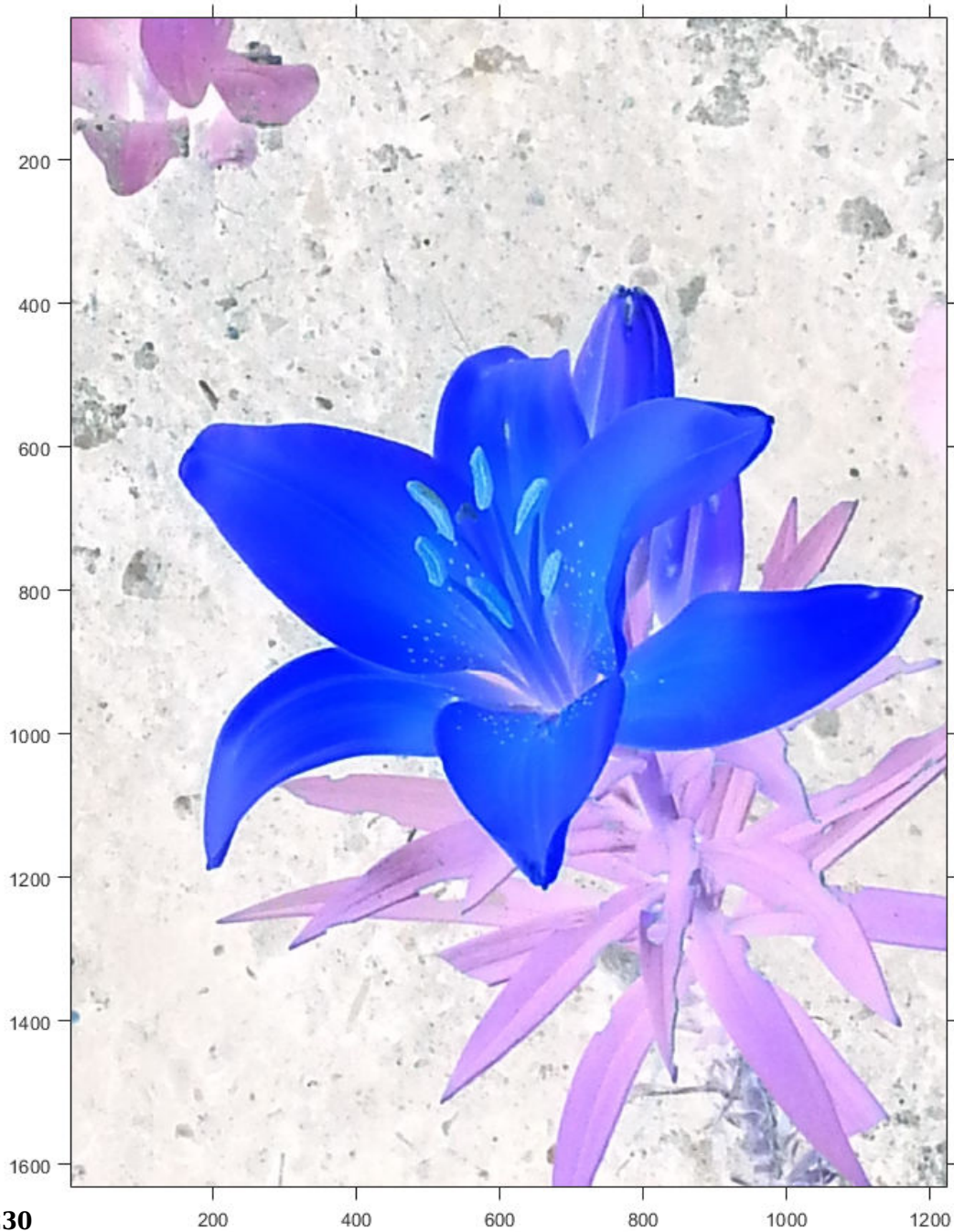
800

1000

1200

Display the complement of the image.

```
c = imcomplement(rgb);  
imshow(c)
```



**1-1230**

Each color channel of the resulting image is the complement of the corresponding color channel in the original image. Regions that were dark, such as dirt, become light. In the original image, the leaves appear green, and petals appear yellow because of a mixture of red and green signals. In the complement image, the leaves appear purple because the red and blue signals are larger than the green signal. The flower petals appear blue because the blue signal is larger than the red and green channels.

### Create the Complement of Intensity Image on a GPU

```
I = imread('glass.png');  
J = imcomplement(gpuArray(I));  
imshowpair(I,J,'montage')
```

## Input Arguments

### I — Input image

RGB image | grayscale image | binary image | gpuArray

Input image, specified as an RGB, grayscale, or binary image.

To perform the computation using a GPU, specify I as a `gpuArray` that contains an RGB, grayscale, or binary image.

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

## Output Arguments

### J — Image complement

RGB image | grayscale image | binary image | gpuArray

Image complement, specified as an RGB, grayscale, or binary image. J has the same size and class as the input image, I.

If the image complement is computed using a GPU, then J is returned as a `gpuArray` containing an RGB, grayscale, or binary image.

## Definitions

### Image Complement

In the complement of a binary image, zeros become ones and ones become zeros. Black and white are reversed.

In the complement of a grayscale or color image, each pixel value is subtracted from the maximum pixel value supported by the class (or 1.0 for double-precision images). 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. For color images, reds become cyan, greens become magenta, blues become yellow, and vice versa.

### Tips

- If `I` is a grayscale or RGB image of class `double`, you can use the expression `1 - I` instead of this function.
- If `I` is a binary image, you can use the expression `~I` 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.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

## **See Also**

`imabsdiff` | `imadd` | `imdivide` | `imlincomb` | `immultiply` | `imsubtract`

**Introduced before R2006a**

## imcontour

Create contour plot of image data

### Syntax

```
imcontour(I)
imcontour(I,levels)
imcontour(I,V)
imcontour(x,y,___)
imcontour(___,LineStyle)
[C,h] = imcontour(___)
```

### Description

`imcontour(I)` draws a contour plot of the grayscale image `I`, choosing the number of levels and the values of levels automatically. `imcontour` automatically sets up the axes so their orientation and aspect ratio match the image.

`imcontour(I,levels)` specifies the number, `levels`, of equally spaced contour levels in the plot.

`imcontour(I,V)` draws 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 image `x`- and `y` coordinates.

`imcontour(___,LineStyle)` draws the contours using the line type and color specified by `LineStyle`. Marker symbols are ignored.

`[C,h] = imcontour(___)` returns the contour matrix, `C`, and the handle, `h`, to the contour patches drawn onto the current axes.

### Examples

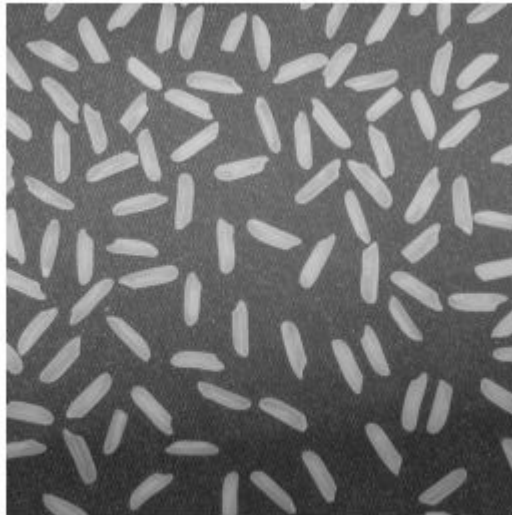


## Create Contour Plot of Image Data

This example shows how to create a contour plot of an image.

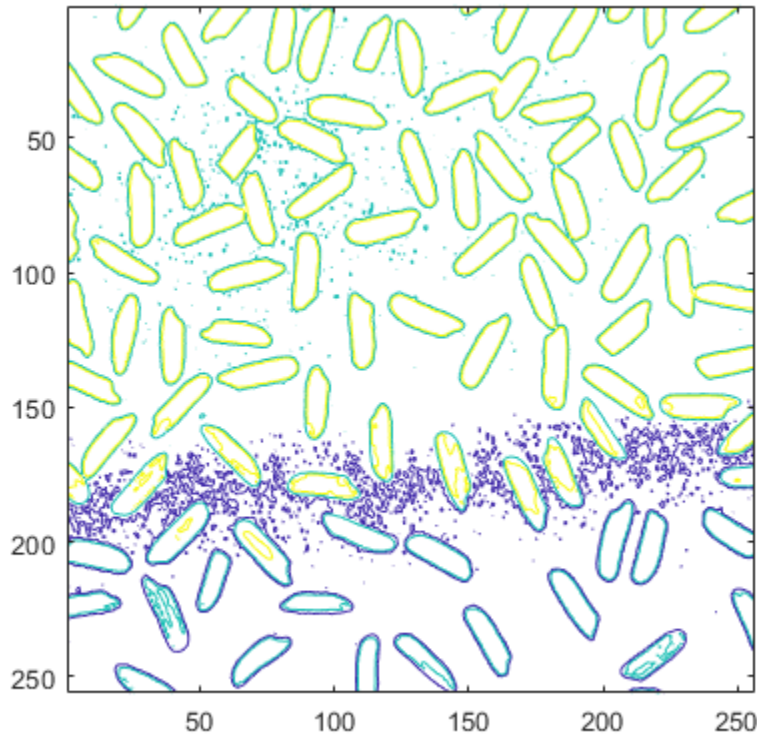
Read grayscale image and display it. The example uses an example image of grains of rice.

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



Create a contour plot of the image using `imcontour` .

```
figure;  
imcontour(I,3)
```



## Input Arguments

### **I** — Grayscale image

*m*-by-*n* matrix

Grayscale image, specified as an *m*-by-*n* matrix.

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

### **levels** — Number of contour levels

numeric scalar

Number of contour levels, specified as a numeric scalar.

### **V — Value of contour levels**

numeric vector

Value of contour levels, specified as a numeric vector with length greater than or equal to two. Use  $V = [v \ v]$  to compute a single contour at level  $v$ .

### **x — Image x values**

2-element numeric vector |  $n$ -element numeric vector

Image x values, specified as one of the following:

- 2-element numeric vector of the form  $[x_{\min} \ x_{\max}]$  — Image extent in the x direction.
- $n$ -element numeric vector — x-coordinate of each column.

### **y — Image y values**

2-element numeric vector |  $m$ -element numeric vector

Image y values, specified as one of the following:

- 2-element numeric vector of the form  $[y_{\min} \ y_{\max}]$  — Image extent in the y direction.
- $m$ -element numeric vector — y-coordinate of each row.

### **LineStyleSpec — Line specification**

LineStyleSpec

Line specification, specified as a LineSpec.

## **Output Arguments**

### **C — Contour matrix**

numeric matrix

Contour matrix, returned as a matrix with two rows. The matrix is defined according to the `ContourMatrix` property of the `Contour` object, `h`.

### **h — Contour patches**

handle

Contour patches, returned as a handle to a `Contour` object.

## See Also

### Functions

LineStyle | xlabel | contour

### Properties

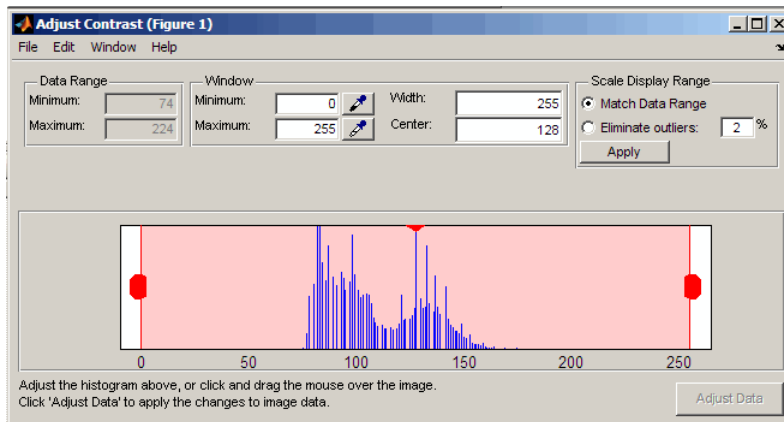
Contour

**Introduced before R2006a**

# imcontrast

## Adjust Contrast tool

Use the `imcontrast` function to create an Adjust Contrast tool. The Adjust Contrast tool is an interactive contrast and brightness adjustment tool that you can use to adjust the black-to-white mapping used to display a grayscale image. For more information about using the tool, see “Tips” on page 1-1241.



## Syntax

```
imcontrast
imcontrast(h)
htool = 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.

`imcontrast(h)` creates the Adjust Contrast tool associated with the image specified by the handle `h`.

`htool = imcontrast( ___ )` returns the handle `htool` to the Adjust Contrast tool figure.

## Examples

### Adjust the Contrast of the Current Image

Read an image into the workspace. Adjust the contrast of the current image.

```
imshow('pout.tif')
imcontrast
```

### Adjust Image Contrast, Specifying Figure Handle

Read an image into the workspace and define the handle of the figure as `h1`. Open a second figure window and define the handle of that figure as `h2`. Adjust the contrast of the first figure by specifying `h1` in the call to `imcontrast`.

```
h1 = figure;
imshow('pout.tif');
h2 = figure;
imshow('coins.png');
imcontrast(h1)
```

## Input Arguments

### **h** — Handle to graphics object

`handle`

Handle to a figure, axes, uipanel, or image graphics object, specified as a handle. If `h` is an axes or figure handle, `imcontrast` uses the first image returned by `findobj(H,'Type','image')`.

## Output Arguments

### **htool** — Handle to Adjust Contrast tool figure

handle

Handle to Adjust Contrast tool figure, returned as a handle.

## 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 pixel value less than the minimum, display 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, display 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.

- Click and drag the mouse within the target image to interactively change the 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.
- 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.
- 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.

## **See Also**

`imadjust` | `imtool` | `stretchlim`

## **Topics**

“Adjust Image Contrast in Image Viewer App”

**Introduced before R2006a**



# imcrop

Crop image

## Syntax

```
J = imcrop
J = imcrop(I)
J = imcrop(X, cmap)
J = imcrop(h)

J = imcrop(I, rect)
J = imcrop(X, cmap, rect)

J = imcrop(x, y, ___ )
[J, rect2] = imcrop( ___ )
[x2, y2, ___ ] = imcrop( ___ )
```

## Description

`J = imcrop` creates an interactive Crop Image tool associated with the image displayed in the current figure. With this syntax and the other interactive syntaxes, the Crop Image tool blocks the MATLAB command line until you complete the operation. `imcrop` returns the cropped image, `J`.

For more information about using the Crop Image tool, see “Interactive Behavior” on page 1-1251.

`J = imcrop(I)` displays the image `I` in a figure window and creates an interactive Crop Image tool associated with the image. `I` can be a grayscale image, a truecolor image, or a logical array.

`J = imcrop(X, cmap)` displays the indexed image `X` in a figure using the colormap `cmap`, and creates an interactive Crop Image tool associated with that image. `imcrop` returns the cropped indexed image, `J`, which also has the colormap `cmap`.

`J = imcrop(h)` creates an interactive Crop Image tool associated with the image specified by the handle `h`.

`J = imcrop(I, rect)` crops the image `I` according to `rect`, which specifies the size and position of the crop rectangle as `[xmin ymin width height]`, in terms of spatial coordinates. The cropped image includes all pixels in the input image that are completely or *partially* enclosed by the rectangle.

The width and height specified by `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.

`J = imcrop(X, cmap, rect)` crops the indexed image `X` with colormap `cmap` according to the vector `rect`, which specifies the size and position of the crop rectangle.

`J = imcrop(x, y, ___)` crops the image using a nondefault coordinate system, where `x` and `y` specify the image limits in the world coordinate system.

`[J, rect2] = imcrop( ___)` also returns the position of the cropping rectangle in `rect2`.

`[x2, y2, ___] = imcrop( ___)` also returns the image limits in `x2` and `y2`.

## Examples


### Crop Image Using Crop Image Interactive Tool

Read image into the workspace.

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

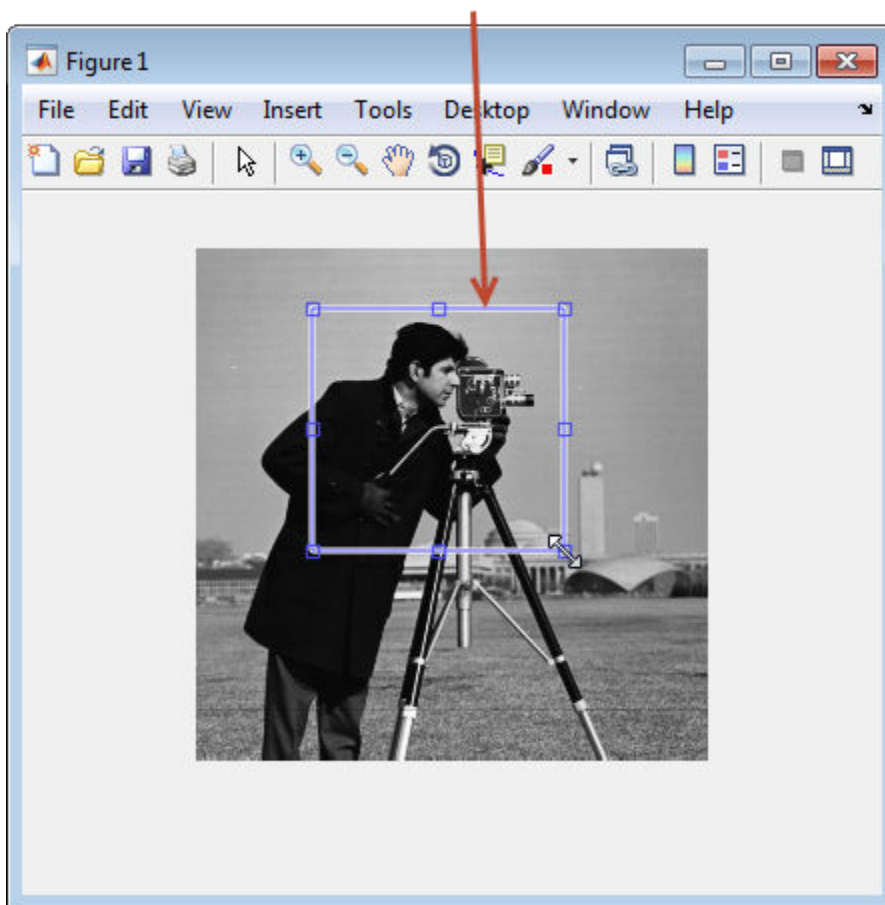
Open Crop Image tool associated with this image. Specify a variable in which to store the cropped image. The example includes the optional return value `rect` in which `imcrop` returns the four-element position vector of the rectangle you draw.

```
[J, rect] = imcrop(I);
```

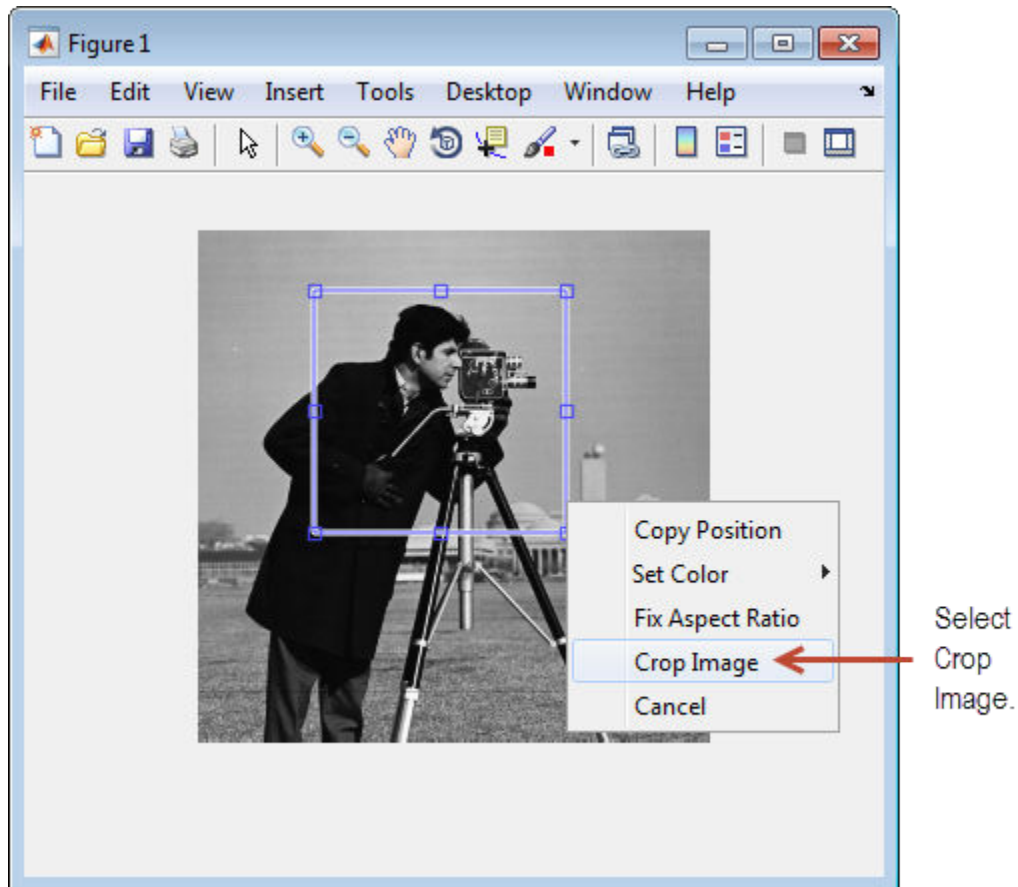
When you move the cursor over the image, it changes to a cross-hairs . The Crop Image tool blocks the MATLAB command line until you complete the operation.

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, J. The variable rect is the four-element position vector describing the crop rectangle you specified.

whos

Name	Size	Bytes	Class	Attributes
I	256x256	65536	uint8	
J	121x126	15246	uint8	
rect	1x4	32	double	

## Crop Image By Specifying Crop Rectangle

Read image into the workspace.

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

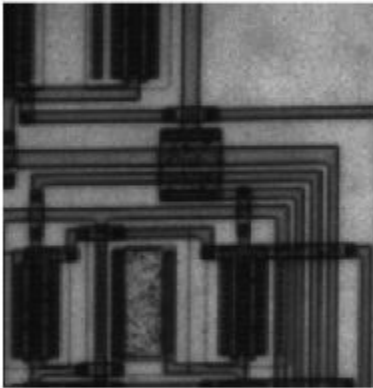
Crop image, specifying crop rectangle.

```
I2 = imcrop(I,[75 68 130 112]);
```

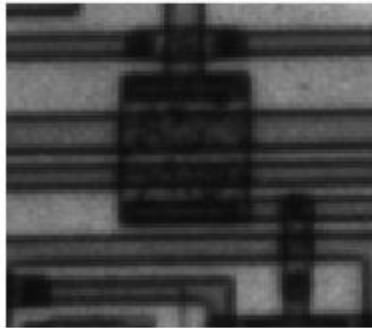
Display original image and cropped image.

```
subplot(1,2,1)
imshow(I)
title('Original Image')
subplot(1,2,2)
imshow(I2)
title('Cropped Image')
```

**Original Image**



**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')
```



## Input Arguments

**I** — Image to be cropped  
real, nonsparse, numeric array

Image to be cropped, specified as a real, nonsparse, numeric array. If you specify `rect` as an input argument, then `I` can also be logical. Otherwise, `imcrop` calls `imshow` and accepts whatever image classes `imshow` accepts.

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

## **X — Indexed image to be cropped**

real, nonsparse, numeric array

Indexed image to be cropped, specified as a real, nonsparse numeric array. If you specify `rect` as an input argument, then `I` can also be logical. Otherwise, `imcrop` calls `imshow` and accepts any image class that `imshow` accepts.

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

## **cmap — Colormap associated with indexed image**

*c*-by-3 numeric array

Colormap associated with indexed image, specified as an *c*-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**

4-element position vector

Size and position of the crop rectangle in spatial coordinates, specified as a 4-element position vector of the form `[xmin ymin width height]`.

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

## **h — Handle to graphics object**

handle

Handle to a figure, axes, uipanel, or image graphics object, specified as a handle. If `h` is an axes or figure handle, `imcrop` uses the first image returned by `findobj(H, 'Type', 'image')`.

## **x, y — Image limits in world coordinates**

2-element numeric vector



Image limits in world coordinates along the x- or y-dimension, specified as a 2-element numeric vector of the form `[min max]`. The values of `x` and `y` set the image XData and YData. The data type of `x` and `y` must match.

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

## Output Arguments

### **J** — Cropped image

real, nonsparse, numeric array

Cropped image, returned as a real, nonsparse, numeric array.

- If you specify an input image, then the output image has the same class as the input image.
- If you do not specify an input image, then the output image generally has the same class as the input image assigned by `imcrop`. However, if the input image is `int16` or `single`, then the output image is `double`.

### **rect2** — Size and position of the crop rectangle

4-element position vector

Size and position of the crop rectangle, returned as a 4-element position vector of the form `[xmin ymin width height]`.

### **x2, y2** — Image limits


2-element numeric vector

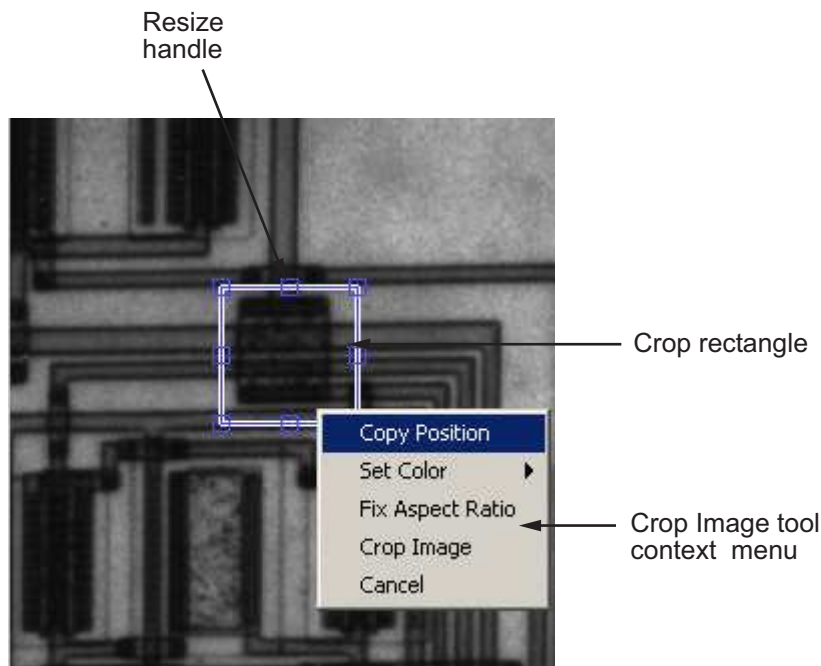
Image limits in the `x` or `y` direction, returned as a 2-element numeric vector of the form `[min max]`. If you specify image limits `x` and `y`, then `x2` and `y2` are equal to these values. Otherwise, `x2` and `y2` are equal to the original image XData and YData.

## Additional Capabilities



### Interactive Behavior

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.

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. The following figure illustrates the Crop Image tool with the context menu displayed.



Interactive Behavior	Description
Delete the Crop Image tool.	Press <b>Backspace</b> , <b>Escape</b> or <b>Delete</b> , or right-click inside the crop rectangle and select <b>Cancel</b> from the context menu.  Note: If you delete the ROI, the function returns empty values.

Interactive Behavior	Description
Resize 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.
Move 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.
Change the color used to display the crop rectangle.	Right-click inside the boundary of the crop rectangle and select <b>Set Color</b> from the context menu.
Crop the image.	Double-click the left mouse button or right-click inside the boundary of the crop rectangle and select <b>Crop Image</b> from the context menu.
Retrieve the coordinates of the crop rectangle.	Right-click inside the boundary of the crop rectangle and select <b>Copy Position</b> from the context menu. <code>imcrop</code> copies a 4-element position vector ( <code>[xmin ymin width height]</code> ) to the clipboard.

## 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:
  - `J = imcrop`
  - `J = imcrop(I)`
  - `X2 = imcrop(X, cmap)`
  - `J = imcrop(h)`

- Indexed images are not supported, including the noninteractive syntax `X2 = imcrop(X, cmap, rect);`

## **See Also**

`imrect` | `zoom`

**Introduced before R2006a**

# imdiffuseest

Estimate parameters for anisotropic diffusion filtering

## Syntax

```
[gradientThreshold,numberOfIterations] = imdiffuseest(I)  
[gradientThreshold,numberOfIterations] = imdiffuseest(I,Name,Value)
```

## Description

`[gradientThreshold,numberOfIterations] = imdiffuseest(I)` estimates the gradient threshold and number of iterations required to filter the grayscale image `I` using anisotropic diffusion.

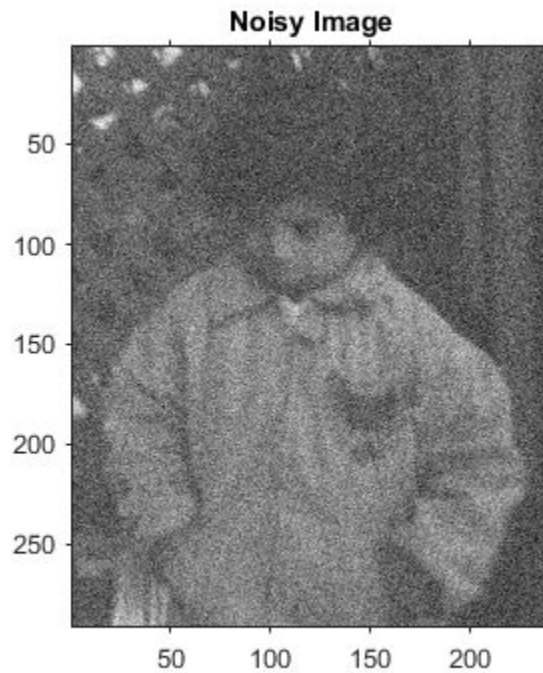
`[gradientThreshold,numberOfIterations] = imdiffuseest(I,Name,Value)` uses name-value pairs to change the behavior of the anisotropic diffusion algorithm.

## Examples

### Estimate Parameters for Anisotropic Diffusion Filtering

Read a grayscale image, then apply strong Gaussian noise to it. Display the noisy image.

```
I = imread('pout.tif');  
Inoisy = imnoise(I,'gaussian',0,0.005);  
imshow(Inoisy)  
title('Noisy Image')
```



Estimate the gradient threshold and number of iterations needed to perform anisotropic diffusion filtering of the image.

```
[gradThresh,numIter] = imdiffuseest(Inoisy)
```

```
gradThresh = 1x5 uint8 row vector
```

```
    64    50    39    34    29
```

```
numIter = 5
```

Filter the noisy image by using anisotropic diffusion with the estimated parameters.

```
Idiffuseest = imdiffusefilt(Inoisy,'GradientThreshold', ...  
    gradThresh,'NumberOfIterations',numIter);
```

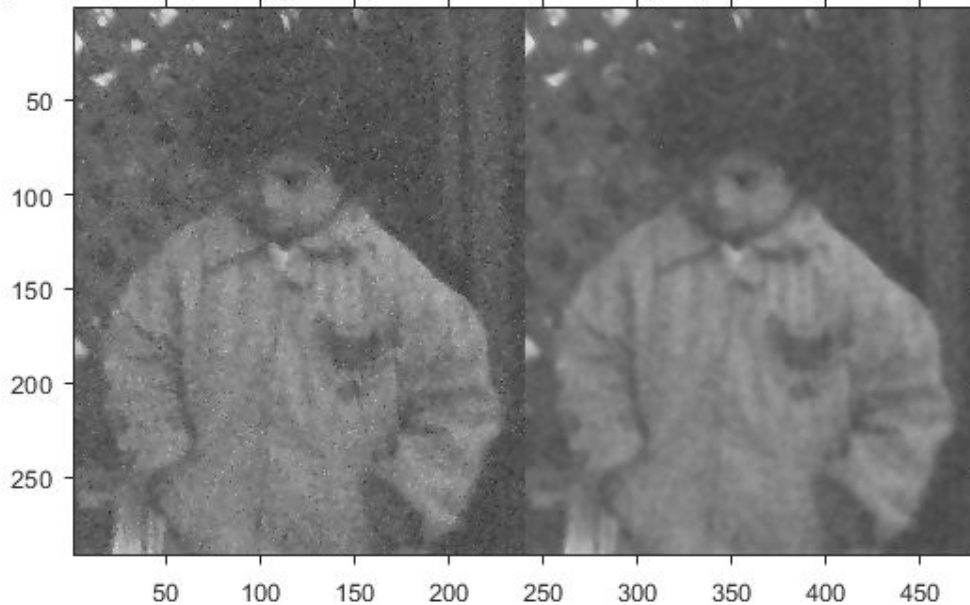
For comparison, also filter the noisy image by using anisotropic diffusion with the default parameters. The default gradient threshold is 25.5 because the data type of the image is uint8, and the default number of iterations is 5.

```
Idiffusedef = imdiffusefilt(Inoisy);
```

Visually compare the two filtered images.

```
montage({Idiffusedef,Idiffuseest},'ThumbnailSize',[1])  
title(['Anisotropic Diffusion Filtering Using ' ...  
      'Default Parameters (Left) vs. Estimated Parameters (Right)'])
```

**anisotropic Diffusion Filtering Using Default Parameters (Left) vs. Estimated Parameters (Right)**



Some noise remains in the image that was filtered using default parameters. The noise is almost completely absent from the image that was filtered using estimated parameters. The sharpness of edges in both images, especially high-contrast edges such as the trellis and white collar, is preserved.

## Input Arguments

### **I** — Image

2-D grayscale image

Image to be filtered, specified as a 2-D grayscale image.

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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `imdiffuseest(I, 'Connectivity', 'minimal')` estimates parameters required for anisotropic diffusion on image `I`, using minimal connectivity.

### **Connectivity** — Connectivity

`'maximal'` (default) | `'minimal'`

Connectivity of a pixel to its neighbors, specified as the comma-separated pair consisting of `'Connectivity'` and `'maximal'` or `'minimal'`. Maximal connectivity considers eight nearest neighbors and minimal connectivity considers four nearest neighbors.

### **ConductionMethod** — Conduction method

`'exponential'` (default) | `'quadratic'`

Conduction method, specified as the comma-separated pair consisting of `'ConductionMethod'` and `'exponential'` or `'quadratic'`. Exponential diffusion favors high-contrast edges over low-contrast edges. Quadratic diffusion favors wide regions over smaller regions.

## Output Arguments

### **gradientThreshold** — Gradient threshold

numeric vector

Gradient threshold, returned as a numeric vector of the same data type as the input image, `I`. The length of the vector is equal to `numberOfIterations`.



**numberOfIterations — Number of iterations**

positive integer

Number of iterations to use in the diffusion process, returned as a positive integer.

**References**

- [1] Perona, P., and J. Malik. "Scale-space and edge detection using anisotropic diffusion." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 12, No. 7, July 1990, pp. 629–639.
- [2] Tsotsios, C., and M. Petrou. "On the choice of the parameters for anisotropic diffusion in image processing." *Pattern Recognition*. Vol. 46, No. 5, May 2013, pp. 1369–1381.

**See Also**

imdiffusefilt

**Introduced in R2018a**

## imdiffusefilt

Anisotropic diffusion filtering of images

### Syntax

```
J = imdiffusefilt(I)
J = imdiffusefilt(I,Name,Value)
```

### Description

`J = imdiffusefilt(I)` applies anisotropic diffusion filtering to image `I` and returns the result in `J`.

`J = imdiffusefilt(I,Name,Value)` uses name-value pairs to change the behavior of the anisotropic diffusion algorithm.

### Examples

#### Perform Edge-Preserving Smoothing Using Anisotropic Diffusion

Read an image into the workspace and display it.

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

Original Image



Smooth the image using anisotropic diffusion. For comparison, also smooth the image using Gaussian blurring. Adjust the standard deviation `sigma` of the Gaussian smoothing kernel so that textured regions, such as the grass, are smoothed a similar amount for both methods.

```
Idiffusion = imdiffusefilt(I);  
sigma = 1.2;  
Igaussian = imgaussfilt(I,sigma);
```

Display the results.

```
montage({Idiffusion,Igaussian},'ThumbnailSize',[1])  
title('Smoothing Using Anisotropic Diffusion (Left) vs. Gaussian Blurring (Right)')
```

**Smoothing Using Anisotropic Diffusion (Left) vs. Gaussian Blurring (Right)**



Anisotropic diffusion preserves the sharpness of edges better than Gaussian blurring.

**Perform Edge-Aware Noise Reduction Using Anisotropic Diffusion**

Read a grayscale image, then apply strong Gaussian noise to it. Display the noisy image.

```
I = imread('pout.tif');  
noisyImage = imnoise(I, 'gaussian', 0, 0.005);  
imshow(noisyImage)  
title('Noisy Image')
```

**Noisy Image**

Compute the structural similarity index (SSIM) to measure the quality of the noisy image. The closer the SSIM value is to 1, the better the image agrees with the noiseless reference image.

```
n = ssim(I,noisyImage);  
disp(['The SSIM value of the noisy image is ',num2str(n),'.'])
```

The SSIM value of the noisy image is 0.26556.

Reduce the noise using anisotropic diffusion. First, try the default parameters for the anisotropic diffusion filter, and display the result.

```
B = imdiffusefilt(noisyImage);  
imshow(B)  
title('Anisotropic Diffusion with Default Parameters')
```

## Anisotropic Diffusion with Default Parameters



```
nB = ssim(I,B);  
disp(['The SSIM value using default anisotropic diffusion is ',num2str(nB),'.'])
```

The SSIM value using default anisotropic diffusion is 0.65665.

The image is still degraded by noise, so refine the filter. Choose the quadratic conduction method because the image is characterized more by wide homogenous regions than by high-contrast edges. Estimate the optimal gradient threshold and number of iterations by using the `imdifuseest` function. Display the resulting image.

```
[gradThresh,numIter] = imdifuseest(noisyImage,'ConductionMethod','quadratic');  
C = imdifusefilt(noisyImage,'ConductionMethod','quadratic', ...  
    'GradientThreshold',gradThresh,'NumberOfIterations',numIter);  
imshow(C)  
title('Anisotropic Diffusion with Estimated Parameters')
```

### Anisotropic Diffusion with Estimated Parameters



```
nC = ssim(I,C);  
disp(['The SSIM value using quadratic anisotropic diffusion is ',num2str(nC),'.'])
```

The SSIM value using quadratic anisotropic diffusion is 0.88135.

Noise is less apparent in the resulting image. The SSIM value, which is closer to 1, confirms that the quality of the image has improved.

### Perform 3-D Edge-Aware Noise Reduction

Load a noisy 3-D grayscale MRI volume.

```
load mrystack
```

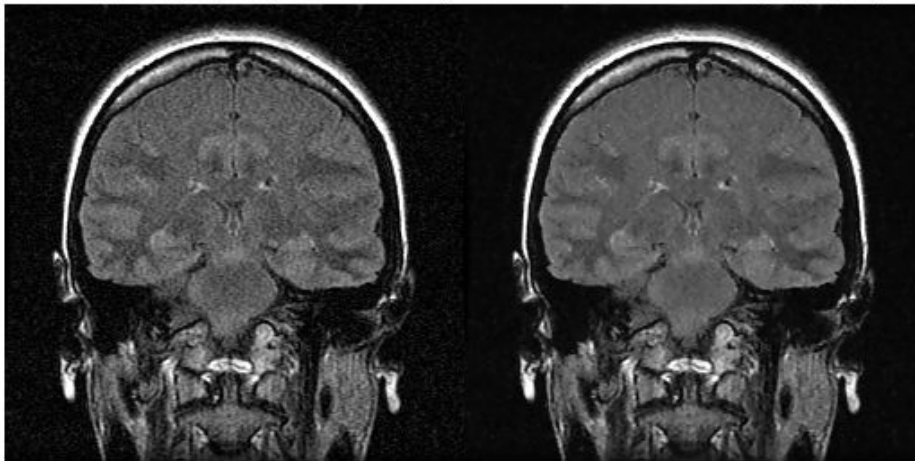
Perform edge-aware noise reduction on the volume using anisotropic diffusion. To prevent over-smoothing the low-contrast features in the brain, decrease the number of iterations from the default number, 5. The tradeoff is that less noise is removed.

```
diffusedImage = imdiffusefilt(mristack,'NumberOfIterations',3);
```

To compare the noisy image and the filtered image in detail, display the tenth slice of both.

```
imshowpair(mristack(:,:,10),diffusedImage(:,:,10),'montage')  
title('Noisy Image (Left) vs. Anisotropic-Diffusion-Filtered Image (Right)')
```

**Noisy Image (Left) vs. Anisotropic-Diffusion-Filtered Image (Right)**



Calculate the Naturalness Image Quality Evaluator (NIQE) score averaged over all slices in the volume. The NIQE score provides a quantitative measure of image quality that does not require a reference image. Lower NIQE scores reflect better perceptual image quality.

```
nframes = size(mristack,3);  
m = 0;  
d = 0;  
for i = 1:nframes  
    m = m + niqe(mristack(:,:,i));
```



```

    d = d + niqe(diffusedImage(:,:,i));
end
mAvg = m/nframes;
dAvg = d/nframes;
disp(['The NIQE score of the noisy volume is ',num2str(mAvg),'.'])

```

The NIQE score of the noisy volume is 5.7794.

```
disp(['The NIQE score using anisotropic diffusion is ',num2str(dAvg),'.'])
```

The NIQE score using anisotropic diffusion is 4.1391.

The NIQE score is consistent with the observation of reduced noise in the filtered image.

## Input Arguments

### I — Image to filter

2-D grayscale image | 3-D grayscale volume

Image to filter, specified as a 2-D grayscale image of size  $m$ -by- $n$  or a 3-D grayscale volume of size  $m$ -by- $n$ -by- $k$ .

---

**Note** To apply anisotropic diffusion filtering to a color image, use `imdiffusefilt` on each color channel independently.

---

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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `imdiffusefilt(I, 'NumberOfIterations', 4, 'Connectivity', 'minimal')` performs anisotropic diffusion on image `I`, using 4 iterations and minimal connectivity.

## **GradientThreshold — Gradient threshold**

numeric scalar | numeric vector

Gradient threshold, specified as the comma-separated pair consisting of 'GradientThreshold' and a numeric scalar or a numeric vector of length `NumberOfIterations`. The value of `GradientThreshold` controls the conduction process by classifying gradient values as an actual edge or as noise. Increasing the value of `GradientThreshold` smooths the image more. The default value is 10% of the dynamic range of the image. You can use the `imdiffuseest` function to estimate a suitable value of `GradientThreshold`.

## **NumberOfIterations — Number of iterations**

5 (default) | positive integer

Number of iterations to use in the diffusion process, specified as the comma-separated pair consisting of 'NumberOfIterations' and a positive integer. You can use the `imdiffuseest` function to estimate a suitable value of `NumberOfIterations`.

## **Connectivity — Connectivity**

'maximal' (default) | 'minimal'

Connectivity of a pixel to its neighbors, specified as the comma-separated pair consisting of 'Connectivity' and one of these values:

- 'maximal' — Considers 8 nearest neighbors for 2-D images, and 26 nearest neighbors for 3-D images
- 'minimal' — Considers 4 nearest neighbors for 2-D images, and 6 nearest neighbors for 3-D images

## **ConductionMethod — Conduction method**

'exponential' (default) | 'quadratic'

Conduction method, specified as the comma-separated pair consisting of 'ConductionMethod' and 'exponential' or 'quadratic'. Exponential diffusion favors high-contrast edges over low-contrast edges. Quadratic diffusion favors wide regions over smaller regions.

## Output Arguments

### **J** — Diffusion-filtered image

numeric array

Diffusion-filtered image, returned as a numeric array of the same size and data type as the input image, **I**.

## References

- [1] Perona, P., and J. Malik. "Scale-space and edge detection using anisotropic diffusion." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 12, No. 7, July 1990, pp. 629-639.
- [2] Gerig, G., O. Kubler, R. Kikinis, and F. A. Jolesz. "Nonlinear anisotropic filtering of MRI data." *IEEE Transactions on Medical Imaging*. Vol. 11, No. 2, June 1992, pp. 221-232.

## See Also

`imdiffuseest` | `imfilter` | `imgaussfilt` | `imguidedfilter` | `imnlmfilt` | `locallapfilt`

**Introduced in R2018a**

## imdilate

Dilate image

### Syntax

```
J = imdilate(I,SE)
J = imdilate(I,nhood)
J = imdilate( ____,packopt)
J = imdilate( ____,shape)
```

### Description

`J = imdilate(I,SE)` dilates the grayscale, binary, or packed binary image `I`, returning the dilated image, `J`. `SE` is a structuring element object or array of structuring element objects, returned by the `strel` or `offsetstrel` functions.

You optionally can perform the dilation using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`J = imdilate(I,nhood)` dilates the image `I`, where `nhood` is a matrix of 0s and 1s that specifies the structuring element neighborhood. `imdilate` determines the center element of the neighborhood by `floor((size(nhood)+1)/2)`.

This syntax is equivalent to `imdilate(I,strel(nhood))`.

`J = imdilate( ____,packopt)` specifies whether `I` is a packed binary image.

This syntax is not supported on a GPU.

`J = imdilate( ____,shape)` specifies the size of the output image.

### Examples

## Dilate Image with Vertical Line Structuring Element

Read a binary image into the workspace.

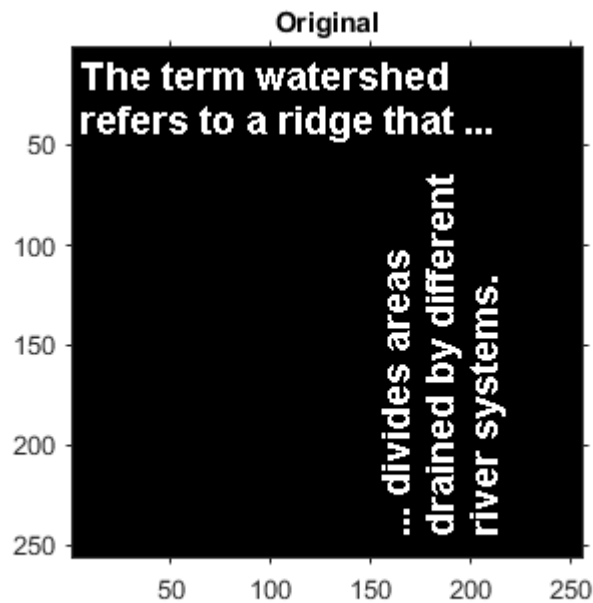
```
BW = imread('text.png');
```

Create a vertical line shaped structuring element.

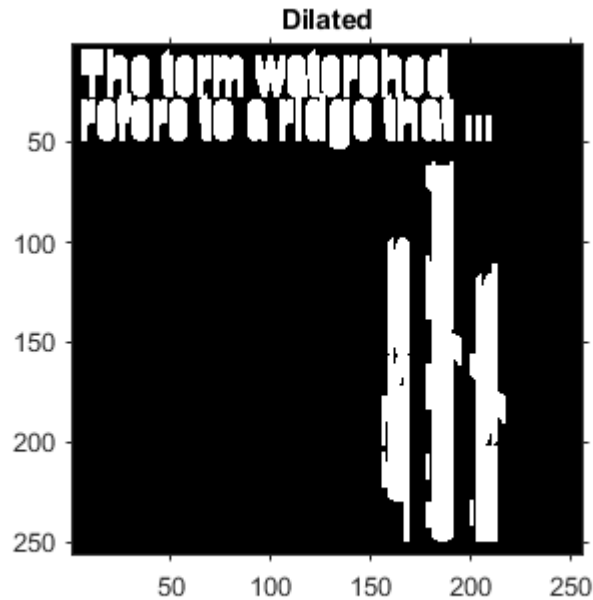
```
se = strel('line',11,90);
```

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 a 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.

```
imshowpair(originalI,dilatedI,'montage')
```



### 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: [3x1 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 = 3x3
```

```
    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;
```

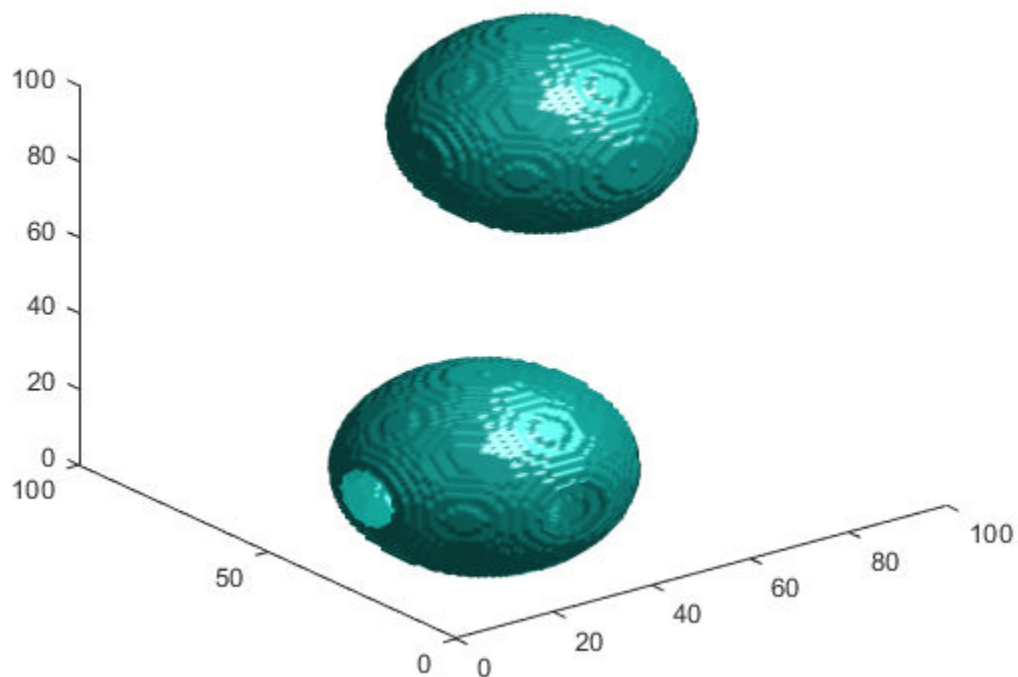
Dilate the 3D volume using a spherical structuring element.

```
se = strel('sphere',25);
dilatedBW = imdilate(BW,se);
```

Visualize the dilated image volume.

```
figure
isosurface(dilatedBW, 0.5)
```





## Input Arguments

### **I** — Input image

grayscale image | binary image | packed binary image | gpuArray

Input image, specified as a grayscale image, binary image, or packed binary image of any dimension.

To perform the dilation using a GPU, specify **I** as a `gpuArray` that contains an image of type `uint8` or `logical`.

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

## **SE — Structuring element**

`strel` object | `offsetstrel` object | array of `strel` objects | array of `offsetstrel` objects

Structuring element, specified as a scalar `strel` object or `offsetstrel` object. SE can also be an array of `strel` object or `offsetstrel` objects, in which case `imdilate` performs multiple dilations of the input image, using each structuring element in succession.

`imdilate` performs grayscale dilation for all images except images of data type `logical`. In this case, the structuring element must be flat and `imdilate` performs binary dilation.

If you perform the dilation using a GPU, then SE must be flat and two-dimensional.

## **nhood — Structuring element neighborhood**

matrix of 0s and 1s

Structuring element neighborhood, specified as a matrix of 0s and 1s.

Example: `[0 1 0; 1 1 1; 0 1 0]`

## **packopt — Indicator of packed binary image**

`'notpacked'` (default) | `'packed'`

Indicator of packed binary image, specified as one of the following.

Value	Description
<code>'notpacked'</code>	I is treated as a normal array.
<code>'ispacked'</code>	I is treated as a packed binary image as produced by <code>bwpack</code> . I must be a 2-D <code>uint32</code> array and SE must be a flat 2-D structuring element. The value of <code>shape</code> must be <code>'same'</code> .

---

**Note** The `packopt` argument is not supported on a GPU.

---

Data Types: `char` | `string`

**shape — Size of output image**

'same' (default) | 'full'

Size of the output image, specified as one of the following.

Value	Description
'same'	The output image is the same size as the input image. If the value of <code>packopt</code> is 'ispacked', then <code>shape</code> must be 'same'.
'full'	Compute the full dilation.

Data Types: char | string

## Output Arguments

**J — Dilated image**

grayscale image | binary image | packed binary image | gpuArray

Dilated image, returned as a grayscale image, binary image, or packed binary image. If the input image `I` is packed binary, then `J` is also packed binary. `J` has the same class as `I`.

If the dilation is performed using a GPU, then `J` is returned as a `gpuArray` that contains a grayscale or binary image of the same class as `I`.

## 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] on page 1-1279.

## Grayscale Dilation

In the general form of *grayscale dilation*, the structuring element has a height. The grayscale 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, grayscale dilation is performed with a flat structuring element ( $B(x, y) = 0$ ). Grayscale 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 “Understand Code Generation with Image Processing Toolbox”.
- The input image, `I`, 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.

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `gpuArray` input must be of type `uint8` or `logical` and the structuring element `SE` must be flat and two-dimensional.
- The `packopt` argument is not supported on the GPU.

For more information, see “Image Processing on a GPU”.

## See Also

### Functions

`bwpack` | `bwunpack` | `conv2` | `filter2` | `imclose` | `imerode` | `imopen`

### Objects

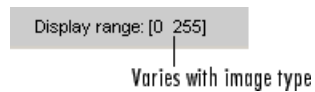
`offsetstrel` | `strel`

**Introduced before R2006a**

# imshow\_range

Display Range tool

Use the `imshow_range` function to create a Display Range tool. The Display Range tool shows the display range of the grayscale image or images in the figure.



## Syntax

```
imshow_range
imshow_range(h)
imshow_range(hparent,himage)
htool = imshow_range( ___ )
```

## Description

`imshow_range` creates a Display Range tool in the current figure.

`imshow_range(h)` creates a Display Range tool in the figure specified by the handle `h`.

`imshow_range(hparent,himage)` creates a Display Range tool in `hparent` that shows the display range of `himage`.

`htool = imshow_range( ___ )` returns a handle to the Display Range tool uipanel.

## Examples

### Create Display Range Tool

Display an image and include the Display Range tool.

```
imshow('bag.png');  
imshow;
```

Import a 16-bit DICOM image and display it with its default range and scaled range in the same figure.

```
dcm = dicomread('CT-MON02-16-ankle.dcm');  
subplot(1,2,1), imshow(dcm);  
subplot(1,2,2), imshow(dcm,[]);  
imshow;
```

## Input Arguments

### **h** — Handle to graphics object

handle

Handle to a figure, axes, uipanel, or image graphics object, specified as a handle. Axes, uipanel, or figure objects must contain at least one image object.

### **hparent** — Handle to figure or uipanel object

handle

Handle to a figure or uipanel object that contains the Display Range tool, specified as a handle.

### **himage** — Handle to images

handle | array of handles

Handle to one or more images, specified as a handle or an array of image handles.

## Output Arguments

### **htool** — Handle to Display Range tool

handle

Handle to Display Range tool uipanel, returned as a handle.



## Tips

- The Display Range 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.
- For an indexed, truecolor, or binary image, the display range is not applicable and is set to empty ( []).
- 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].

## See Also

imshow

**Introduced before R2006a**

# imdistline

Distance tool

## Description

An `imdistline` object is a type of `imline` that encapsulates a Distance tool, which consists of an interactive line over an image, paired with a text label that displays the distance between the line endpoints.

You can adjust the size and position of the line by using the mouse. The line also has a context menu that controls aspects of its appearance and behavior. For more information, see “Usage” on page 1-1512.

## Creation

## 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.

`h = imdistline(hparent)` creates a draggable Distance tool on the object specified by `hparent`.

`h = imdistline( ____,x,y)` creates a Distance tool with endpoints at the positions specified by `x` and `y`.

## Input Arguments

### **hparent** — Handle to parent object

handle

Handle to parent object, specified as a handle. The parent is typically an axes object, but can also be any other object that can be the parent of an `hgroup` object.

### **x** — x-coordinates of endpoints

2-element numeric vector

x-coordinates of endpoints, specified as a 2-element numeric vector.

Example: `h = imdistline(gca,[10 100],[20 40]);` sets the first endpoint at the (x, y) coordinate (10, 20) and the second endpoint at the coordinate (100, 40).

### **y** — y-coordinates of endpoints

2-element numeric vector

y-coordinates of endpoints, specified as a 2-element numeric vector.

Example: `h = imdistline(gca,[10 100],[20 40]);` sets the first endpoint at the (x, y) coordinate (10, 20) and the second endpoint at the coordinate (100, 40).

## Properties

### **Deletable** — ROI can be deleted


true (default) | false


ROI can be deleted, specified as `true` or `false`.

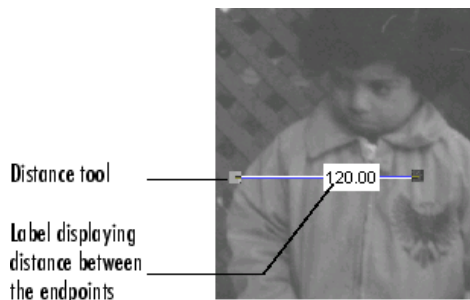
Data Types: `logical`

## Usage

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. Right-click the line to access the context menu.

Distance Tool Behavior	Context Menu Item
Export endpoint and distance data to the workspace	Select <b>Export to Workspace</b> from the context menu.
Toggle the distance label on/off.	Select <b>Show Distance Label</b> from the context menu.
Specify horizontal and vertical drag constraints	Select <b>Constrain Drag</b> from the context menu.
Change the color used to display the line.	Select <b>Set Color</b> from the context menu.
Delete the Distance tool object	Select <b>Delete</b> from the context menu.

## Object Functions

addNewPositionCallback	Add new-position callback to ROI object
createMask	Create mask within image
delete	Delete handle object
getAngleFromHorizontal	Return angle between Distance tool and horizontal axis
getColor	Get color used to draw ROI object
getDistance	Return distance between endpoints of Distance tool
getLabelHandle	Return handle to text label of Distance tool
getLabelTextFormatter	Return format of text label of Distance tool
getLabelVisible	Return visibility of text label of Distance tool

getPosition	Return current position of ROI object
getPositionConstraintFcn	Return function handle to current position constraint function
removeNewPositionCallback	Remove new-position callback from ROI object
resume	Resume execution of MATLAB command line
setColor	Set color used to draw ROI object
setConstrainedPosition	Set ROI object to new position
setLabelTextFormatter	Set format used to display text label of Distance tool
setLabelVisible	Set visibility of text label of Distance tool
setPosition	Move ROI object to new position
setPositionConstraintFcn	Set position constraint function of ROI object
wait	Block MATLAB command line until ROI creation is finished

---

**Note** Each function that accepts an `imline` also accepts an `imdistline`. These objects define position in the same way.

---

## Examples

### Insert Distance Tool into an Image

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.

```
imshow('pout.tif');
h = imdistline;
fcn = makeConstrainToRectFcn('imline',...
    get(gca,'XLim'),get(gca,'YLim'));
setDragConstraintFcn(h,fcn);
```

### Position Endpoints of Distance Tool

Position endpoints of the Distance tool at the specified locations.

```
imshow('pout.tif');
h = imdistline(gca,[10 100],[10 100]);
```

Delete the Distance tool.

```
delete(h)
```

## Use Distance Tool with Spatial Referencing

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 the initial position of distance tool on Harvard Bridge.

```
hline = imdistline(gca,[271 471],[108 650]);
setLabelTextFormatter(hline,'%02.0f meters');
```

Repeat the 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 the image to reflect desired units.

```
set(hImg,'XData',XDataInMeters,'YData',YDataInMeters);
set(gca,'XLim',XDataInMeters,'YLim',YDataInMeters);
```

Specify the initial position of distance tool on Harvard Bridge.

```
hline = imdistline(gca,[271 471],[108 650]);
setLabelTextFormatter(hline,'%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.

## See Also

`imline` | `imroi` | `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`.

If `X` is an integer array, elements in the output that exceed the range of integer type are truncated, and fractional values are rounded.

### 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 = 2x3 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 = 2x3 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.

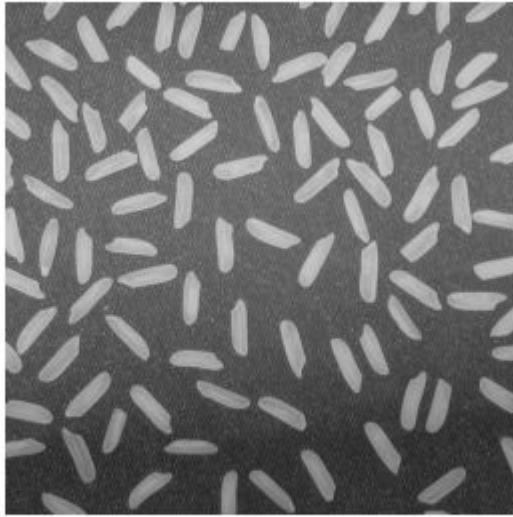
```
background = imopen(I,strel('disk',15));
```

Divide out the background from the image.

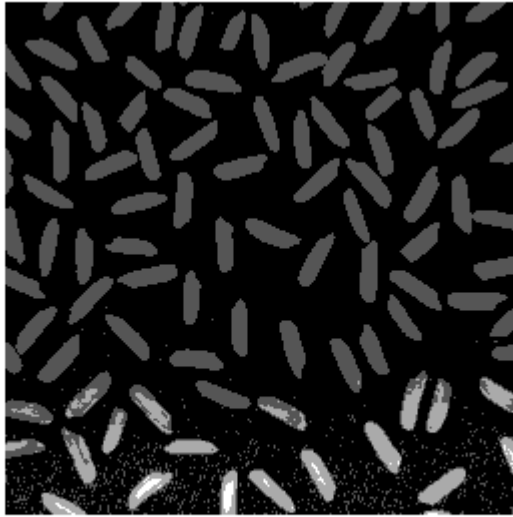
```
J = imdivide(I,background);
```

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.

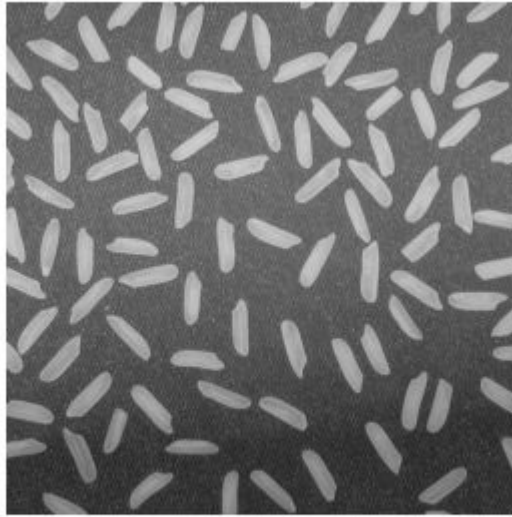
```
I = imread('rice.png');
```

Divide each value of the image by a constant factor of 2.

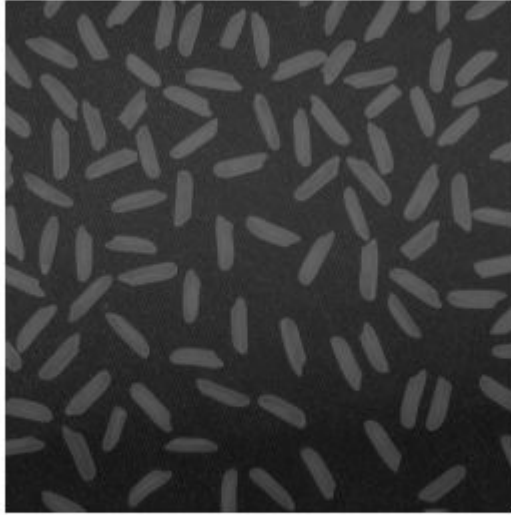
```
J = imdivide(I,2);
```

Display the original image and the processed image.

```
imshow(I)
```



```
figure  
imshow(J)
```



## Input Arguments

### **X — First array**

real, nonsparse, numeric or logical array

First array (dividend), specified as a real, nonsparse, numeric or logical array of any dimension.

### **Y — Second array**

real, nonsparse, numeric or logical array

Second array (divisor) to be divided from X, specified as a real, nonsparse, numeric or logical array. Y either has the same size and class as X, or Y is a scalar of type double.

## Output Arguments

### **Z — Quotient**

numeric array

Quotient, returned as a numeric array of the same size as *X*. *Z* is the same class as *X* unless *X* is logical, in which case *Z* is data type `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.

### **See Also**

`imabsdiff` | `imadd` | `imcomplement` | `imlincomb` | `immultiply` | `imsubtract`

**Introduced before R2006a**

# imellipse

Create draggable ellipse

---

**Note** `imellipse` is not recommended. Use `drawellipse` instead.

---

## Description

An `imellipse` object encapsulates an interactive ellipse over an image.

You can adjust the size and position of the ellipse by using the mouse. The ellipse also has a context menu that controls aspects of its appearance and behavior. For more information, see “Usage” on page 1-1299.

## Creation

## Syntax

```
h = imellipse
h = imellipse(hparent)
h = imellipse(hparent,position)
h = imellipse( ____,Name,Value)
```

## Description

`h = imellipse` begins interactive placement of an ellipse on the current axes, and returns an `imellipse` object.

`h = imellipse(hparent)` begins interactive placement of an ellipse on the object specified by `hparent`.

`h = imellipse(hparent,position)` creates a draggable ellipse at the position `position` on the object specified by `hparent`.

`h = imellipse( ____, Name, Value)` specifies name-value pairs that control the behavior of the ellipse.

## Input Arguments

### **hparent** — Handle to parent object

handle

Handle to parent object, specified as a handle. The parent is typically an axes object, but can also be any other object that can be the parent of an `hggroup` object.

### **position** — Position of ellipse

4-element vector

Position of the ellipse as defined by a bounding rectangle, specified as a 4-element vector of the form `[xmin ymin width height]`. The initial size of the bounding rectangle is width-by-height, and the upper-left corner of the rectangle is at the (x,y) coordinate (xmin,ymin).

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

### **PositionConstraintFcn** — Position constraint function

function handle

Position constraint function, specified as the comma-separated pair consisting of 'PositionConstraintFcn' and a function handle. `fcn` is called whenever the mouse is dragged. You can use this function to control where the ellipse can be dragged. See the help for the `setPositionConstraintFcn` function for information about valid function handles.

## Properties

### **Deletable** — ROI can be deleted

true (default) | false

ROI can be deleted, specified as `true` or `false`.

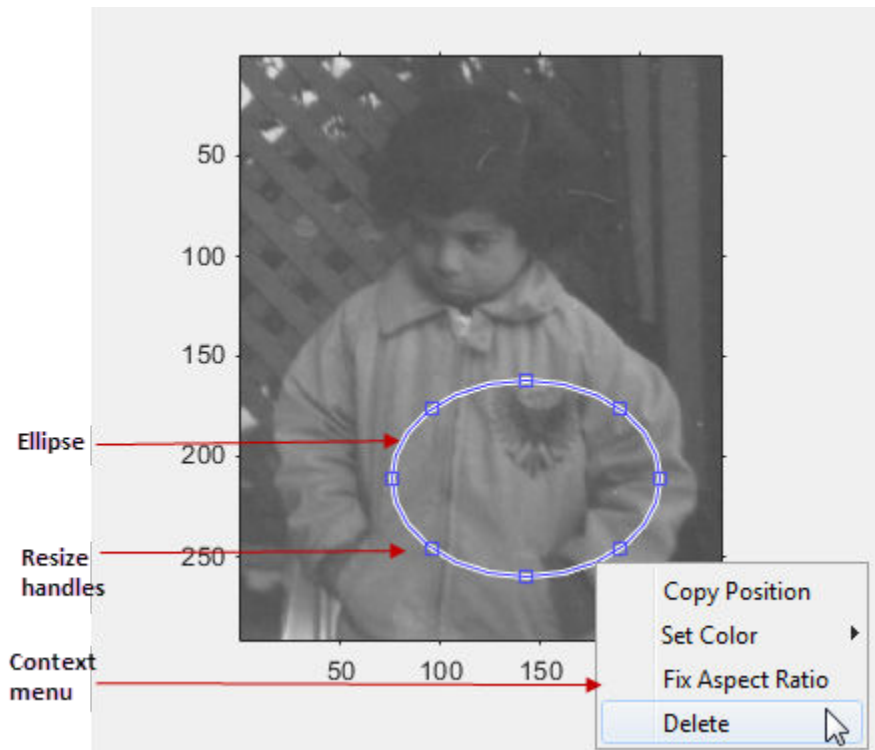


Data Types: logical



## Usage

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. Right-click on the ellipse to access this context menu.



The 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 <b>Set Color</b> from the context menu.
Retrieving the current position of the ellipse.	Move the pointer inside the ellipse. Right-click and select <b>Copy Position</b> from the context menu. <code>imellipse</code> copies a four-element position vector <code>[xmin ymin width height]</code> to the clipboard.
Preserving the current aspect ratio of the ellipse during resizing.	Move the pointer inside the ellipse. Right-click and select <b>Fix Aspect Ratio</b> from the context menu.
Deleting the ellipse	Move the pointer inside the ellipse. Right-click and select <b>Delete</b> 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>

## Object Functions

Each `imellipse` object supports a number of methods. Type methods `imellipse` to see a complete list.

<code>addNewPositionCallback</code>	Add new-position callback to ROI object
<code>createMask</code>	Create mask within image
<code>delete</code>	Delete handle object
<code>getColor</code>	Get color used to draw ROI object
<code>getPosition</code>	Return current position of ROI object
<code>getPositionConstraintFcn</code>	Return function handle to current position constraint function
<code>getVertices</code>	Return vertices on perimeter of ellipse ROI object
<code>removeNewPositionCallback</code>	Remove new-position callback from ROI object
<code>resume</code>	Resume execution of MATLAB command line

<code>setColor</code>	Set color used to draw ROI object
<code>setConstrainedPosition</code>	Set ROI object to new position
<code>setFixedAspectRatioMode</code>	Preserve aspect ratio when resizing ROI object
<code>setPosition</code>	Move ROI object to new position
<code>setPositionConstraintFcn</code>	Set position constraint function of ROI object
<code>setResizable</code>	Set resize behavior of ROI object
<code>wait</code>	Block MATLAB command line until ROI creation is finished

## Examples

### Update Title when Ellipse Moves

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.

```
imshow('coins.png')
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);
```

### Click and Drag to Place Ellipse

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.

```
imshow('coins.png')
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` | `imline` | `impoint` | `impoly` | `imrect` | `imroi` | `iptgetapi` |  
`makeConstrainToRectFcn`

**Introduced in R2007b**

# imerode

Erode image

## Syntax

```
J = imerode(I,SE)
J = imerode(I,nhood)
J = imerode( ____,packopt,m)
J = imerode( ____,shape)
```

## Description

`J = imerode(I,SE)` erodes the grayscale, binary, or packed binary image `I`, returning the eroded image, `J`. `SE` is a structuring element object or array of structuring element objects, returned by the `strel` or `offsetstrel` functions.

You optionally can perform the erosion using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`J = imerode(I,nhood)` erodes the image `I`, where `nhood` is a matrix of 0s and 1s that specifies the structuring element neighborhood. The `imerode` function determines the center element of the neighborhood by `floor((size(nhood)+1)/2)`.

This syntax is equivalent to `imerode(I,strel(nhood))`.

`J = imerode( ____,packopt,m)` specifies whether input image `I` is a packed binary image. `m` specifies the row dimension of the original unpacked image.

This syntax is not supported on a GPU.

`J = imerode( ____,shape)` specifies the size of the output image.

## 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.

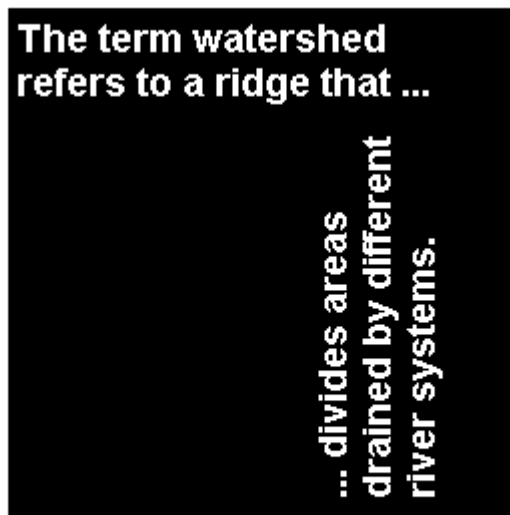
```
se = strel('line',11,90);
```

Erode the image with the structuring element.

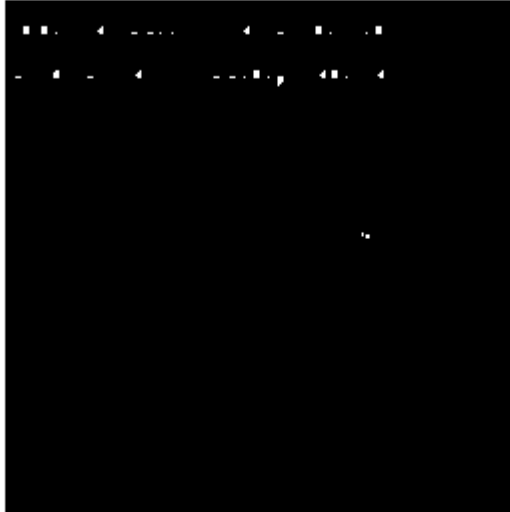
```
erodedBW = imerode(originalBW,se);
```

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 MRI Stack Volume Using Cubic Structuring Element

Create a binary volume.

```
load mristack  
BW = mristack < 100;
```

Create a cubic structuring element.

```
se = strel('cube',3)
```

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

```
    Neighborhood: [3x3x3 logical]  
    Dimensionality: 3
```

Erode the volume with a cubic structuring element.

```
erodedBW = imerode(BW, se);
```

## Input Arguments

### **I** — Input image

grayscale image | binary image | packed binary image | `gpuArray`

Input image, specified as a grayscale image, binary image, or packed binary image of any dimension.

To perform the erosion using a GPU, specify **I** as a `gpuArray` that contains an image of type `uint8` or `logical`.

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

### **SE** — Structuring element

`strel` object | `offsetstrel` object | array of `strel` objects | array of `offsetstrel` objects

Structuring element, specified as a scalar `strel` object or `offsetstrel` object. **SE** can also be an array of `strel` object or `offsetstrel` objects, in which case `imerode` performs multiple erosions of the input image, using each structuring element in succession.

`imerode` performs grayscale erosion for all images except images of data type `logical`. In this case, the structuring element must be flat and `imerode` performs binary erosion.

If you perform the erosion using a GPU, then **SE** must be flat and two-dimensional.

### **nhood** — Structuring element neighborhood

matrix of 0s and 1s

Structuring element neighborhood, specified as a matrix of 0s and 1s.

Example: `[0 1 0; 1 1 1; 0 1 0]`

### **packopt** — Indicator of packed binary image

'notpacked' (default) | 'packed'

Indicator of packed binary image, specified as one of the following.

Value	Description
'notpacked'	I is treated as a normal array.
'ispacked'	I is treated as a packed binary image as produced by <code>bwpack</code> . I must be a 2-D <code>uint32</code> array and SE must be a flat 2-D structuring element. The value of <code>shape</code> must be 'same'.

---

**Note** The `packopt` argument is not supported on a GPU.

---

Data Types: `char` | `string`

### **m** — Row dimension of original unpacked image

positive integer

Row dimension of the original unpacked image, specified as a positive integer.

Data Types: `double`

### **shape** — Size of output image

'same' (default) | 'full'

Size of the output image, specified as one of the following.

Value	Description
'same'	The output image is the same size as the input image. If the value of <code>packopt</code> is 'ispacked', then <code>shape</code> must be 'same'.
'full'	Compute the full erosion.

Data Types: `char` | `string`

## Output Arguments

### **J** — Eroded image

grayscale image | binary image | packed binary image | `gpuArray`

Eroded image, returned as a grayscale image, binary image, or packed binary image. If the input image *I* is packed binary, then *J* is also packed binary. *J* has the same class as *I*.

If the erosion is performed using a GPU, then *J* is returned as a `gpuArray` that contains a grayscale or binary image of the same class as *I*.

## Definitions

### Binary Erosion

The *binary erosion* of *A* by *B*, denoted  $A \ominus B$ , is defined as the set operation  $A \ominus B = \{z | (B_z \subseteq A)\}$ . 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] on page 1-1311.

### Grayscale Erosion

In the general form of *grayscale erosion*, the structuring element has a height. The grayscale erosion of  $A(x, y)$  by  $B(x, y)$  is defined as:

$$(A \ominus B)(x, y) = \min \{A(x + x', y + y') - B(x', y') \mid (x', y') \in D_B\},$$

$D_B$  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, grayscale erosion is performed with a flat structuring element ( $B(x, y) = 0$ ). Grayscale 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 erosion with a structuring element object that has a decomposition, imerode automatically uses binary image packing to speed up the erosion.

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 “Understand Code Generation with Image Processing Toolbox”.
- The input image, `I`, 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.

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `gpuArray` input must be of type `uint8` or `logical` and the structuring element `SE` must be flat and two-dimensional.
- The `packopt` argument is not supported on the GPU.

For more information, see “Image Processing on a GPU”.

## See Also

### Functions

`bwpack` | `bwunpack` | `conv2` | `filter2` | `imclose` | `imdilate` | `imopen`

### Objects

`offsetstrel` | `strel`

**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.

`BW = imextendedmax(I,H,conn)` computes the extended-maxima transform, where `conn` specifies the pixel connectivity.

## Examples

### Perform Extended-Maxima transform

Read image into workspace.

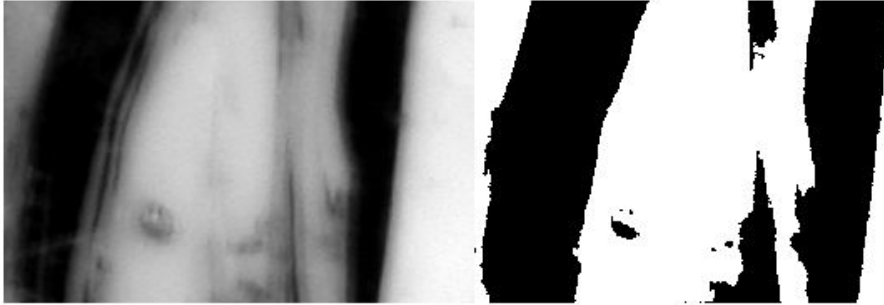
```
I = imread('glass.png');
```

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

numeric array

Input image, specified as a numeric array of any dimension.

Example: `I = imread('glass.png');`

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);`

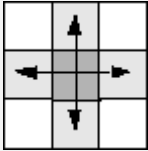
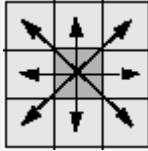
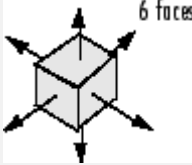
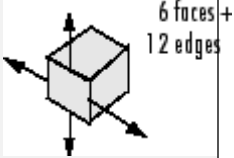
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

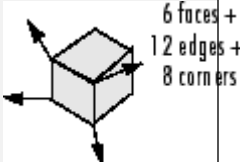
### **conn** — Pixel connectivity

`4` | `8` | `6` | `18` | `26` | 3-by-3-by- ... -by-3 matrix of 0s and 1s



Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	
18-connected	Pixels are connected if their faces or edges touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	

Value	Meaning	
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	

For higher dimensions, `imextendedmax` uses the default value `conndef(ndims(I), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, the optional third input argument, `conn`, must be a compile-time constant.

### 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.

`BW = imextendedmin(I,H,conn)` computes the extended-minima transform, which is the regional minima of the H-minima transform.

## Examples

### Perform Extended-Minima transform

Read image into the workspace.

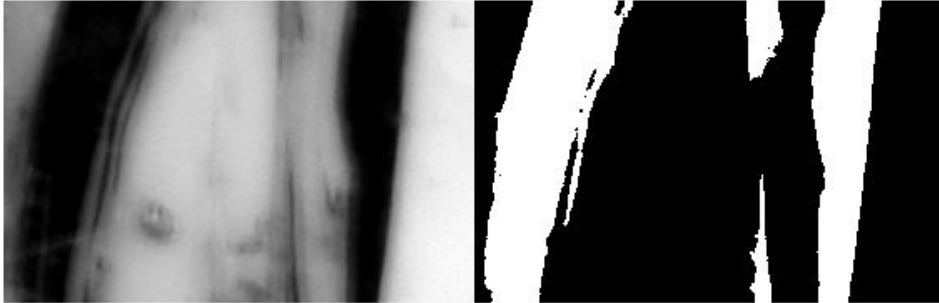
```
I = imread('glass.png');
```

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

numeric array

Input array, specified as a numeric array of any dimension.

Example: `I = imread('glass.png');`

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.

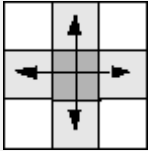
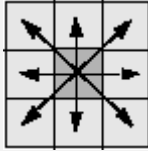
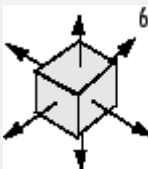
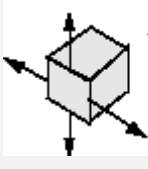
Example: `BW = imextendedmin(I,80);`

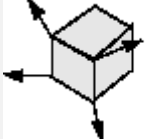
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **conn** — Pixel connectivity

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	 <p>6 faces</p>
18-connected	Pixels are connected if their faces or edges touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	 <p>6 faces + 12 edges</p>

Value	Meaning	
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	 <p>6 faces + 12 edges + 8 corners</p>

For higher dimensions, `imextendedmin` uses the default value `conndef(ndims(I), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, the optional third input argument, `conn`, must be a compile-time constant.

### See Also

`conndef` | `imextendedmax` | `imhmin` | `imreconstruct` | `imregionalmin`

**Introduced before R2006a**



# imfill

Fill image regions and holes

## Syntax

```
BW2 = imfill(BW,locations)
BW2 = imfill(BW,locations,conn)
BW2 = imfill(BW,'holes')
BW2 = imfill(BW,conn,'holes')

I2 = imfill(I)
I2 = imfill(I,conn)

BW2 = imfill(BW)
BW2 = imfill(BW,0,conn)
[BW2, locations_out] = imfill(BW)
```

## 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`.

You optionally can perform the flood-fill operation using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`BW2 = imfill(BW,locations,conn)` fills the area defined by `locations`, where `conn` specifies the connectivity.

`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.

`BW2 = imfill(BW,conn,'holes')` fills holes in the binary image `BW`, where `conn` specifies the connectivity.

`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.

`I2 = imfill(I,conn)` fills holes in the grayscale image `I`, where `conn` specifies the connectivity.

`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.

This syntax is not supported on a GPU.

`BW2 = imfill(BW,0,conn)` lets you override the default connectivity as you interactively specify locations.

This syntax is not supported on a GPU.

`[BW2, locations_out] = imfill(BW)` returns the locations of points selected interactively in `locations_out`. To use this syntax, `BW` must be a 2-D image.

This syntax is not supported on a GPU.

## 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')
```

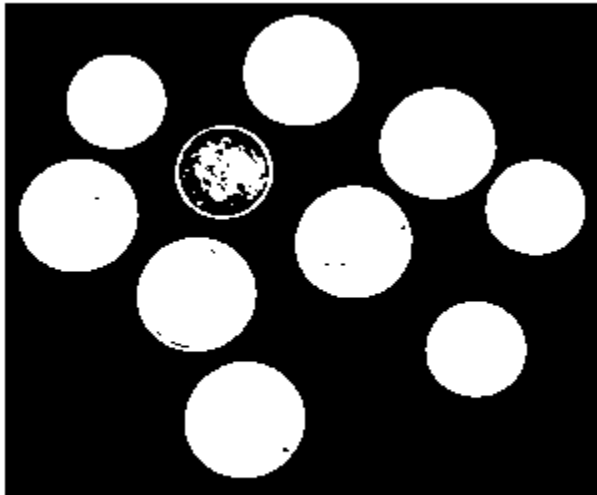
Original Image



Convert image to binary image.

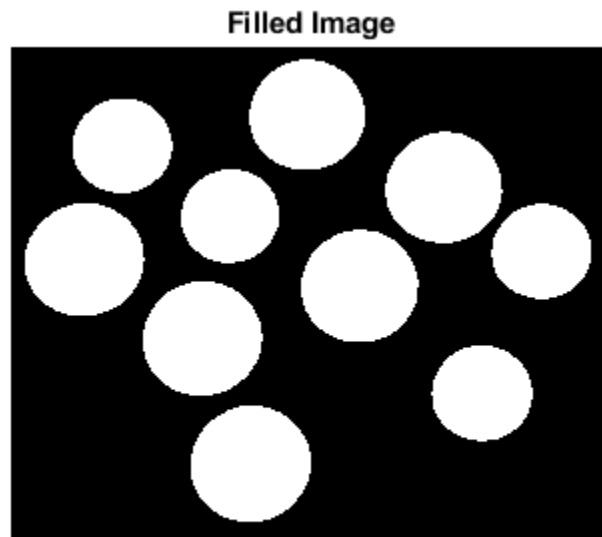
```
BW = imbinarize(I);  
figure  
imshow(BW)  
title('Original Image Converted to Binary Image')
```

**Original Image Converted to Binary Image**



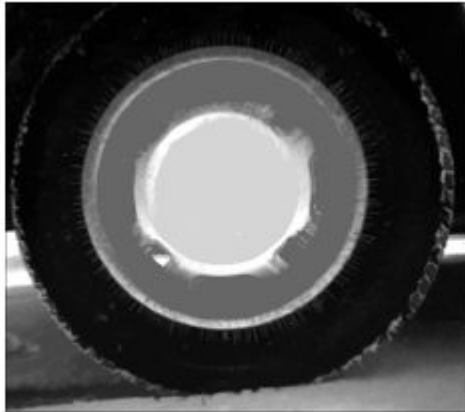
Fill holes in the binary image and display the result.

```
BW2 = imfill(BW, 'holes');  
figure  
imshow(BW2)  
title('Filled Image')
```



### 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

logical array | gpuArray

Input binary image, specified as a logical array of any dimension.

To fill image regions and holes using a GPU, specify **BW** as a `gpuArray` that contains a real, nonsparse, logical array of any dimension.

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

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

Data Types: `logical`

### **locations** — Linear indices identifying pixel locations

numeric vector of positive integers | 2-D numeric matrix of positive integers

Linear indices identifying pixel locations, specified as a numeric vector or 2-D numeric matrix of positive integers. If `locations` is a  $p$ -by-1 vector, then it contains the linear indices of the starting locations. If `locations` is a  $p$ -by-`ndims`(**BW**) matrix, then each row contains the array indices of one of the starting locations.

Example: `[3 3]`

Data Types: `double`

### **I** — Input grayscale image

numeric array | gpuArray

Input grayscale image, specified as a numeric array of any dimension.

To fill image regions and holes using a GPU, specify **I** as a `gpuArray` that contains a real, nonsparse, numeric array of any dimension.

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

Example: `I = gpuArray(imread('cameraman.tif'));`

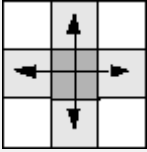
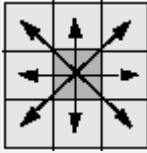
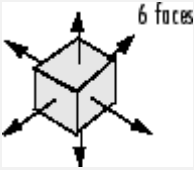
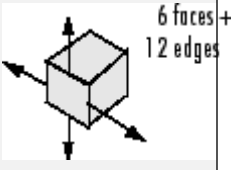
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

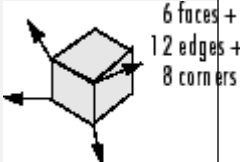


**conn — Pixel connectivity**

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 4 for 2-D images, and 6 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	
18-connected	Pixels are connected if their faces or edges touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	

Value	Meaning	
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	

For higher dimensions, `imfill` uses the default value `conndef(ndims(BW), 'minimal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

Data Types: `double` | `logical`

## Output Arguments

### **BW2 — Filled image**

`logical array` | `gpuArray`

Filled image, returned as logical array.

If image regions and holes are filled using a GPU, then `BW2` is returned as a `gpuArray` that contains a logical array.

### **locations\_out — Linear indices of pixel locations**

`numeric vector` | `numeric matrix`

Linear indices of pixel locations, returned as a numeric vector or matrix.

### **I2 — Filled grayscale image**

`numeric array` | `gpuArray`

Filled grayscale image, returned as a numeric array.

If image regions and holes are filled using a GPU, then `I2` is returned as a `gpuArray` that contains a numeric array.

## 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 “Understand Code Generation with Image Processing Toolbox”.
- The optional input arguments, `conn` and `'holes'`, must be 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.

## **GPU Arrays**

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Inputs must be 2-D, supporting only the 2-D connectivities (4 and 8). Does not support the interactive hole filling syntax.

For more information, see “Image Processing on a GPU”.

## **See Also**

`bwselect` | `conndef` | `imreconstruct` | `regionfill`

**Introduced before R2006a**

# imfilter

N-D filtering of multidimensional images

## Syntax

```
B = imfilter(A,h)
B = imfilter(A,h,options,...)
```

## Description

`B = imfilter(A,h)` filters the multidimensional array `A` with the multidimensional filter `h` and returns the result in `B`.

You optionally can filter a multidimensional array with a 2-D filter using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`B = imfilter(A,h,options,...)` performs multidimensional filtering according to one or more specified options.

## Examples

### Create Filter and Apply It

Read a color image into the workspace and display it.

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

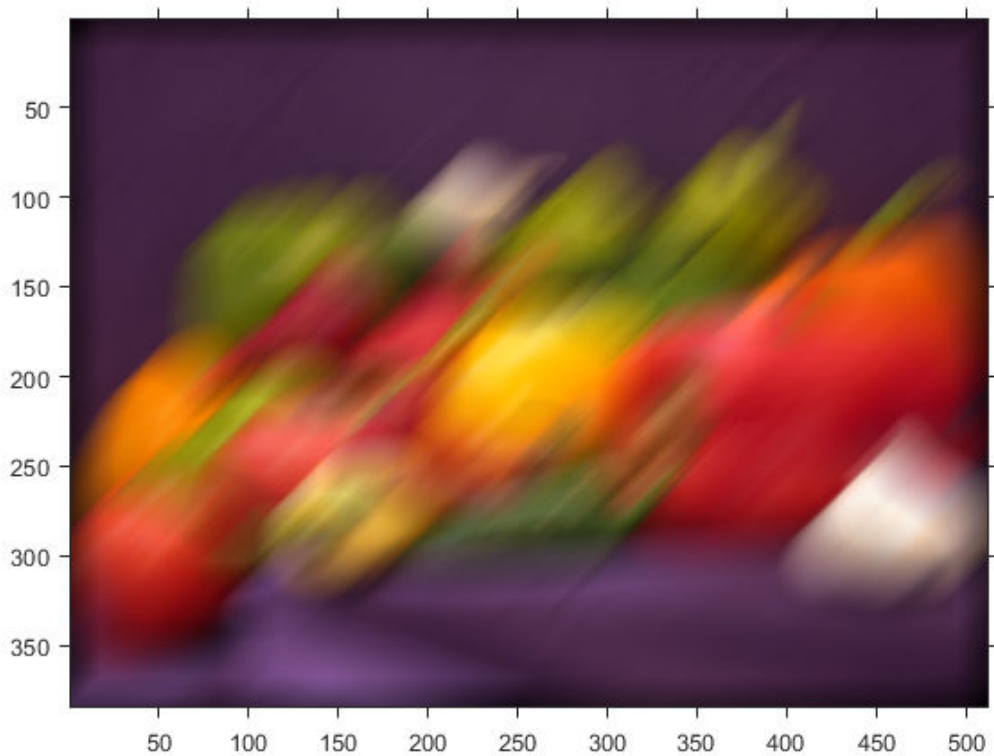


Create a motion-blur filter using the `fspecial` function.

```
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)
```



### **Filter Images Using `imfilter` with Convolution**

By default, `imfilter` uses correlation because the toolbox filter design functions produce correlation kernels. Use the optional parameter to use convolution.

Create a sample matrix.

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

```
A = 5×5
```



```

17   24   1   8   15
23   5   7   14  16
4    6   13  20  22
10   12  19  21   3
11   18  25   2   9

```

Create a filter.

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

Filter using correlation, the default.

```
imfilter(A,h)
```

```
ans = 5x5
```

```

24  -16  -16  14  -8
5   -16   9   9  -14
6    9   14   9  -20
12   9   9  -16  -21
18   14  -16  -16  -2

```

Filter using convolution, specifying `imfilter` with the optional parameter.

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

```
ans = 5x5
```

```

-24  16  16  -14  8
-5   16  -9  -9  14
-6   -9  -14  -9  20
-12  -9  -9  16  21
-18  -14  16  16  2

```

### Convert Image Class to Avoid Negative Output Values

In this example, the output of `imfilter` has negative values when the input is of class `double`. To avoid negative values, convert the image to a different data type before calling `imfilter`. For example, when the input type is `uint8`, `imfilter` truncates output values to 0. It might also be appropriate to convert the image to a signed integer type.

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

```
A = 5×5
```

```
    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
```

Filter the image with `imfilter`.

```
h = [-1 0 1];
imfilter(A,h)
```

```
ans = 5×5
```

```
    24   -16   -16    14    -8
     5   -16     9     9   -14
     6     9    14     9   -20
    12     9     9   -16   -21
    18    14   -16   -16    -2
```

Notice that the result has negative values. To avoid negative values in the output image, convert the input image to `uint8` before performing the filtering. Since the input to `imfilter` is of class `uint8`, the output also is of class `uint8`, and `imfilter` truncates negative values to 0.

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

```
ans = 5×5 uint8 matrix
```

```
    24     0     0    14     0
     5     0     9     9     0
     6     9    14     9     0
    12     9     9     0     0
    18    14     0     0     0
```

## 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)
```



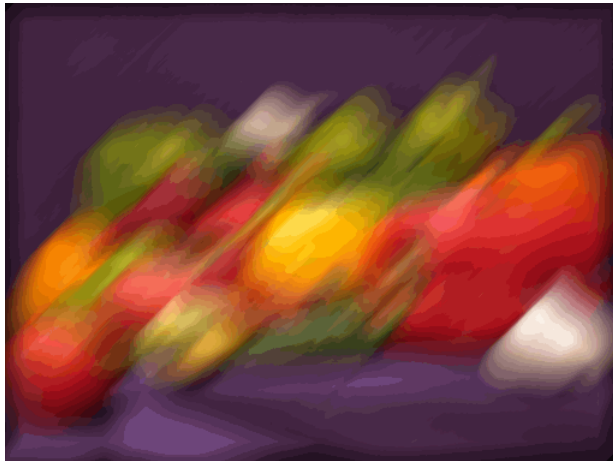
### Original Image

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

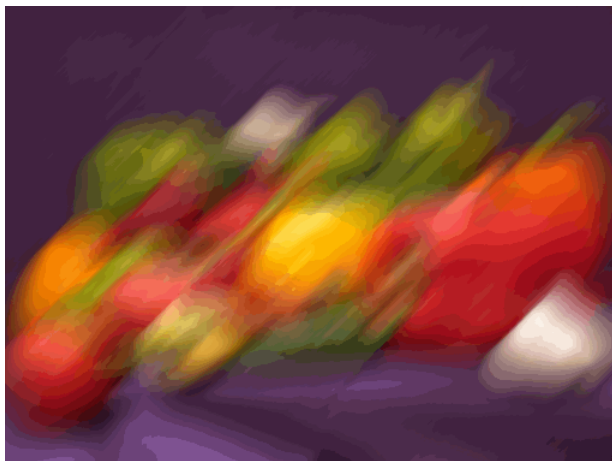
Note that `imfilter` is more memory efficient than some other filtering operations in that it outputs an array of the same data type as the input image array. In this example, the output is an array of `uint8`.

`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**

numeric array

Image to be filtered, specified as a numeric array of any class and dimension.

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

### **h — Multidimensional filter**

N-D array of data type `double`

Multidimensional filter, specified as an N-D array of data type `double`.

To perform the image filtering using a GPU, specify `h` as a vector or 2-D matrix of data type `double`.

Data Types: `double`

### **options — Options that control the filtering operation**

character vector | string scalar | numeric scalar

Options that control the filtering operation, specified as a character vector, string scalar, or numeric scalar. The following table lists all supported options.

**Boundary Options**

Option	Description
<b>Padding Options</b>	
numeric scalar, X	Input array values outside the bounds of the array are assigned the value X. When no padding 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.
<b>Output Size</b>	
'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.
<b>Correlation and Convolution Options</b>	
'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

Filtered image, returned as a numeric array of the same size and class as the input image, A.

## Tips

- This function may take advantage of hardware optimization for data types `uint8`, `uint16`, `int16`, `single`, and `double` to run faster.
- If you specify a large kernel, a kernel that contains large values, or specify an image containing large values, you can see different results between MATLAB and generated code using codegen for floating point data types. This happens because of accumulation errors due to different algorithm implementations.

## Algorithms

The `imfilter` function computes the value of each output pixel using double-precision, floating-point arithmetic. If the result exceeds the range of the data type, then `imfilter` truncates the result to the allowed range of the data type. If it is an integer data type, then `imfilter` rounds fractional values.

If the image is filtered using a GPU, then `imfilter` computes the value of each output pixel using either single- or double-precision floating point, depending on the data type of `A`. If `A` contains double-precision or `uint32` values, then `imfilter` uses double-precision values. For all other data types, `imfilter` uses single-precision. If `A` is an integer or logical array, then `imfilter` truncates output elements that exceed the range of the given type, and rounds fractional 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 “Understand 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.

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Input kernel must be 2-D.

For more information, see “Image Processing on a GPU”.

## See Also

`conv2` | `convn` | `filter2` | `fspecial`

## Topics

“Filter Grayscale and Truecolor (RGB) Images using `imfilter` Function”

“`imfilter` Boundary Padding Options”

“What Is Image Filtering in the Spatial Domain?”

**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].

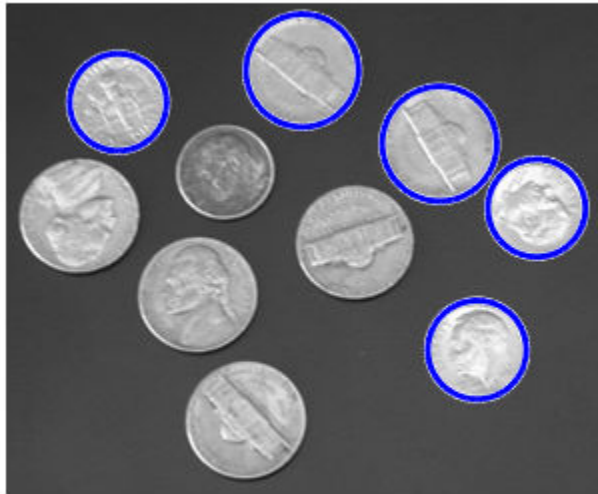
```
[centers, radii, metric] = imfindcircles(A,[15 30]);
```

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.

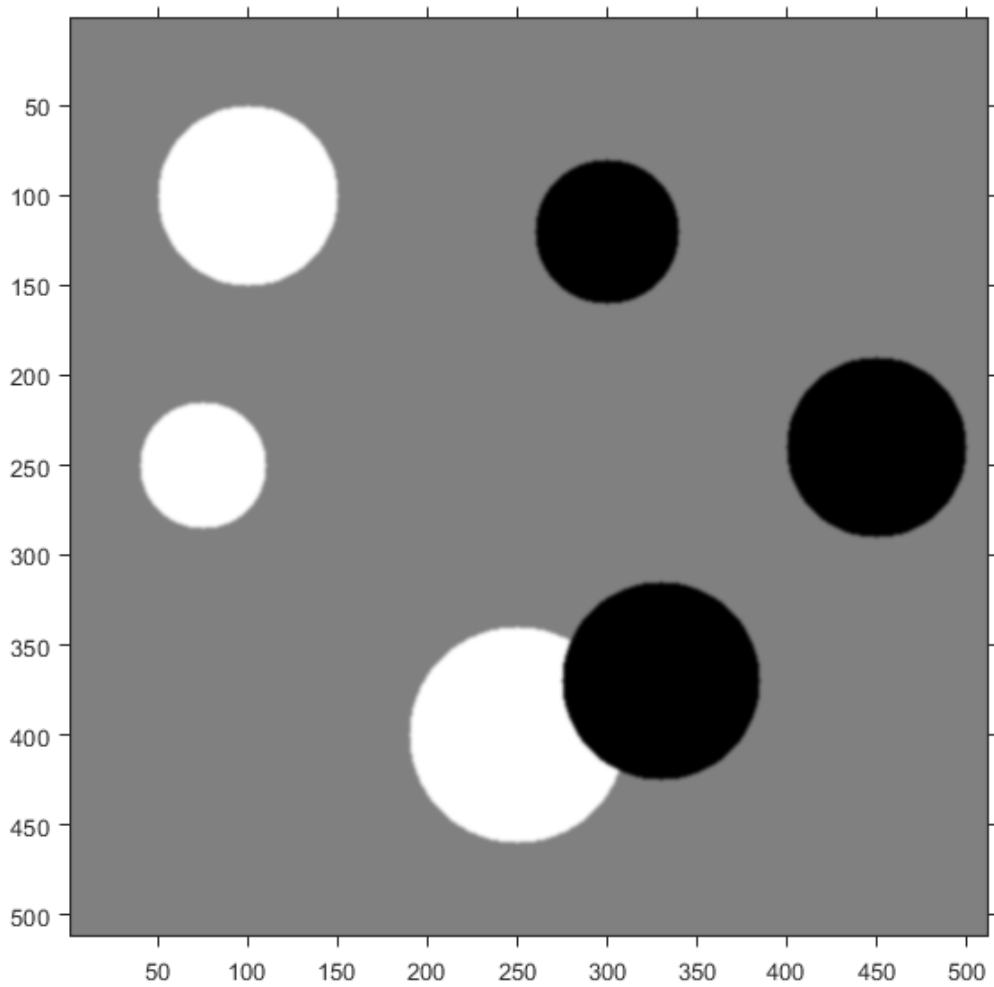
```
viscircles(centersStrong5, radiiStrong5, 'EdgeColor', 'b');
```



### Draw Lines Around Bright and Dark Circles in Image

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.

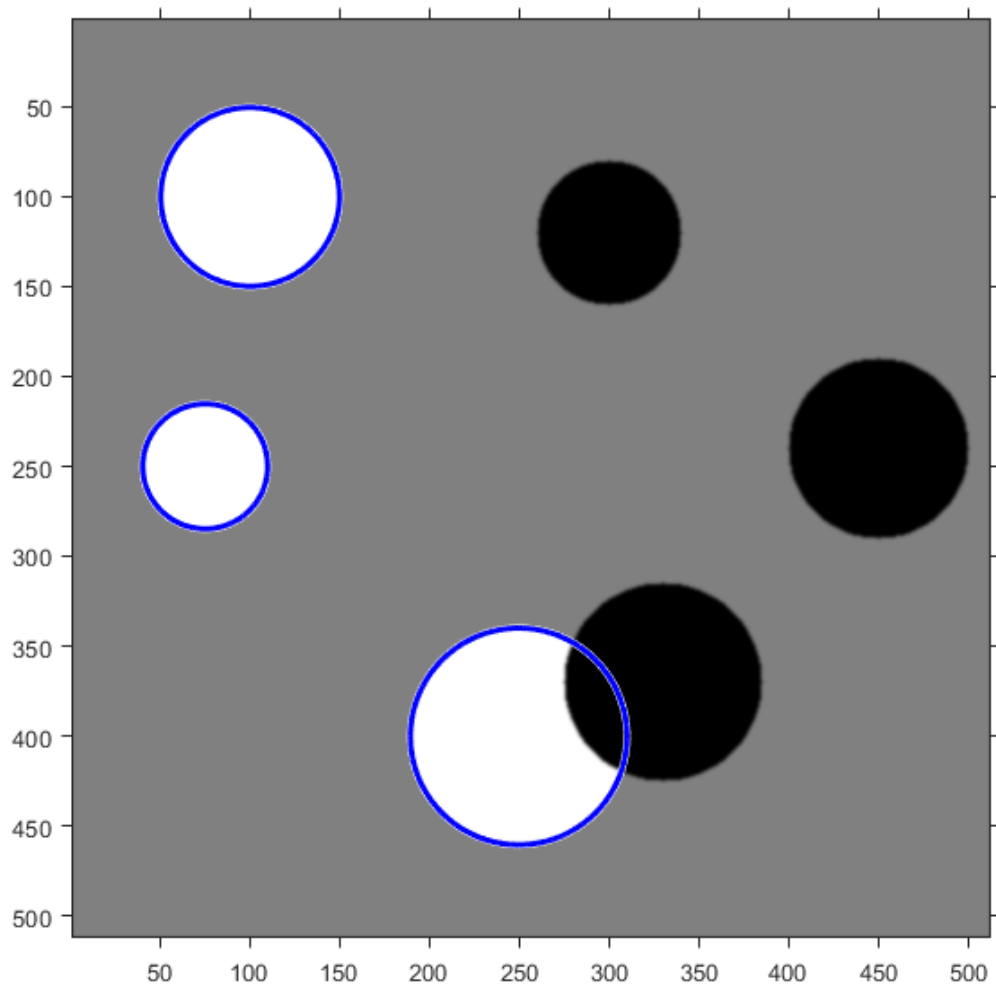
```
[centersBright, radiiBright] = imfindcircles(A,[Rmin Rmax], 'ObjectPolarity', 'bright');
```

Find all the dark circles in the image within the radius range.

```
[centersDark, radiiDark] = imfindcircles(A,[Rmin Rmax], 'ObjectPolarity', 'dark');
```

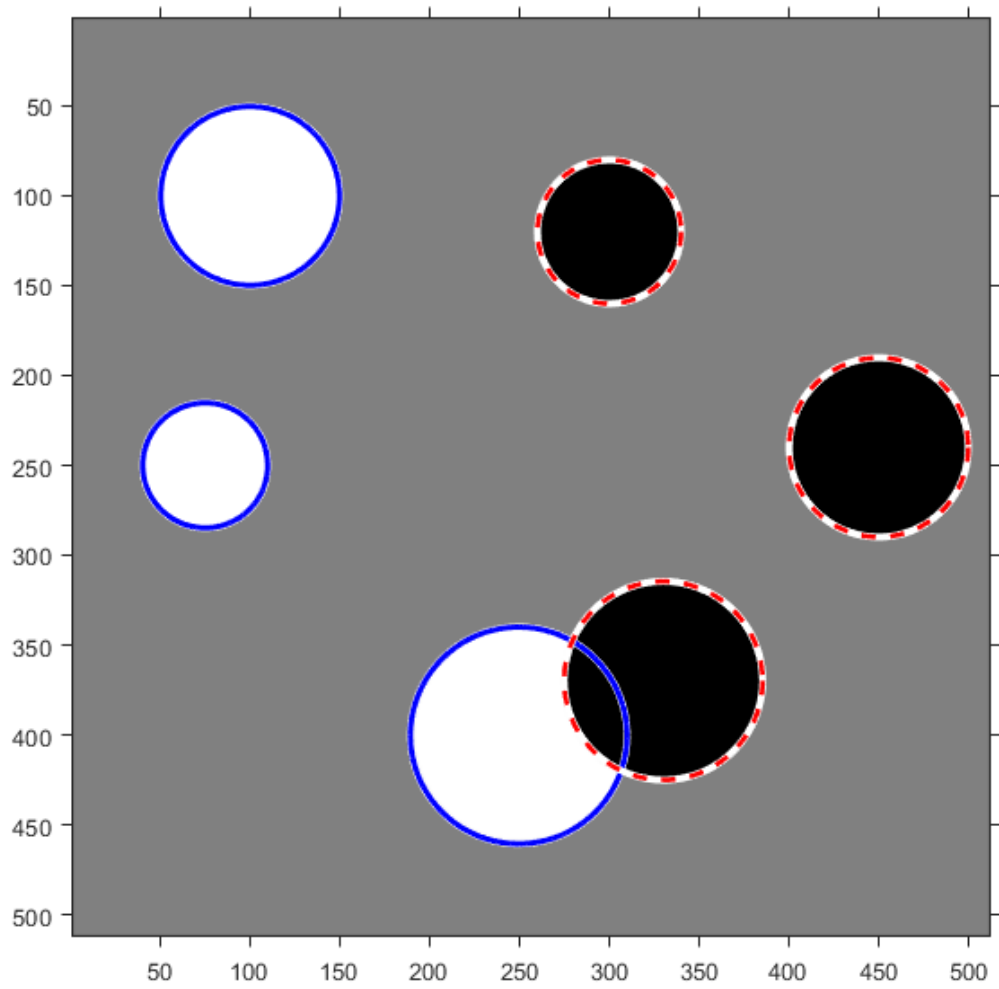
Draw blue lines around the edges of the bright circles.

```
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 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 `rmax < 3*rmin` and `(rmax-rmin) < 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.

## Algorithms

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.

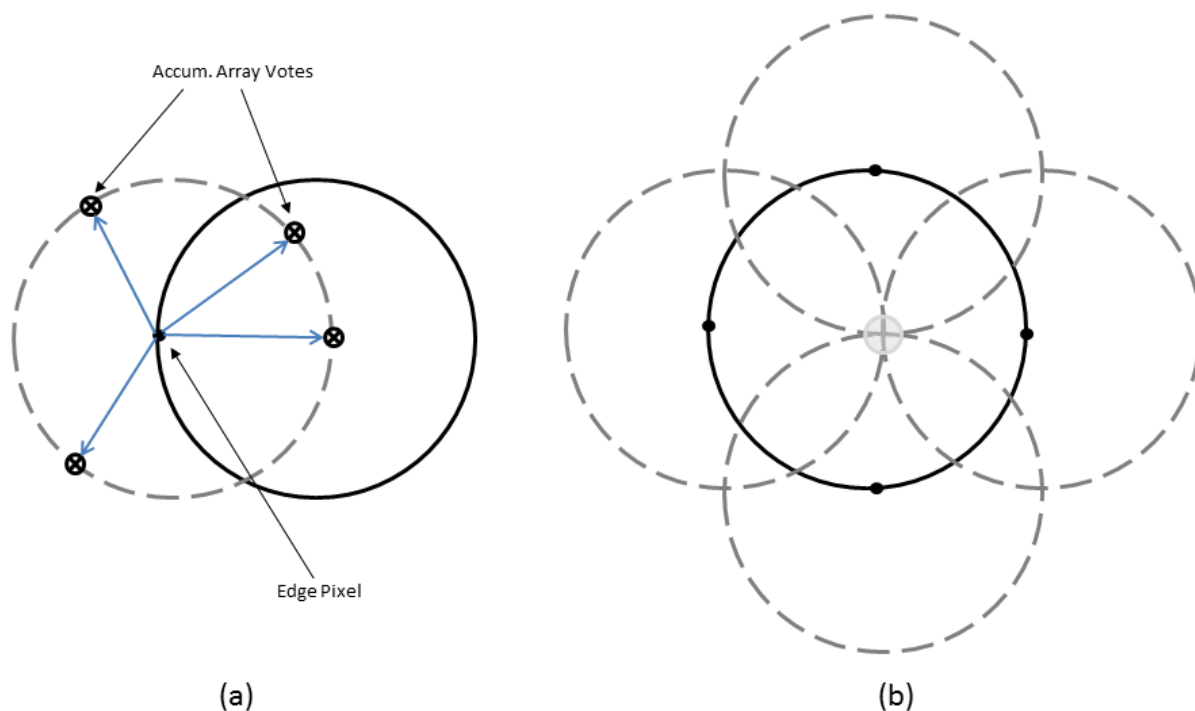


Figure 1: classical CHT voting pattern

## 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).

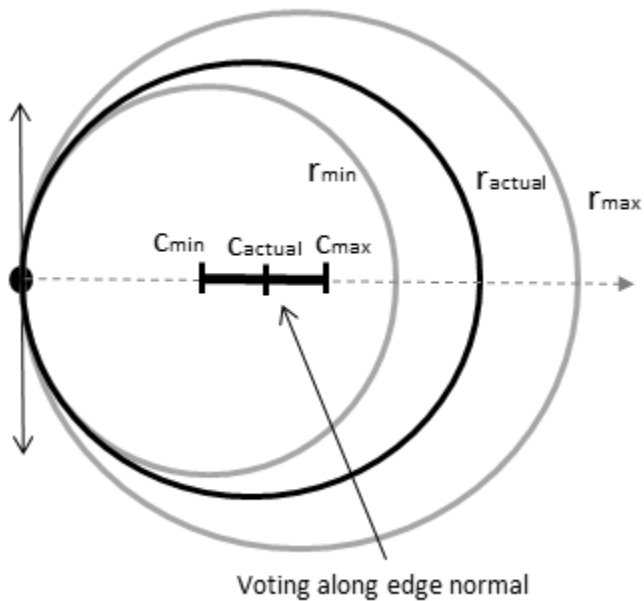


Figure 2: Voting mode: multiple radii, along direction of the gradient

$r_{\min}$	Minimum search radius
$r_{\max}$	Maximum search radius
$r_{\text{actual}}$	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 $r_{\max}$
$C_{\text{actual}}$	Center of the circle of radius $r_{\text{actual}}$

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 2 and 3 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 “Understand Code Generation with Image Processing Toolbox”.

- When generating code, all character vector input parameters and values must be a compile-time constant.

## **See Also**

`hough` | `houghlines` | `houghpeaks` | `viscircles`

**Introduced in R2012a**



# imflatfield

2-D image flat-field correction

## Syntax

```
J = imflatfield(I,sigma)
J = imflatfield(I,sigma,mask)
J = imflatfield(___, 'FilterSize',filterSize)
```

## Description

`J = imflatfield(I,sigma)` applies flat-field correction to the grayscale or RGB image `I`. The correction uses Gaussian smoothing with a standard deviation of `sigma` to approximate the shading component of `I`. The corrected image is returned in `J`.

`J = imflatfield(I,sigma,mask)` applies flat-field correction to image `I` only where the binary mask is `true`. Where the mask is `false`, the output image `J` contains the unmodified values of image `I`.

`J = imflatfield(___, 'FilterSize',filterSize)` specifies the size of the Gaussian smoothing filter.

## Examples

### Correct Shading Distortion in Grayscale Image

Load a grayscale image. This image has severe shading distortion on the left side and in the upper-right corner.

```
I = imread('printedtext.png');
imshow(I)
title('Distorted Image')
```

Distorted 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]
```

Perform the flat-field correction.

```
sigma = 30;
Iflatfield = imflatfield(I,sigma);
```

Display the result. The corrected image has more uniform brightness.

```
imshow(Iflatfield)
title(['Flat-Field Corrected Image, \sigma = ',num2str(sigma)])
```

Flat-Field Corrected Image,  $\sigma = 30$ 

## 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 25
```

## Correct Vignetting Defect in Color Image

Load a color image that has vignetting, or darkening of the corners.

```
I = imread('fabric.png');
imshow(I)
title('Image with Vignetting')
```

Image with Vignetting

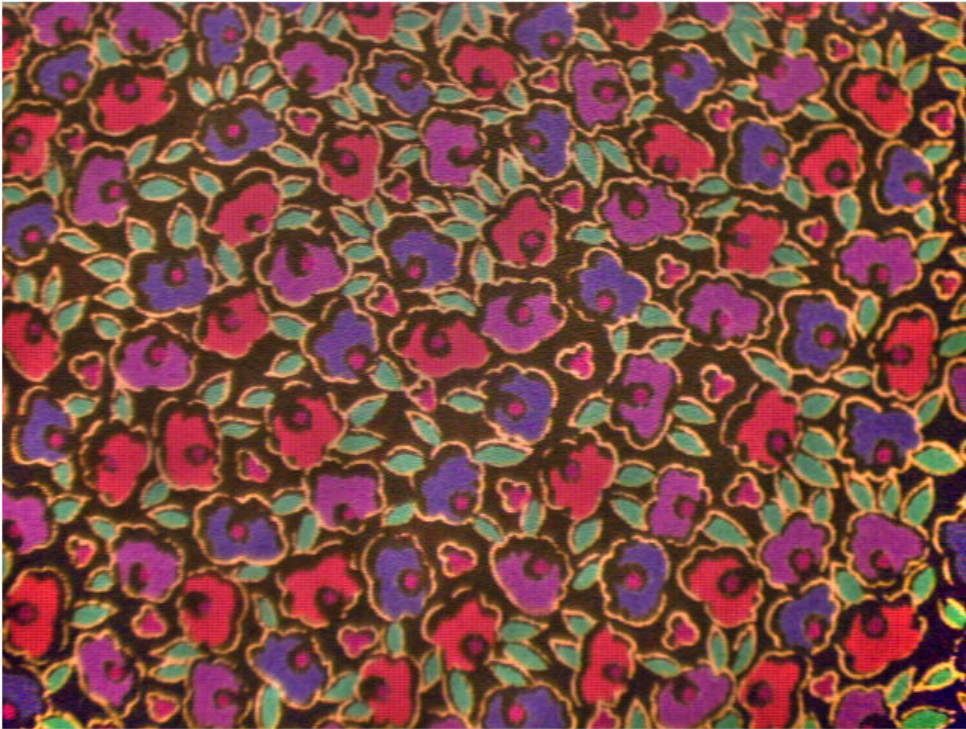


Perform the flat-field correction.

```
sigma = 20;  
Iflatfield = imflatfield(I,sigma);
```

Display the result. The corrected image has more uniform brightness.

```
imshow(Iflatfield)  
title(['Flat-Field Corrected Image, \sigma = ',num2str(sigma)])
```

Flat-Field Corrected Image,  $\sigma = 20$ 

### Apply Flat-Field Correction Using Binary Mask

Load a color image. This image has a shading defect in the lower right corner.

```
I = imread('hands1.jpg');  
imshow(I)  
title('Image with Dark Corner')
```

**Image with Dark Corner**



Try applying flat-field correction to the entire image.

```
sigma = 25;  
Iflatfield = imflatfield(I,25);  
imshow(Iflatfield)  
title(['Flat-Field Corrected Image, \sigma = ', num2str(sigma)])
```

**Flat-Field Corrected Image,  $\sigma = 25$** 

The shading defect in the corner is corrected, but the center of the image is too bright and the hand has changed color. To avoid this brightening artifact, apply flat-field correction just to the background of the image.

Load the mask of this image. In the original mask, `maskHand`, the segmented hand is the region of interest (ROI). Invert the mask so that the background is the ROI. Display the mask, which shows the ROI in white.

```
maskHand = imread('hands1-mask.png');  
maskBackground = ~maskHand;  
imshow(maskBackground)  
title('Background Mask')
```

**Background Mask**



Perform the flat-field correction on the background of the image using the mask `maskBackground`. The hand is not a region of interest in the mask, therefore flat-field correction is not applied to pixels in the hand.

```
Iflatfield2 = imflatfield(I,sigma,maskBackground);
```

Display the corrected image. The shading defect in the corner is corrected, and the hand retains its original color.

```
imshow(Iflatfield2)  
title(['Flat-Field Corrected Background, \sigma = ',num2str(sigma)])
```



**Flat-Field Corrected Background,  $\sigma = 25$** 

## Input Arguments

### **I — Distorted image**

2-D grayscale image | 2-D RGB image

Distorted image, specified as a 2-D grayscale image of size  $m$ -by- $n$  or a 2-D RGB image of size  $m$ -by- $n$ -by-3.

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

### **sigma — Standard deviation of Gaussian smoothing filter**

positive number | 2-element vector of positive numbers

Standard deviation of the Gaussian smoothing filter, specified as a positive number or a 2-element vector of positive numbers. If you specify a scalar, then `imflatfield` uses a square Gaussian kernel.

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

## **mask — Binary mask**

2-D numeric matrix | 2-D logical matrix

Binary mask, specified as a 2-D numeric or logical matrix of size  $m$ -by- $n$ . For numeric input, any nonzero pixels are considered to be 1 (`true`).

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

## **filterSize — Size of Gaussian filter**

positive, odd integer | 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, then `imflatfield` uses a square filter. The default filter size is  $2*\text{ceil}(2*\text{sigma})+1$ .

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

## **Output Arguments**

### **J — Corrected image**

2-D grayscale image | 2-D RGB image

Corrected image, returned as a 2-D grayscale or RGB image of the same size and data type as the input image, `I`.

## **Tips**

- When `I` is an RGB image, then `imflatfield` converts the image to the HSV color space using `rgb2hsv` and applies the flat-field correction to the HSV Value channel. The image is converted back to RGB color space by using `hsv2rgb`.
- If you specify a `mask`, then `imflatfield` dilates the mask and pads the image boundaries to reduce edge artifacts during the flat-field estimation.

## See Also

[hsv2rgb](#) | [rgb2hsv](#)

**Introduced in R2018b**

## imfreehand

Create draggable freehand region

**Note** `imfreehand` is not recommended. Use `drawfreehand` or `drawassisted` instead.

### Description

An `imfreehand` object encapsulates an interactive freehand region over an image.

You can add vertices and adjust the size and position of the polygon by using the mouse. The polygon also has a context menu that controls aspects of its appearance and behavior. For more information, see “Usage” on page 1-1376.

### Creation

### Syntax

```
h = imfreehand
h = imfreehand(hparent)
h = imfreehand( ____,Name,Value)
```

### Description

`h = imfreehand` begins interactive placement of a freehand region on the current axes, and returns an `imfreehand` object.

`h = imfreehand(hparent)` begins interactive placement of a freehand region on the object specified by `hparent`.

`h = imfreehand( ____,Name,Value)` specifies name-value pairs that control the behavior of the freehand region.

## Input Arguments

### **hparent — Handle to parent object**

handle

Handle to parent object, specified as a handle. The parent is typically an axes object, but can also be any other object that can be the parent of an hggroup object.

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

### **PositionConstraintFcn — Position constraint function**

function handle

Position constraint function, specified as the comma-separated pair consisting of 'PositionConstraintFcn' and a function handle. `fcn` is called whenever the mouse is dragged. You can use this function to control where the freehand region can be dragged. See the help for the `setPositionConstraintFcn` function for information about valid function handles.

### **Closed — Freehand region is closed**

true (default) | false

Freehand region is closed, specified as the comma-separated pair consisting of 'Closed' and true or false. When set to true (the default), `imfreehand` draws a straight line to connect the endpoints of the freehand line to create a closed region. If set to false, `imfreehand` leaves the region open.

Data Types: logical

## Properties

### **Deletable — ROI can be deleted**

true (default) | false

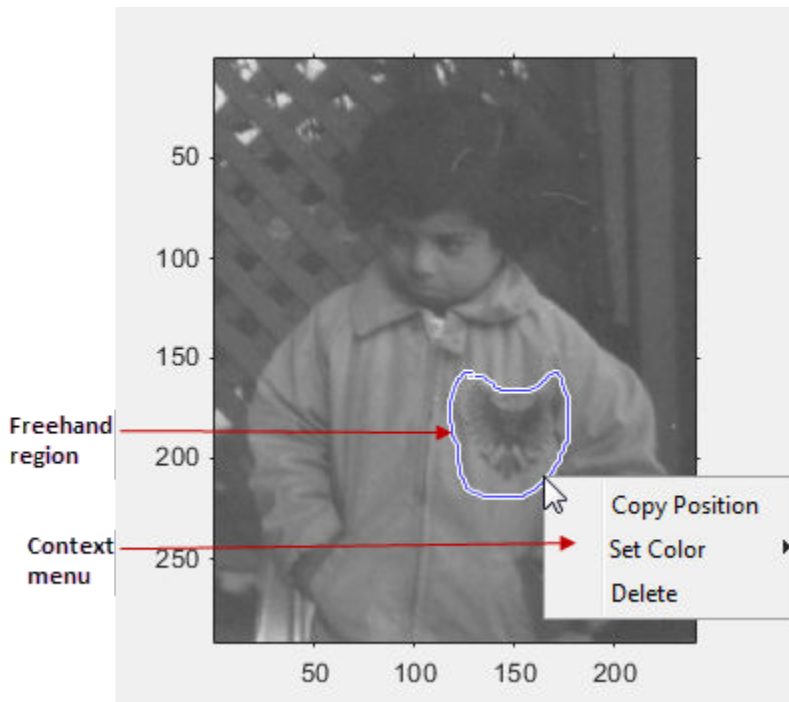
ROI can be deleted, specified as true or false.

Data Types: logical


## Usage

When you call `imfreehand` with an interactive syntax, the pointer changes to a cross hairs  $\oplus$  when positioned over an image. Click and drag the mouse to draw the freehand region and adjust the position of the 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 freehand region also supports a context menu that you can use to control aspects of its appearance and behavior.



The 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 <b>Set Color</b> from the context menu.
Retrieving the current position of the freehand region.	Move the pointer inside the freehand region. Right-click and select <b>Copy Position</b> from the context menu. <code>imfreehand</code> copies an $n$ -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 <b>Delete</b> 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>

## Object Functions

Each `imfreehand` object supports a number of methods. Type methods `imfreehand` to see a complete list.

<code>addNewPositionCallback</code>	Add new-position callback to ROI object
<code>createMask</code>	Create mask within image
<code>delete</code>	Delete handle object
<code>getColor</code>	Get color used to draw ROI object
<code>getPosition</code>	Return current position of ROI object
<code>getPositionConstraintFcn</code>	Return function handle to current position constraint function
<code>removeNewPositionCallback</code>	Remove new-position callback from ROI object
<code>resume</code>	Resume execution of MATLAB command line
<code>setClosed</code>	Set closure behavior of ROI object
<code>setColor</code>	Set color used to draw ROI object
<code>setConstrainedPosition</code>	Set ROI object to new position
<code>setPositionConstraintFcn</code>	Set position constraint function of ROI object
<code>wait</code>	Block MATLAB command line until ROI creation is finished

## Examples

### Click and Drag to Place Freehand Region

Interactively place a closed freehand region of interest by clicking and dragging over an image.

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

Interactively move the freehand region by clicking and dragging. Use the `wait` function to block the MATLAB command line. Double-click on the freehand region to resume execution of the MATLAB command line.

```
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

`imellipse` | `imline` | `impoint` | `impoly` | `imrect` | `imroi` | `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.

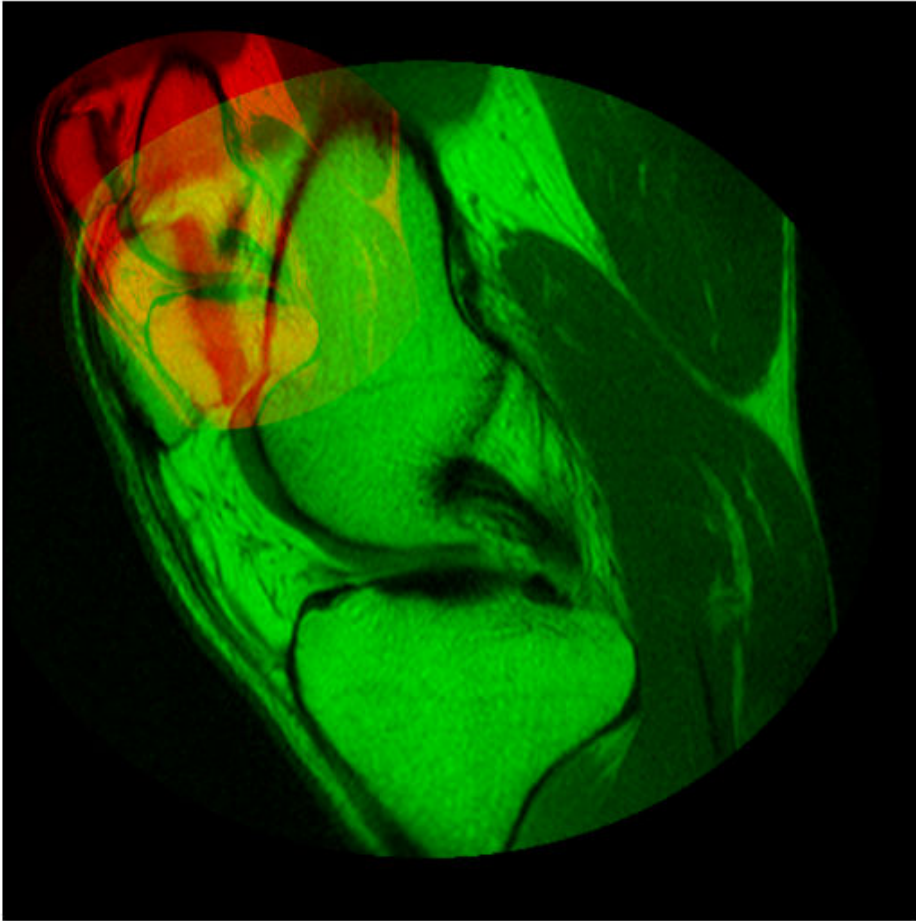
```
RB.XWorldLimits = RA.XWorldLimits;  
RB.YWorldLimits = RA.YWorldLimits;
```

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)
```

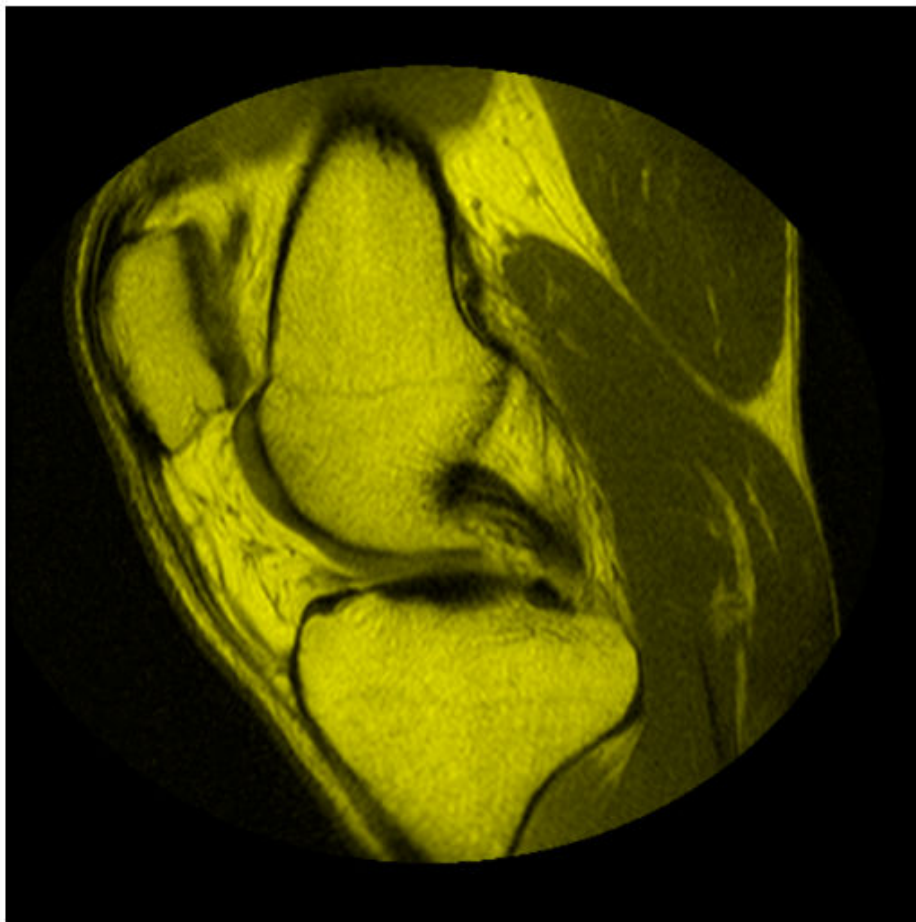


Create a new fused image, this time using the spatial referencing information in RA and RB.

```
[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.
'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.

## 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 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:

<code>'independent'</code>	Scales the intensity values of A and B independently when C is created.
<code>'joint'</code>	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.
<code>'none'</code>	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:

[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

## 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

`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,wavelength,orientation,Name,Value,...)
[mag, phase] = imgaborfilt(A,gaborbank)
```

## 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,wavelength,orientation,Name,Value,...)` applies a single Gabor filter using name-value pairs to control various aspects of filtering.

`[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)`.

## Examples

### Apply Single Gabor Filter to Input Image

Read image into the workspace.

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

Convert image to grayscale.

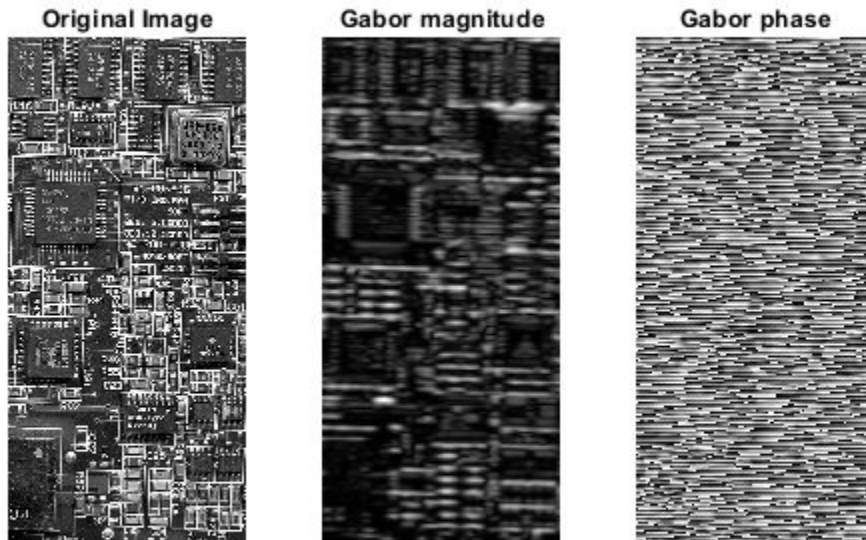
```
I = rgb2gray(I);
```

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);
```

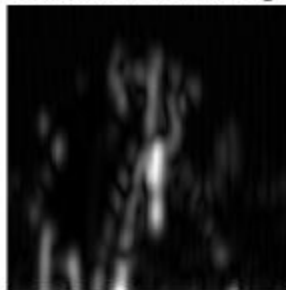
Display results. The figure shows the magnitude response for each filter.

```
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
```

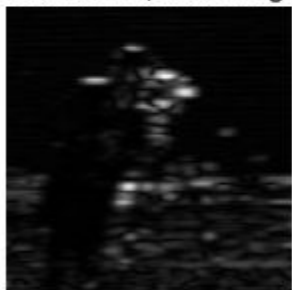
**Orientation=0, Wavelength=4**



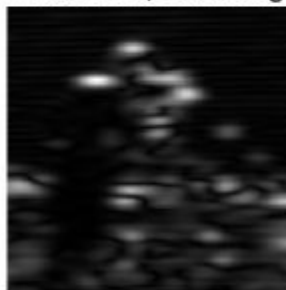
**Orientation=0, Wavelength=8**



**Orientation=90, Wavelength=4**



**Orientation=90, Wavelength=8**



## 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 `Infs` or `NaNs`, 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 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

`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)
```

## Description

`B = imgaussfilt(A)` filters image `A` with a 2-D Gaussian smoothing kernel with standard deviation of 0.5, and returns the filtered image in `B`.

You optionally can perform the filtering using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`B = imgaussfilt(A, sigma)` filters image `A` with a 2-D Gaussian smoothing kernel with standard deviation specified by `sigma`.

`B = imgaussfilt( ____, Name, Value)` uses name-value pair arguments to control aspects of the filtering.

## 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

numeric array | `gpuArray`

Image to be filtered, specified as a numeric array of any dimension.

To perform the filtering using a GPU, specify **A** as a `gpuArray` that contains a numeric array.

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

### **sigma** — Standard deviation of the Gaussian distribution

0.5 (default) | positive number | 2-element vector of positive numbers

Standard deviation of the Gaussian distribution, specified as a positive number or a 2-element vector of positive numbers. If you specify a scalar, then `imgaussfilt` uses a square Gaussian kernel.

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 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**

positive, odd integer | 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, then `imgaussfilt` uses a square filter. The default filter size is  $2 * \text{ceil}(2 * \text{sigma}) + 1$ .

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

**Padding — Image padding**

'replicate' (default) | numeric scalar | 'circular' | 'symmetric'

Image padding, specified as one of the following.

Value	Description
numeric scalar	Pad image with elements of constant value.
'circular'	Pad with circular repetition of elements within the dimension.
'replicate'	Pad by repeating border elements of array.
'symmetric'	Pad image with mirror reflections of itself.

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

**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
'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.

Data Types: `char` | `string`

## Output Arguments

### **B — Filtered image**

`numeric array` | `gpuArray`

Filtered image, returned as a numeric array of the same class and size as the input image, A.

If the image is filtered using a GPU, then B is returned as a `gpuArray` containing a numeric array.

## Tips

- If image A contains `Inf`s or `NaN`s, then 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 `Inf`s and `NaN`s in the output in a manner similar to `imfilter`, consider setting the `'FilterDomain'` parameter to `'spatial'`.
- If you set the `'FilterDomain'` parameter to `'auto'`, then `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, then `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.

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

## 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)
```

## Description

`B = imgaussfilt3(A)` filters 3-D image `A` with a 3-D Gaussian smoothing kernel with standard deviation of 0.5, and returns the filtered image in `B`.

You optionally can perform the filtering using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`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)` uses name-value pair arguments to control aspects of the filtering.

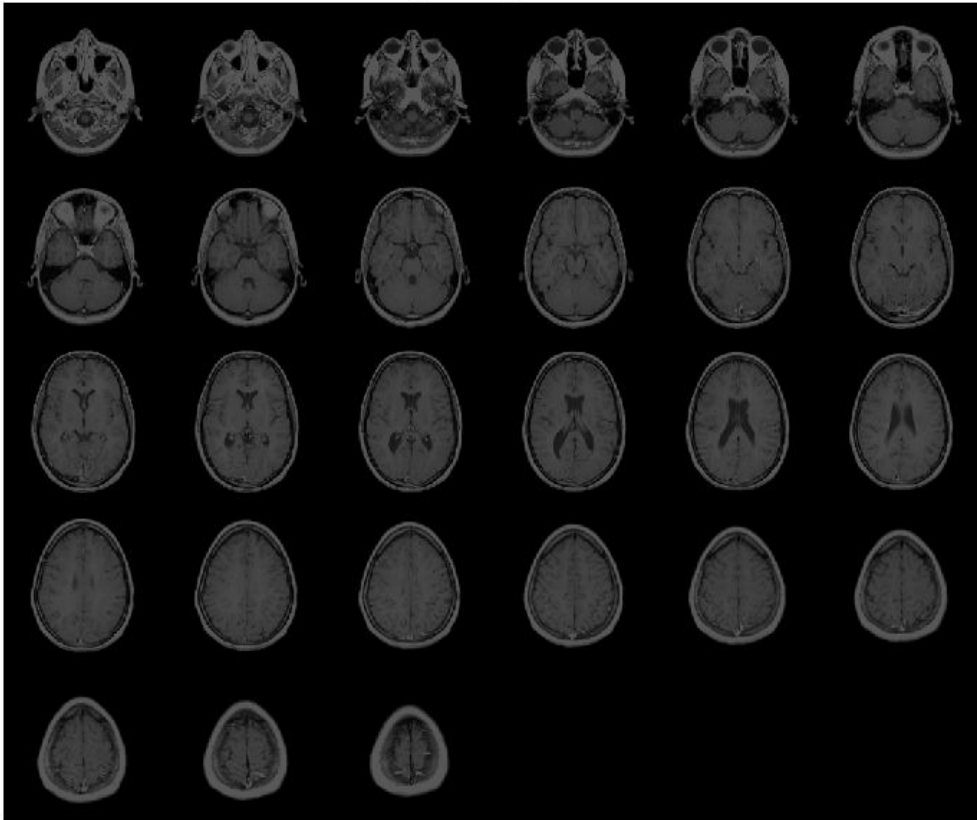
## Examples

### 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')
```

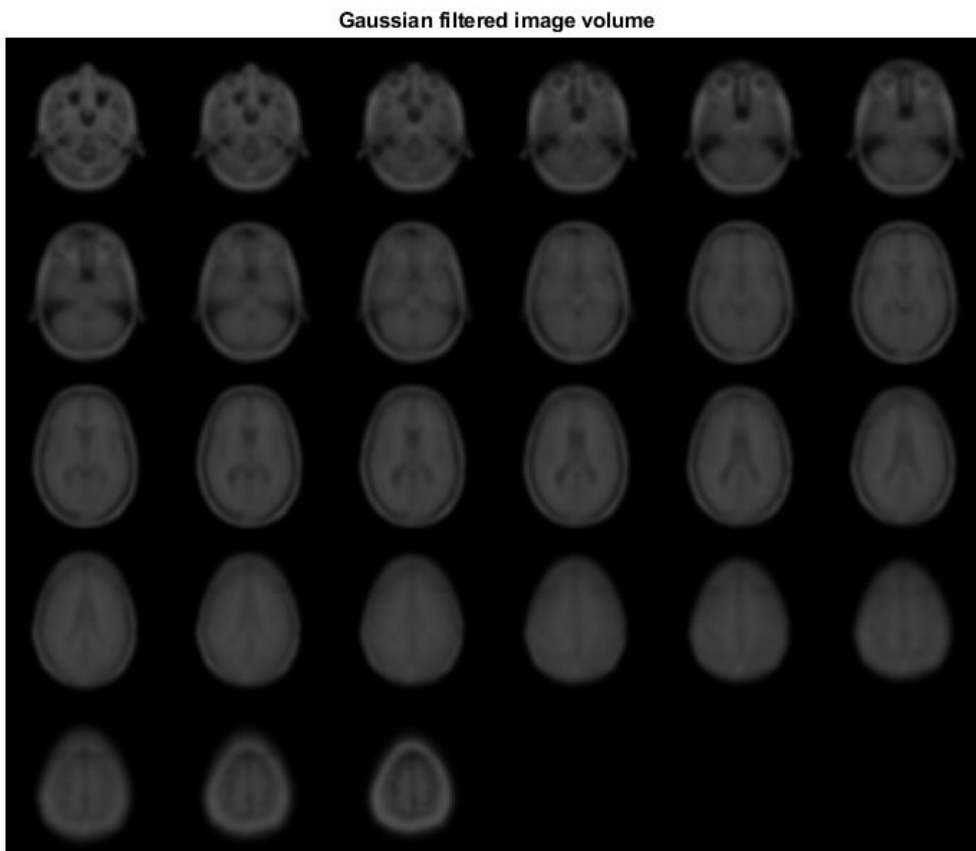
Original image volume



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 numeric array | `gpuArray`

Image to be filtered, specified as a 3-D numeric array.

To perform the filtering using a GPU, specify **A** as a `gpuArray` that contains a 3-D numeric array.

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

### **sigma** — Standard deviation of the Gaussian distribution

0.5 (default) | positive number | 3-element vector of positive numbers

Standard deviation of the Gaussian distribution, specified as positive number or a 3-element vector of positive numbers. If **sigma** is a scalar, then `imgaussfilt3` uses a cubic Gaussian kernel.

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 quotes.

You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: `volSmooth = imgaussfilt3(vol,sigma,'padding','circular');`

### FilterSize — Size of the Gaussian filter

positive, odd, integer | 3-element vector of positive, odd, integers

Size of the Gaussian filter, specified as a scalar or 3-element vector of positive, odd, integers. If you specify a scalar, then `imgaussfilt3` uses a cubic filter. The default filter size is  $2*\text{ceil}(2*\text{sigma})+1$ .

Example: `volSmooth = imgaussfilt3(vol,sigma,'FilterSize',5);`

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

### Padding — Image padding

'replicate' (default) | 'circular' | 'symmetric' | numeric scalar

Image padding, specified as one of the following.

Value	Description
numeric scalar	Pad image with elements of constant value.
'circular'	Pad with circular repetition of elements within the dimension.
'replicate'	Pad by repeating border elements of array.
'symmetric'	Pad image 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` | `string`

### 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.

Filter Domain	Description
'frequency'	Perform convolution in the frequency domain.
'spatial'	Perform convolution in the spatial domain.

```
Example: volSmooth =  
imgaussfilt3(vol,sigma,'FilterDomain','frequency');
```

Data Types: char | string

## Output Arguments

### **B** — Filtered image

numeric array | gpuArray

Filtered image, returned as a numeric array of the same class and size as input image.

If the image is filtered using a GPU, then **B** is returned as a `gpuArray` containing a numeric array.

## Tips

- If image **A** contains `Inf`s or `NaN`s, then 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 `Inf`s and `NaN`s in the output in a manner similar to `imfilter`, consider setting the `'FilterDomain'` parameter to `'spatial'`.
- If you set the `'FilterDomain'` parameter to `'auto'`, then `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, then `imgaussfilt3` uses `'replicate'` padding by default, which is different from the default used by `imfilter`.

## Extended Capabilities

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

### 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`.

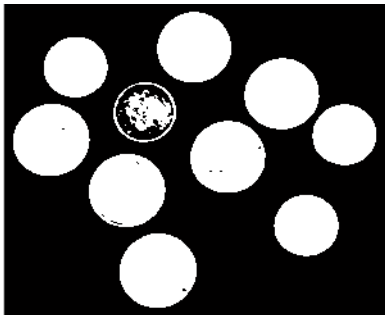
```
I = imread('coins.png');  
figure, imshow(I)
```





### Original Image

```
bw = im2bw(I, graythresh(getimage));  
figure, imshow(bw)
```

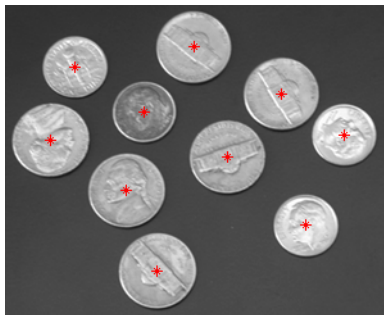


### Binary Image

```
bw2 = imfill(bw, 'holes');  
s = regionprops(bw2, 'centroid');  
centroids = cat(1, s.Centroid);
```

Display original image I and superimpose centroids:

```
imtool(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  
imtool rice.png  
sz = get(groot,'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.

---

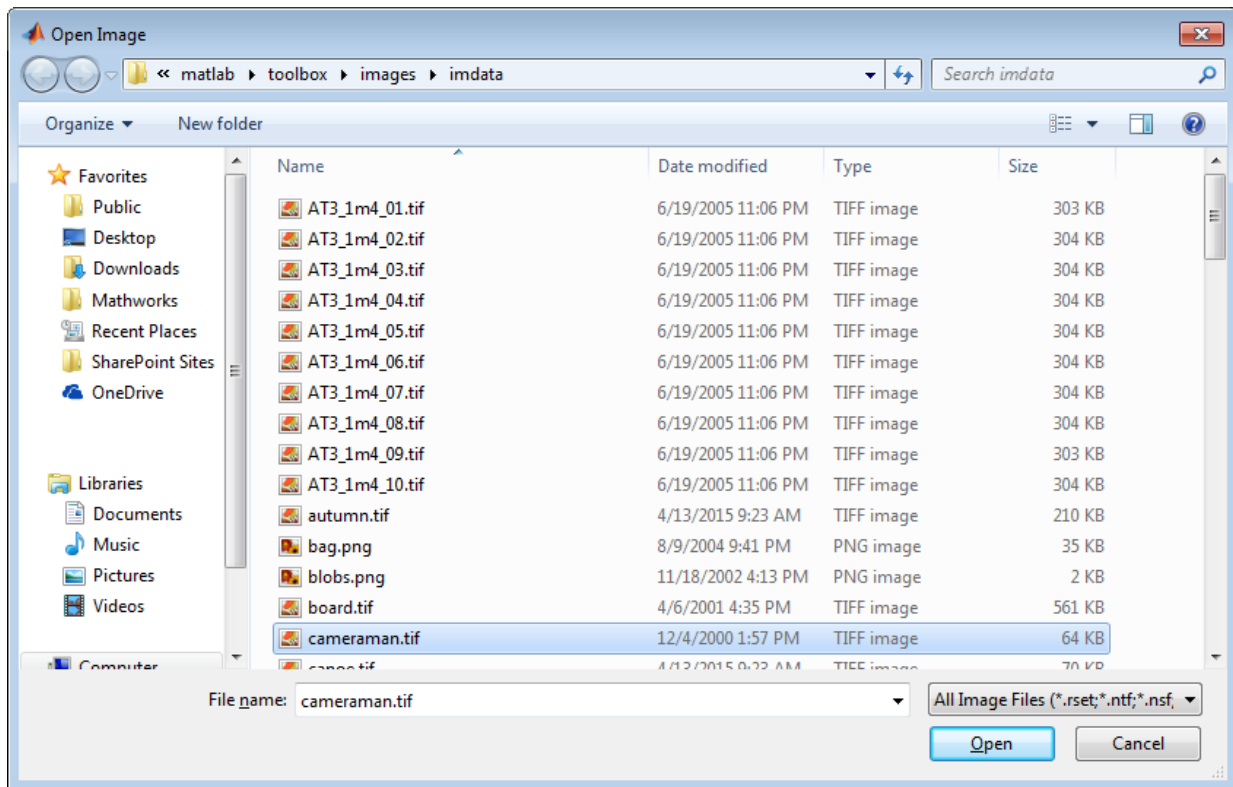
`[filename,user_canceled] = imgetfile(Name,Value)` supports name-value parameter arguments that you can use to control aspects of its behavior.

## 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`

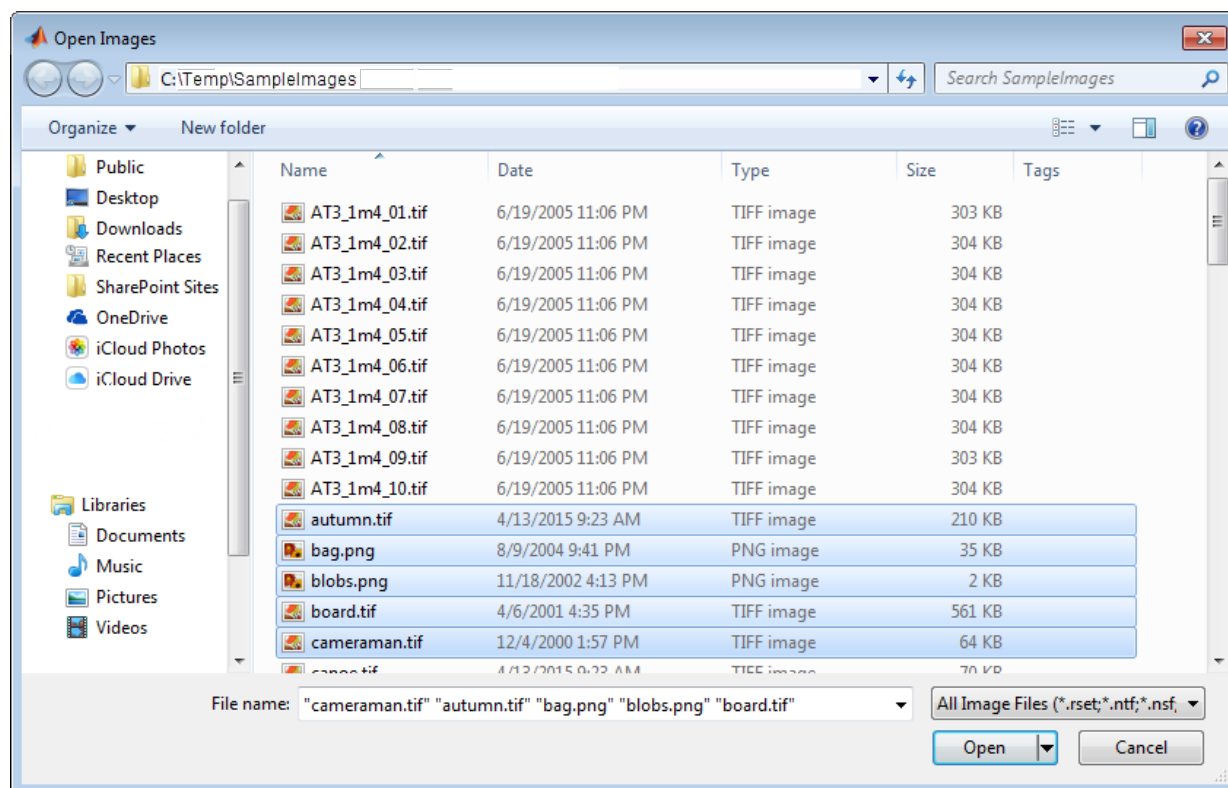
0

## 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 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 | string scalar

Folder displayed when the Open Image dialog box opens, specified as a string scalar or 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 'on' or 'off', or a Boolean value `true` or `false`. 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

`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)
[Gmag,Gdir] = imgradient(Gx,Gy)
```

## Description

`[Gmag,Gdir] = imgradient(I)` returns the gradient magnitude, `Gmag`, and the gradient direction, `Gdir`, for the grayscale or binary image `I`.

You optionally can compute the gradient magnitude and direction using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`[Gmag,Gdir] = imgradient(I,method)` returns the gradient magnitude and direction using specified method.

`[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.

## Examples

### Calculate Gradient Magnitude and Direction Using Prewitt Method

Read an image into workspace.

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

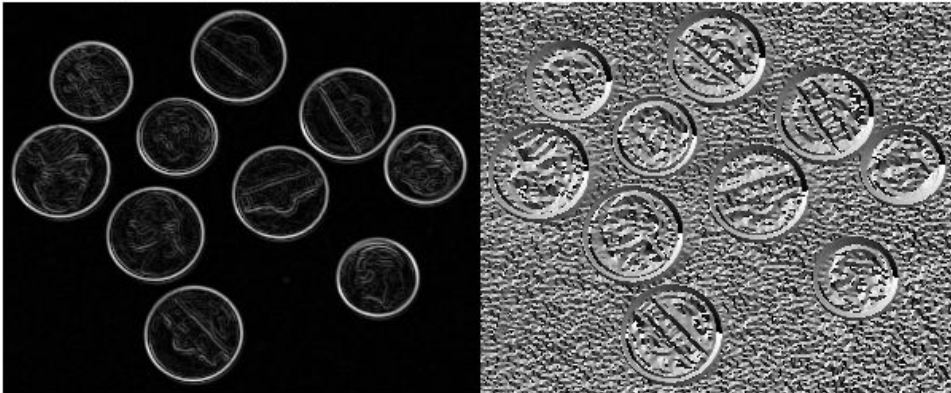
Calculate the gradient magnitude and direction, specifying the Prewitt gradient operator.

```
[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 the x- and y-directional gradients. By default, `imgradientxy` uses the Sobel gradient operator.

```
[Gx,Gy] = imgradientxy(I);
```

Display the directional gradients.

```
imshowpair(Gx,Gy,'montage')  
title('Directional Gradients Gx and Gy, Using Sobel Method')
```

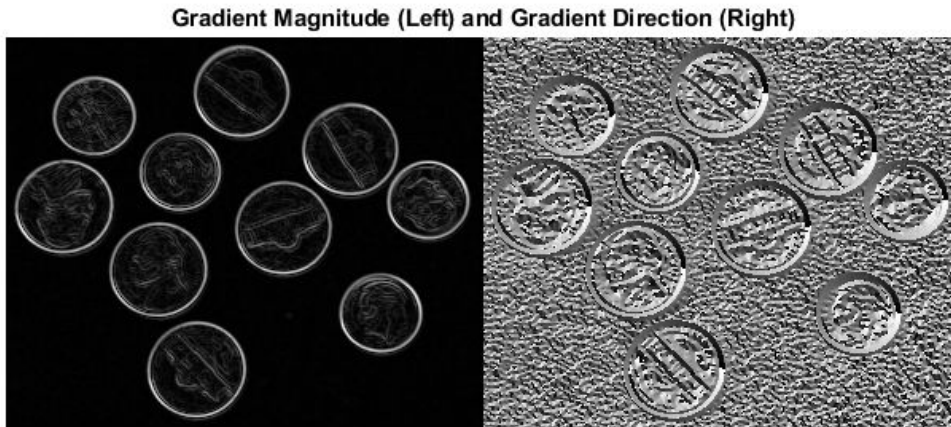


Calculate the gradient magnitude and direction using the directional gradients.

```
[Gmag,Gdir] = imgradient(Gx,Gy);
```

Display the gradient magnitude and direction.

```
imshowpair(Gmag,Gdir,'montage')  
title('Gradient Magnitude (Left) and Gradient Direction (Right)')
```



## Input Arguments

### **I** — Input image

grayscale image | binary image | gpuArray

Input image, specified as a grayscale or binary image, that is, a nonsparse numeric or logical 2-D matrix.

To calculate the gradient magnitude and direction using a GPU, specify I as a gpuArray that contains a grayscale or binary 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` | `string`

### **Gx — Directional gradients along x-axis (horizontal)**

`numeric matrix` | `gpuArray`

Directional gradient along x-axis (horizontal), specified as a numeric matrix equal in size to image I, typically returned by `imgradientxy`.

To calculate the gradient magnitude and direction using a GPU, specify Gx as a `gpuArray` that contains a numeric matrix.

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

### **Gy — Directional gradients along y-axis (vertical)**

`numeric matrix` | `gpuArray`

Directional gradient along y-axis (vertical), specified as a numeric matrix equal in size to image I, typically returned by `imgradientxy`.

To calculate the gradient magnitude and direction using a GPU, specify Gy as a `gpuArray` that contains a numeric matrix.

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

## **Output Arguments**

### **Gmag — Gradient magnitude**

`numeric matrix` | `gpuArray`

Gradient magnitude, returned as a numeric 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`.

If you use a GPU to compute the gradient magnitude and direction, then `Gmag` is returned as a `gpuArray` that contains a numeric matrix.

Data Types: `double` | `single`

### **Gdir — Gradient direction**

numeric matrix | `gpuArray`

Gradient direction, returned as a numeric 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`.

If you use a GPU to compute the gradient magnitude and direction, then `Gdir` is returned as a `gpuArray` that contains a numeric matrix.

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 `Gx` and `Gy`.

`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.

## Extended Capabilities

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

### See Also

`edge` | `fspecial` | `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.

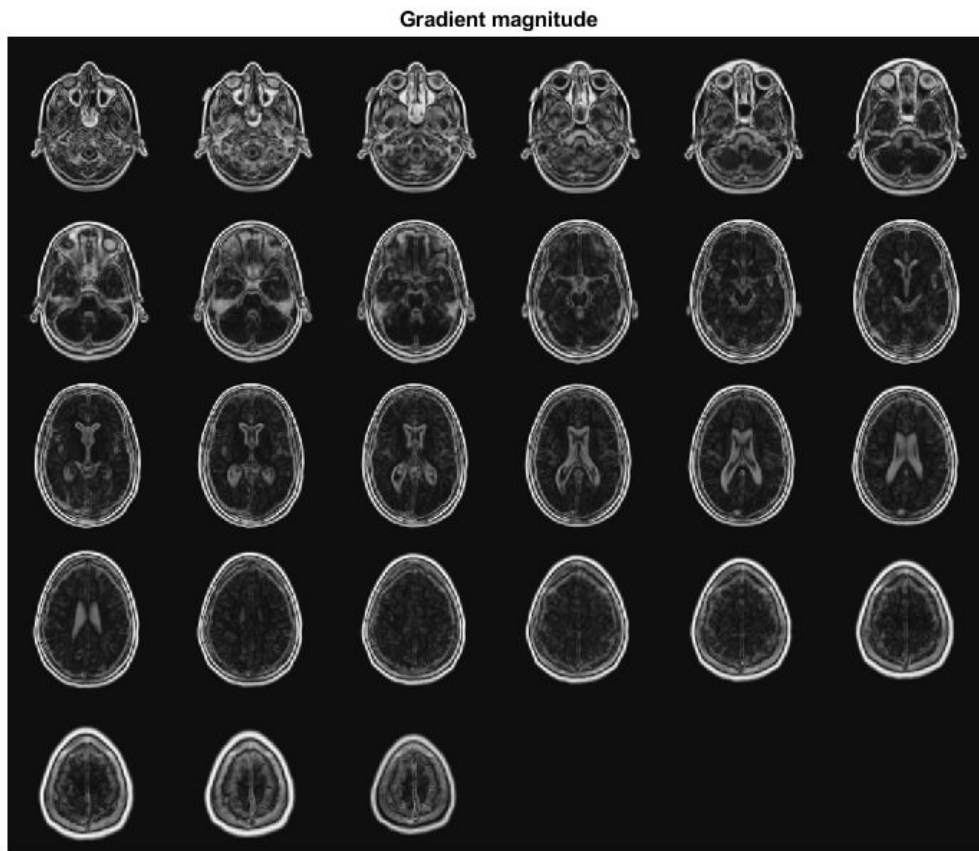
```
volData = load('mri');
sz = volData.size;
vol = squeeze(volData.D);
```

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',[1])  
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. $dI/dx = (I(x+1) - I(x-1))/2$
'intermediate'	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` | `string`

### 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 Gx and Gy. `imgradientxyz` returns Gz.

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.

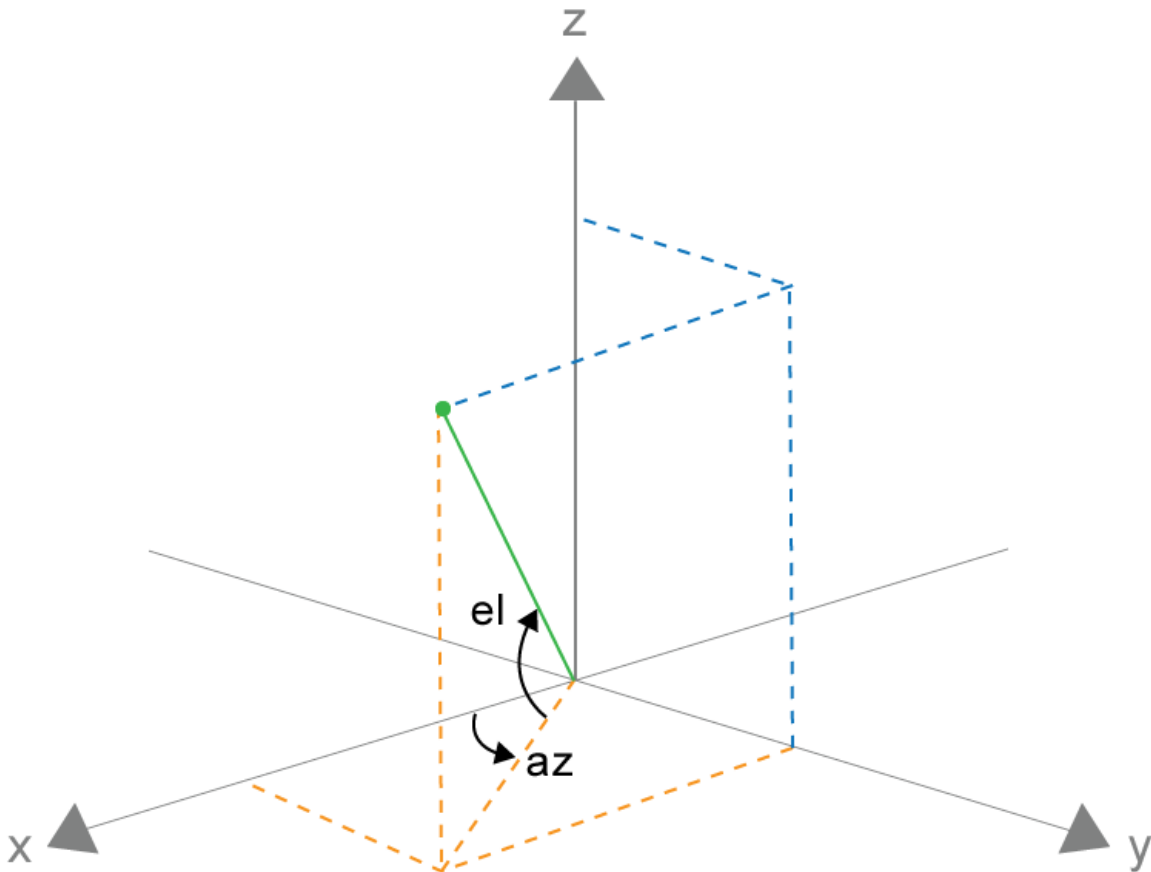
Gmag is of class `double`, unless the input image I or any of the directional gradients Gx, Gy, or Gz are of class `single`. In this case, Gmag is of class `single`.

### **Gazimuth — Azimuthal angle**

nonsparse matrix

Azimuthal angle, returned as a nonsparse matrix the same size as image I. `Gazimuth` 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.

Gazimuth 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, Gazimuth 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

`imgradient` | `imgradientxy` | `imgradientxyz`

**Introduced in R2016a**

# imgradientxy

Directional gradients of an image

## Syntax

```
[Gx,Gy] = imgradientxy(I)  
[Gx,Gy] = imgradientxy(I,method)
```

## 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.

You optionally can compute the directional gradients using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`[Gx,Gy] = imgradientxy(I,method)` returns the directional gradients using the specified `method`.

## Examples

### Calculate Directional Gradients Using Prewitt Method

Read an image into workspace.

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

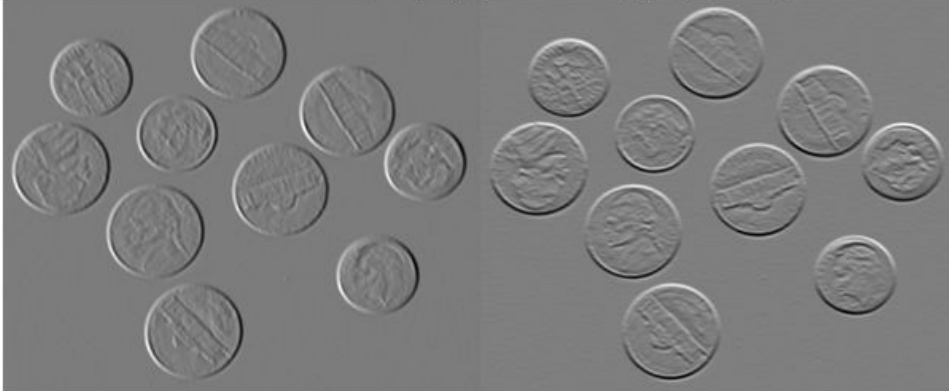
Calculate the *x*- and *y*-directional gradients using the Prewitt gradient operator.

```
[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 Using Prewitt Method on a GPU

Read an image into a gpuArray.

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

Calculate gradient magnitude and gradient direction using Prewitt's gradient operator, and display the 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 the x- and y-directional gradients. By default, `imgradientxy` uses the Sobel gradient operator.

```
[Gx,Gy] = imgradientxy(I);
```

Display the directional gradients.

```
imshowpair(Gx,Gy,'montage')  
title('Directional Gradients Gx and Gy, Using Sobel Method')
```

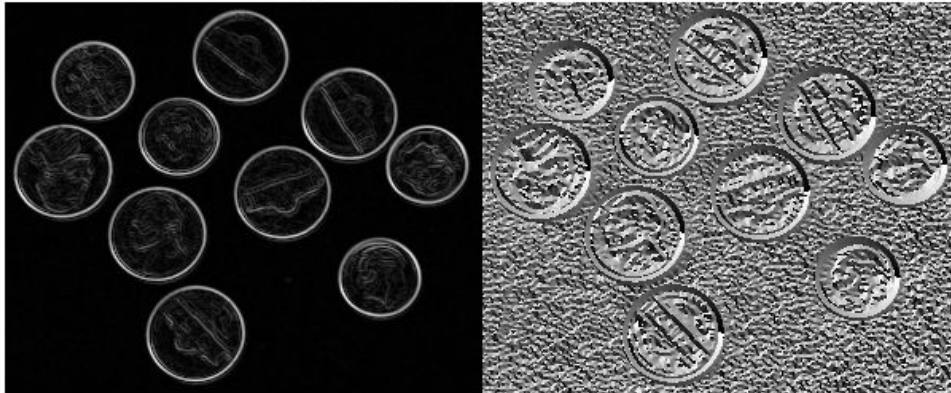


Calculate the gradient magnitude and direction using the directional gradients.

```
[Gmag,Gdir] = imgradient(Gx,Gy);
```

Display the gradient magnitude and direction.

```
imshowpair(Gmag,Gdir,'montage')  
title('Gradient Magnitude (Left) and Gradient Direction (Right)')
```

**Gradient Magnitude (Left) and Gradient Direction (Right)**

## Input Arguments

### **I** — Input image

grayscale image | binary image | gpuArray

Input image, specified as a grayscale or binary image, that is, a non-sparse numeric or logical 2-D matrix.

To calculate the directional gradients using a GPU, specify **I** as a `gpuArray` that contains a grayscale or binary 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)

Method	Description
'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 | string

## Output Arguments

### Gx — Directional gradients along x-axis

numeric matrix | gpuArray

Directional gradient along the x-axis, returned as a numeric 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`.

If you use a GPU to compute the directional gradients, then Gx is returned as a `gpuArray` that contains a numeric matrix.

Data Types: single | double

### Gy — Directional gradient along y-axis

numeric matrix | gpuArray

Directional gradients along the y-axis, returned as a numeric 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`.

If you use a GPU to compute the directional gradients, then Gy is returned as a `gpuArray` that contains a numeric matrix.

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  $1/6$ .

## Extended Capabilities

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

### See Also

`edge` | `fspecial` | `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.

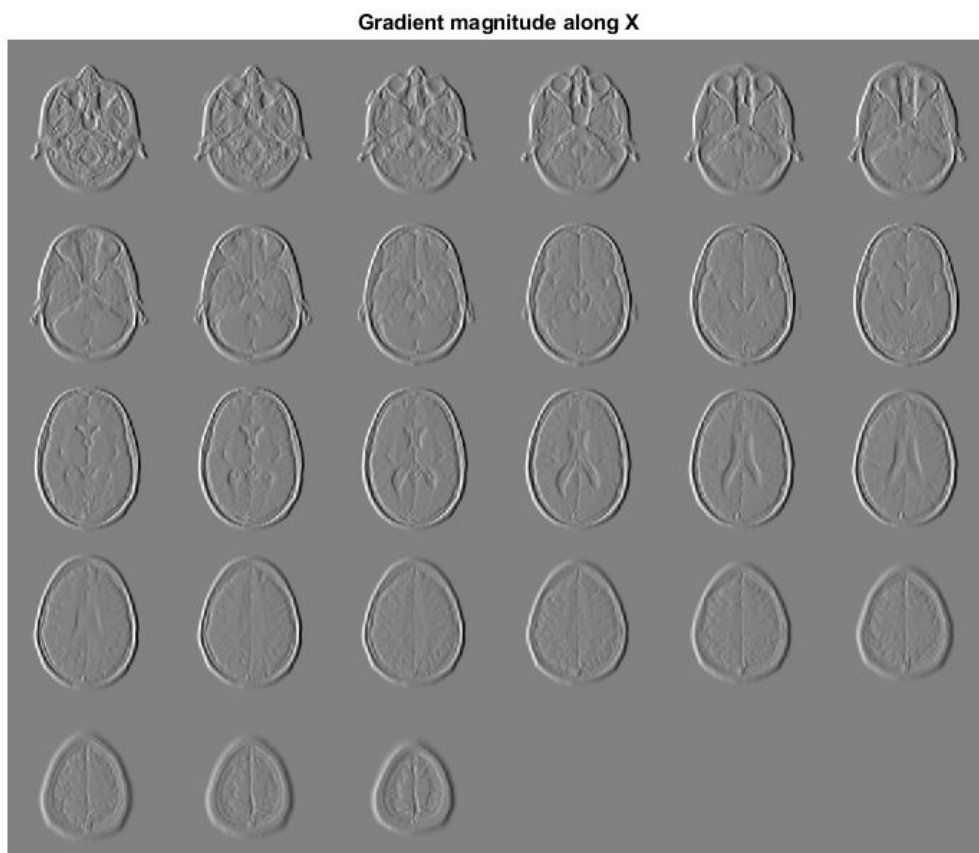
```
volData = load('mri');  
sz = volData.siz;  
vol = squeeze(volData.D);
```

Calculate the directional gradients.

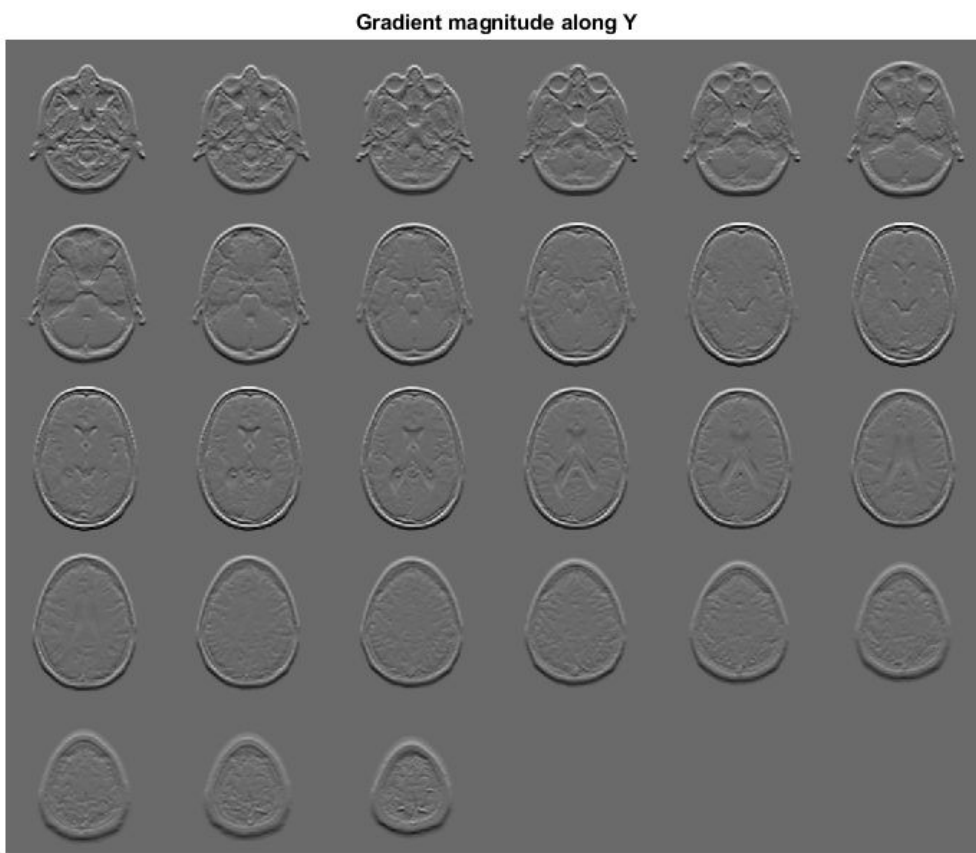
```
[Gx, Gy, Gz] = imgradientxyz(vol);
```

Visualize the directional gradients as a montage.

```
figure, montage(reshape(Gx,sz(1),sz(2),1,sz(3)), 'DisplayRange', [])  
title('Gradient magnitude along X')
```

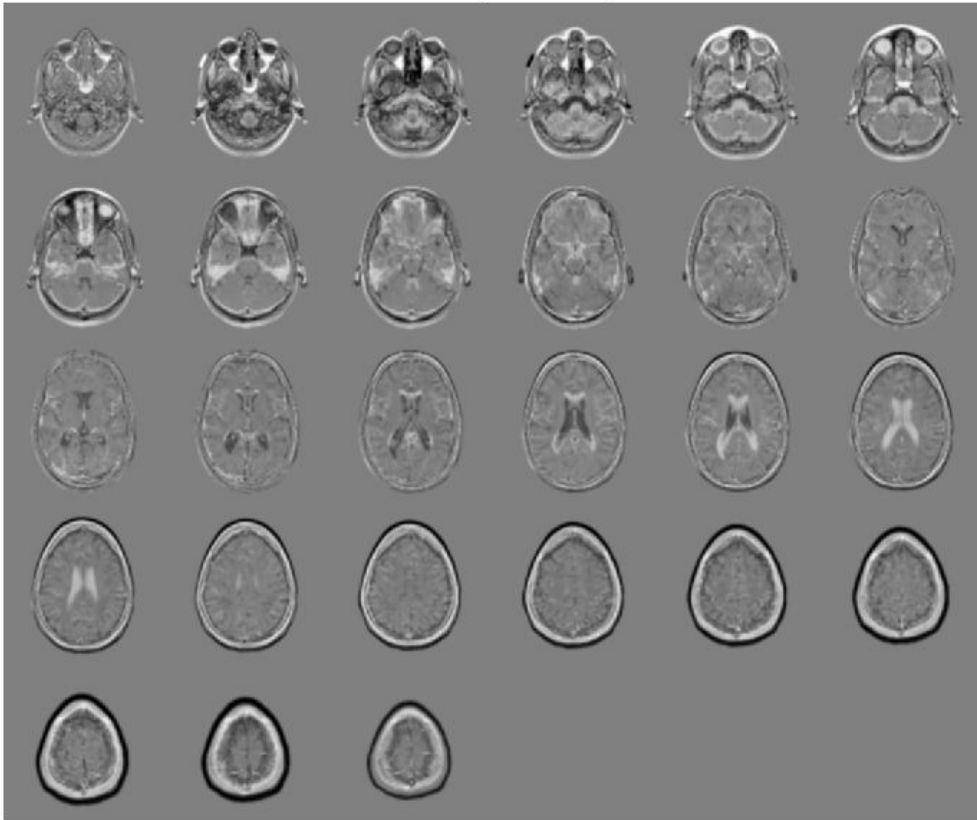


```
figure, montage(reshape(Gy,sz(1),sz(2),1,sz(3)), 'DisplayRange', [])  
title('Gradient magnitude along Y')
```



```
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. $dI/dx = (I(x+1) - I(x-1))/2$
'intermediate'	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` | `string`

## 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**

`imgradient` | `imgradient3` | `imgradientxy`

**Introduced in R2016a**

# imguiddfilter

Guided filtering of images

## Syntax

```
B = imguiddfilter(A,G)
B = imguiddfilter(A)
B = imguiddfilter(__,Name,Value,...)
```

## Description

`B = imguiddfilter(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 = imguiddfilter(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 = imguiddfilter(__,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 guided filter.

Read an image into the workspace. Display the image.

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



Smooth the image using `imguidedfilter`. In this syntax, `imguidedfilter` uses the image itself as the guidance image.

```
Iguided = imguidedfilter(A);
```

For comparison, smooth the original image using a gaussian filter defined by `imgaussfilt`. Set the standard deviation of the filter to 2.5 so that the degree of smoothing approximately matches that of the guided filter.

```
Igaussian = imgaussfilt(A,2);
```

Display the result of guided filtering and the result of gaussian filtering.

```
imshowpair(Iguided,Igaussian,'montage');
```



Observe that the flat regions of the two filtered images, such as the jacket and the face, have similar amounts of smoothing. However, the guided filtered image better preserves the sharpness of edges, such as around the trellis and the collar of the white shirt.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `Ismooth = imguifilter(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]`. Do not specify a value greater than the size of the image.

Example: `Ismooth = imguifilter(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, `imguidedfilter` 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, `imguidedfilter` uses G for guidance for all the channels of A independently.
  - If both A and G are RGB images, `imguidedfilter` 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, `imguidedfilter` 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

`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

```
[counts,binLocations] = imhist(I)
[counts,binLocations] = imhist(I,n)
[counts,binLocations] = imhist(X,map)
```

```
imhist( ___ )
```

## Description

`[counts,binLocations] = imhist(I)` calculates the histogram for the grayscale image `I`. The `imhist` function returns the histogram counts in `counts` and the bin locations in `binLocations`. The number of bins in the histogram is determined by the image type.

You optionally can compute the histogram counts and bin locations using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`[counts,binLocations] = imhist(I,n)` specifies the number of bins, `n`, used to calculate the histogram.

`[counts,binLocations] = imhist(X,map)` calculates the histogram for the indexed image `X` with colormap `map`. The histogram has one bin for each entry in the colormap.

This syntax is not supported on a GPU.

`imhist( ___ )` displays a plot of the histogram. If the input image is an indexed image, then the histogram shows the distribution of pixel values above a colorbar of the colormap `map`.

If you use this syntax when `I` is a `gpuArray`, then no plot is displayed. `imhist` returns the histogram counts in `ans` and does not return the histogram bin locations.

## Examples

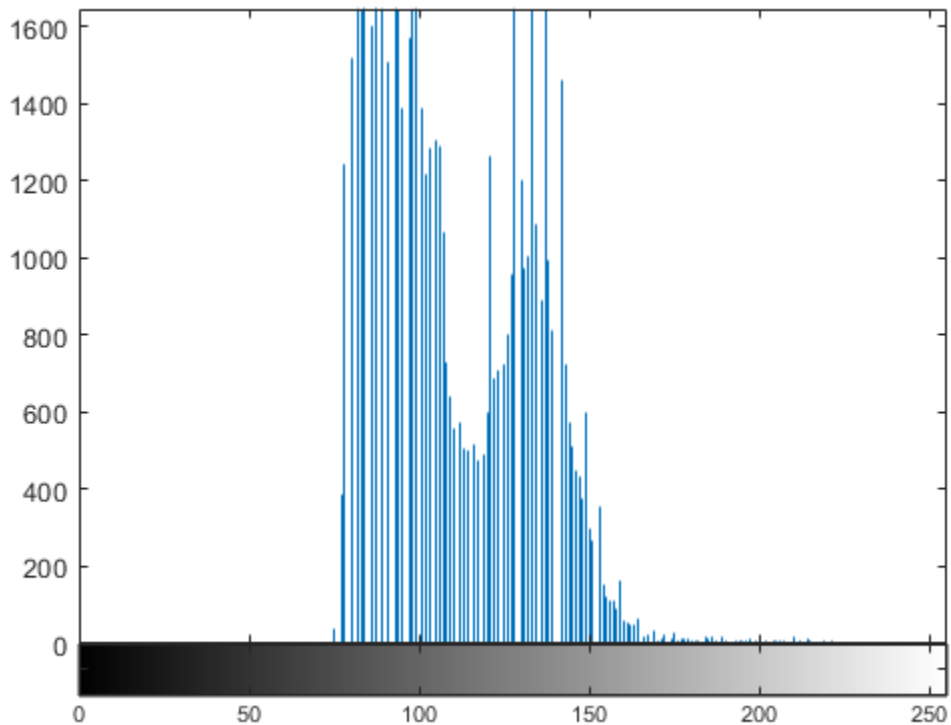
### Calculate Histogram

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)
```



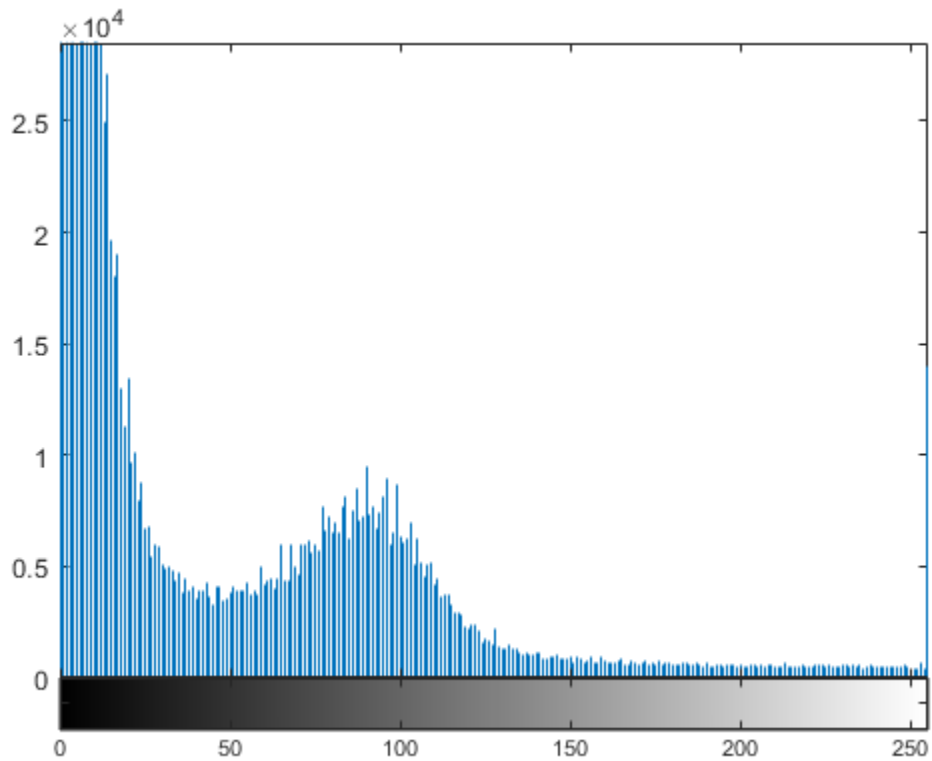
## Display the Histogram of a 3-D Intensity Image

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** — Grayscale image

numeric array | `gpuArray`

Grayscale image, specified as a numeric array of any dimension.

To compute the histogram counts and bin locations using a GPU, specify **I** as a `gpuArray` that contains a numeric array.

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

Example: `I = gpuArray(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, then `imhist` uses a default value of 256 bins. If **I** is a binary image, then `imhist` uses two bins.

Example: `[counts,x] = imhist(I,50);`

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

### **X** — Indexed image

numeric array

Indexed image, specified as a numeric array of any dimension.

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

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

### **map — Colormap associated with indexed image**

*c*-by-3 matrix

Colormap associated with indexed image *X*, specified as a *c*-by-3 matrix. The colormap must be at least as long as the largest index in *X*.

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

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

## Output Arguments

### **counts — Histogram counts**

numeric array | `gpuArray`

Histogram counts, returned as a numeric array. If the histogram is computed for an indexed image, *X*, then the length of *counts* is the same as the length of the colormap, *map*.

If the histogram is computed using a GPU, then *counts* is returned as a `gpuArray` containing a numeric array.

### **binLocations — Bin locations**

numeric array | `gpuArray`

Bin locations, returned as a numeric array.

If the histogram is computed using a GPU, then *binLocations* is returned as a `gpuArray` containing a numeric array.

## Tips

- For grayscale 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:

	<b>doubl e</b>	<b>single</b>	<b>int8</b>	<b>int16</b>	<b>int32</b>	<b>uint8</b>	<b>uint16</b>	<b>uint32</b>	<b>logical</b>
$A$	1	1	255	65535	42949 67295	255	65535	42949 67295	1
$B$	0	0	128	32768	21474 83648	0	0	0	0

- To display the histogram from counts and binLocations, use the command `stem(binLocations,counts)`.

## 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 “Understand 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.



## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When running on a GPU, `imhist` does not display the histogram. To display the histogram, use `stem(binLocations,counts)`.

For more information, see “Image Processing on a GPU”.

## See Also

`histeq` | `histogram` | `stem`

**Introduced before R2006a**

## imhistmatch

Adjust histogram of 2-D image to match histogram of reference image

### Syntax

```
J = imhistmatch(I,ref)
J = imhistmatch(I,ref,nbins)
J = imhistmatch( ____,Name,Value)
[J,hgram] = imhistmatch( ____ )
```

### Description

`J = imhistmatch(I,ref)` transforms the 2-D grayscale or truecolor image `I` returning output image `J` whose histogram approximately matches the histogram of the reference image `ref`.

- If both `I` and `ref` are truecolor RGB images, then `imhistmatch` matches each color channel of `I` independently to the corresponding color channel of `ref`.
- If `I` is a truecolor RGB image and `ref` is a grayscale image, then `imhistmatch` matches each channel of `I` against the single histogram derived from `ref`.
- If `I` is a grayscale image, then `ref` must also be a grayscale image.

Images `I` and `ref` can be any of the permissible data types and need not be equal in size.

`J = imhistmatch(I,ref,nbins)` uses `nbins` equally spaced bins within the appropriate range for the given image data type. The returned image `J` has no more than `nbins` discrete levels.

- If the data type of the image is either `single` or `double`, then the histogram range is `[0, 1]`.
- If the data type of the image is `uint8`, then the histogram range is `[0, 255]`.
- If the data type of the image is `uint16`, then the histogram range is `[0, 65535]`.
- If the data type of the image is `int16`, then the histogram range is `[-32768, 32767]`.

`J = imhistmatch( ____, Name, Value)` uses name-value pairs to change the behavior of the histogram matching algorithm.

`[J,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 = 1×3  
  
2036      3060      3
```

Get the size of `Ref`.

```
size(Ref)  
  
ans = 1×2  
  
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.

```
imshow(A)  
title('RGB Image with Color Cast')
```



```
imshow(Ref)  
title('Reference Grayscale Image')
```

Reference Grayscale Image



```
imshow(B)  
title('Histogram Matched RGB Image')
```

Histogram Matched RGB Image

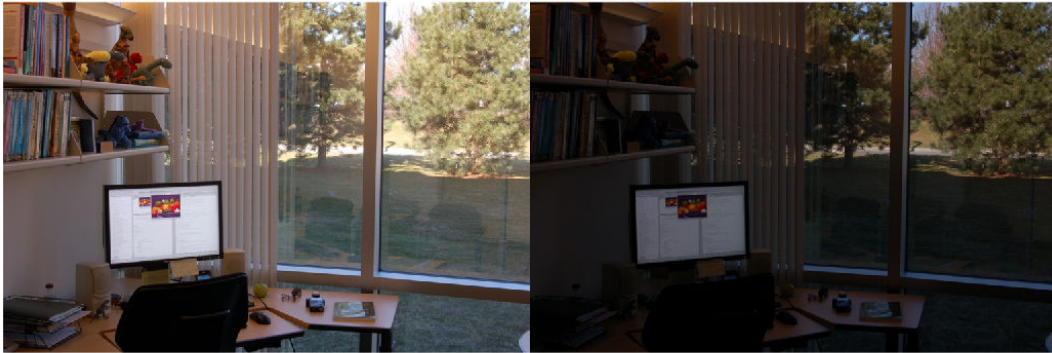


## Histogram-Match Image Using Polynomial Method

Read a color image and a reference image. To demonstrate the polynomial method, assign the reference image to be the darker of the two images.

```
I = imread('office_4.jpg');  
ref = imread('office_2.jpg');  
montage({I,ref})  
title('Input Image (Left) vs Reference Image (Right)');
```

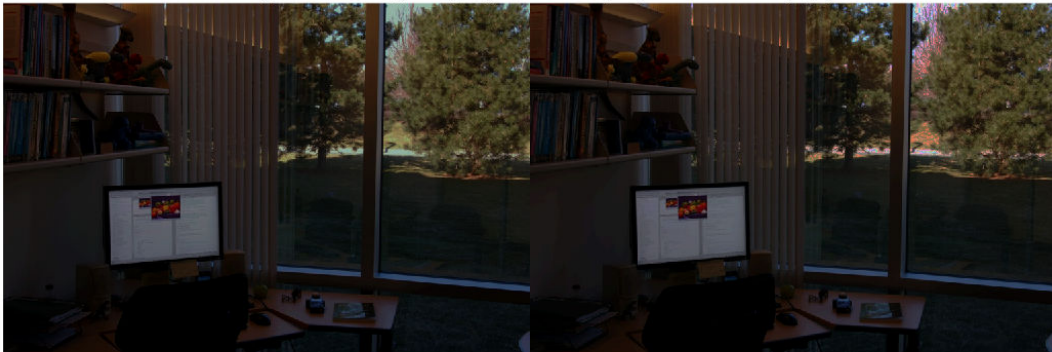
Input Image (Left) vs Reference Image (Right)



Use the polynomial method to adjust the intensity of image I so that it matches the histogram of reference image ref. For comparison, also adjust the intensity of image I using the uniform method.

```
J = imhistmatch(I,ref,'method','polynomial');  
K = imhistmatch(I,ref,'method','uniform');  
montage({J,K})  
title('Histogram-Matched Image Using Polynomial Method (Left) vs Uniform Method (Right)')
```

Histogram-Matched Image Using Polynomial Method (Left) vs Uniform Method (Right)



The histogram-matched image using the uniform method introduces false colors in the sky and road. The histogram-matched image using the polynomial method does not exhibit this artifact.

## Multiple N Values Applied to RGB Images

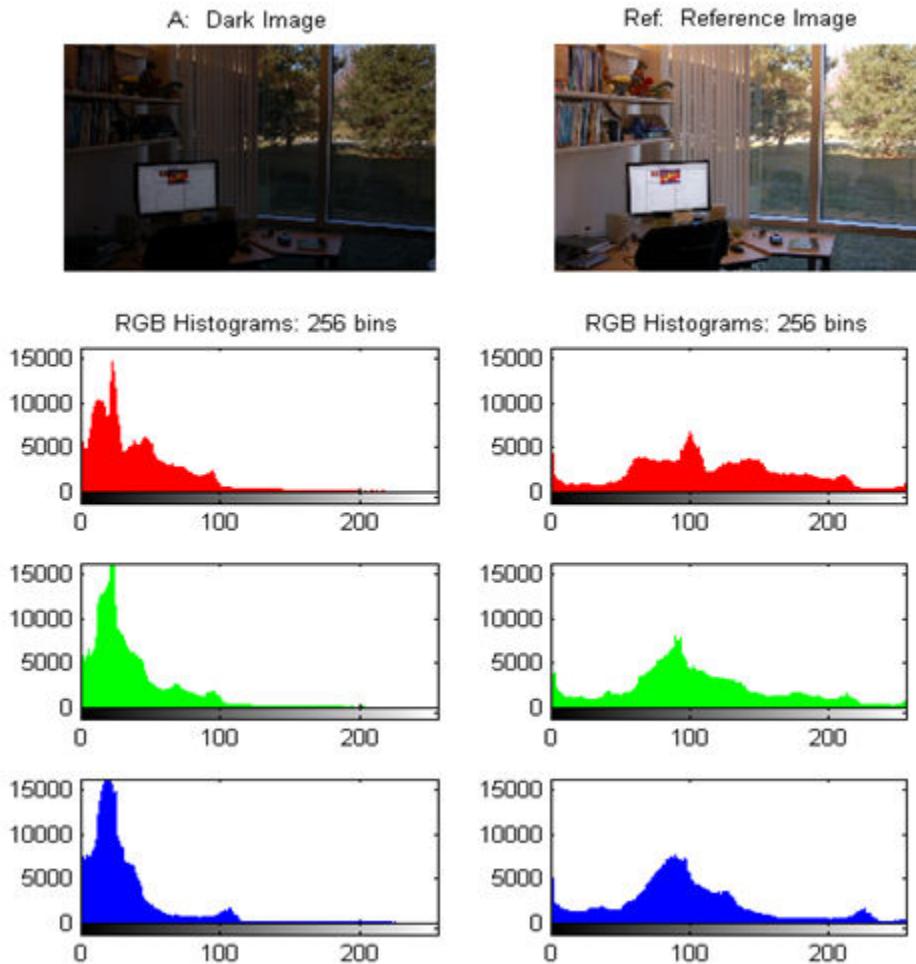
In this example, you will see the effect on output image `J` of varying the number of equally spaced bins in the target histogram of image `Ref`, from its default value 64 to the maximum value of 256 for `uint8` pixel data.

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);
```

Output Image B64



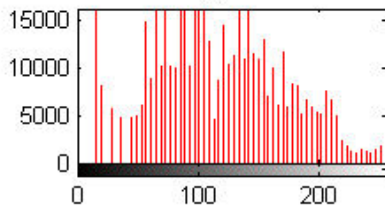
Output Image B128



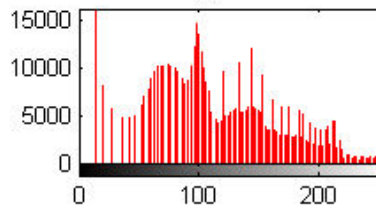
Output Image B256



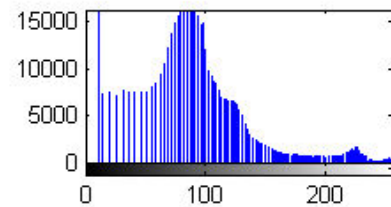
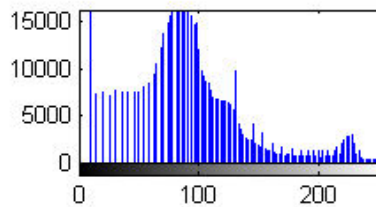
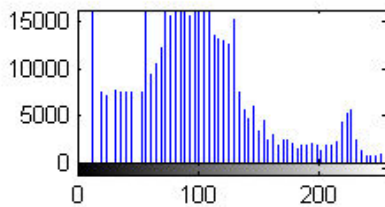
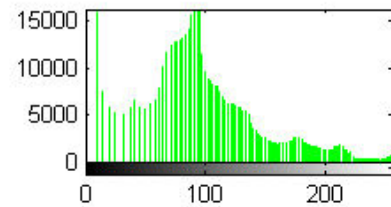
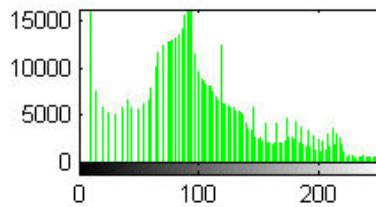
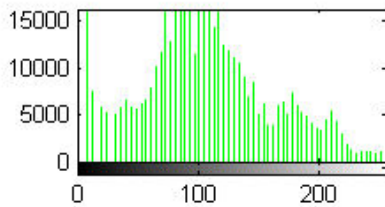
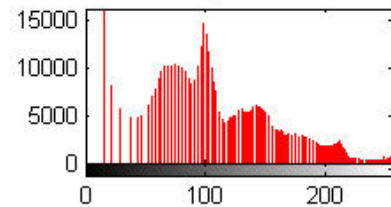
RGB Histograms: 64 bins



RGB Histograms: 128 bins



RGB Histograms: 256 bins



The number of unique 8-bit level values for each output image is described in the table. Note that as `nbins` increases, the number of levels in each RGB channel of output image B also increases.

<b>nbins</b>	<b>Output Image</b>	<b>Number of unique histogram values</b>
64	B64	57
128	B128	101
256	B256	134

### 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
```

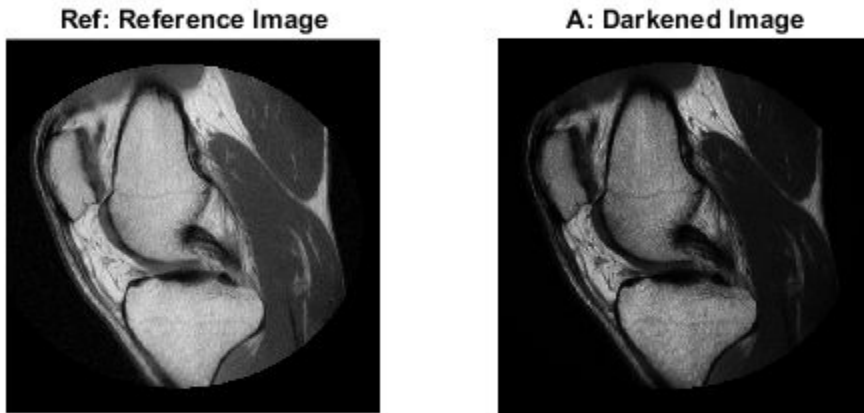
Darken the reference image `Ref` to create an image `A` that can be used in the histogram matching operation.

```
%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');
```



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
```

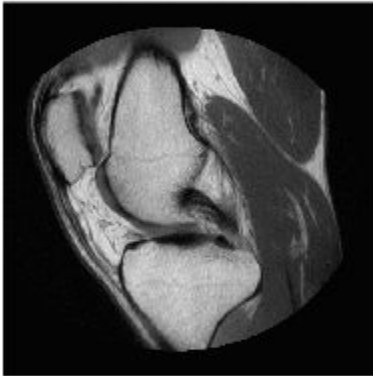
```
N = length(LevelsK); % number of unique 16-bit code values in image A.  
B16bitUniq = imhistmatch(A(:,:,1),Ref(:,:,1),N);
```

View the results of the two histogram matching operations.

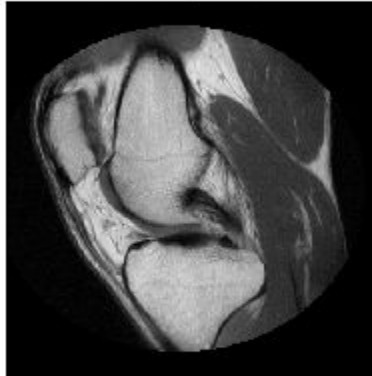
```
figure  
subplot(1,2,1)  
imshow(B16bit64)
```

```
title('B16bit64: 64 bins')
subplot(1,2,2)
imshow(Ref)
title(['B16bitUniq: ', num2str(N), ' bins'])
```

**B16bit64: 64 bins**



**B16bitUniq: 448 bins**



## Input Arguments

### **I** — 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 `J` 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 `J`.

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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `imhistmatch(I, ref, 'Method', 'polynomial')` matches the histogram of image `I` to that of reference image `ref` using the polynomial mapping technique.

### **Method** — Mapping technique

'uniform' (default) | 'polynomial'

Mapping technique used to map the histogram of `ref` to image `I`, specified as the comma-separated pair consisting of 'Method' and one of these values:

- 'uniform' — Use a histogram-based intensity function and histogram equalization.
- 'polynomial' — Calculate a cubic Hermite polynomial mapping function from the cumulative histograms of the source and reference images. The polynomial method is useful when the reference image is darker than the input image. In this situation, the polynomial method gives a smoother color transition than the uniform method.

## Output Arguments

### **J** — 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 **I** whose histogram is an approximate match to the histogram of input image **ref** built with **nbins** equally-spaced bins. Image **J** is of the same size and data type as input image **I**. Input argument **nbins** represents the upper limit of the number of discrete levels contained in image **J**.

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

### **hgram** — Histogram counts derived from reference image **ref**

1-by-**nbins** vector | 3-by-**nbins** 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.

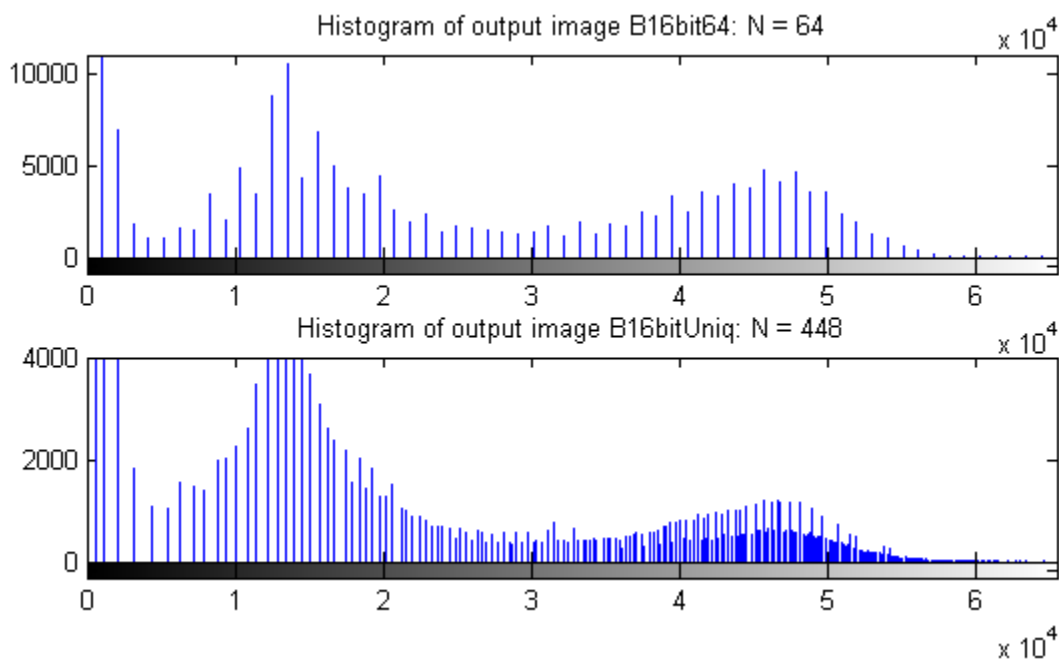
Data Types: `double`

## Algorithms

The objective of `imhistmatch` is to transform image **I** such that the histogram of image **J** 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 **J**.

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 **J** 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

`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

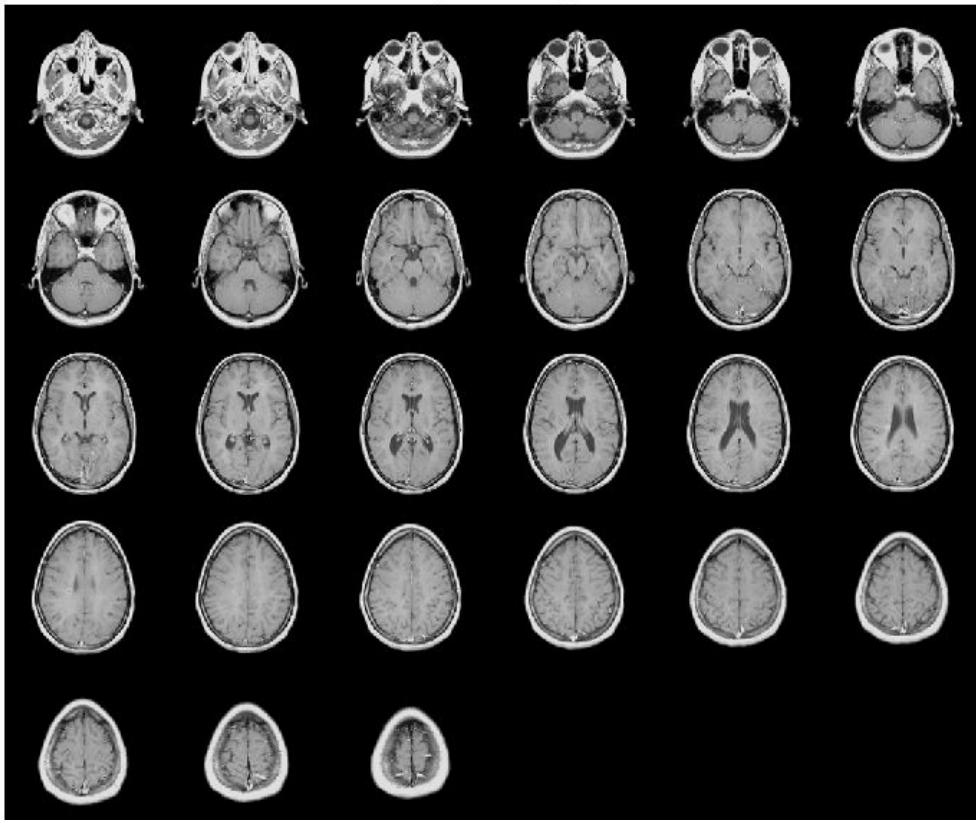
Load an N-D grayscale image into the workspace. Also load a grayscale image to provide a reference histogram.

```
load mri D  
load mrystack
```

Display the original volume as slices.

```
figure  
montage(D, 'DisplayRange', [])  
title('Original 3-D Image')
```

Original 3-D Image



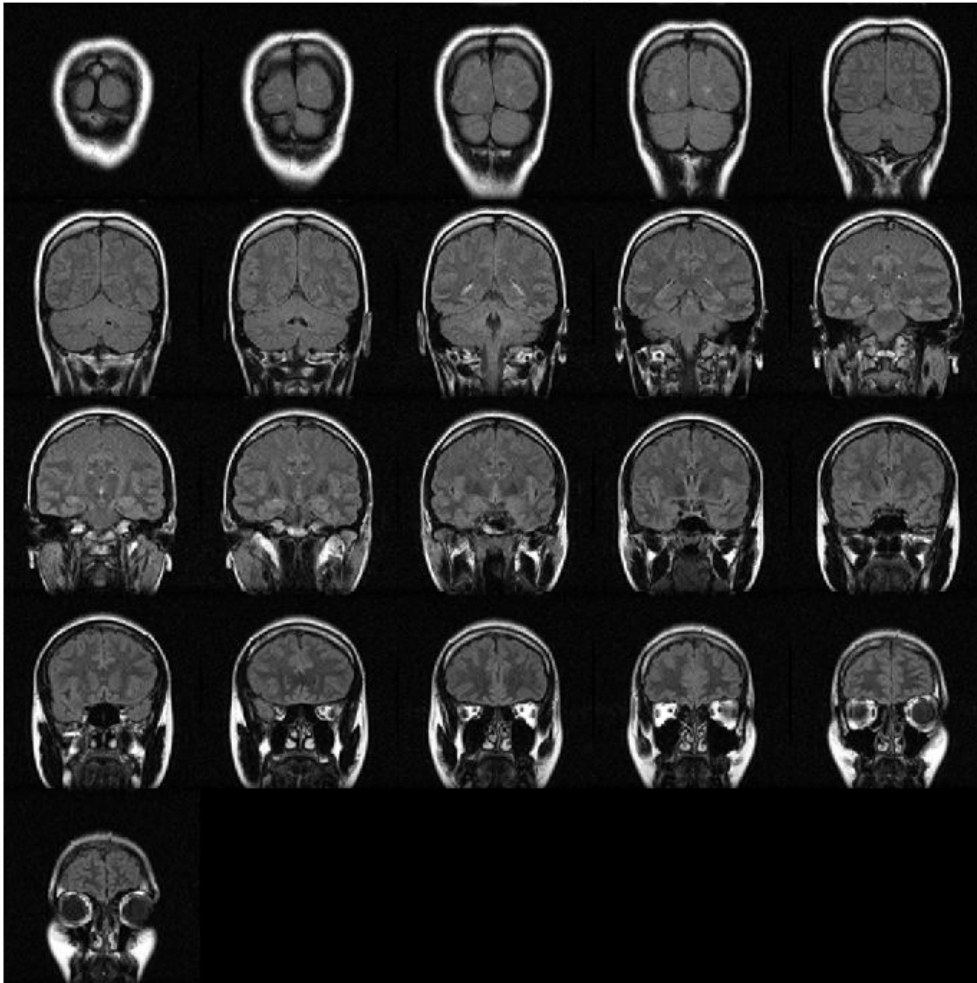
Reshape the reference as a stack of grayscale slices for display.

```
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')
```

Reference 3-D Image

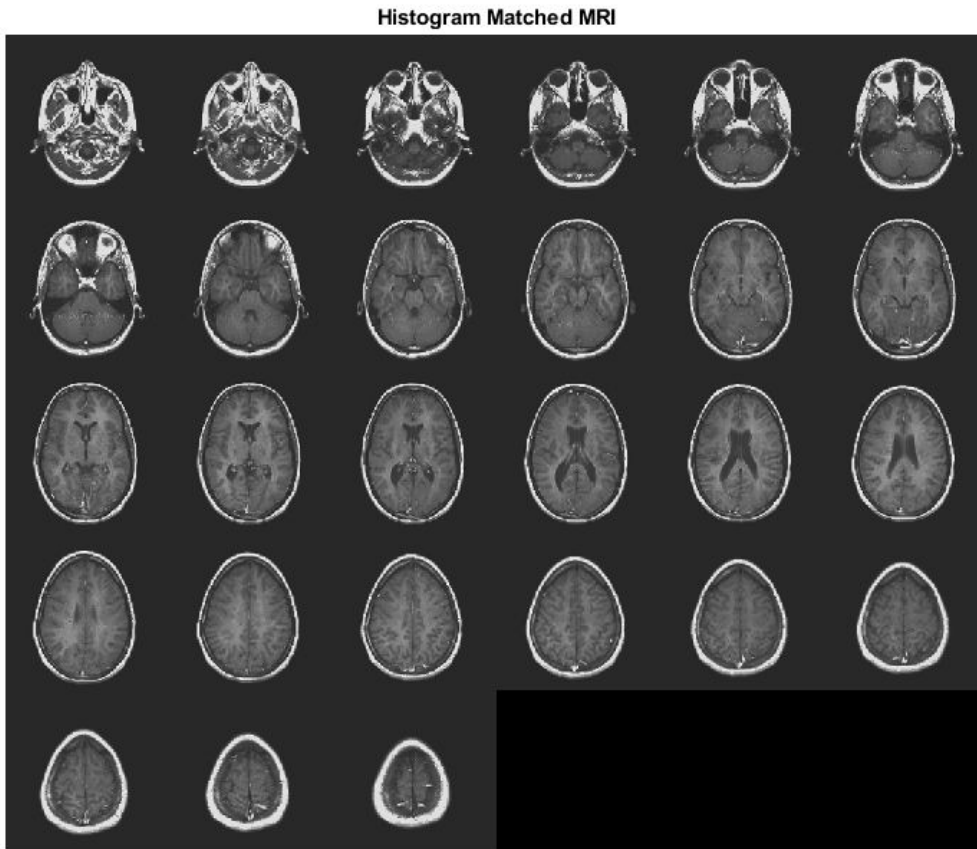


Match the histogram of D to the histogram of the fullsize ref.

```
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**

`histeq` | `imadjust` | `imhist` | `imhistmatch`

**Introduced in R2017a**



# imhmax

H-maxima transform

## Syntax

```
J = imhmax(I,H)
J = imhmax(I,H,conn)
```

## Description

`J = imhmax(I,H)` suppresses all maxima in the intensity image `I` whose height is less than `H`. Regional maxima are connected components of pixels with a constant intensity value, and whose external boundary pixels all have a lower value.

`J = 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 = 10×10
```

```

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 = 10x10
```

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 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

numeric array

Input image, specified as a numeric array of any dimension.

```
Example: I = imread('glass.png');
```

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

### **H** — H-maxima transform

nonnegative scalar

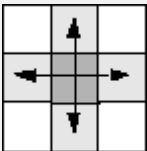
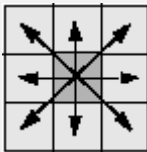
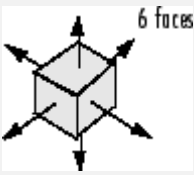
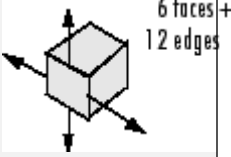
H-maxima transform, specified as a nonnegative scalar.

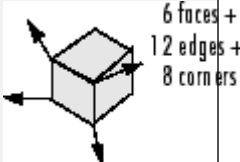
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### conn — Pixel connectivity

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	
18-connected	Pixels are connected if their faces or edges touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	

Value	Meaning	
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	 <p>6 faces + 12 edges + 8 corners</p>

For higher dimensions, `imhmax` uses the default value `conndef(ndims(I), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

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

## Output Arguments

### J — Transformed image

numeric array

Transformed image, returned as a numeric array of the same size and class 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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, the optional third input argument, `conn`, must be a compile-time constant.

### See Also

`conndef` | `imextendedmax` | `imhmin` | `imreconstruct` | `imregionalmax`

**Introduced before R2006a**

## imhmin

H-minima transform

### Syntax

```
J = imhmin(I,H)
J = imhmin(I,H,conn)
```

### Description

`J = imhmin(I,H)` suppresses all minima in the grayscale image `I` whose depth is less than `H`. Regional minima are connected components of pixels with a constant intensity value,  $t$ , whose external boundary pixels all have a value greater than  $t$ .

`J = 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  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  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

numeric array

Input image, specified as a numeric array of any dimension.

Example: `I = imread('glass.png');`

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

### H — H-minima transform

nonnegative scalar

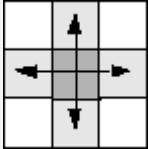
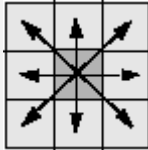
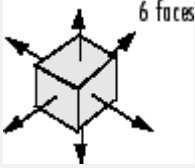
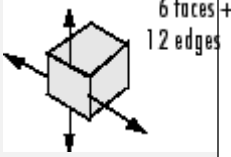
H-minima transform, specified as a nonnegative scalar.

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

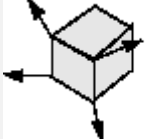
**conn — Pixel connectivity**

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	<p>Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction.</p>	
8-connected	<p>Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.</p>	
<b>Three-Dimensional Connectivities</b>		
6-connected	<p>Pixels are connected if their faces touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	
18-connected	<p>Pixels are connected if their faces or edges touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	



Value	Meaning	
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	 <p>6 faces + 12 edges + 8 corners</p>

For higher dimensions, `imhmin` uses the default value `conndef(ndims(I), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

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

## Output Arguments

### J — Transformed image

numeric array

Transformed image, returned as a nonsparse numeric array of the same size and class 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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, the optional third input argument, `conn`, must be a compile-time constant.

### See Also

`conndef` | `imextendedmin` | `imhmax` | `imreconstruct` | `imregionalmin`

**Introduced before R2006a**

# imimposemin

Impose minima

## Syntax

```
J = imimposemin(I,BW)
J = imimposemin(I,BW,conn)
```

## Description

`J = imimposemin(I,BW)` modifies the grayscale mask image `I` using morphological reconstruction so it only has regional minima wherever binary marker image `BW` is nonzero.

`J = imimposemin(I,BW,conn)` specifies the pixel connectivity for the morphological reconstruction.

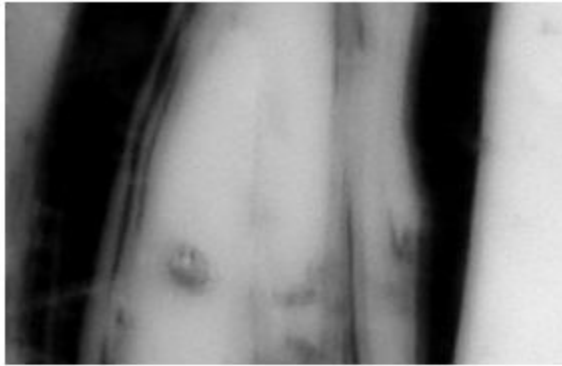
## Examples

### 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)
```

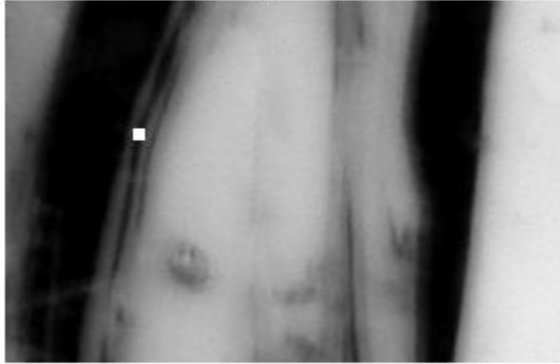


Create a binary image that is the same size as the mask image and sets a small area of the binary image to 1. These pixels define the location in the mask image where a regional minimum will be imposed. The resulting image is called the *marker* image.

```
marker = false(size(mask));  
marker(65:70,65:70) = true;
```

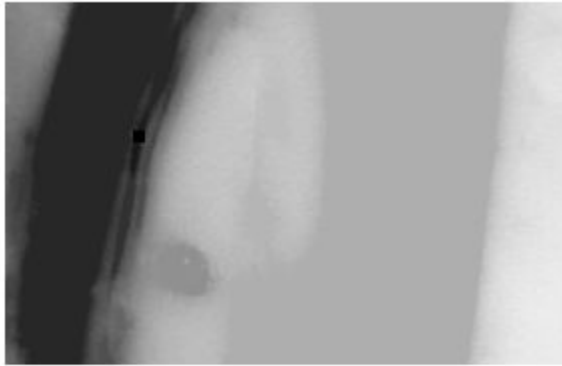
Superimpose the marker over the mask to show where these pixels of interest fall on the original image. The small white square marks the spot. This code is not essential to the impose minima operation.

```
J = mask;  
J(marker) = 255;  
figure  
imshow(J)  
title('Marker Image Superimposed on Mask')
```

**Marker Image Superimposed on Mask**

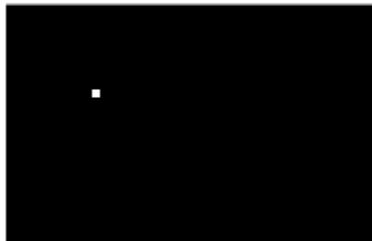
Impose the regional minimum on the input image using the `imimposemin` function. Note how all the dark areas of the original image, except the marked area, are lighter.

```
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**

## Input Arguments

### **I** — Grayscale mask image

numeric array | logical array

Grayscale mask image, specified as a numeric or logical array of any dimension.

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

### **BW** — Binary marker image

numeric array | logical array

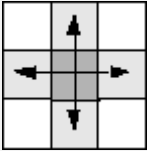
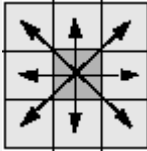
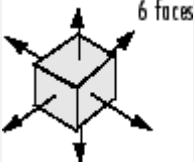
Binary marker image, specified as a numeric or logical array of the same size as the grayscale mask image I. For numeric input, any nonzero pixels are considered to be 1 (true).

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

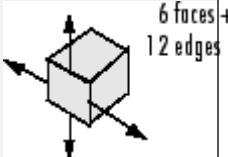
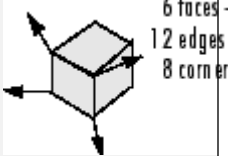
**conn — Pixel connectivity**

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	



Value	Meaning	
18-connected	<p>Pixels are connected if their faces or edges touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	

For higher dimensions, `imimposemin` uses the default value `conndef(ndims(I), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

Data Types: `double` | `logical`

## Output Arguments

### **J** — Reconstructed image

numeric array

Reconstructed image, returned as a numeric or logical array of the same size and data type as `I`.

## Algorithms

`imimposemin` uses a technique based on morphological reconstruction.

## See Also

`conndef` | `imreconstruct` | `imregionalmin`

## Topics

“Understanding Morphological Reconstruction”

**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( ___,outputClass)
```

## Description

`Z = imlincomb(K1,A1,K2,A2,...,Kn,An)` computes the linear combination of images, `A1`, `A2`, ... , `An`, with weights `K1`, `K2`, ... , `Kn` according to:

$$Z = K1*A1 + K2*A2 + \dots + Kn*An$$

You optionally can compute the linear combination using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`Z = imlincomb(K1,A1,K2,A2,...,Kn,An,K)` adds an offset, `K`, to the linear combination:

$$Z = K1*A1 + K2*A2 + \dots + Kn*An + K$$

`Z = imlincomb( ___,outputClass)` specifies the output class of `Z`.

## Examples

### Scale an Image Using Linear Combinations

Read an image into the workspace.

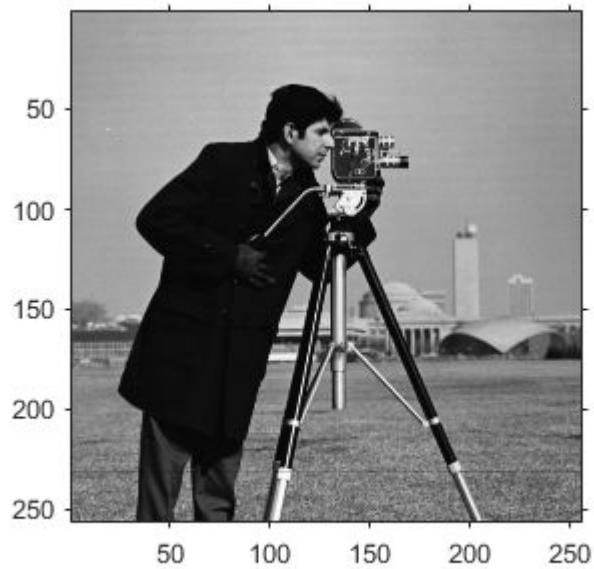
```
I = imread('cameraman.tif');
```

Scale the image using a coefficient of 1.5 in the linear combination.

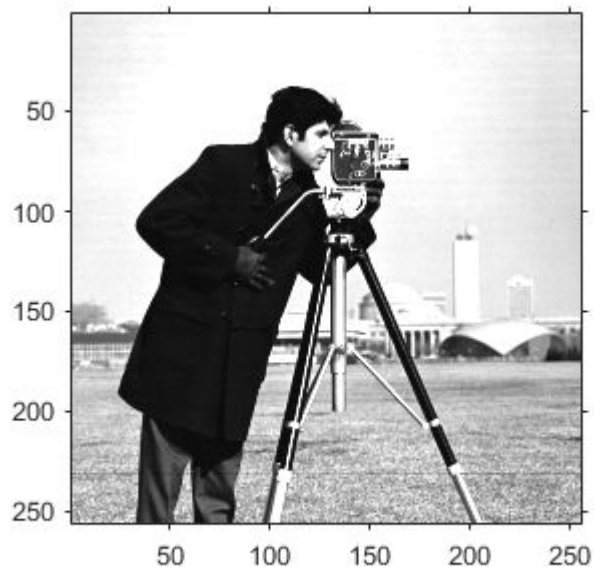
```
J = imlincomb(1.5,I);
```

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 = 2x3 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 = 2x3 uint8 matrix
```

```
153    25    63
 47   138    75
```



## Input Arguments

### **K1, K2, Kn — Image coefficients**

numeric scalar | gpuArray

Image coefficients, specified as real, numeric scalars.

To compute the linear combination using a GPU, specify the image coefficients as `gpuArrays` that each contain a numeric scalar.

Data Types: `double`

### **A1, A2, An — Input images**

numeric array | gpuArray

Input images, specified as numeric arrays of the same size and class.

To compute the linear combination using a GPU, specify the input images as `gpuArrays` that each contain a numeric array of the same size and class.

### **K — Offset**

numeric scalar

Offset, specified as a numeric scalar.

Data Types: `double`

### **outputClass — Output class**

string scalar | character vector

Output class of `Z`, specified as a string scalar or character vector containing the name of a numeric class.

Example: `'uint16'`

Example: `"double"`

## Output Arguments

### **Z — Linearly combined image**

numeric array | gpuArray

Linearly combined image, returned as a numeric array of the same size as `A1`. If `A1` is `logical`, then `Z` is `double`, otherwise `Z` has the same class as `A1`.

If the linear combination is computed using a GPU, then `Z` is returned as a `gpuArray` containing a numeric array.

### Tips

- 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` clips elements of `Z` that exceed the range of the integer type and rounds off fractional 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 “Understand 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.

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

## See Also

`imadd` | `imcomplement` | `imdivide` | `immultiply` | `imsubtract`

**Introduced before R2006a**

## **imline**

Create draggable, resizable line

---

**Note** `imline` is not recommended. Use `drawline` instead.

---

### **Description**

An `imline` object encapsulates an interactive line over an image.

You can adjust the size and position of the line by using the mouse. The line also has a context menu that controls aspects of its appearance and behavior. For more information, see “Usage” on page 1-1512.

### **Creation**

### **Syntax**

```
h = imline
h = imline(hparent)
h = imline(hparent,position)
h = imline(hparent,x,y)
h = imline( ____,Name,Value)
```

### **Description**

`h = imline` begins interactive placement of a line on the current axes, and returns an `imline` object.

`h = imline(hparent)` begins interactive placement of a line on the object specified by `hparent`.

`h = imline(hparent,position)` creates a draggable, resizeable line, with coordinates defined by `position`.

`h = imline(hparent,x,y)` creates a draggable, resizeable line, with  $x$ - and  $y$ -coordinates of the endpoints defined by  $x$  and  $y$ .

`h = imline(____,Name,Value)` specifies name-value pairs that control the behavior of the line.

## Input Arguments

### **hparent** — Handle to parent object

handle

Handle to parent object, specified as a handle. The parent is typically an axes object, but can also be any other object that can be the parent of an `hggroup` object.

### **position** — Position of line endpoints

2-element vector

Position of line endpoints, specified as a 2-by-2 array of the form `[x1 y1; x2 y2]`.

### **x** — x-coordinates of line endpoints

2-element vector

x-coordinates of line endpoints, specified as a 2-element vector of the form `x = [x1 x2]`.

### **y** — y-coordinates of line endpoints

2-element vector

y-coordinates of line endpoints, specified as a 2-element vector of the form `y = [y1 y2]`.

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

### **PositionConstraintFcn** — Position constraint function

function handle

Position constraint function, specified as the comma-separated pair consisting of `'PositionConstraintFcn'` and a function handle. `fcn` is called whenever the mouse is dragged. You can use this function to control where the line can be dragged. See the help

for the `setPositionConstraintFcn` function for information about valid function handles.

## Properties

### **Deletable — ROI can be deleted**


`true` (default) | `false`

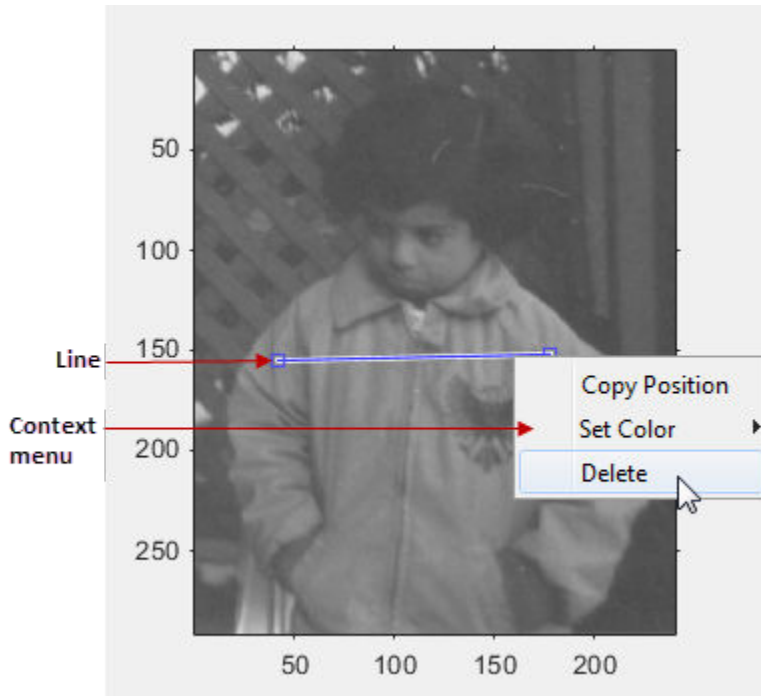
ROI can be deleted, specified as `true` or `false`.

Data Types: `logical`



## Usage

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.



The table describes the interactive behavior supported by `imline`.

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 <b>Set Color</b> from the context menu.

Interactive Behavior	Description
Retrieving the coordinates of the endpoints of the line.	Move the pointer over the line. Right-click and select <b>Copy Position</b> 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 <code>[X1 Y1; X2 Y2]</code> .
Deleting the line	Move the pointer on top of the line. Right-click and select <b>Delete</b> from the context menu. To remove this option from the context menu, set the <code>Deletable</code> property to <code>false</code> : <code>h = imline(); h.Deletable = false;</code>

## Object Functions

Each `imline` object supports a number of methods. Type methods `imline` to see a complete list.

<code>addNewPositionCallback</code>	Add new-position callback to ROI object
<code>createMask</code>	Create mask within image
<code>delete</code>	Delete handle object
<code>getColor</code>	Get color used to draw ROI object
<code>getPosition</code>	Return current position of ROI object
<code>getPositionConstraintFcn</code>	Return function handle to current position constraint function
<code>removeNewPositionCallback</code>	Remove new-position callback from ROI object
<code>resume</code>	Resume execution of MATLAB command line
<code>setColor</code>	Set color used to draw ROI object
<code>setConstrainedPosition</code>	Set ROI object to new position
<code>setPosition</code>	Move ROI object to new position
<code>setPositionConstraintFcn</code>	Set position constraint function of ROI object
<code>wait</code>	Block MATLAB command line until ROI creation is finished

## Examples

### Update Title when Line Moves

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.



```
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` function.

```
removeNewPositionCallback(h,id);
```

## Click and Drag to Place Line

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.

```
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.
- Use `imdistline` to create an interactive line with a text box that displays the distance between line endpoints.

## See Also

`imdistline` | `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
setMagnification	<p>Sets the magnification in units of screen pixels per image pixel.</p> <pre>setMagnification(new_mag)</pre> <p>where <code>new_mag</code> is a scalar magnification factor. Multiply <code>new_mag</code> by 100 to get percent magnification. For example if you call <code>setMagnification(2)</code>, the magnification box will show '200%'.</p>

## Examples

Add a magnification box to a scrollable image. Because the toolbox scrollable navigation is incompatible with standard MATLAB figure window navigation tools, the example suppresses the toolbar and menu bar in the figure window. The example positions the scroll panel in the figure window to allow room for the magnification box.

```
hFig = figure('Toolbar','none',...
             'Menubar','none');
hIm = imshow('pears.png');
hSP = imscrollpanel(hFig,hIm);
set(hSP,'Units','normalized',...
     'Position',[0 .1 1 .9])

hMagBox = immagbox(hFig,hIm);
pos = get(hMagBox,'Position');
set(hMagBox,'Position',[0 0 pos(3) pos(4)])
```

Change the magnification of the image in the scroll panel, using the scroll panel API function `setMagnification`. Notice how the magnification box updates.

```
apiSP = iptgetapi(hSP);
apiSP.setMagnification(2)
```

## See also

`imscrollpanel`, `iptgetapi`

**Introduced before R2006a**

## immovie

Make movie from multiframe image

### Syntax

```
mov = immovie(X,cmap)
mov = immovie(RGB)
```

### Description

`mov = immovie(X,cmap)` returns the movie structure array `mov` from the images in the multiframe indexed image `X` with colormap `cmap`.

`mov = immovie(RGB)` returns the movie structure array `mov` from the images in the multiframe truecolor image `RGB`.

### Examples

#### Make Movie from Indexed Image Sequence

```
load mri
mov = immovie(D,map);
imshow(mov)
```

### Input Arguments

#### **X** — Multiframe indexed image

*m*-by-*n*-by-1-by-*k* numeric array

Multiframe indexed image, specified as an *m*-by-*n*-by-1-by-*k* numeric array, where *k* is the number of frames. Each frame uses the same colormap, `cmap`.

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

**cmap — Color map***c*-by-3 numeric matrix

Color map associated with multiframe indexed image *X*, specified as a *c*-by-3 numeric matrix containing the RGB values of *c* colors.

**RGB — Multiframe truecolor image***m*-by-*n*-by-3-by-*k* numeric array

Multiframe truecolor image, specified as an *m*-by-*n*-by-3-by-*k* numeric array, where *k* is the number of frames.

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

## Output Arguments

**mov — Movie***k*-by-1 array of movie frame structures

Movie, returned as an *k*-by-1 array of movie frame structures. For details about the movie frame structure, see `getframe`.

## Tips

- To play the movie, call `implay`.
- To create a movie that can be played outside the MATLAB environment, use a `VideoWriter` object.

## See Also

`VideoWriter` | `getframe` | `implay` | `montage` | `movie`

**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**

`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`.

`immultiply` computes each element of `Z` individually in double-precision floating point. If `X` or `Y` 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)
```



## Input Arguments

### **X — First array**

real, nonsparse, numeric or logical array

First array, specified as a real, nonsparse, numeric or logical array of any dimension.

### **Y — Second array**

real, nonsparse, numeric or logical array

Second array to be multiplied with X, specified as a real, nonsparse, numeric or logical array.

- If X is numeric, then the size and class of Y can have one of the following values:
  - Y is the same size and class as X.
  - Y is the same size as X and is logical.
  - Y is a scalar of type `double`.
- If X is logical, then Y must have the same size as X. Y can be any class.

## Output Arguments

### **Z — Product**

numeric array

Product, returned as a numeric array.

- If X is numeric, then Z has the same size and class as X.
- If X is logical, then Z has the same size and class as Y.

If X or Y is an integer array, elements of the output that exceed the range of the integer type are truncated, and fractional values are rounded.

## See Also

`imabsdiff` | `imadd` | `imcomplement` | `imdivide` | `imlincomb` | `imsubtract`

**Introduced before R2006a**

# imnlfilt

Non-local means filtering of image

## Syntax

```
J = imnlfilt(I)
J = imnlfilt(I,Name,Value)
[J,estDoS] = imnlfilt( ___ )
```

## Description

`J = imnlfilt(I)` applies a non-local means-based filter to the grayscale or color image `I` and returns the resulting image in `J`.

`J = imnlfilt(I,Name,Value)` uses name-value pairs to change the behavior of the non-local means filter.

`[J,estDoS] = imnlfilt( ___ )` also returns the degree of smoothing, `estDoS`, used to estimate the denoised pixel value.

## Examples

### Denoise Grayscale Image Using Non-Local Means Filter

Read a grayscale image.

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

Add white Gaussian noise with zero mean and 0.0015 variance to the image using the `imnoise` function.

```
noisyImage = imnoise(I,'gaussian',0,0.0015);
```

Remove noise from the image through non-local means filtering. The `imnlfilt` estimates degree of smoothing based on the standard deviation of noise in the image.

```
[filteredImage,estDoS] = imnlmfilt(noisyImage);
```

Display the noisy image (left) and the non-local means filtered image (right) as a montage. Display estimated degree of smoothing, `estDoS` as the figure title.

The non-local means filter removes noise from the input image but preserves the sharpness of strong edges, such as the silhouette of the man and buildings. This function also smooths textured regions, such as the grass in the foreground of the image, resulting in less detail when compared to the noisy image.

```
montage({noisyImage,filteredImage})  
title(['Estimated degree of smoothing, ', 'estDoS = ', num2str(estDoS)]);
```

Estimated degree of smoothing, estDoS = 11.4833



## Denoise Color Image Using Non-Local Means Filter

Read a color image.

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

Add white Gaussian noise with zero mean and 0.0015 variance to the image using the `imnoise` function. Display the noisy RGB image.

```
noisyRGB = imnoise(imRGB, 'gaussian', 0, 0.0015);  
imshow(noisyRGB)
```



Convert the noisy RGB image to the L\*a\*b color space, so that the non-local means filter smooths perceptually similar colors.

```
noisyLAB = rgb2lab(noisyRGB);
```

Extract a homogeneous L\*a\*b patch from the noisy background to compute the noise standard deviation.

```
roi = [210,24,52,41];  
patch = imcrop(noisyLAB,roi);
```

In this  $L^*a^*b$  patch, compute the Euclidean distance from the origin, `edist`. Then, calculate the standard deviation of `edist` to estimate the noise.

```
patchSq = patch.^2;  
edist = sqrt(sum(patchSq,3));  
patchSigma = sqrt(var(edist(:)));
```

Set the `'DegreeOfSmoothing'` value to be higher than the standard deviation of the patch. Filter the noisy  $L^*a^*b$  image using non-local means filtering.

```
DoS = 1.5*patchSigma;  
denoisedLAB = imnlmfilt(noisyLAB, 'DegreeOfSmoothing',DoS);
```

Convert the filtered  $L^*a^*b$  image to the RGB color space. Display the filtered RGB image.

```
denoisedRGB = lab2rgb(denoisedLAB, 'Out', 'uint8');  
imshow(denoisedRGB)
```





Compare a patch from the noisy RGB image (left) and the same patch from the non-local means filtered RGB image (right).

```
roi2 = [178,68,110,110];  
montage({imcrop(noisyRGB,roi2),imcrop(denoisedRGB,roi2)})
```



## Input Arguments

### **I** — Image to filter

2-D grayscale image | 2-D color image

Image to filter, specified as a 2-D grayscale image of size  $m$ -by- $n$  or a 2-D color image of size  $m$ -by- $n$ -by-3. The size of **I** must be greater than or equal to 21-by-21.

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 quotes. You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example: `J = imnlmfilt(I,'DegreeOfSmoothing',10);`

**DegreeOfSmoothing — Degree of smoothing**

positive number

Degree of smoothing, specified as the comma-separated pair consisting of 'DegreeOfSmoothing' and a positive number. As this value increases, the smoothing in the resulting image *J* increases. If you do not specify 'DegreeOfSmoothing', then the default value is the standard deviation of noise estimated from the image. For more information, see “Default Degree of Smoothing” on page 1-1542.

**SearchWindowSize — Search window size**

21 (default) | odd-valued positive integer

Search window size, specified as the comma-separated pair consisting of 'SearchWindowSize' and an odd-valued positive integer, *s*. The search for similar neighborhoods to a pixel is limited to the *s*-by-*s* region surrounding that pixel. SearchWindowSize affects the performance linearly in terms of time. SearchWindowSize cannot be larger than the size of the input image, *I*.

**ComparisonWindowSize — Comparison window size**

5 (default) | odd-valued positive integer

Comparison window size, specified as the comma-separated pair consisting of 'ComparisonWindowSize' and an odd-valued positive integer, *c*. The `imnImfilt` function computes similarity weights using the *c*-by-*c* neighborhood surrounding pixels. ComparisonWindowSize must be less than or equal to SearchWindowSize. For more information, see “Estimate Denoised Pixel Value” on page 1-1542.

## Output Arguments

**J — Non-local means filtered image**

2-D grayscale image | 2-D color image

Non-local means filtered image, returned as a 2-D grayscale image or 2-D color image of the same size and data type as the input image, *I*.

**estDoS — Estimated degree of smoothing**

positive number

Estimated degree of smoothing, returned as a positive number. If you specify DegreeOfSmoothing, then `imnImfilt` returns the same value in estDoS. Otherwise,

`imnlmfilt` returns the default degree of smoothing estimated using “Default Degree of Smoothing” on page 1-1542.

### Tips

- To smooth perceptually close colors in an RGB image, convert the image to the CIE L\*a\*b\* color space using `rgb2lab` before applying the non-local means filter. To view the results, first convert the filtered L\*a\*b\* image to the RGB color space using `lab2rgb`.
- If the data type of `I` is `double`, then computations are performed in data type `double`. Otherwise, computations are performed in data type `single`.

## Algorithms

### Default Degree of Smoothing

The default value of `'DegreeOfSmoothing'` is the standard deviation of noise estimated from the image. To estimate the standard deviation, `imnlmfilt` convolves the image with a 3-by-3 filter proposed by J. Immerkær [2]. When `I` is a color image, the default value of `'DegreeOfSmoothing'` is the standard deviations of noise averaged across the channels.

### Estimate Denoised Pixel Value

The non-local means filtering algorithm estimates the denoised value of pixel  $p$  using these steps.

- 1 For a specific pixel,  $q$ , in the search window, calculate the weighted Euclidean distance between pixel values in the  $c$ -by- $c$  comparison windows surrounding  $p$  and  $q$ . For color images, include all channels in the Euclidean distance calculation.

The weight is a decreasing exponential function whose rate of decay is determined by the square of `'DegreeOfSmoothing'`. When an image is noisy, `'DegreeOfSmoothing'` is large and only pixels with similar values contribute to the Euclidean distance calculation. When an image has little noise, `'DegreeOfSmoothing'` is small and all pixels contribute to the Euclidean distance calculation.

The result is a numeric scalar that indicates the similarity between the neighborhood of  $p$  and the neighborhood of  $q$ .

---

**Note** In the implementation by A. Buades et al. [1], the Euclidean distance between two comparison windows is convolved with a Gaussian kernel of size  $c$ -by- $c$ . This convolution gives more weight to the Euclidean distance between pixel values for pixels near the center of the comparison window. The `imnlmfilt` function omits this step for computational efficiency.

---

- 2 Repeat this computation for each of the other pixels in the  $s$ -by- $s$  search window, finding the weighted Euclidean distance between pixel  $p$  and each of those pixels. The result is an  $s$ -by- $s$  similarity matrix that indicates similarity between the neighborhood of  $p$  and the other neighborhoods in the search window.
- 3 Normalize the similarity matrix.
- 4 Using the weights in the normalized similarity matrix, compute the weighted average of pixel values in the  $s$ -by- $s$  search window around pixel  $p$ . The result is the denoised value of  $p$ .

## References

- [1] Buades, A., B. Coll, and J.-M. Morel. "A Non-Local Algorithm for Image Denoising." *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Vol. 2, June 2005, pp. 60-65.
- [2] Immerkær, J. "Fast Noise Variance Estimation." *Computer Vision and Image Understanding*. Vol. 64, Number 2, Sept. 1996, pp. 300-302.

## See Also

`imbilatfilt` | `imdifusefilt` | `imguidedfilter` | `locallapfilt`

**Introduced in R2018b**

## imnoise

Add noise to image

### Syntax

```
J = imnoise(I, 'gaussian')
J = imnoise(I, 'gaussian', m)
J = imnoise(I, 'gaussian', m, var_gauss)
J = imnoise(I, 'localvar', var_local)
J = imnoise(I, 'localvar', intensity_map, var_local)
J = imnoise(I, 'poisson')
J = imnoise(I, 'salt & pepper')
J = imnoise(I, 'salt & pepper', d)
J = imnoise(I, 'speckle')
J = imnoise(I, 'speckle', var_speckle)
```

### Description

`J = imnoise(I, 'gaussian')` adds zero-mean, Gaussian white noise with variance of 0.01 to grayscale image `I`.

You optionally can add noise using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`J = imnoise(I, 'gaussian', m)` adds Gaussian white noise with mean `m` and variance of 0.01.

`J = imnoise(I, 'gaussian', m, var_gauss)` adds Gaussian white noise with mean `m` and variance `var_gauss`.

`J = imnoise(I, 'localvar', var_local)` adds zero-mean, Gaussian white noise of local variance `var_local`.

`J = imnoise(I, 'localvar', intensity_map, var_local)` adds zero-mean, Gaussian white noise. The local variance of the noise, `var_local`, is a function of the

image intensity values in  $I$ . The mapping of image intensity value to noise variance is specified by the vector `intensity_map`.

`J = imnoise(I, 'poisson')` generates Poisson noise from the data instead of adding artificial noise to the data. See “Algorithms” on page 1-1549 for more information.

`J = imnoise(I, 'salt & pepper')` adds salt and pepper noise, with default noise density 0.05. This affects approximately 5% of pixels.

`J = imnoise(I, 'salt & pepper', d)` adds salt and pepper noise, where  $d$  is the noise density. This affects approximately  $d \times \text{numel}(I)$  pixels.

`J = imnoise(I, 'speckle')` adds multiplicative noise using the equation  $J = I + n * I$ , where  $n$  is uniformly distributed random noise with mean 0 and variance 0.04.

`J = imnoise(I, 'speckle', var_speckle)` adds multiplicative noise with variance `var_speckle`.

## Examples

### Add Noise to an Image

Read a grayscale image and display it.

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



Add salt and pepper noise, with a noise density of 0.02, to the image. Display the result.

```
J = imnoise(I, 'salt & pepper', 0.02);  
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);
```

## Input Arguments

### **I** — Grayscale image

numeric matrix | gpuArray

Grayscale image, specified as a numeric matrix. If I has more than two dimensions, it is treated as a multidimensional intensity image and not as an RGB image.

---

**Note** For Poisson noise, images of data type `int16` are not allowed.

---

To add noise using a GPU, specify `I` as a `gpuArray` that contains a numeric matrix.

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

### **m — Mean of Gaussian noise**

0 (default) | numeric scalar

Mean of Gaussian noise, specified as a numeric scalar.

### **var\_gauss — Variance of Gaussian noise**

0.01 (default) | numeric scalar

Variance of Gaussian noise, specified as a numeric scalar.

### **var\_local — Local variance of Gaussian noise**

numeric matrix | numeric vector

Local variance of Gaussian noise, specified as one of the following:

- A numeric matrix of the same size as `I`.
- A numeric vector the same length of `intensity_map`.

### **intensity\_map — Intensity values**

numeric vector

Intensity values that are mapped to Gaussian noise variance, specified as a numeric vector. The values are normalized to the range [0, 1].

You can plot the functional relationship between noise variance `var_local` and image intensity using the command `plot(intensity_map, var_local)`.

### **d — Noise density**

0.05 (default) | numeric scalar

Noise density for salt and pepper noise, specified as a numeric scalar. The noise is applied to approximately `d*numel(I)` pixels.

### **var\_speckle — Variance of multiplicative noise**

0.04 (default) | numeric scalar

Variance of multiplicative noise, specified as a numeric scalar.

## Output Arguments

### **J** — Noisy image

numeric matrix | gpuArray

Noisy image, returned as a numeric matrix of the same data type as input image **I**.

If noise is added using a GPU, then **J** is returned as a `gpuArray` containing a numeric matrix.

## Algorithms

- 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 a different class, 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.
- The Poisson distribution depends on the data type of input image **I**:
  - 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.

## Extended Capabilities

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

### See Also

rand | randn

**Introduced before R2006a**

# imopen

Morphologically open image

## Syntax

```
J = imopen(I,SE)
J = imopen(I,nhood)
```

## Description

`J = imopen(I,SE)` performs morphological opening on the grayscale or binary image `I`, returning the opened image, `J`. `SE` is a single structuring element object returned by the `strel` or `offsetstrel` functions. The morphological open operation is an erosion followed by a dilation, using the same structuring element for both operations.

You optionally can perform the opening using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

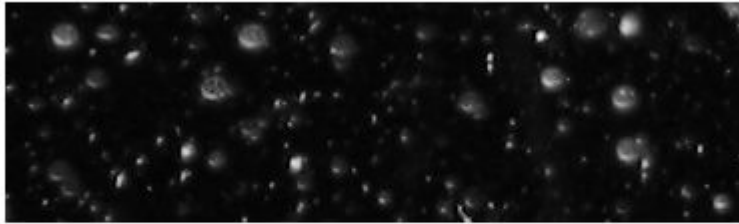
`J = imopen(I,nhood)` opens the image `I`, where `nhood` is a matrix of 0s and 1s that specifies the structuring element neighborhood. This is equivalent to the syntax `imopen(I,strel(nhood))`. The `imopen` function determines the center element of the neighborhood by `floor((size(nhood)+1)/2)`.

## 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 Using 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,[])
```

## Input Arguments

### I — Input image

grayscale image | binary image | `gpuArray`

Input image, specified as a grayscale image or binary image. I can have any dimension, and must be real and nonparse.

To perform the opening using a GPU, specify I as a `gpuArray` that contains an image of type `uint8` or `logical`.

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

### SE — Structuring element

`strel` object | `offsetstrel` object

Structuring element, specified as a single `strel` object or `offsetstrel` object. If the image I is data type `logical`, the structuring element must be flat.

If you perform the opening using a GPU, then SE must be flat and two-dimensional.

### nhood — Structuring element neighborhood

matrix of 0s and 1s

Structuring element neighborhood, specified as a matrix of 0s and 1s.

Example: `[0 1 0; 1 1 1; 0 1 0]`

## Output Arguments

### J — Opened image

grayscale image | binary image | gpuArray

Opened image, returned as a grayscale image or binary image. J has the same class as input image I.

If the opening is performed using a GPU, then J is returned as a `gpuArray` that contains a grayscale or binary image of the same class 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 “Understand 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.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:



- `gpuArray` input must be of type `uint8` or `logical` and the structuring element must be flat and two-dimensional.

For more information, see “Image Processing on a GPU”.

## See Also

### Functions

`imclose` | `imdilate` | `imerode`

### Objects

`offsetstrel` | `strel`

**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**

`boundarymask` | `superpixels`

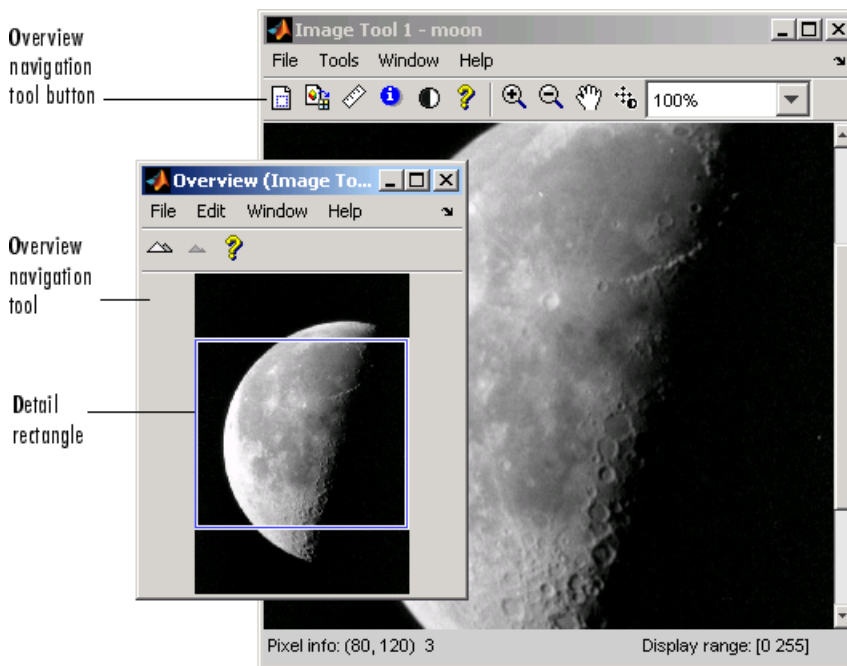
**Introduced in R2016a**

# imoverview

Overview tool for image displayed in scroll panel

Use the `imoverview` function to create an Overview tool. The Overview tool is a navigation aid for images displayed in a scroll panel.

`imoverview` creates the Overview 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.



## Syntax

```
imoverview(himage)
```

```
htool = 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`.

`htool = imoverview( ___ )` returns `htool`, a handle to the Overview tool figure.

## Examples

### Create Overview Tool

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)
```

## Input Arguments

**himage** — Handle to image

handle

Handle to image, specified as a handle.

## Output Arguments

**htool** — Handle to Overview tool

handle

Handle to Overview tool figure, returned as a handle.

## Tips

- To create an Overview tool that can be embedded in an existing figure or uipanel object, use `imoverviewpanel`.

## 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

```
P = impixel
```

```
P = impixel(I)
```

```
P = impixel(X,map)
```

```
P = impixel(I,c,r)
```

```
P = impixel(X,map,c,r)
```

```
P = impixel(x,y,I,xi,yi)
```

```
P = impixel(x,y,X,map,xi,yi)
```

```
[xi2,yi2,P] = impixel( ___ )
```

## Description

`P = impixel` lets you select pixels interactively from the image in the current axes. With this syntax and the other interactive syntaxes, the pixel selection tool blocks the MATLAB command line until you complete the operation. When you finish selecting pixels, `impixel` returns the pixel values in `P`.

Use normal button clicks to select pixels. Press **Backspace** or **Delete** to remove the previously selected pixel. To add a final pixel and finish pixel selection in one step, press shift-click, or right-click or double-click. To finish selecting pixels without adding a final pixel, press **Return**.

`P = impixel(I)` returns the value of pixels in image `I`. The `impixel` function displays the image and waits for you to select pixels in the image using the mouse.

`P = impixel(X,map)` returns the value of pixels in indexed image `X` with corresponding colormap, `map`. The `impixel` function displays the image and waits for you to select pixels in the image using the mouse.

`P = impixel(I, c, r)` returns the values of specified pixels in image `I`. The column and row coordinates of the sampled pixels are specified by `c` and `r`. The  $k$ th row of `P` contains the RGB values for the pixel  $(r(k), c(k))$ .

`P = impixel(X, map, c, r)` returns the values of specified pixels in indexed image `X`. The column and row coordinates of the sampled pixels are specified by `c` and `r`.

`P = impixel(x, y, I, xi, yi)` returns the values of pixels in the specified image, `I`, using a nondefault coordinate system where `x` and `y` specify the image limits. `xi` and `yi` specify the spatial coordinates of the pixels whose values are returned in `P`.

`P = impixel(x, y, X, map, xi, yi)` returns the values of pixels in the specified indexed image, `X`, with corresponding colormap, `map`, using a nondefault coordinate system.

`[xi2, yi2, P] = impixel( ___ )` additionally returns the coordinates of the selected pixels.

## 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 = 4×3
```

```
    62    29    64  
    62    34    63  
   166    54    60  
    59    28    47
```

## Input Arguments

### **I** — Input image

RGB image | grayscale image | binary image

Input image, specified as an RGB, grayscale, or binary image.

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

### **X** — Indexed image

numeric array

Indexed image, specified as a numeric array.

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

### **map** — Colormap

*m*-by-3 numeric array

Colormap associated with the indexed image *X*, specified as an *m*-by-3 numeric array. Values with data type `single` or `double` must be in the range [0, 1].

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

### **c, r** — Column or row index of pixels to sample

vector of positive integers

Column or row index of pixels to sample, specified as a vector of positive integers. The length and data type of *c* and *r* must match.

Data Types: `single` | `double`

### **x, y** — Image limits in world coordinates

2-element numeric vector

Image limits in world coordinates along the *x* or *y* direction, specified as a 2-element numeric vector of the form [min max]. The values of *x* and *y* set the image *XData* and *YData*. The data type of *x* and *y* must match.

Data Types: `single` | `double`

### **xi, yi** — *x*- or *y*-coordinates of pixels to sample

numeric vector

$x$ - or  $y$ -coordinates of pixels to sample, in the world coordinate system defined by  $x$  and  $y$ , specified as a numeric vector. The length and data type of  $x_i$  and  $y_i$  must match.

Data Types: `single` | `double`

## Output Arguments

### **P** — Sampled pixel values

$p$ -by-3 matrix

Sampled pixel values, returned as a  $p$ -by-3 matrix. `impixel` always returns pixel values as RGB triplets, regardless of the image type. The values in each row of the matrix depends on the image type.

Image Type	Result
RGB	Returns the actual RGB data for the pixel. The values are data type <code>double</code> .
Grayscale	Returns the intensity value as an RGB triplet, where $R=G=B$ . The values are data type <code>double</code> .
Indexed	Returns the RGB triplet stored in the row of the colormap that the pixel value points to. The values have the same data type as the colormap, <code>map</code> .
Binary	Returns the intensity value as an RGB triplet, where $R=G=B$ . The values are data type <code>double</code> .

### **$x_i2$ , $y_i2$** — $x$ - or $y$ -coordinates of sampled pixels

numeric vector

$x$ - or  $y$ -coordinates of sampled pixels, returned as a numeric vector.

- If you select pixels interactively using the mouse, or if you specify pixels using row and column indices  $c$  and  $r$ , then  $x_i2$  and  $y_i2$  are interpreted as pixel indices.
- If you specify the image limits,  $x$  and  $y$ , in a nondefault spatial coordinate system, then  $x_i2$  and  $y_i2$  are interpreted as  $x$ - and  $y$ -coordinates in that coordinate system.

## See Also

`getpts` | `improfile`

**Introduced before R2006a**

# impixelinfo

Pixel Information tool

Use the `impixelinfo` function to create a Pixel Information tool. 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. For more information about using the tool, see [Tips](#) on page 1-1571.

X and Y coordinates	Pixel Value
Pixel info: (418, 261)	143

## Syntax

```
impixelinfo
impixelinfo(h)
impixelinfo(hparent,himage)
htool = impixelinfo( ___ )
```

## Description

`impixelinfo` creates a Pixel Information tool in the current figure.

`impixelinfo(h)` creates a Pixel Information tool in the figure specified by the handle `h`.

`impixelinfo(hparent,himage)` creates a Pixel Information tool in `hparent` that provides information about the pixels in `himage`.

`htool = impixelinfo( ___ )` returns a handle to the Pixel Information tool uipanel.

## Examples

### Add Pixel Information Tool to Figure

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;
```

## Input Arguments

### **h** — Handle to graphics object

handle

Handle to a figure, axes, uipanel, or image graphics object, specified as a handle. Axes, uipanel, or figure objects must contain at least one image object.

### **hparent** — Handle to figure or uipanel object

handle

Handle to a figure or uipanel object that contains the Pixel Information tool, specified as a handle.

### **himage** — Handle to images

handle | array of handles

Handle to one or more images, specified as a handle or an array of image handles.



## Output Arguments

### htool — Handle to Pixel Information tool

handle

Handle to Pixel Information tool uipanel, returned as a handle.

## Tips

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 CDataMapping property set to direct	(X,Y) value <index> [R G B]	(19,10) 82 <4> [15 255 10]

- If you want to display the pixel information without the “Pixel Info” label, use the `impixelinfoval` function.
- 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**.

## See Also

`impixelinfoval` | `imtool`

**Introduced before R2006a**

# impixelinfoval

Pixel Information tool without text label

Use the `impixelinfoval` function to create a Pixel Information tool without the **Pixel info:** text label. 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. The information displayed depends on the image type. See `impixelinfo` for more details about using the Pixel Information tool.

X and Y coordinates	Pixel Value
(167, 251)	114

## Syntax

```
htool = impixelinfoval(hparent,himage)
```

## Description

`htool = impixelinfoval(hparent,himage)` creates a Pixel Information tool in `hparent` that provides information about the pixels in `himage`.

## Examples

### Add Pixel Information Tool Without Text Label

Add a Pixel Information tool to a figure, excluding the text label. 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)
```

## Input Arguments

### **hparent** — Handle to figure or uipanel object

handle

Handle to a figure or uipanel object that contains the Pixel Information tool, specified as a handle.

### **himage** — Handle to images

handle | array of handles

Handle to one or more images, specified as a handle or an array of image handles.

## Output Arguments

### **htool** — Handle to Pixel Information tool

handle

Handle to Pixel Information tool uipanel, returned as a handle.

## See Also

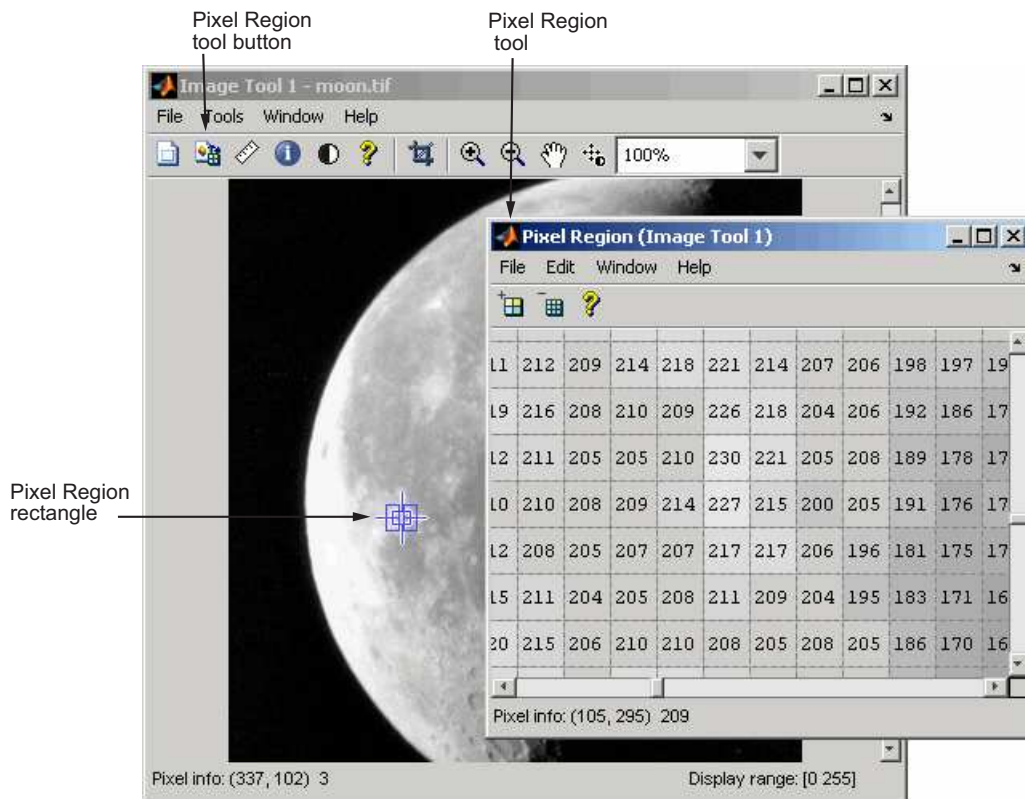
[impixelinfo](#)

**Introduced before R2006a**

# impixelregion

Pixel Region tool

Use the `impixelregion` function to create a Pixel Region tool. 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. For more information about using the tool, see Tips on page 1-1577.



## Syntax

`impixelregion`

```
impixelregion(h)  
htool = impixelregion( ___ )
```

## Description

`impixelregion` creates a Pixel Region tool associated with the image displayed in the current figure, called the target image.

`impixelregion(h)` creates a Pixel Region tool in the figure specified by the handle `h`.

`htool = impixelregion( ___ )` returns `htool`, a handle of the Pixel Region tool figure.

## Examples

### Create Pixel Region Tool

Display an image and then create a Pixel Region tool associated with the image.

```
imshow peppers.png  
impixelregion
```

## Input Arguments

**h — Handle to graphics object**

handle

Handle to a figure, axes, uipanel, or image graphics object, specified as a handle. If `h` is an axes or figure handle, `impixelregion` uses the first image returned by `findobj(H, 'Type', 'image')`.

## Output Arguments

**htool — Handle to Pixel Region tool**

handle

Handle to Pixel Region tool figure, returned as a handle.

## Tips

- 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**.
- To create a Pixel Region tool that can be embedded in an existing figure window or uipanel, use `impixelregionpanel`.

## See Also

`impixelinfo` | `impixelregionpanel` | `imtool`

**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](#)

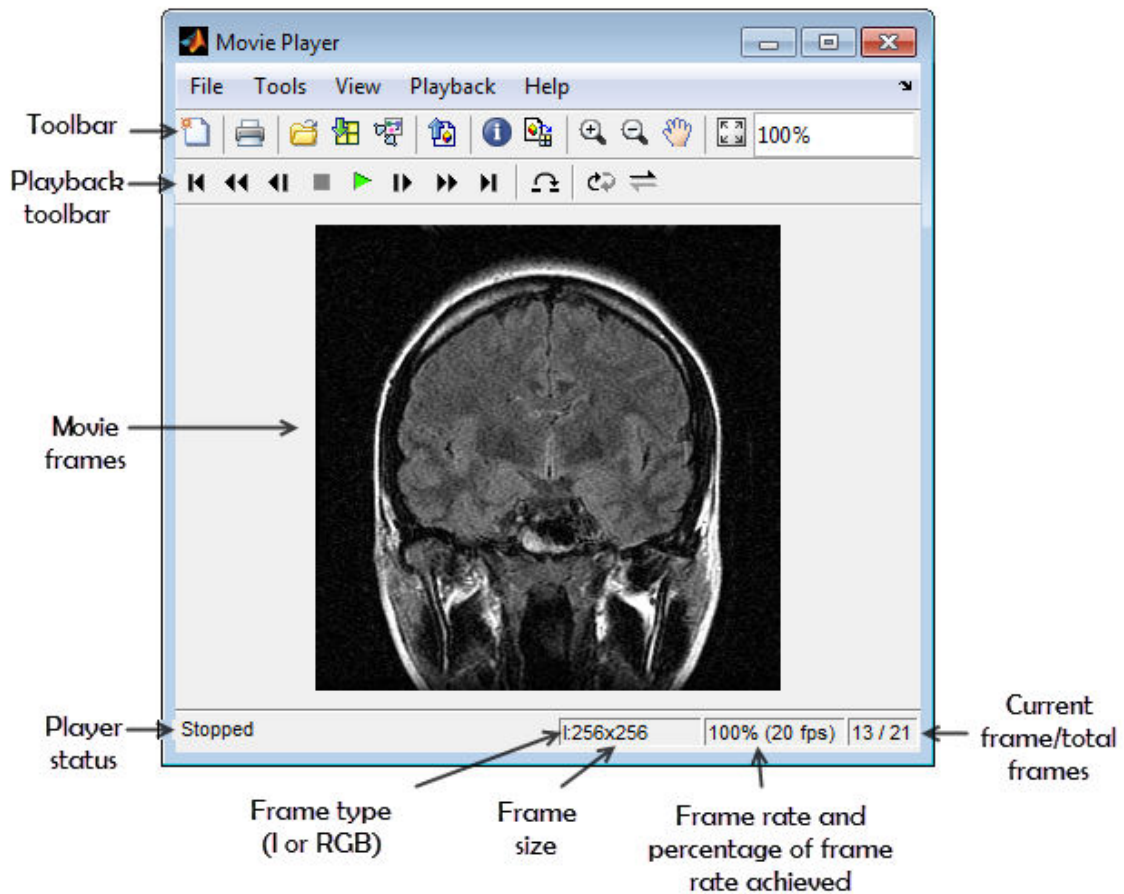
**Introduced before R2006a**

## **implay**

Play movies, videos, or image sequences

Use the `implay` function to open the **Video Viewer** app, which plays MATLAB movies, videos, or image sequences (also called image stacks). 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. You can open multiple Video Viewers to view different movies simultaneously.

The figure shows the Video Viewer app containing an image sequence.



## Syntax

```

implay
implay(filename)
implay(I)
implay( __ , fps)

```

## Description

`implay` opens the Video Viewer app. To select the movie or image sequence that you want to play, use the Video Viewer **File** menu.

`implay(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.

`implay(I)` opens the Video Viewer app, displaying the first frame in the multiframe image specified by `I`.

`implay( ____, fps)` specifies the frame rate at which you want to view the movie or image sequence.

## Examples

### Play Three Types of Videos

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');
```

## Input Arguments

**filename** — File name

character vector | string scalar

File name of AVI file, specified as a character vector or string scalar.

Data Types: `char` | `string`

### **I** — **Multiframe image sequence**

MATLAB movie structure | sequence of binary, grayscale, or truecolor images

Multiframe image sequence, specified as one of the following.

- A MATLAB movie structure
- A binary or grayscale image sequence, specified as an  $m$ -by- $n$ -by-1-by- $k$  array or an  $m$ -by- $n$ -by- $k$  array
- A truecolor (RGB) image sequence, specified as an  $M$ -by- $N$ -by-3-by- $K$  array.

I can be numeric but `uint8` is preferred. The actual data type used to display pixels may differ from the source data type.

### **fps** — **Frames per second**

20 (default) | positive number

Frame rate, in frames-per-second, specified as a positive number. If omitted, the Video Viewer uses the frame rate specified in the file or the default value, 20.

## 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.

## See Also

`VideoWriter` | **Volume Viewer** | `immovie` | `montage` | `movie`

**Introduced in R2007b**

## impoint

Create draggable point

---

**Note** `impoint` is not recommended. Use `drawpoint` instead.

---

### Description

An `impoint` object encapsulates an interactive point over an image.

You can adjust the position of the point by using the mouse. The point also has a context menu that controls aspects of its appearance and behavior. For more information, see “Usage” on page 1-1586.

### Creation

### Syntax

```
h = impoint
h = impoint(hparent)
h = impoint(hparent,position)
h = impoint(hparent,x,y)
h = impoint( ____,Name,Value)
```

### Description

`h = impoint` begins interactive placement of a point on the current axes, and returns an `impoint` object.

`h = impoint(hparent)` begins interactive placement of a point on the object specified by `hparent`.

`h = impoint(hparent,position)` creates a draggable point with coordinates defined by `position`.

`h = impoint(hparent,x,y)` creates a draggable point with  $x$ - and  $y$ -coordinates of the point defined by  $x$  and  $y$ .

`h = impoint( ____,Name,Value)` specifies name-value pairs that control the behavior of the point.

## Input Arguments

### **hparent** — Handle to parent object

handle

Handle to parent object, specified as a handle. The parent is typically an axes object, but can also be any other object that can be the parent of an `hggroup` object.

### **position** — Position of point

2-element vector

Position of point, specified as a 2-element vector of the form `[x y]`.

### **x, y** — $x$ - or $y$ -coordinate of point

numeric scalar

$x$ - or  $y$ -coordinate of the point, specified as a numeric scalar.

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

### **PositionConstraintFcn** — Position constraint function

function handle

Position constraint function, specified as the comma-separated pair consisting of `'PositionConstraintFcn'` and a function handle. `fcn` is called whenever the mouse is dragged. You can use this function to control where the point can be dragged. See the help for the `setPositionConstraintFcn` function for information about valid function handles.

## Properties

### Deletable — ROI can be deleted

true (default) | false

ROI can be deleted, specified as `true` or `false`.

Data Types: `logical`

## Usage


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 table describes the interactive behavior supported by `impoint`.

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 <b>Set Color</b> 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 <b>Copy Position</b> 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 <b>Delete</b> 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>

## Object Functions

Each `impoint` object supports a number of methods. Type `methods impoint` to see a complete list.

<code>addNewPositionCallback</code>	Add new-position callback to ROI object
<code>createMask</code>	Create mask within image
<code>delete</code>	Delete handle object
<code>getColor</code>	Get color used to draw ROI object
<code>getPosition</code>	Return current position of ROI object
<code>getPositionConstraintFcn</code>	Return function handle to current position constraint function
<code>removeNewPositionCallback</code>	Remove new-position callback from ROI object
<code>resume</code>	Resume execution of MATLAB command line
<code>setColor</code>	Set color used to draw ROI object
<code>setConstrainedPosition</code>	Set ROI object to new position
<code>setPosition</code>	Move ROI object to new position
<code>setPositionConstraintFcn</code>	Set position constraint function of ROI object
<code>setString</code>	Set text label for point ROI object

`wait` Block MATLAB command line until ROI creation is finished

## Examples

### Enforce Boundary Constraint when Point Moves

Use `impoint` methods to set custom color, set a label, enforce a boundary constraint, and update position in title as point moves.

```
imshow('rice.png')
h = impoint(gca,100,200);
```

Update the title with the new position by using `addNewPositionCallback`.

```
addNewPositionCallback(h,@(h) title(sprintf('%1.0f,%1.0f',h(1),h(2))));
```

Construct a boundary constraint function by using `makeConstrainToRectFcn`.

```
fcn = makeConstrainToRectFcn('impoint',get(gca,'XLim'),get(gca,'YLim'));
```

Enforce the boundary constraint function using `setPositionConstraintFcn`.

```
setPositionConstraintFcn(h,fcn);
setColor(h,'r');
setString(h,'Point label');
```

### Click and Drag to Move Point

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

```
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, then 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` | `imfreehand` | `imline` | `impoly` | `imrect` | `imroi` |  
`makeConstrainToRectFcn`

**Introduced before R2006a**

## impoly

Create draggable, resizable polygon

---

**Note** `impoly` is not recommended. Use `drawpolygon` instead.

---

### Description

An `impoly` object encapsulates an interactive polygon over an image.

You can add vertices and adjust the size and position of the polygon by using the mouse. The polygon also has a context menu that controls aspects of its appearance and behavior. For more information, see “Usage” on page 1-1592.

### Creation

### Syntax

```
h = impoly
h = impoly(hparent)
h = impoly(hparent,position)
h = impoly( ____,Name,Value)
```

### Description

`h = impoly` begins interactive placement of a polygon on the current axes, and returns an `impoly` object.

`h = impoly(hparent)` begins interactive placement of a polygon on the object specified by `hparent`.

`h = impoly(hparent,position)` creates a draggable, resizeable polygon with vertices at coordinates defined by `position`.

`h = impoly( ____, Name, Value)` specifies name-value pairs that control the behavior of the polygon.

## Input Arguments

### **hparent** — Handle to parent object

handle

Handle to parent object, specified as a handle. The parent is typically an axes object, but can also be any other object that can be the parent of an `hggroup` object.

### **position** — Position of polygon vertices

*n*-by-2 matrix

Position of polygon vertices, specified as an *n*-by-2 matrix. The two columns define the *x*- and *y*-coordinate, respectively, of each of the *n* vertices.

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

### **PositionConstraintFcn** — Position constraint function

function handle

Position constraint function, specified as the comma-separated pair consisting of `'PositionConstraintFcn'` and a function handle. `fcn` is called whenever the mouse is dragged. You can use this function to control where the polygon can be dragged. See the help for the `setPositionConstraintFcn` function for information about valid function handles.

### **Closed** — Polygon is closed

`true` (default) | `false`

Polygon is closed, specified as the comma-separated pair consisting of `'Closed'` and `true` or `false`. When set to `true` (the default), `impoly` creates a closed polygon, that is, it draws a straight line between the last vertex specified and the first vertex specified to create a closed region. When `Closed` is `false`, `impoly` does not connect the last vertex with the first vertex, creating an open polygon (or polyline).

Data Types: `logical`

## Properties

### Deletable — ROI can be deleted


true (default) | false

ROI can be deleted, specified as `true` or `false`.

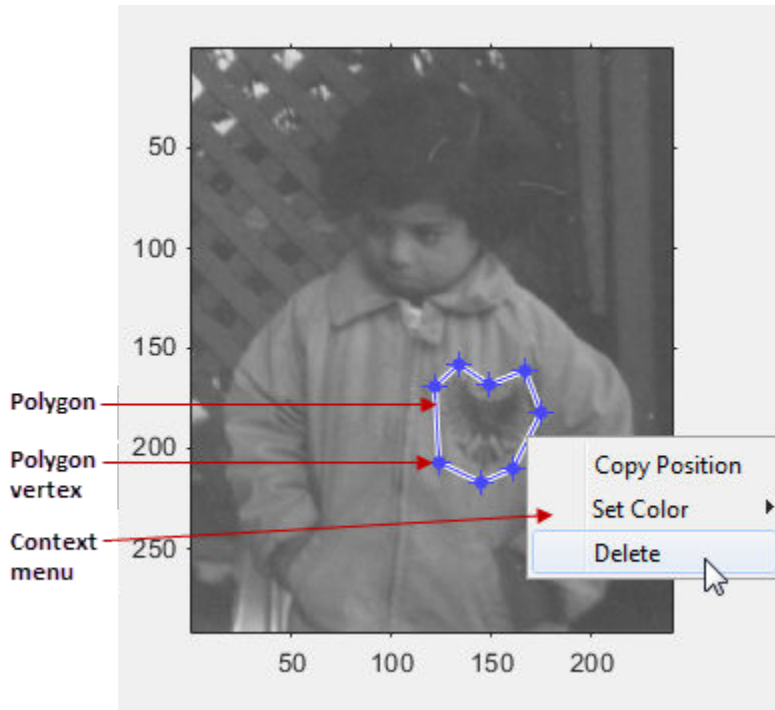
Data Types: `logical`

## Usage


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. By default, `impoly` 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 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 figure shows the context menu when the pointer is on the polygon but not on a vertex.



The table lists the interactive behaviors supported by `impoly`.

Interactive Behavior	Description
Closing the polygon.	Use any of the following mechanisms: <ul style="list-style-type: none"> <li>• Move the pointer over the initial vertex of the polygon that you selected. The pointer changes to a circle . Click either mouse button.</li> <li>• 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.</li> <li>• 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.</li> </ul>

Interactive Behavior	Description
Adding a new vertex.	Move the pointer over an edge of the polygon. Press and hold the <b>A</b> 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 <b>Delete Vertex</b> 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 <b>Delete</b> from the context menu. To remove this option from the context menu, set the <code>Deletable</code> 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 <b>Set Color</b> from the context menu.
Retrieving the coordinates of the vertices	Move the pointer inside the polygon. Right-click and select <b>Copy Position</b> from the context menu. <code>impoly</code> copies an $n$ -by-2 array containing the $x$ - and $y$ -coordinates of each vertex to the clipboard. $n$ is the number of vertices you specified.

## Object Functions

Each `impoly` object supports a number of methods. Type methods `impoly` to see a complete list.



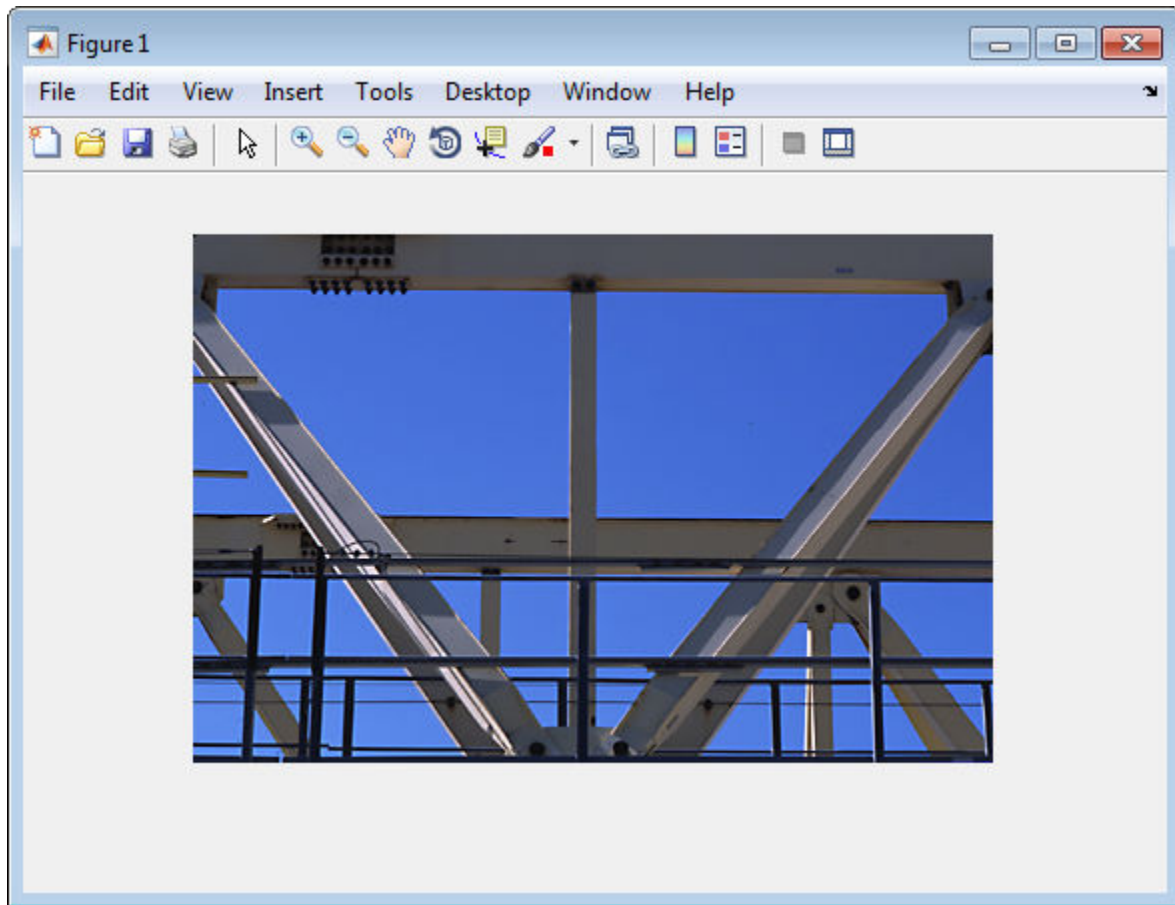
<code>addNewPositionCallback</code>	Add new-position callback to ROI object
<code>createMask</code>	Create mask within image
<code>delete</code>	Delete handle object
<code>getColor</code>	Get color used to draw ROI object
<code>getPosition</code>	Return current position of ROI object
<code>getPositionConstraintFcn</code>	Return function handle to current position constraint function
<code>removeNewPositionCallback</code>	Remove new-position callback from ROI object
<code>resume</code>	Resume execution of MATLAB command line
<code>setClosed</code>	Set closure behavior of ROI object
<code>setColor</code>	Set color used to draw ROI object
<code>setConstrainedPosition</code>	Set ROI object to new position
<code>setPosition</code>	Move ROI object to new position
<code>setPositionConstraintFcn</code>	Set position constraint function of ROI object
<code>setVerticesDraggable</code>	Set vertex behavior of ROI object
<code>wait</code>	Block MATLAB command line until ROI creation is finished

## Examples

### Draw Polygon on Image and Specify Position Constraint Function

Display an image.

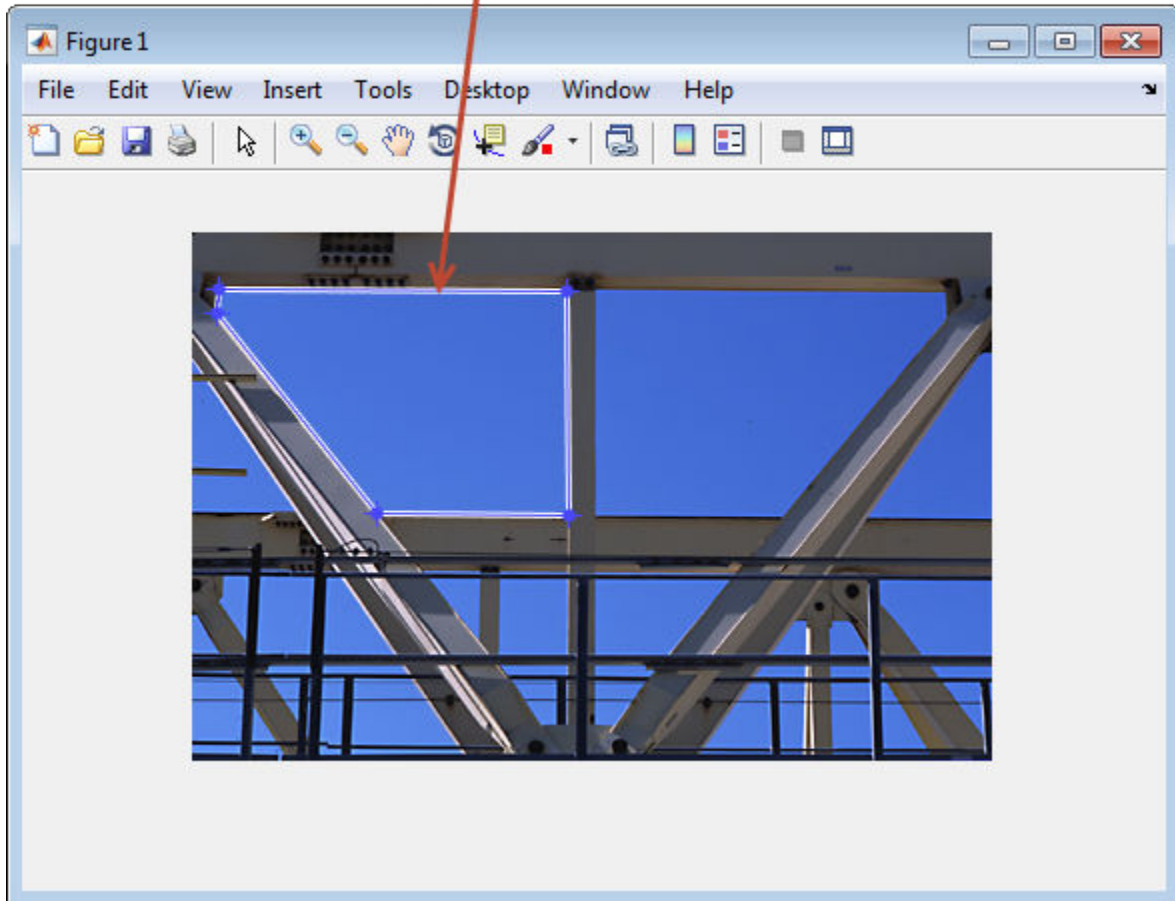
```
imshow('gantrycrane.png')
```



Draw a polygon on the image, specifying the location of five vertices.

```
h = impoly(gca,[188,30; 189,142; 93,141; 13,41; 14,29]);
```

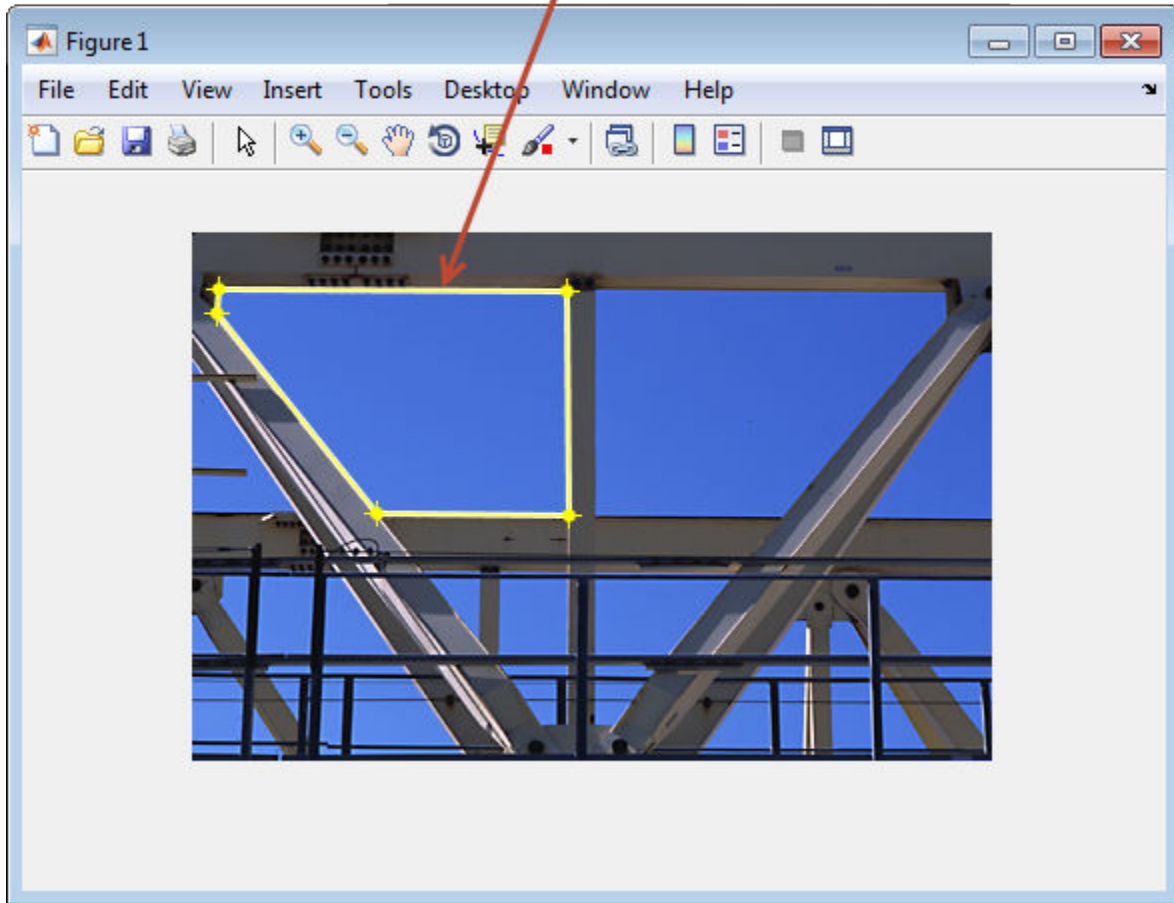
Polygon



Set the color of the polygon to yellow.

```
setColor(h, 'yellow');
```

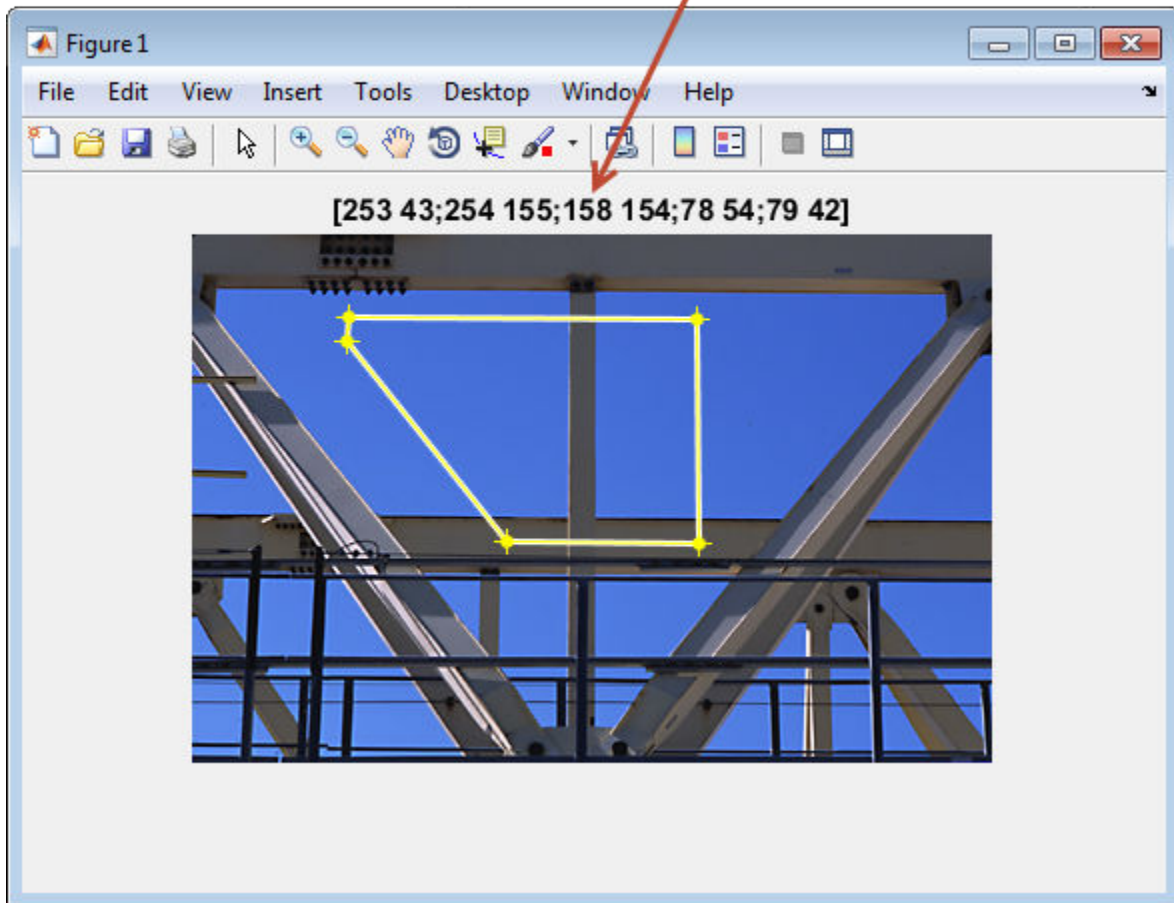
Color of polygon changed to yellow



Define a function for the new position callback. This function displays the current position of the polygon whenever it is moved.

```
addNewPositionCallback(h, @(p) title(mat2str(p,3)));
```

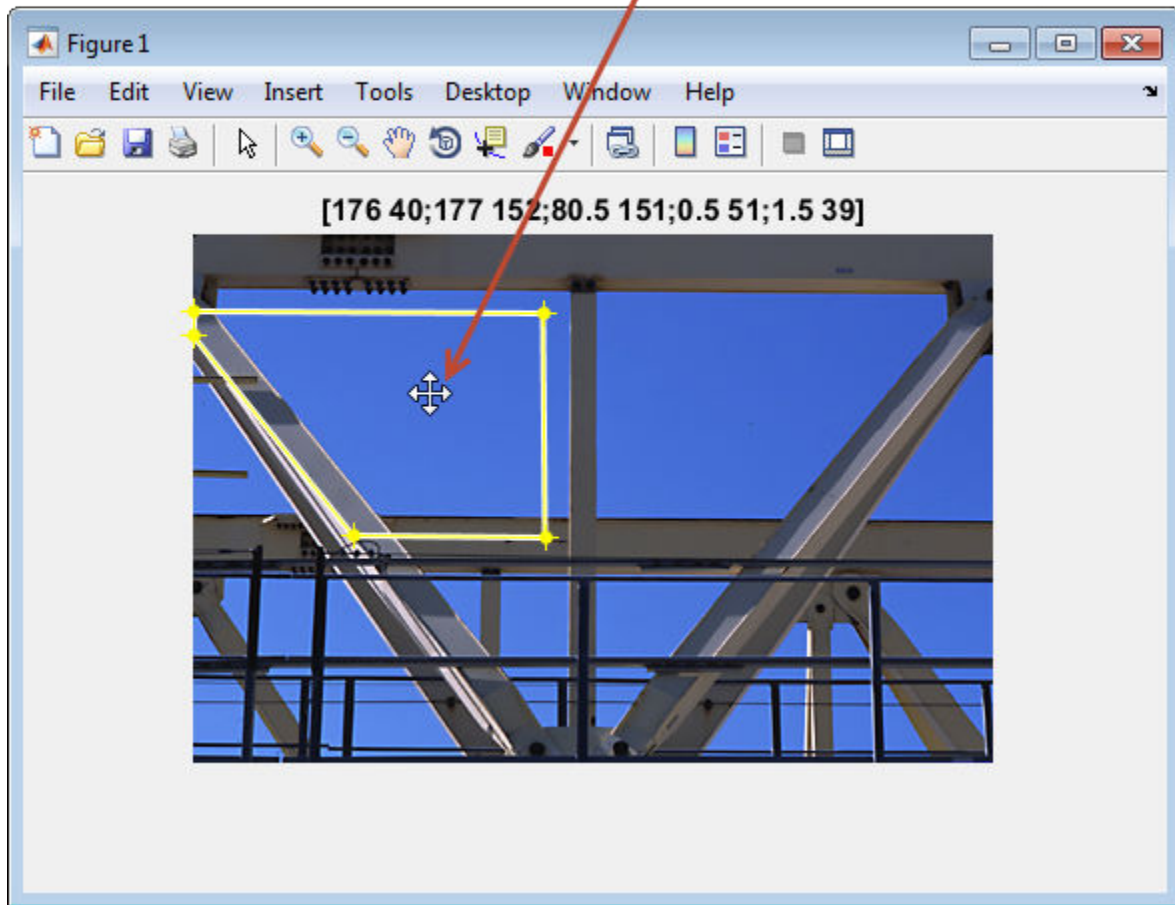
Title with current position of polygon



Create the function that constrains the movement of the polygon by using `makeConstrainToRectFcn`, specifying the boundary of the image as the limits. Enforce the boundary constraint function using `setPositionConstraintFcn`.

```
fcn = makeConstrainToRectFcn('impoly',get(gca,'XLim'),get(gca,'YLim'));
setPositionConstraintFcn(h,fcn);
```

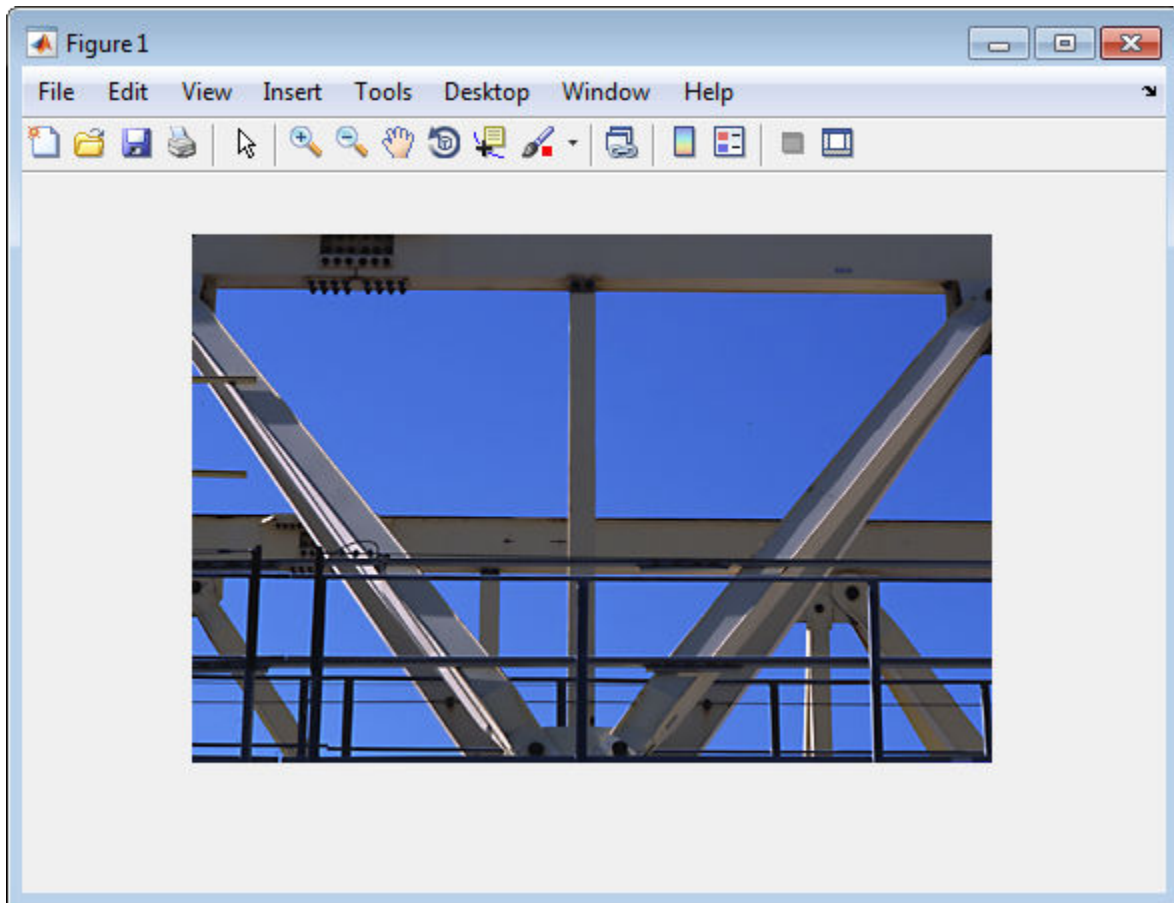
Can't move polygon past image border



### Interactively Create a Polygon by Clicking to Specify Vertex Locations

Display image.

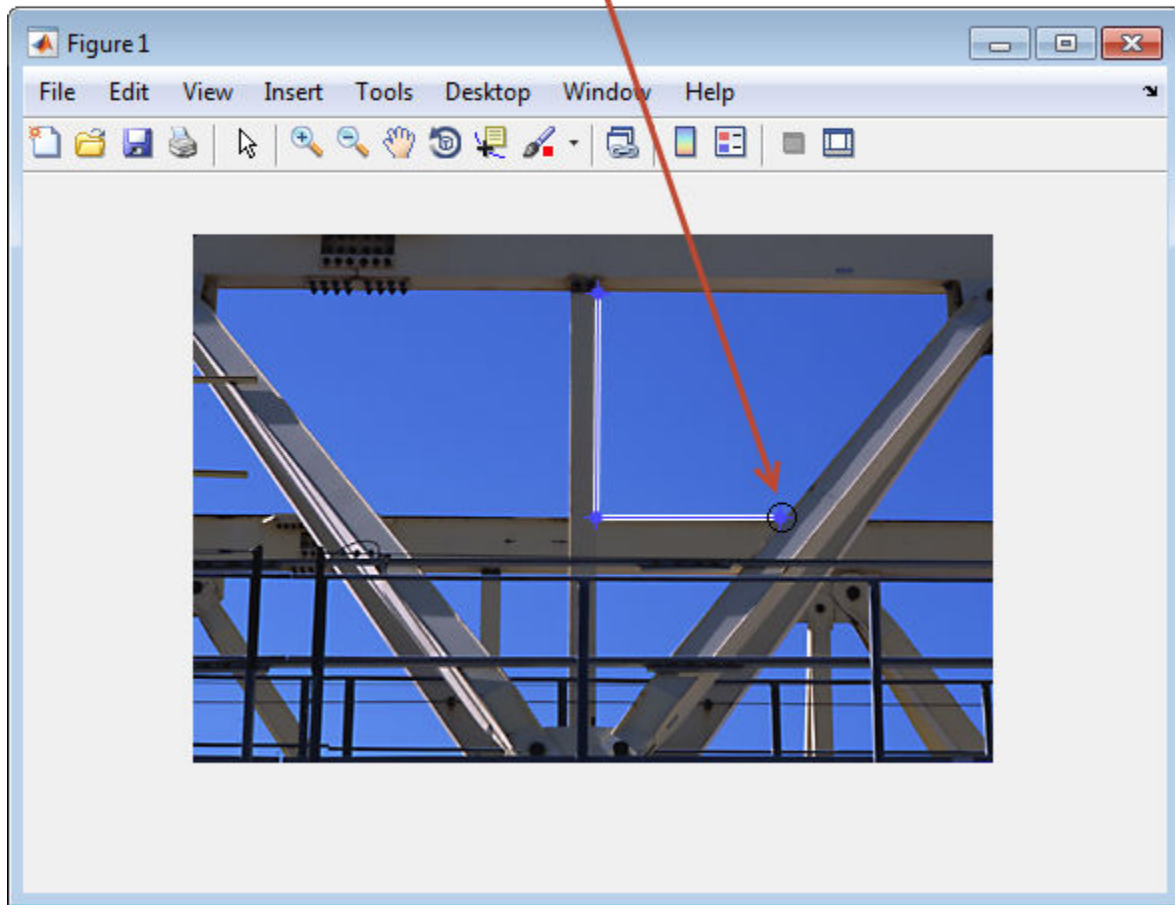
```
figure  
imshow('gantrycrane.png');
```



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);
```

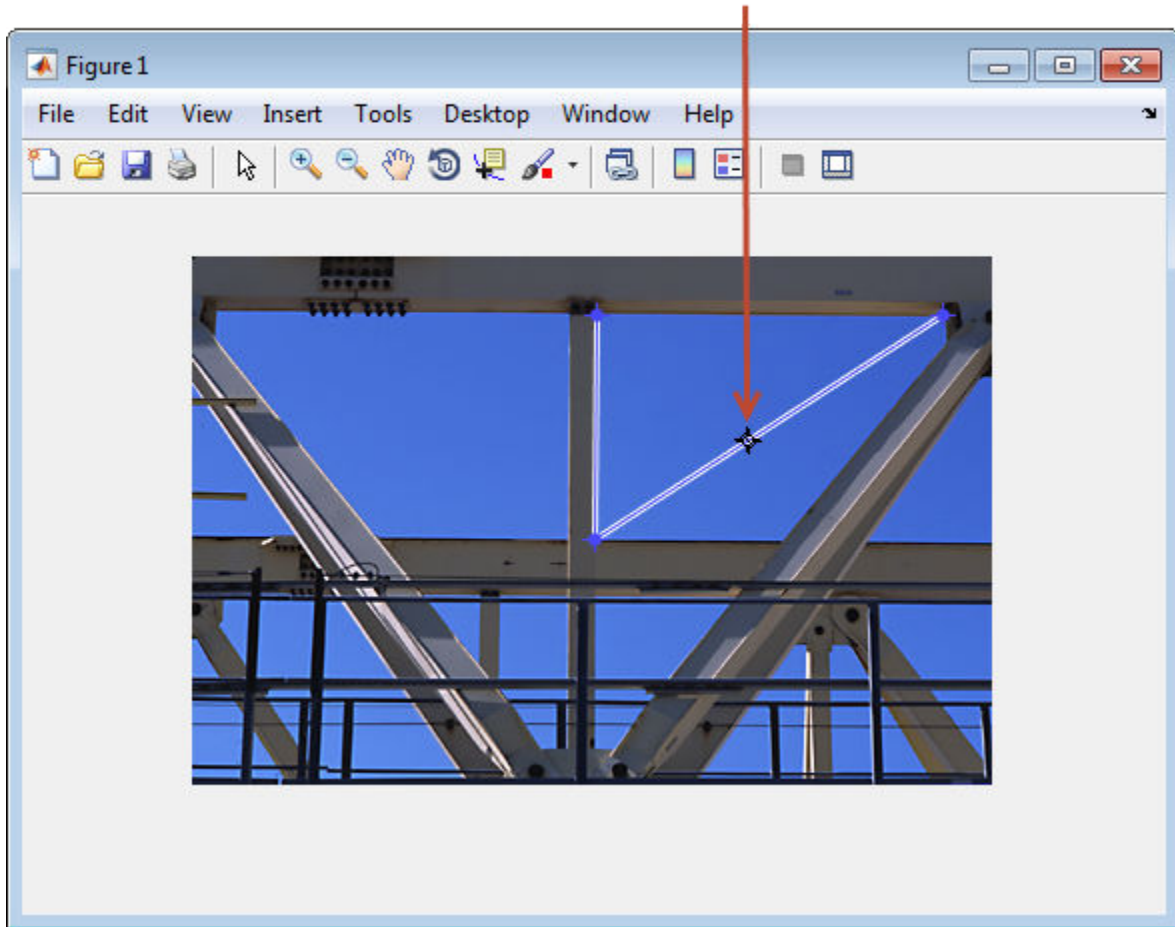
Cursor changes to circle.



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 and hold 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 “Usage” on page 1-1592.



Press and hold the A key to get the add vertex cursor. Then drag the new vertex where desired.



## 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.

## See Also

`imellipse` | `imfreehand` | `imline` | `impoint` | `imrect` | `imroi` |  
`makeConstrainToRectFcn`

**Introduced in R2007b**

# 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.

`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`.

`[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)` retrieves pixel intensity values using a nondefault coordinate system where `x` and `y` specify 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.

## Examples

### Plot Multisegment Line from Image

Read an image into the workspace, and display it.

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

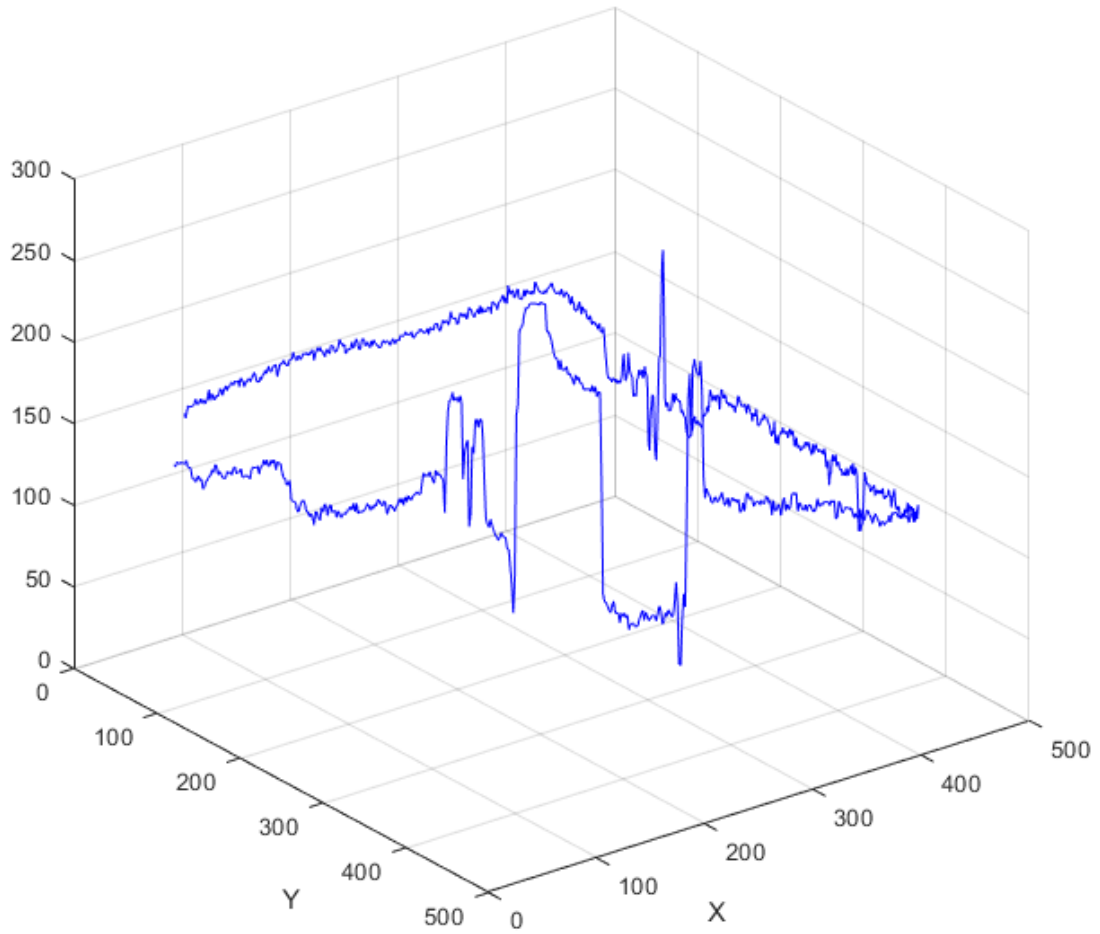


Specify  $x$ - and  $y$ -coordinates that define connected line segments.

```
x = [19 427 416 77];  
y = [96 462 37 33];
```

Display a 3-D plot of the pixel values of these line segments.

```
improfile(I,x,y),grid on;
```



## Input Arguments

### **n — Number of points**

positive integer

Number of points to include along the path, specified as a positive integer. If you do not provide this argument, `improfile` chooses a value for `n`, roughly equal to the number of pixels the path traverses.

Data Types: `double`

### **I — Input image**

RGB image | grayscale image | binary image

Input image, specified as an RGB, grayscale, or binary image.

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

### **xi, yi — Spatial coordinates of the endpoints of line segments**

numeric vector

Spatial coordinates of the endpoints of line segments in the `x` or `y` dimension, specified as a numeric vector. `xi` and `yi` have equal length.

Data Types: `double`

### **x, y — Image limits in world coordinates**

2-element numeric vector

Image limits in world coordinates along the `x`- or `y`-dimension, specified as a 2-element numeric vector of the form `[min max]`. `x` and `y` specify the image `XData` and `YData`. The data type of `x` and `y` must match.

Data Types: `single` | `double`

### **method — Interpolation method**

'nearest' (default) | 'bilinear' | 'bicubic'

Interpolation method, specified as 'nearest' for nearest-neighbor interpolation, 'bilinear', or 'bicubic'.

Data Types: `char` | `string`

## Output Arguments

### **c — Pixel intensity values**

n-by-1 numeric vector | n-by-1-by-3 numeric array

Pixel intensity values, returned as an n-by-1 numeric vector when I is a grayscale intensity image, or an n-by-1-by-3 numeric array when I is an RGB image.

Data Types: double

### **cx, cy — Spatial coordinates of sampled pixels**

n-by-1 numeric vector

Spatial coordinates of sampled pixels in the x or y dimension, specified as an n-by-1 numeric vector.

Data Types: double

## 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.

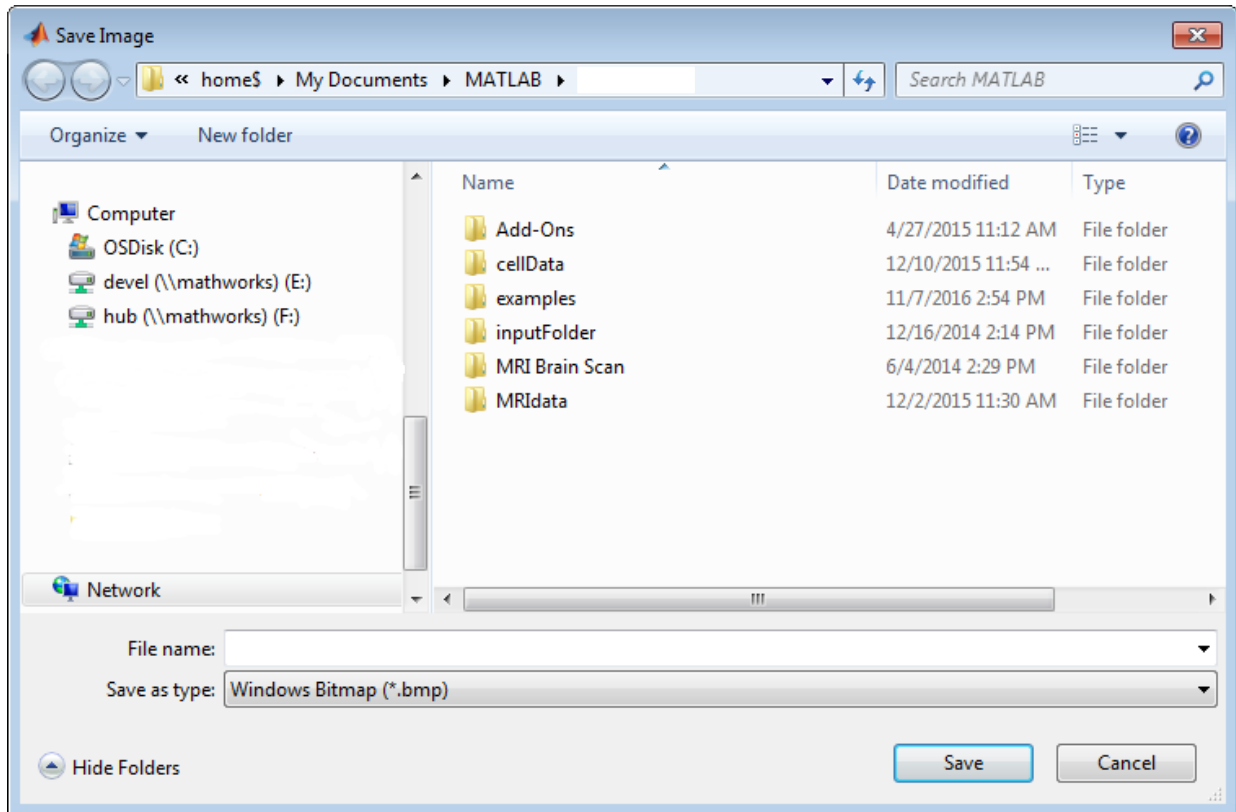
---

## Examples

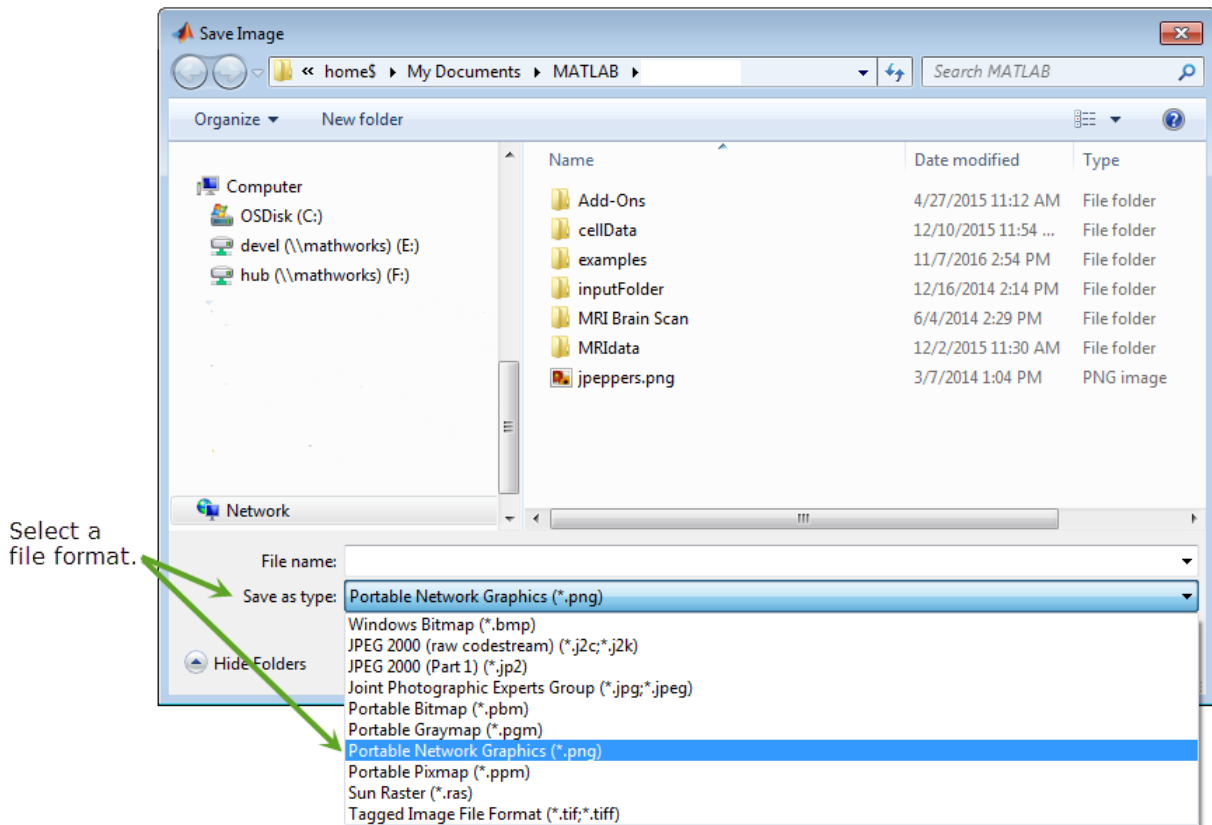
### Get User-Specified File Name

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.

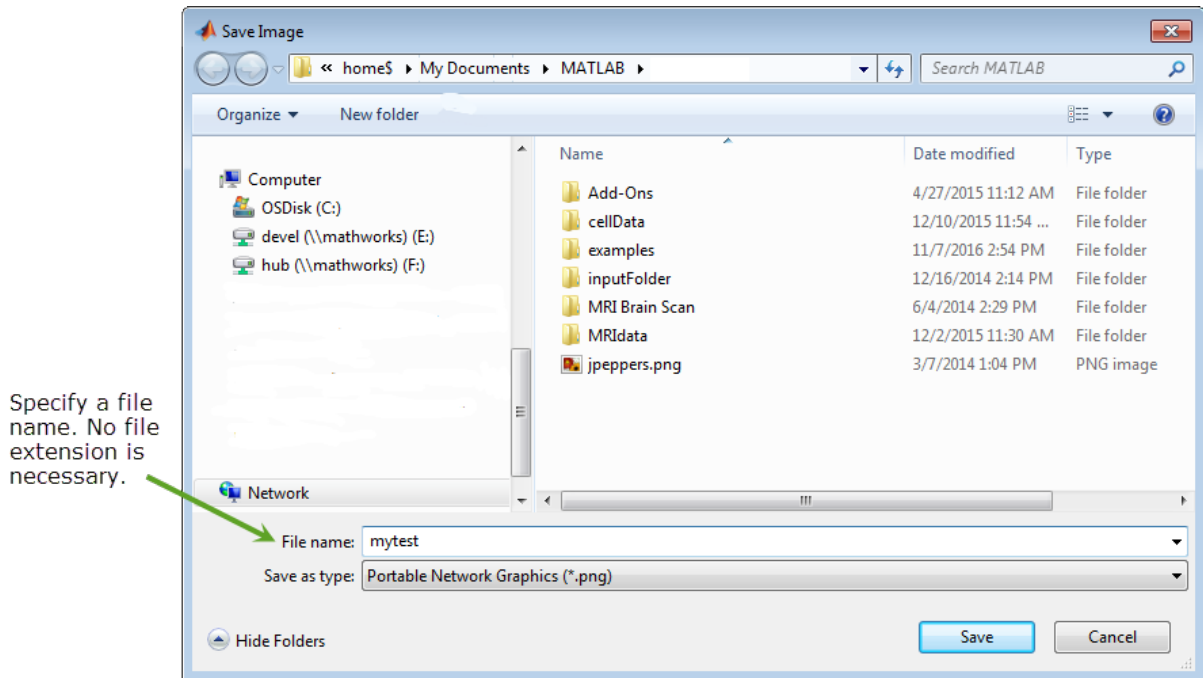
```
[fn, ext, ucancel] = imputfile
```



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**. `inputfile` 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 `inputfile` 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

`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 | string

## 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 on page 1-1619:

$w = \left[ \frac{1-a}{4}, \frac{a}{2}, \frac{1}{4}, \frac{1}{4}, \frac{a}{2}, \frac{1-a}{4} \right]$ , 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

`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  $N = \text{length}(\text{levels})$ . 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.

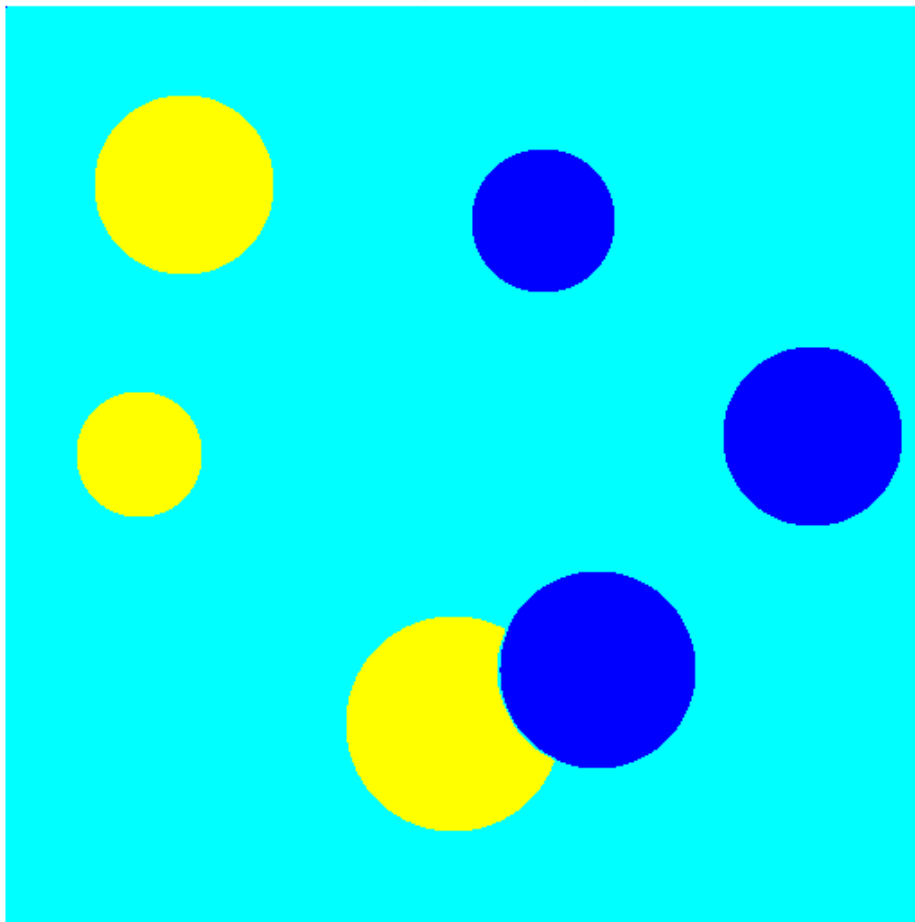
```
thresh = multithresh(I,2);
```

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);

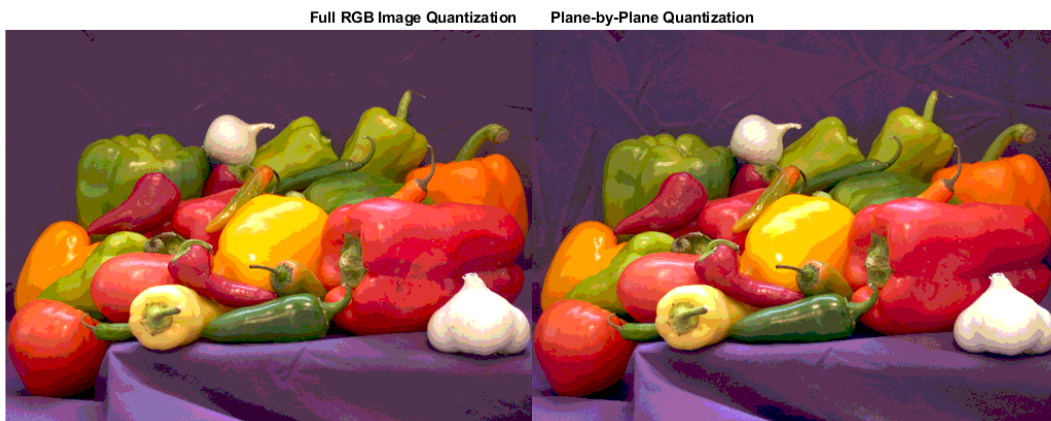
```

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 = 1x8 uint8 row vector
```

```
    65    88   119   149   169   189   215   255
```

```
[quant8_I_max, index] = imquantize(I,thresh,valuesMax);
```

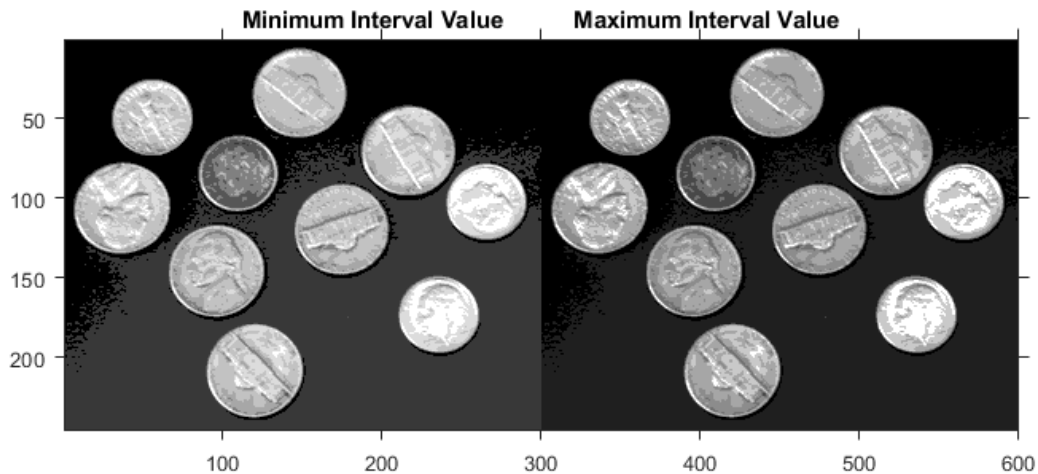
Similarly, construct the `valuesMin` vector such that the minimum value in each quantization interval is assigned to the eight levels of the output image. Instead of calling `imquantize` again with the vector `valuesMin`, use the output argument `index` to assign those values to the output image.

```
valuesMin = [min(I(:)) thresh]
valuesMin = 1x8 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

`label2rgb` | `multithresh` | `rgb2ind`

**Introduced in R2012b**

# imreconstruct

Morphological reconstruction

## Syntax

```
J = imreconstruct(marker,mask)
J = imreconstruct(marker,mask,conn)
```

## Description

`J = imreconstruct(marker,mask)` performs morphological reconstruction of the image `marker` under the image `mask`, and returns the reconstruction in `J`. The elements of `marker` must be less than or equal to the corresponding elements of `mask`. If the values in `marker` are greater than corresponding elements in `mask`, then `imreconstruct` clips the values to the `mask` level before starting the procedure.

You optionally can perform morphological reconstruction of 2-D images using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

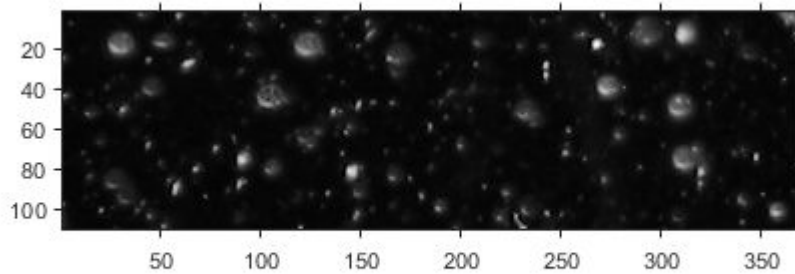
`J = imreconstruct(marker,mask,conn)` performs morphological reconstruction with the specified connectivity, `conn`.

## Examples

### Perform Opening-by-Reconstruction to Identify High Intensity Objects

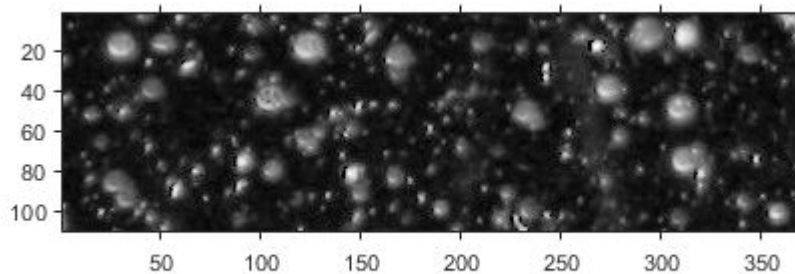
Read a grayscale image and display it.

```
I = imread('snowflakes.png');
figure
imshow(I)
```



Adjust the contrast of the image to create the mask image and display results.

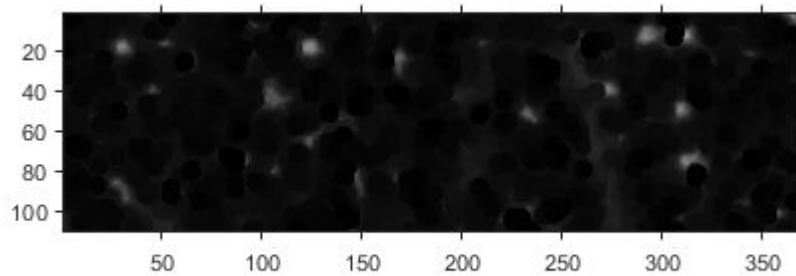
```
mask = adapthisteq(I);  
figure  
imshow(mask)
```



Create a marker image that identifies high-intensity objects in the image using morphological erosion and display results.

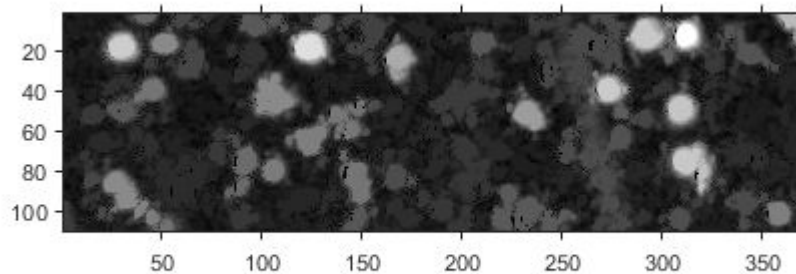
```
se = strel('disk',5);  
marker = imerode(mask,se);  
imshow(marker)
```





Perform morphological opening on the mask image, using the marker image to identify high-intensity objects in the mask. Display results.

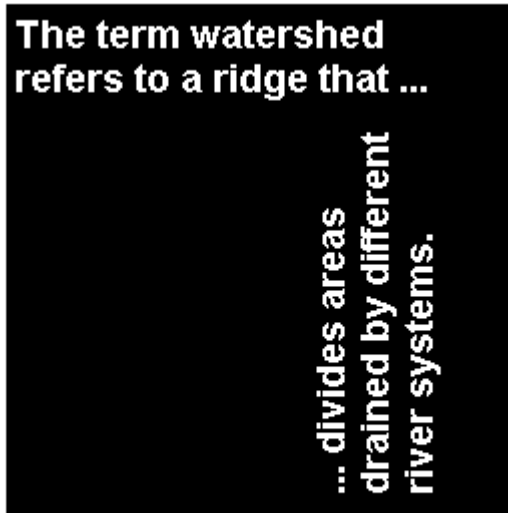
```
obr = imreconstruct(marker,mask);  
figure  
imshow(obr,[])
```



### 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)
```

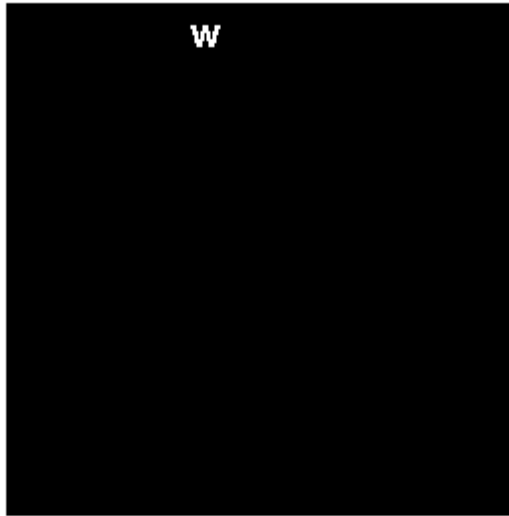


Create a marker image that identifies the object in the image you want to extract through segmentation. For this example, identify the "w" in the word "watershed".

```
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.

```
J = imreconstruct(marker,mask);  
figure, imshow(J)
```

## Input Arguments

### **marker — Input image**

numeric array | logical array | gpuArray

Input image, specified as a numeric or logical array.

To perform the morphological reconstruction using a GPU, specify `marker` as a `gpuArray` that contains a 2-D numeric or logical matrix. `imreconstruct` does not support RGB images and 3-D images on a GPU.

Example: `se = strel('disk',5); marker = imerode(mask,se);`

Example: `marker = gpuArray(imread('text.png'));`

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

### **mask — Mask image**

numeric array | logical array | gpuArray

Mask image, specified as a numeric or logical array of the same size and data type as `marker`.

To perform the morphological reconstruction using a GPU, specify `mask` as a `gpuArray` that contains a 2-D numeric or logical matrix. `imreconstruct` does not support RGB images and 3-D mask images on a GPU.

Example: `mask = imread('text.png');`

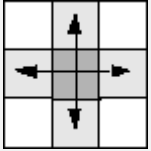
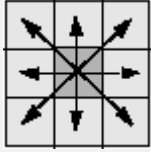
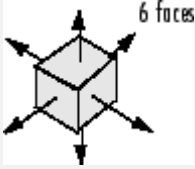
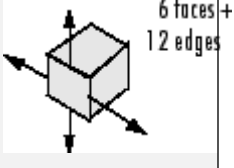
Example: `mask = gpuArray(imread('text.png'));`

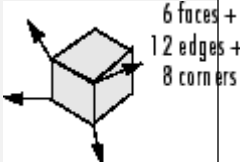
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### **conn — Pixel connectivity**

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning
<b>Two-Dimensional Connectivities</b>	
4-connected	<p>Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction.</p> 
8-connected	<p>Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.</p> 
<b>Three-Dimensional Connectivities</b>	
6-connected	<p>Pixels are connected if their faces touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul> 
18-connected	<p>Pixels are connected if their faces or edges touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul> 

Value	Meaning	
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	

For higher dimensions, `imreconstruct` uses the default value `conndef(ndims(marker), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

Data Types: `double` | `logical`

## Output Arguments

### J — Reconstructed image

`numeric array` | `logical array` | `gpuArray`

Reconstructed image, returned as a numeric or logical array, depending on the input image, that is the same size as the input image.

If the morphological reconstruction is performed using a GPU, then J is returned as a `gpuArray` that contains a numeric or logical matrix.

## 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 “Understand 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.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Inputs must be 2-D, supporting only the 2-D connectivities (4 and 8).

For more information, see “Image Processing on a GPU”.

## See Also

`imclearborder` | `imextendedmax` | `imextendedmin` | `imfill` | `imhmax` | `imhmin` | `imimposemin`

## Topics

“Understanding Morphological Reconstruction”

**Introduced before R2006a**



# imrect

Create draggable rectangle

---

**Note** `imrect` is not recommended. Use `drawrectangle` instead.

---

## Description

An `imrect` object encapsulates an interactive rectangle over an image.

You can adjust the size and position of the rectangle by using the mouse. The rectangle also has a context menu that controls aspects of its appearance and behavior. For more information, see “Usage” on page 1-1645.

## Creation

### Syntax

```
h = imrect
h = imrect(hparent)
h = imrect(hparent,position)
h = imrect( ____,Name,Value)
```

### Description

`h = imrect` begins interactive placement of a rectangle on the current axes, and returns an `imrect` object.

`h = imrect(hparent)` begins interactive placement of a rectangle on the object specified by `hparent`.

`h = imrect(hparent,position)` creates a draggable rectangle at the position `position` on the object specified by `hparent`.

`h = imrect( ____, Name, Value)` specifies name-value pairs that control the behavior of the rectangle.

## Input Arguments

### **hparent** — Handle to parent object

handle

Handle to parent object, specified as a handle. The parent is typically an axes object, but can also be any other object that can be the parent of an `hggroup` object.

### **position** — Position of rectangle

4-element vector

Position of the rectangle, specified as a 4-element vector of the form `[xmin ymin width height]`. The initial size of the rectangle is `width-by-height`, and the upper-left corner of the rectangle is at the `(x,y)` coordinate `(xmin,ymin)`.

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

### **PositionConstraintFcn** — Position constraint function

function handle

Position constraint function, specified as the comma-separated pair consisting of `'PositionConstraintFcn'` and a function handle. `fcn` 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 `setPositionConstraintFcn` function for information about valid function handles.

## Properties

### **Deletable** — ROI can be deleted

`true` (default) | `false`

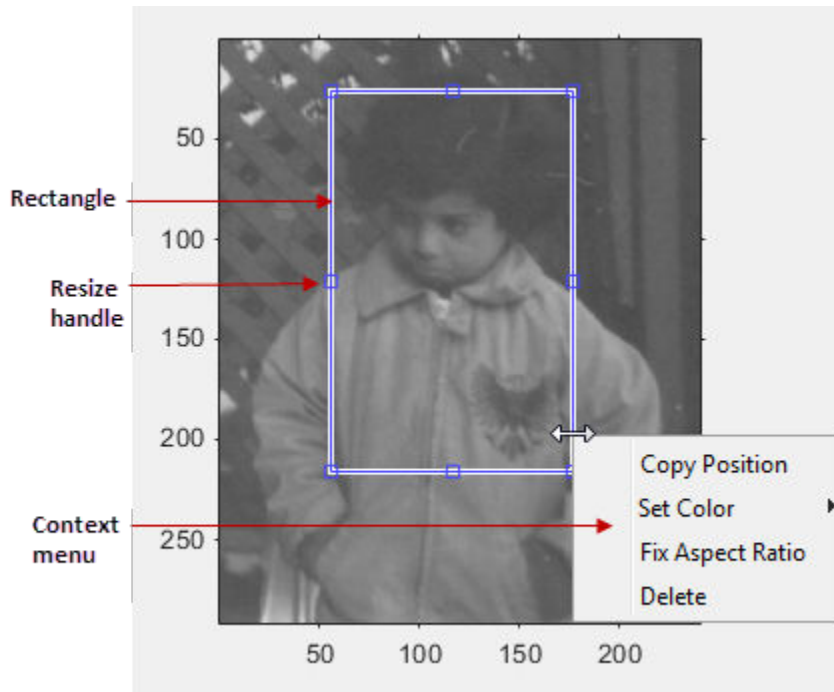
ROI can be deleted, specified as `true` or `false`.

Data Types: `logical`


## Usage

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. Right-click on the rectangle to access this context menu.



The 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, $\leftrightarrow$ . 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 <b>Set Color</b> from the context menu.
Retrieving the coordinates of the current position	Move the pointer inside the polygon. Right-click and select <b>Copy Position</b> 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 <b>Fix Aspect Ratio</b> from the context menu.
Deleting the rectangle	Move the pointer inside the rectangle or on an edge of the rectangle. Right-click and select <b>Delete</b> 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>

When you use `setResizable` to make the rectangle non-resizable, the **Fix Aspect Ratio** context menu item is not provided.

## Object Functions

Each `imrect` object supports a number of methods. Type `methods imrect` to see a complete list.

<code>addNewPositionCallback</code>	Add new-position callback to ROI object
<code>createMask</code>	Create mask within image
<code>delete</code>	Delete handle object
<code>getColor</code>	Get color used to draw ROI object
<code>getPosition</code>	Return current position of ROI object
<code>getPositionConstraintFcn</code>	Return function handle to current position constraint function
<code>removeNewPositionCallback</code>	Remove new-position callback from ROI object
<code>resume</code>	Resume execution of MATLAB command line
<code>setColor</code>	Set color used to draw ROI object
<code>setConstrainedPosition</code>	Set ROI object to new position

setFixedAspectRatioMode	Preserve aspect ratio when resizing ROI object
setPosition	Move ROI object to new position
setPositionConstraintFcn	Set position constraint function of ROI object
setResizable	Set resize behavior of ROI object
wait	Block MATLAB command line until ROI creation is finished

## Examples

### Update Title when Rectangle Moves

Display a rectangle ROI over an image. Display the position of the rectangle in the title. The title updates when you move the rectangle. Try dragging one side of the rectangle outside the boundary of the image.

```
imshow('cameraman.tif')
h = imrect(gca,[10 10 100 100]);
addNewPositionCallback(h,@(p) title(mat2str(p,3)));
```

Specify a position constraint function using `makeConstrainToRectFcn` to keep the rectangle inside the original `XLim` and `YLim` ranges of the image.

```
fcn = makeConstrainToRectFcn('imrect',get(gca,'XLim'),get(gca,'YLim'));
setPositionConstraintFcn(h,fcn);
```

Now drag the rectangle using the mouse. Observe that the rectangle can no longer extend past the image boundary.

### Click and Drag to Place Rectangle

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.

```
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.

## See Also

`imellipse` | `imfreehand` | `imline` | `impoint` | `impoly` | `imroi` | `makeConstrainToRectFcn`

**Introduced before R2006a**

# imreducehaze

Reduce atmospheric haze

## Syntax

```
[D,T,L] = imreducehaze(X)
[ ___ ] = imreducehaze(X,amount)
[ ___ ] = imreducehaze( ___,Name,Value)
```

## Description

`[D,T,L] = imreducehaze(X)` reduces atmospheric haze in `X`, which is an RGB or grayscale image. `D` is the dehazed image. `T` contains an estimate of the haze thickness at each pixel. `L` is the estimated atmospheric light, which represents the value of the brightest non-specular haze.

`[ ___ ] = imreducehaze(X,amount)` reduces atmospheric haze in image `X`, where `amount` specifies the amount of haze removed.

`[ ___ ] = imreducehaze( ___,Name,Value)` changes the behavior of the dehazing algorithm using name-value pairs.

## Examples

### Reduce Haze Using Default Parameters

Read hazy image into the workspace.

```
A = imread('foggysf1.jpg');
```

Reduce the haze and display the result along side the original image.

```
B = imreducehaze(A);
```

```
figure, imshowpair(A, B, 'montage')
```



## Reduce Haze Using `approxdcp` Contrast Stretching

Read hazy image into the workspace.

```
A = imread('foggysf2.jpg');
```

Reduce 90% of the haze using the `approxdcp` method.

```
B = imreducehaze(A, 0.9, 'method', 'approxdcp');
```

Display the result along side the original image.

```
figure, imshowpair(A, B, 'montage')
```





### Estimate Haze Thickness and Image Depth

Read hazy image into the workspace.

```
A = imread('foggyroad.jpg');
```

Reduce haze in the image using default parameter values.

```
[~, T] = imreducehaze(A);
```

Display the result along side the original image.

```
figure, imshowpair(A, T, 'montage')
```



The haze thickness provides a rough approximation of the depth of the scene, defined up to an unknown multiplication factor. Add `eps` to avoid  $\log(0)$ .

```
D = -log(1-T+eps);
```

For display purposes, scale the depth so that it is in  $[0,1]$ .

```
D = mat2gray(D);
```

Display the original image next to the estimated depth in false color.

```
figure
subplot(1,2,1)
imshow(A), title('Hazy image')
subplot(1,2,2)
imshow(D), title('Depth estimate')
colormap(gca, hot(256))
```

Hazy image



Depth estimate



## Input Arguments

### **X** — Input RGB or grayscale image

real, non-sparse,  $m$ -by- $n$ -by-3 (RGB) or  $m$ -by- $n$  (grayscale) array

Input RGB or grayscale image, specified as a real, non-sparse,  $m$ -by- $n$ -by-3 (RGB) or  $m$ -by- $n$  (grayscale) array.

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

### **amount** — How much haze to remove

1 (default) | scalar in the range  $[0, 1]$

How much haze to remove, specified as a scalar in the range  $[0, 1]$ . When amount is 1 (the default), `imreducehaze` reduces the maximum amount of haze. When the amount is 0, the input image is unchanged.

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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `B = imreducehaze(A, 0.9, 'method', 'approxdcp');`

### Method — Technique used to reduce haze

`'simplifiedcp'` (default) | `'approxdcp'`

Technique used to reduce haze, specified as `'simplifiedcp'` or `'approxdcp'`. The `'simplifiedcp'` method employs a per-pixel dark channel prior to haze estimation and quadtree decomposition to compute the atmospheric light. `'approxdcp'` uses both per-pixel and spatial blocks when computing the dark channel prior to haze reduction and does not use quadtree decomposition.

Data Types: `char` | `string`

### AtmosphericLight — Maximum value to be treated as haze

1-by-3 vector (for RGB images) | scalar (for grayscale images)

Maximum value to be treated as haze, specified a 1-by-3 vector (for RGB images) or a scalar (for grayscale images). Values must be less than or equal to 1. When not specified, this value depends on the value of `'method'`. For `'approxdcp'`, the brightest 0.1% pixels of the dark channel are considered to estimate the value. For `'simplifiedcp'`, `imreducehaze` uses quadtree decomposition to compute the value.

Data Types: `double`

### ContrastEnhancement — Post-processing technique to improve image contrast

`'global'` (default) | `'boost'` | `'none'`

Post-processing technique to improve image contrast, specified as `'global'` (default), `'boost'`, or `'none'`.

Data Types: char | string

**BoostAmount — Amount of per-pixel gain**

[0, 1] (default) | scalar in the range [0 1]

Amount of per-pixel gain to apply as post-processing, specified as a scalar in the range [0, 1]. This parameter is only allowed if 'ContrastEnhancement' is set to 'boost'.

Data Types: double

## Output Arguments

**D — Dehazed image**

numeric array

Dehazed image, returned as numeric array.

**T — Estimate of haze thickness at each pixel**

numeric array

Estimate of haze thickness at each pixel, returned as a numeric array.

**L — Estimated atmospheric light**

numeric array

Estimated atmospheric light, returned as a numeric array. L represents the brightest non-specular haze.

## Tips

- Atmospheric light values should be greater than 0.5 for better results
- Use contrast enhancement as none for applying user's customized contrast enhancement technique
- Amount value should be near to 1 for reducing more haze.
- Reduce Amount value if the result looks color distortion

## Algorithms

Function `imreducehaze` uses two different dehazing methods, `simplifiedcp` and `approxdcp`, to reduce haze in an image. These two dehazing techniques can be described in five main steps

- Atmospheric light Estimation using dark channel prior
- Estimation of transmission map
- Refine the estimated transmission map
- Restoration
- Post processing

The widely used model to describe the formation of hazy image [1] is

$$I(x) = J(x)t(x) + A(1 - t(x))$$

Where  $I$  is the observed intensity,  $J$  is the scene radiance,  $A$  is atmospheric light, and  $t$  is the transmission medium describing the portion of light that is not scattered and reaches the camera. With the estimation of the transmission map  $t$  and atmospheric light  $A$  we can recover scene radiance  $J$ .

$$J(x) = (I(x) - A) / (\max(t(x), t_0)) + A$$

The Approximate Dark Channel Prior (ApproxDCP) method [1] uses dark channel to reduce haze from a single hazy image. The dark channel is based on the key observation that most local patches in outdoor haze-free images contain some pixels whose intensity is very low in at least one color channel. Atmospheric light is estimated using the top 0.1% bright pixels of dark channel. The transmission map is then estimated using dark channel and atmospheric light. This method used a guided filter to refine the estimated transmission map. After getting atmospheric light and transmission map, scene radiance is recovered using inverse Koschmieder law. The resulting dehazed image has low contrast (dim) so contrast enhancement as post-processing step is used to improve the contrast.

The Simple Dark Channel Prior (SimpleDCP) method uses dark channel to reduce haze from the image. The dark channel can be estimated by considering minimum intensity across R, G, B channels. Atmospheric light is estimated using quadtree decomposition [2] of dark channel. Then estimation of transmission map using dark channel is done. The guided filter is used to refine the transmission map. The scene radiance is recovered using estimated atmospheric light and transmission map. Contrast enhancement

technique is applied as a post processing step to improve contrast of an image. The major difference between SimpleDCP and ApproxDCP is the estimation of dark channel and atmospheric light.

## References

- [1] He, Kaiming. "Single Image Haze Removal Using Dark Channel Prior." *Thesis, The Chinese University of Hong Kong, 2011*.
- [2] Dubok, et al. "Single Image Dehazing with Image Entropy and INformation Fidelity." *ICIP, 2014*.

## See Also

**Introduced in R2017b**

## imref2d

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 resolution in each dimension can be different.

### Creation

### Syntax

```
R = imref2d
R = imref2d(imageSize)
R = imref2d(imageSize,pixelExtentInWorldX,pixelExtentInWorldY)
R = imref2d(imageSize,xWorldLimits,yWorldLimits)
```

### Description

`R = imref2d` creates an `imref2d` object with default property settings.

`R = imref2d(imageSize)` sets the optional `ImageSize` on page 1-0 property.

`R = imref2d(imageSize,pixelExtentInWorldX,pixelExtentInWorldY)` sets the optional `ImageSize` on page 1-0 , `PixelExtentInWorldX` on page 1-0 , and `PixelExtentInWorldY` on page 1-0 properties.



`R = imref2d(imageSize,xWorldLimits,yWorldLimits)` sets the optional `ImageSize` on page 1-0 , `XWorldLimits` on page 1-0 , and `YWorldLimits` on page 1-0 properties.

## 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 sets 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 sets 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

## Object Functions

contains	Determine if image contains points in world coordinate system
intrinsicToWorld	Convert from intrinsic to world coordinates
sizesMatch	Determine if object and image are size-compatible
worldToIntrinsic	Convert from world to intrinsic coordinates
worldToSubscript	Convert world coordinates to row and column subscripts

## Examples

### Create 2-D Spatial Referencing Object Knowing Image Size and World Limits

Read a 2-D grayscale image into the workspace.

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

Create an `imref2d` object, specifying the size and world limits of the image associated with the object.

```
xWorldLimits = [2 5];
yWorldLimits = [3 6];
RA = imref2d(size(A),xWorldLimits,yWorldLimits)
```

```
RA =
```

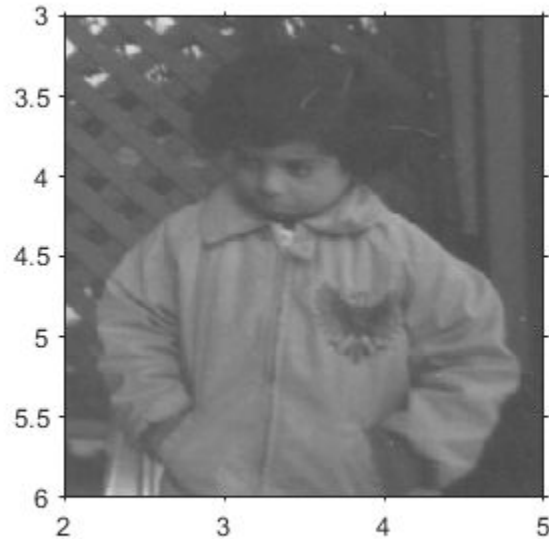
```
imref2d with properties:
```

```

    XWorldLimits: [2 5]
    YWorldLimits: [3 6]
    ImageSize: [291 240]
PixelExtentInWorldX: 0.0125
PixelExtentInWorldY: 0.0103
ImageExtentInWorldX: 3
ImageExtentInWorldY: 3
    XIntrinsicLimits: [0.5000 240.5000]
    YIntrinsicLimits: [0.5000 291.5000]
```

Display the image, specifying the spatial referencing object. The axes coordinates reflect the world coordinates.

```
figure
imshow(A,RA);
```



## Create 2-D Spatial Referencing Object Knowing Image Size and Resolution

Read a 2-D grayscale image into the workspace.

```
m = dicominfo('knee1.dcm');  
A = dicomread(m);
```

Create an `imref2d` object, specifying the size and the resolution of the pixels. The DICOM file contains a metadata field `PixelSpacing` that specifies the image resolution in each dimension in millimeters per pixel.

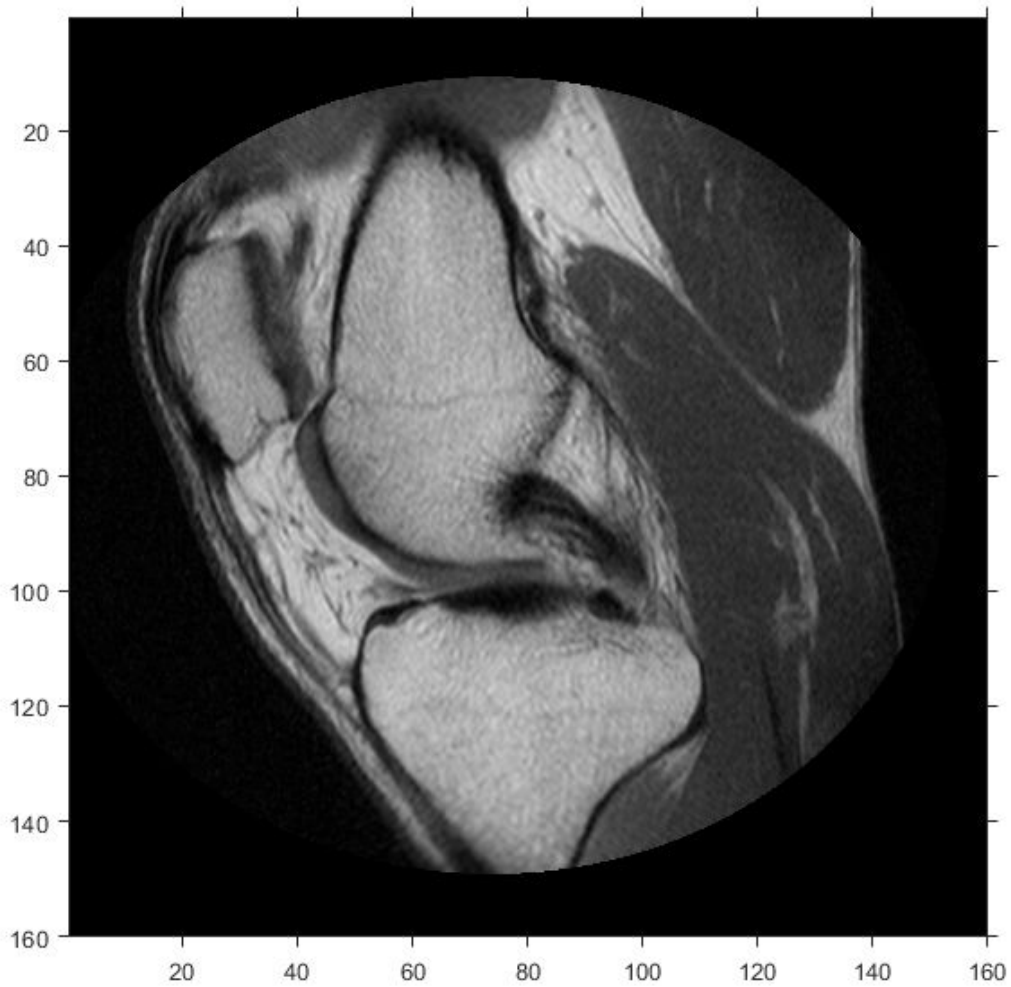
```
RA = imref2d(size(A),m.PixelSpacing(2),m.PixelSpacing(1))
```

```
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]
```

Display the image, specifying the spatial referencing object. The axes coordinates reflect the world coordinates.

```
figure
imshow(A,RA, 'DisplayRange', [0 512])
```



Compare the width of the image in world coordinates and intrinsic coordinates. This image width in intrinsic coordinates, with units of pixels, is:

```
RA.ImageSize(1)
```

```
ans = 512
```

The image width in world coordinates, with units of millimeters, is:

```
RA.ImageExtentInWorldX
```

```
ans = 160
```

## Definitions

### Intrinsic Coordinate System

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  $x = 3.0$ ,  $y = 5.0$ .

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.

### Tips

- You can create an `imref2d` object for an RGB image. If you create the object specifying the `ImageSize` on page 1-0 property as a three-element vector (such as that returned by the `size` function), only the first two elements are used to set `ImageSize`.

## 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**

`imref3d` | `imshow` | `imwarp`

### **Topics**

“Specify Fill Values in Geometric Transformation Output”

**Introduced in R2013a**



# imref3d

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*, -*y* and -*z* values align with world-*x*, -*y*, and -*z* values, respectively. The resolution in each dimension can be different.

## Creation

### Syntax

```
R = imref3d
R = imref3d(imageSize)
R =
imref3d(imageSize,pixelExtentInWorldX,pixelExtentInWorldY,pixelExtentInWorldZ)
R = imref3d(imageSize,xWorldLimits,yWorldLimits,zWorldLimits)
```

### Description

`R = imref3d` creates an `imref2d` object with default property settings.

`R = imref3d(imageSize)` sets the optional `ImageSize` on page 1-0 property.

`R = imref3d(imageSize,pixelExtentInWorldX,pixelExtentInWorldY,pixelExtentInWorldZ)` sets the optional `ImageSize` on page 1-0 , `PixelExtentInWorldX`

on page 1-0 , PixelExtentInWorldY on page 1-0 , and PixelExtentInWorldZ on page 1-0 properties.

`R = imref3d(imageSize,xWorldLimits,yWorldLimits,zWorldLimits)` sets the optional `ImageSize` on page 1-0 , `XWorldLimits` on page 1-0 , `YWorldLimits` on page 1-0 , and `ZWorldLimits` on page 1-0 properties.

## 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

## **Object Functions**

contains	Determine if image contains points in world coordinate system
intrinsicToWorld	Convert from intrinsic to world coordinates
sizesMatch	Determine if object and image are size-compatible
worldToIntrinsic	Convert from world to intrinsic coordinates
worldToSubscript	Convert world coordinates to row and column subscripts

## **Examples**

## Create imref3d Object Knowing 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

## Definitions

### Intrinsic Coordinate System

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  $x = 3.0$ ,  $y = 5.0$ ,  $z = 4.0$ .

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.

## 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

[imref2d](#) | [imregister](#)

**Introduced in R2013a**

# imregionalmax

Regional maxima

## Syntax

```
BW = imregionalmax(I)
BW = imregionalmax(I,conn)
```

## Description

`BW = imregionalmax(I)` returns the binary image `BW` that identifies the regional maxima in grayscale image `I`. Regional minima are connected components of pixels with a constant intensity value, surrounded by pixels with a lower value.

You optionally can identify regional maxima of 2-D images using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`BW = imregionalmax(I,conn)` specifies the pixel connectivity, `conn`.

## 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 = 10x10
```

```
    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 = 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  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** — Grayscale image

numeric array | gpuArray

Input array, specified as a numeric array of any dimension.

To calculate regional maxima using a GPU, specify `I` as a `gpuArray` that contains a 2-D numeric matrix.

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

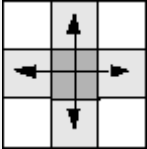
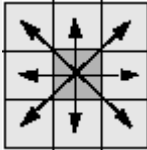
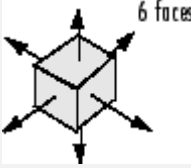
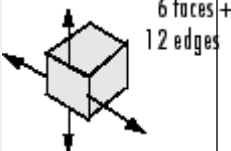
```
Example: I = gpuArray(imread('cameraman.tif'));
```

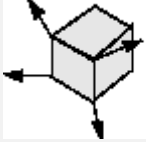
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

**conn — Pixel connectivity**

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	<p>Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction.</p>	
8-connected	<p>Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.</p>	
<b>Three-Dimensional Connectivities</b>		
6-connected	<p>Pixels are connected if their faces touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	
18-connected	<p>Pixels are connected if their faces or edges touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	

Value	Meaning	
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	 <p>6 faces + 12 edges + 8 corners</p>

For higher dimensions, `imregionalmax` uses the default value `conndef(ndims(I), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

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

## Output Arguments

### **BW** — Locations of regional maxima

logical array | `gpuArray`

Locations of regional maxima, returned as a logical array of the same size as `I`. Pixels with the value 1 indicate regional maxima; all other pixels are set to 0.

If the regional maxima are computed using a GPU, then `BW` is returned as a `gpuArray` that contains a logical matrix of the same size as `I`.

Data Types: `logical`

## 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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, the optional second input argument, `conn`, must be a compile-time constant.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Inputs must be 2-D, supporting only the 2-D connectivities (4 and 8).

For more information, see “Image Processing on a GPU”.

### See Also

`conndef` | `imextendedmax` | `imhmax` | `imreconstruct` | `imregionalmin`

**Introduced before R2006a**

# imregionalmin

Regional minima

## Syntax

```
BW = imregionalmin(I)
BW = imregionalmin(I,conn)
```

## Description

`BW = imregionalmin(I)` returns the binary image `BW` that identifies the regional minima in grayscale image `I`. Regional minima are connected components of pixels with a constant intensity value, surrounded by pixels with a lower value.

You optionally can identify regional minima of 2-D images using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`BW = imregionalmin(I,conn)` specifies the desired connectivity, `conn`.

## 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 = 10x10
```

```

10  10  10  10  10  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   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

```

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 surround
A(6:8,6:8) = 7;

```

```
A =
```

```

10  10  10  10  10  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   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** — Grayscale image

numeric array | gpuArray

Input array, specified as a numeric array of any dimension.

To calculate regional minima using a GPU, specify `I` as a `gpuArray` that contains a 2-D numeric matrix.

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

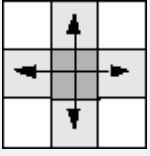
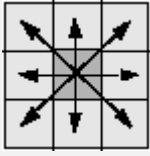
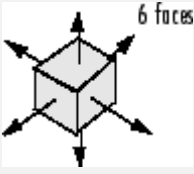
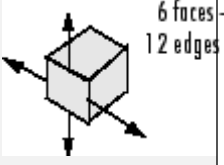
```
Example: I = gpuArray(imread('cameraman.tif'));
```

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

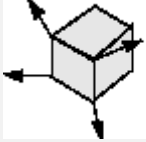
**conn — Pixel connectivity**

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	
18-connected	Pixels are connected if their faces or edges touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	



Value	Meaning	
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	 <p>6 faces + 12 edges + 8 corners</p>

For higher dimensions, `imregionalmin` uses the default value `conndef(ndims(I), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

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

## Output Arguments

### **BW** — Locations of regional minima

logical array | `gpuArray`

Locations of regional minima, returned as a logical array of the same size as `I`. Pixels with the value 1 indicate regional maxima; all other pixels are set to 0.

If the regional minima are computed using a GPU, then `BW` is returned as a `gpuArray` that contains a logical matrix of the same size as `I`.

Data Types: `logical`

## 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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, the optional second input argument, `conn`, must be a compile-time constant.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Inputs must be 2-D, supporting only the 2-D connectivities (4 and 8).

For more information, see “Image Processing on a GPU”.

### 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, where `modality` specifies the image capture modality. `imregconfig` returns `optimizer` and `metric` with default settings to provide a basic registration configuration.

## Examples

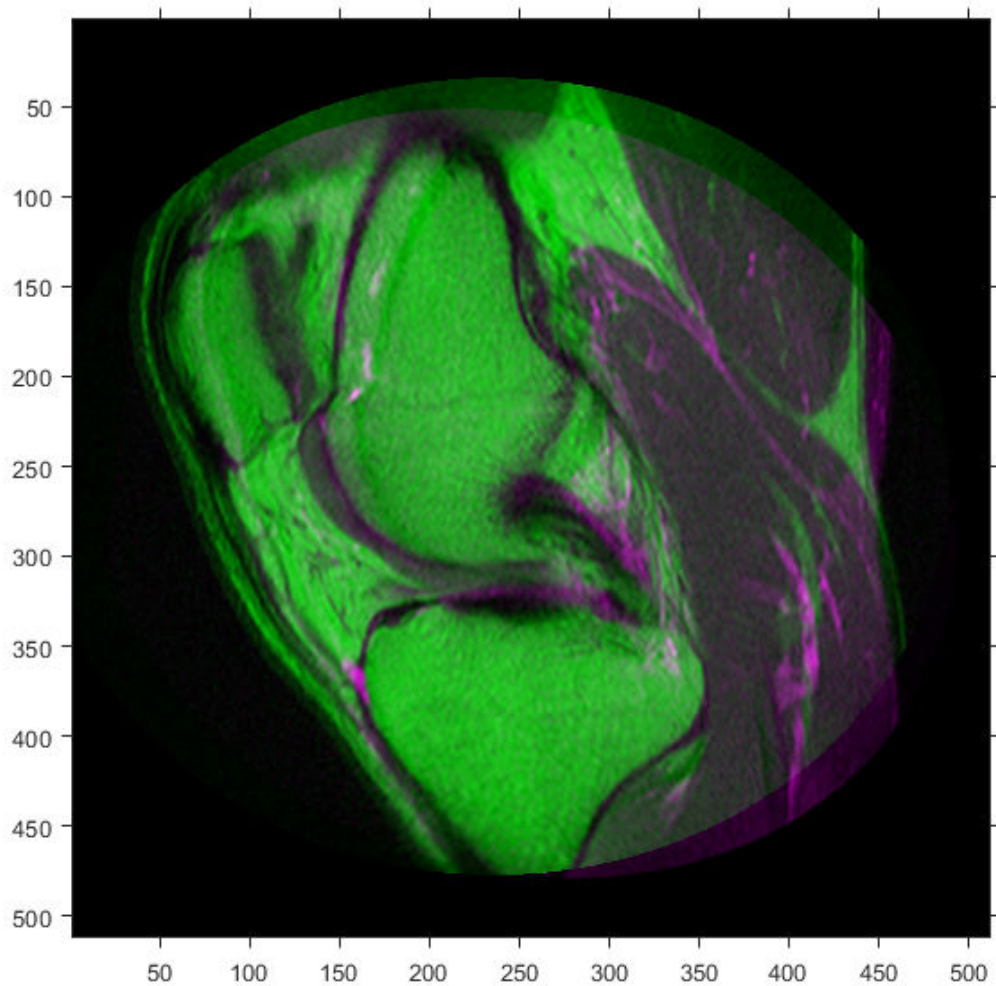
### Register Multimodal MRI Images with Optimizer

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.

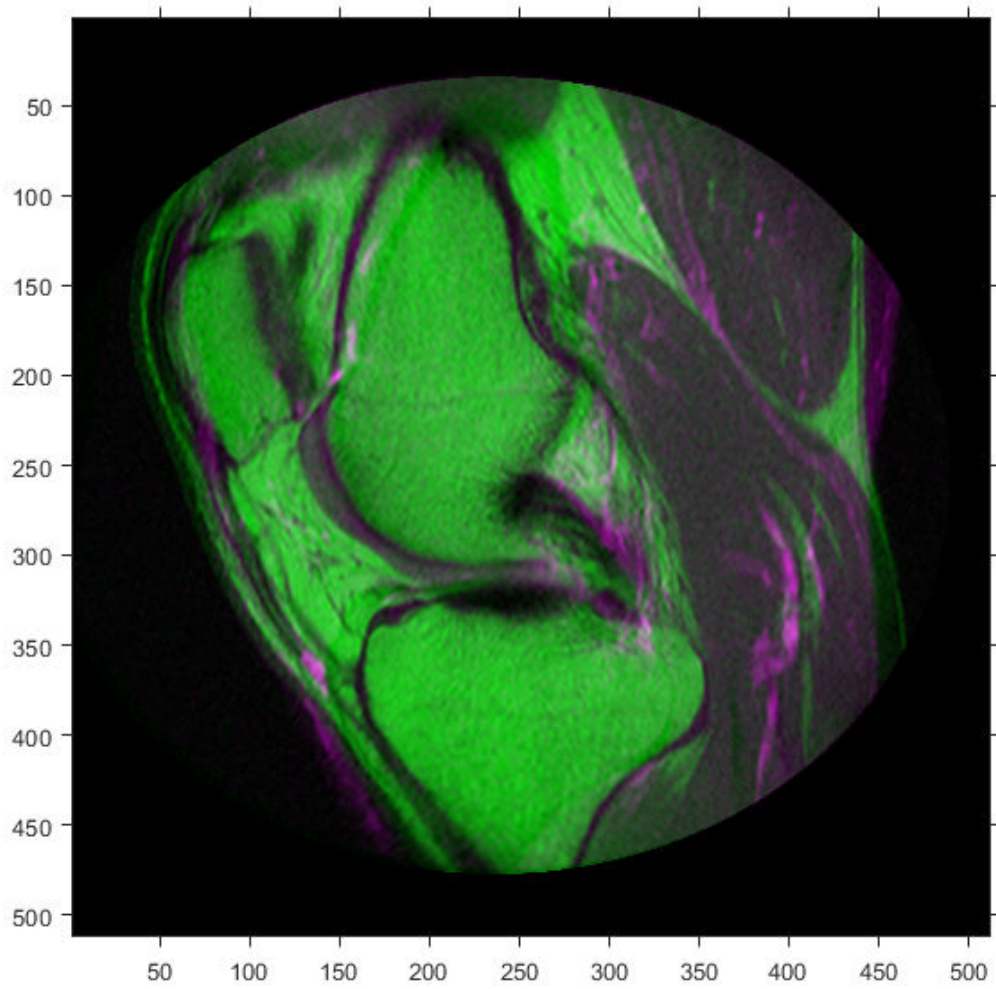
```
optimizer.InitialRadius = 0.009;
optimizer.Epsilon = 1.5e-4;
optimizer.GrowthFactor = 1.01;
optimizer.MaximumIterations = 300;
```

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' on page 1-1689 (with similar brightness and contrast) or 'multimodal' on page 1-1689 (with different brightness or contrast).

Data Types: char | string

## Output Arguments

### **optimizer** — Optimization configuration

RegularStepGradientDescent or OnePlusOneEvolutionary optimizer object

Optimization configuration, returned as a RegularStepGradientDescent or OnePlusOneEvolutionary optimizer object.

### **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.

## 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

#### Apps

**Registration Estimator**

#### Functions

`imregister` | `imshowpair`

#### Objects

`MattesMutualInformation` | `MeanSquares` | `OnePlusOneEvolutionary` | `RegularStepGradientDescent`

### 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 string scalar or 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.

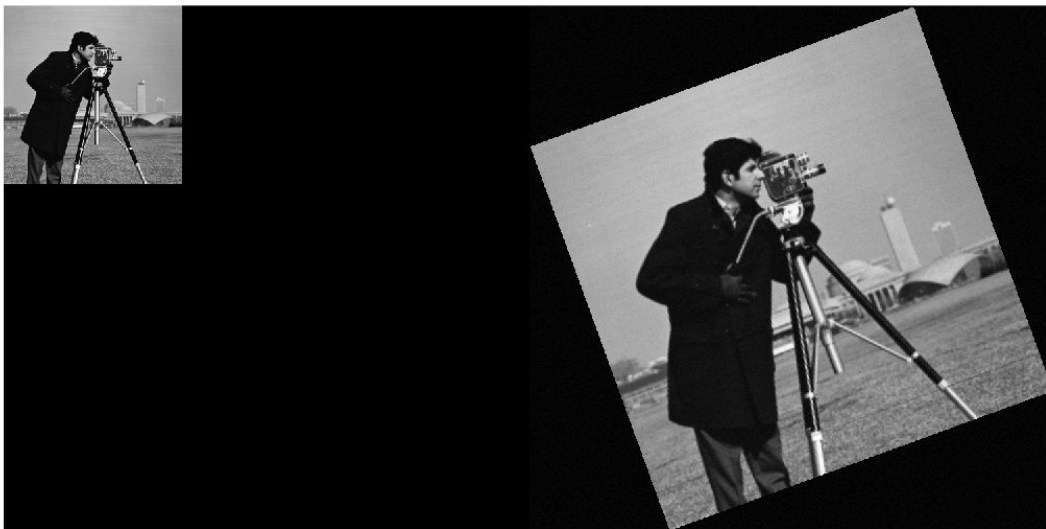
```
fixed = imread('cameraman.tif');
```

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.

```
imshowpair(fixed,movingReg,'montage')
```



```
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` | `string`

**moving** — Spatial referencing information associated with the image to be registered

imref2d object

Spatial referencing information associated with the image to be registered, specified as an `imref2d` object.

## **Rfixed** — Spatial referencing information associated with the reference (fixed) image

imref2d object

Spatial referencing information associated with the reference (fixed) image, specified as an imref2d object.

## **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 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

### 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)
[D,moving_reg] = imregdemons(moving, fixed, N)
[D,moving_reg] = 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`. 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.

You optionally can estimate the displacement field using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`[D,moving_reg] = 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.

`[D,moving_reg] = 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.

```
imshowpair(fixed,moving,'montage')
```



Overlay the two images to make it easy to see where the images differ. The differences are highlighted in green and magenta.

```
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.

```
imshowpair(fixed, movingReg)
```



```
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 | gpuArray

Image to be registered, specified as a 2-D or 3-D grayscale image.

To perform the computation using a GPU, specify `moving` as a `gpuArray` that contains 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 | gpuArray

Reference image in the target orientation, specified as a 2-D or 3-D grayscale image.

To perform the computation using a GPU, specify `fixed` as a `gpuArray` that contains 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, then 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`

## 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 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 | `gpuArray`

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- $n$ , then 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- $p$ , then 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.

If the displacement field is computed using a GPU, then `D` is returned as a `gpuArray` containing a numeric array.

Data Types: `double`

### **moving\_reg** — Aligned image

2-D or 3-D grayscale image | `gpuArray`

Registered image, returned as a 2-D or 3-D grayscale image, warped according to the displacement field `D` and resampled using linear interpolation.

If the displacement field is computed using a GPU, then `moving_reg` is returned as a `gpuArray` containing a 2-D or 3-D grayscale image.

## Tips

- To transform an image using the displacement field `D`, use `imwarp`.

## References

- [1] Thirion, J.-P. "Image matching as a diffusion process: an analogy with Maxwell's demons". *Medical Image Analysis*. Vol. 2, Number 3, 1998, pp. 243-260.
- [2] Vercauteren, T, X. Pennec, A. Perchant, N. Ayache, "Diffeomorphic Demons: Efficient Non-parametric Image Registration", *NeuroImage*. Vol. 45, Number 1, Supplement 1, March 2009, pp. 61-72.

## Extended Capabilities

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The parameter 'DisplayWaitbar' is not supported on the GPU.

For more information, see "Image Processing on a GPU".

## 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 string scalar or 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

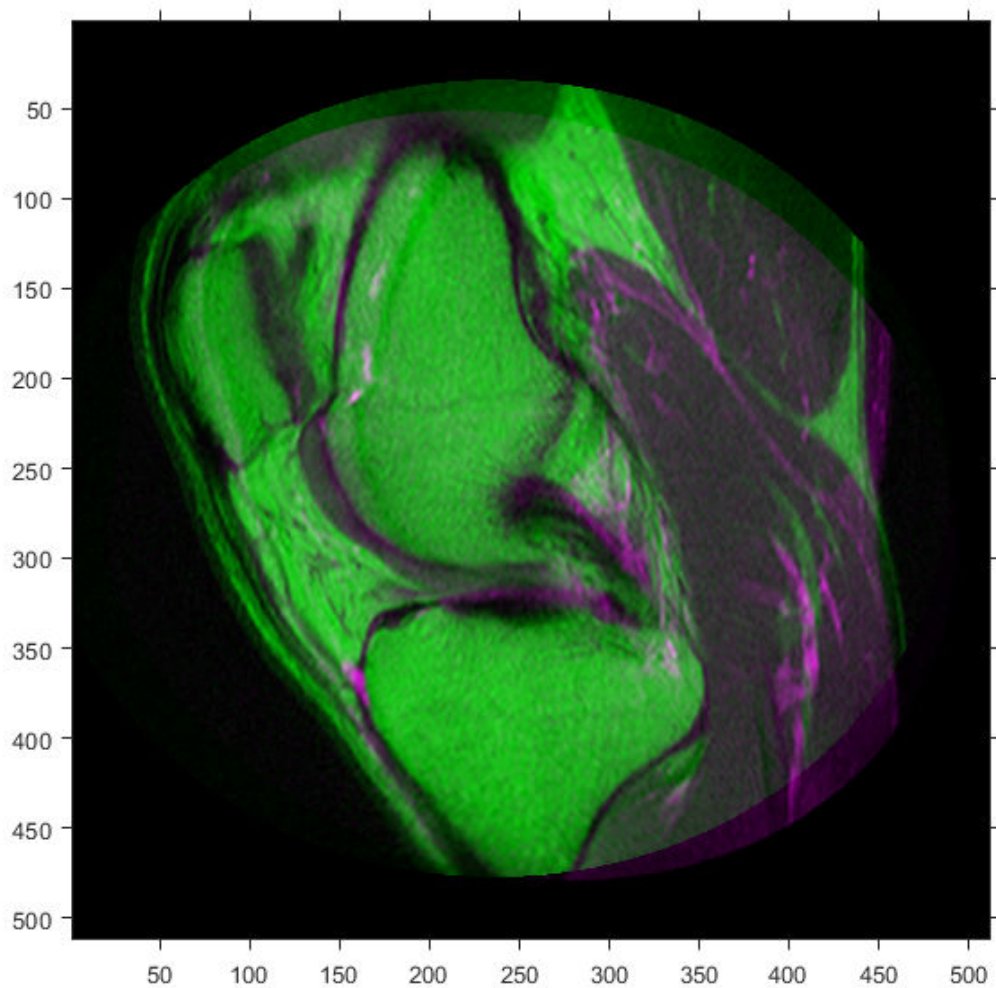
Read two images. This example uses two magnetic resonance (MRI) images of a knee. The fixed image is a spin echo image, while the moving image is a spin echo image with

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.

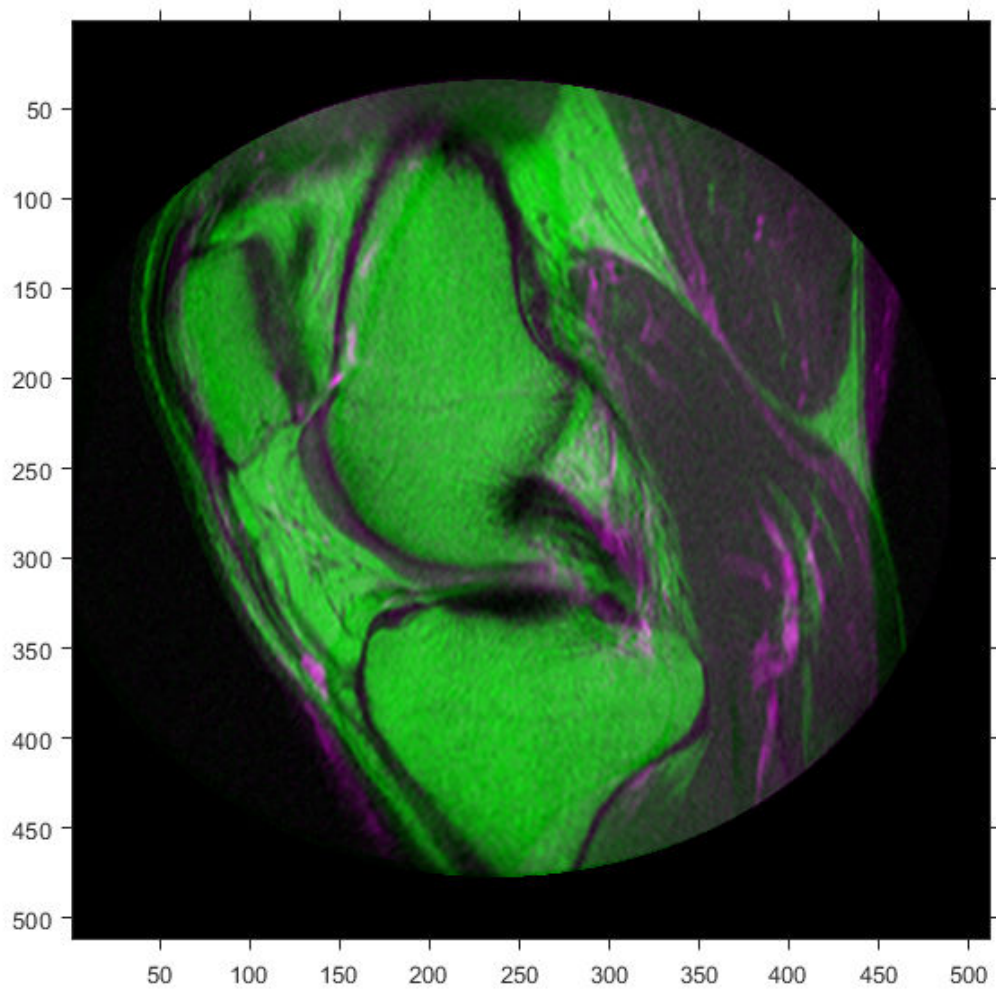
```
optimizer.InitialRadius = 0.009;
optimizer.Epsilon = 1.5e-4;
optimizer.GrowthFactor = 1.01;
optimizer.MaximumIterations = 300;
```

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

`imref2d` or `imref3d` object

Spatial referencing information associated with the image to be registered, specified as an `imref2d` or `imref3d` object.

### **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

`imref2d` or `imref3d` object

Spatial referencing information associated with the reference (fixed) image, specified as an `imref2d` or `imref3d` object.

### **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
<code>'translation'</code>	$(x,y)$ translation in 2-D, or $(x,y,z)$ translation in 3-D.
<code>'rigid'</code>	Rigid transformation consisting of translation and rotation.
<code>'similarity'</code>	Nonreflective similarity transformation consisting of translation, rotation, and scale.

Value	Description
'affine'	Affine transformation consisting of translation, rotation, scale, and shear.

The 'similarity' and 'affine' transformation types always involve nonreflective transformations.

Data Types: `char` | `string`

### **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.

### **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.

## **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 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` object

Starting geometric transformation, specified as the comma-separated pair consisting of 'InitialTransformation' and an affine2d or affine3d 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**

imref2d or imref3d object

Spatial referencing information associated with output image, returned as an imref2d or imref3d object.

## **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

### 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**

## imregmtb

Register 2-D images using median threshold bitmaps

### Syntax

```
[R1,R2,...,Rn,shift] = imregmtb(M1,M2,...,Mn,F)
```

### Description

`[R1,R2,...,Rn,shift] = imregmtb(M1,M2,...,Mn,F)` registers an arbitrary number of moving images `M1,M2,...,Mn` with respect to the fixed (reference) image, `F`, using the median threshold bitmap technique. The registered images are returned in `R1,R2,...,Rn`, and the estimated displacement of the registered images is returned in `shift`.

The median threshold bitmap technique is effective for registering images captured with variable exposures. `imregmtb` considers only translations, not rotations or other types of geometric transformations.

### Examples

#### Register Images with Jitter Using Median Threshold Bitmaps

Read a series of images with different exposures.

```
I1 = imread('office_1.jpg');  
I2 = imread('office_2.jpg');  
I3 = imread('office_3.jpg');  
I4 = imread('office_4.jpg');  
I5 = imread('office_5.jpg');  
I6 = imread('office_6.jpg');
```

The images were captured from a fixed camera, and there are no moving objects in the scene. For this example, simulate camera motion, or jitter, by translating each image

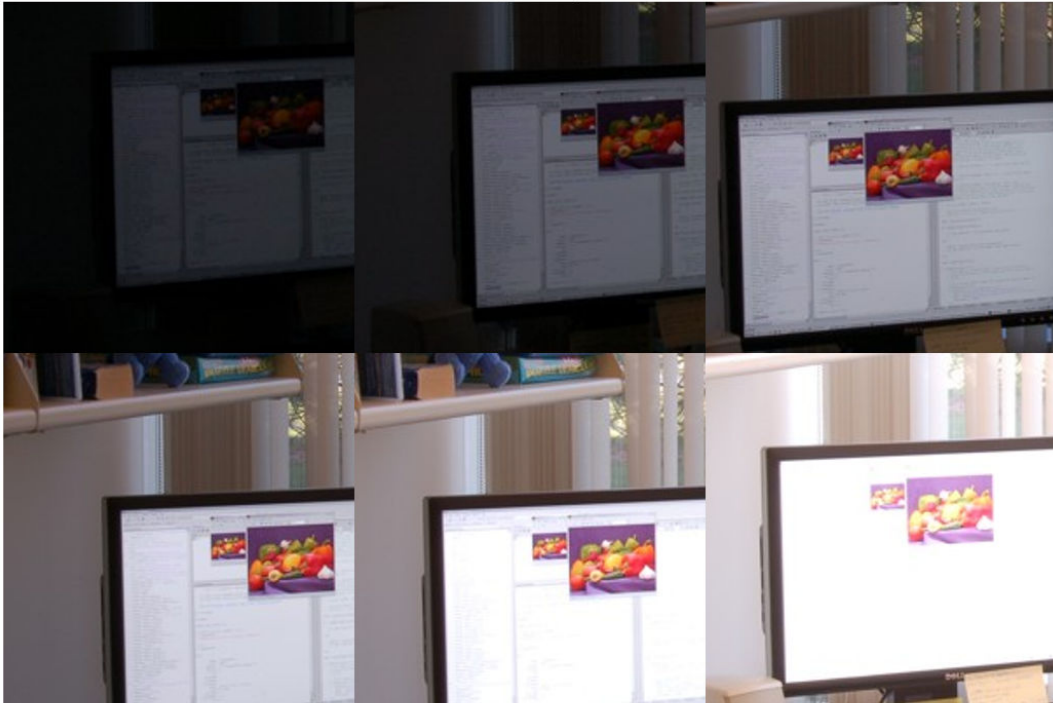
horizontally and vertically by a random amount in the range [-30, 30] pixels. Store the translation values for all five moving images in the 5-by-2 matrix `t`. Designate the sixth image, `I6`, as the fixed (or reference) image. Do not apply jitter to this image.

```
t = randi([-30 30],5,2);
I1 = imtranslate(I1,t(1,:));
I2 = imtranslate(I2,t(2,:));
I3 = imtranslate(I3,t(3,:));
I4 = imtranslate(I4,t(4,:));
I5 = imtranslate(I5,t(5,:));
```

To compare the image positions, display a region of interest (ROI) from the center of each image. The vector `roi` specifies the x- and y-coordinate of the top left corner, and the width and height of the ROI.

```
roi = [140 260 200 200];
montage({imcrop(I1,roi),imcrop(I2,roi),imcrop(I3,roi), ...
        imcrop(I4,roi),imcrop(I5,roi),imcrop(I6,roi)})
title('Misaligned Images')
```

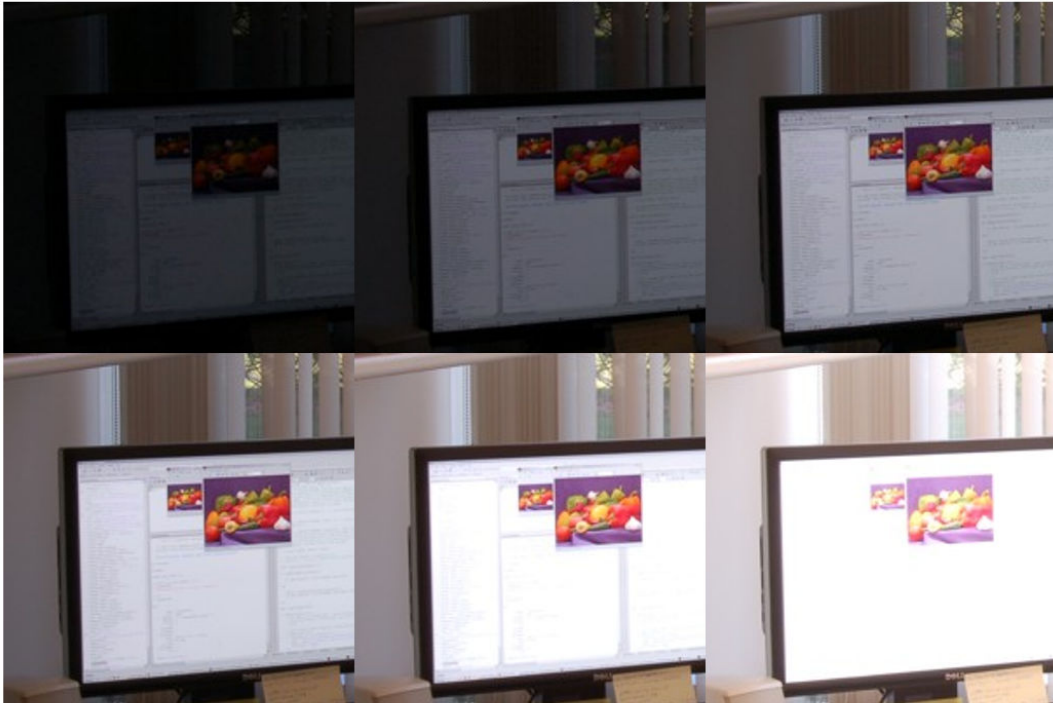
Misaligned Images



Register the spatially shifted images using median threshold bitmaps. Display an ROI from the center of each image.

```
[R1,R2,R3,R4,R5,shift] = imregmtb(I1,I2,I3,I4,I5,I6);  
montage({imcrop(R1,roi),imcrop(R2,roi),imcrop(R3,roi), ...  
        imcrop(R4,roi),imcrop(R5,roi),imcrop(I6,roi)})  
title('Registered Images')
```

Registered Images



The images look well-aligned.

Examine the estimated displacement, `shift`, of each moving image with respect to the fixed image. `shift` represents the estimated transformation that must be applied to the moving image to align it with the fixed image.

`shift`

`shift = 5x2`

-26	25
-25	14
23	-3
-25	-28
-8	-28

Compare the estimated displacement to the actual displacement. Recall that the transformation `t` was applied to the fixed image to simulate the jitter of each moving image. Therefore, the transformation `-t` is analogous to the transformation returned by `shift`.

`-t`

`ans = 5×2`

```
-19    25
-25    14
 23    -3
-25   -28
  -8   -28
```

The `imregmtb` function does a good job estimating the displacement of each frame.

## Input Arguments

### **M1, M2, . . . , Mn — Moving images**

grayscale images | RGB images

Moving images, specified as a series of grayscale images or RGB images with identical or variable exposures. The images must have the same size and data type.

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

### **F — Fixed image**

grayscale image | RGB image

Fixed image, specified as a grayscale image or RGB image. `F` must have the same size and data type as the moving images, `M1, M2, . . . , Mn`.

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

## Output Arguments

### **R1, R2, . . . , Rn — Registered images**

grayscale images | RGB images

Registered images, returned as a series of grayscale images or RGB images. The registered images have the same size and data type as the moving images,  $M_1, M_2, \dots, M_n$ .

**shift — Estimated displacement**

*n*-by-2 numeric matrix

Estimated displacement in the horizontal and vertical direction of the *n* registered images, returned as an *n*-by-2 numeric matrix.

**References**

- [1] Reinhard, E., W. Heidrich, P. Debevec, S. Pattanaik, G. Ward, K. Myszkowski. *High Dynamic Range Imaging, Second Edition*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2010, pp. 155–170.

**See Also**

`blendexposure` | `imregcorr` | `imregister` | `imtranslate`

**Introduced in R2018a**

## 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 string scalar or 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

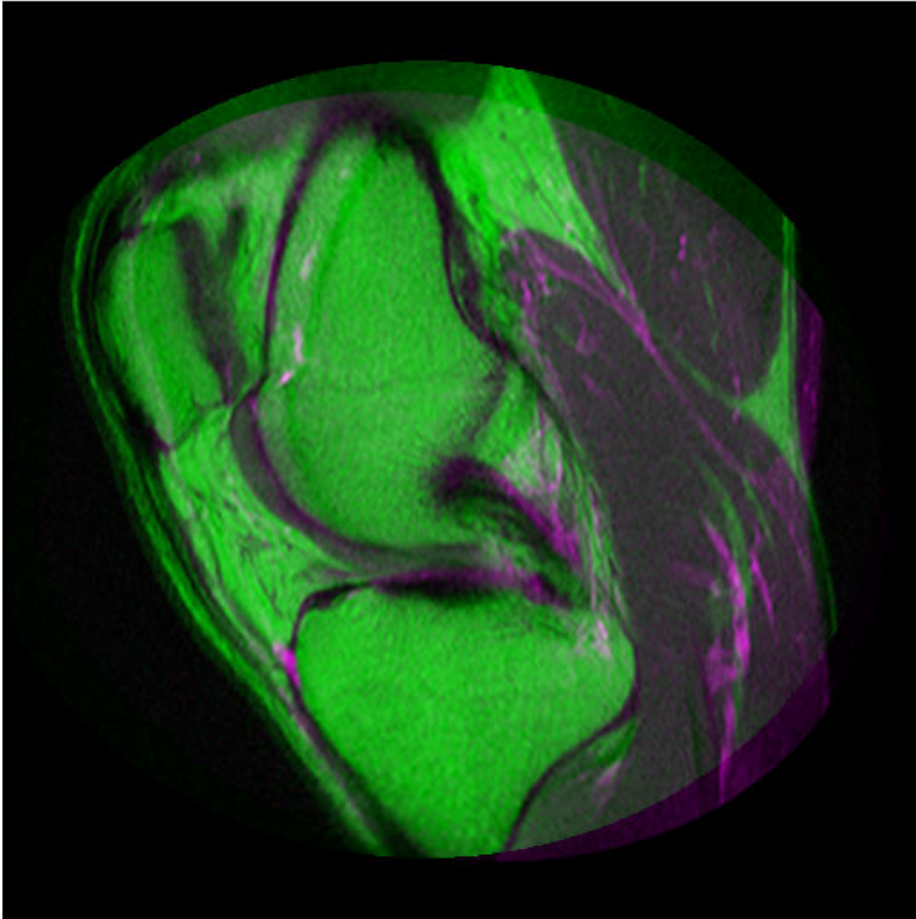


inversion recovery. The two sagittal slices were acquired at the same time but are slightly out of alignment.

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

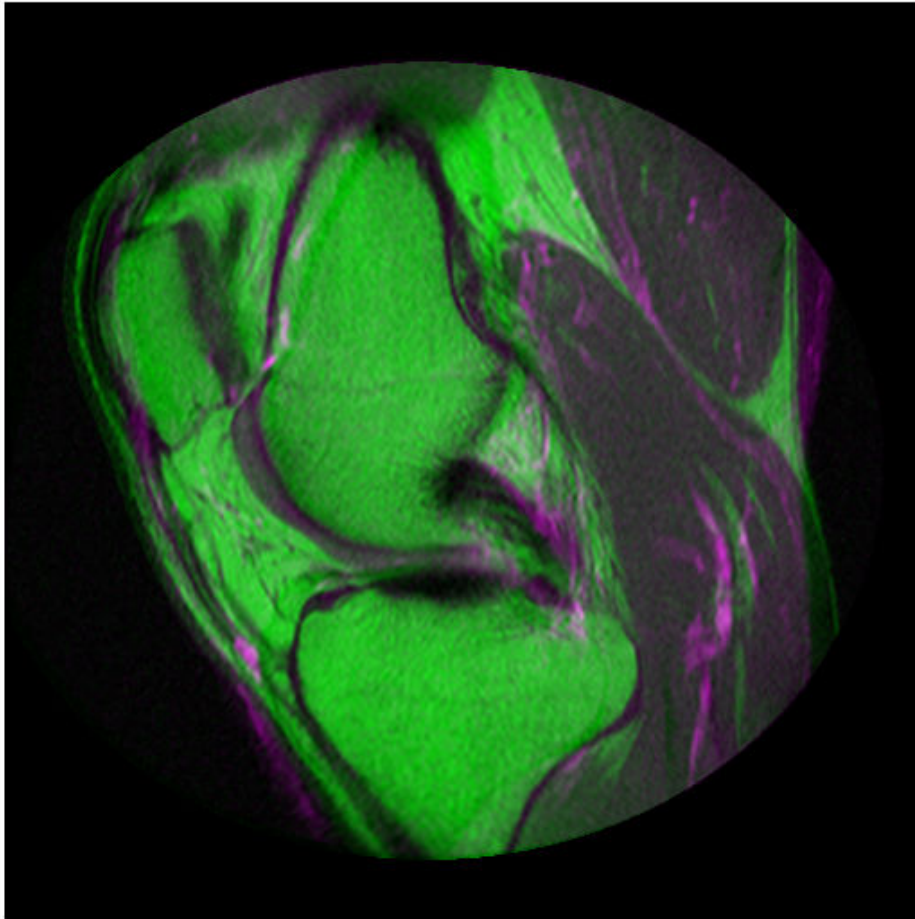
```

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

`imref2d` or `imref3d` object

Spatial referencing information associated with the image to be registered, specified as an `imref2d` or `imref3d` object.

### **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

`imref2d` or `imref3d` object

Spatial referencing information associated with the reference (fixed) image, specified as an `imref2d` or `imref3d` object.

### **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.

Value	Description
'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.

Data Types: `char` | `string`

### **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.

### **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.

## **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 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**

affine2d or affine3d object

Starting geometric transformation, specified as the comma-separated pair consisting of 'InitialTransformation' and an affine2d or affine3d 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**

affine2d or affine3d object

Geometric transformation, returned as an affine2d or affine3d object. 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

### Apps

**Registration Estimator**

### Functions

`imregconfig` | `imregister` | `imshowpair` | `imwarp`

### Objects

`affine2d` | `affine3d`

## Topics

“Create an Optimizer and Metric for Intensity-Based Image Registration”

**Introduced in R2013a**



# imresize

Resize image

## Syntax

```
J = imresize(I, scale)
J = imresize(I, [numrows numcols])
[Y, newmap] = imresize(X, map, ___)
___ = imresize(___, method)
___ = imresize(___, Name, Value)
```

## Description

`J = imresize(I, scale)` returns image `J` that is `scale` times the size of grayscale, RGB, or binary image `I`. If `I` has more than two dimensions, then `imresize` only resizes the first two dimensions. By default, `imresize` uses bicubic interpolation and performs antialiasing.

You optionally can resize images using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`J = imresize(I, [numrows numcols])` returns image `J` that has the number of rows and columns specified by the vector `[numrows numcols]`.

`[Y, newmap] = imresize(X, map, ___)` resizes the indexed image `X` with colormap `map`. By default, `imresize` returns an optimized colormap, `newmap`, with the resized image. To return a colormap that is the same as the original colormap, use the `Colormap` name-value pair argument.

This syntax is not supported on a GPU.

`___ = imresize(___, method)` specifies the interpolation method used.

Only the 'bicubic' and 'cubic' interpolation methods are supported on a GPU.

`___ = imresize(___, Name, Value)` uses name-value pair arguments to control various aspects of the resizing operation.

This syntax is not supported on a GPU.

## Examples

### Resize Image Specifying Scale Factor

Read image into the workspace.

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

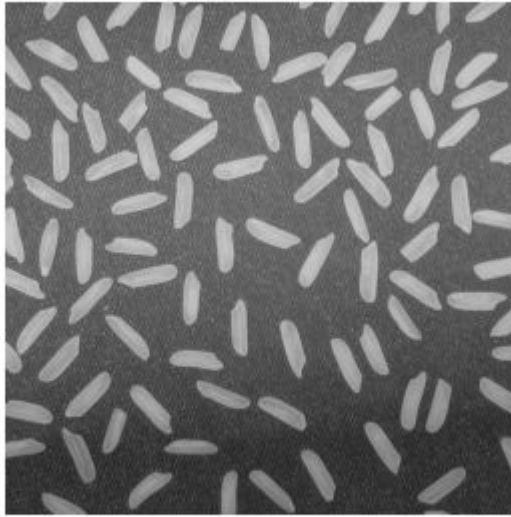
Resize the image, specifying scale factor and using default interpolation method and antialiasing.

```
J = imresize(I, 0.5);
```

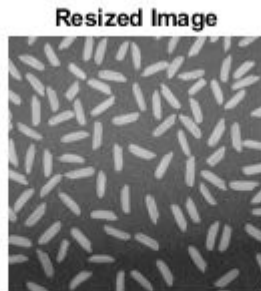
Display the original and the resized image.

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

Original Image



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



## 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.

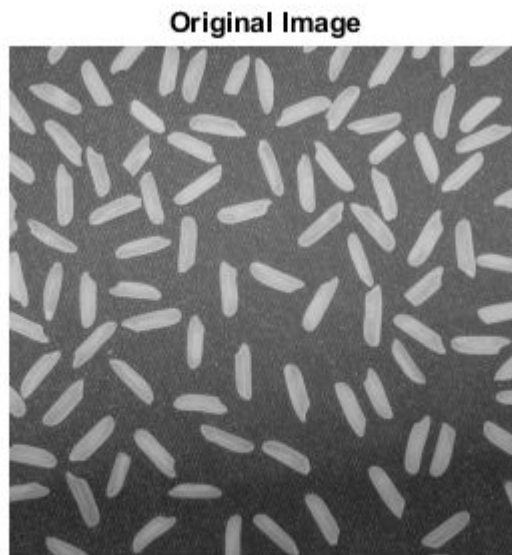
```
I = imread('rice.png');
```

Resize the image, specifying scale factor and the interpolation method.

```
J = imresize(I, 0.5, 'nearest');
```

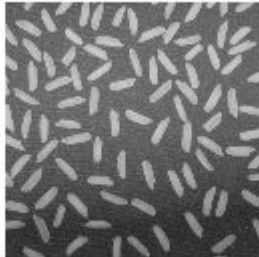
Display the original and the resized image.

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



```
figure  
imshow(J)  
title('Resized Image Using Nearest-Neighbor')
```

**Resized Image Using Nearest-Neighbor**



## **Resize Indexed Image**

Read image into the workspace.

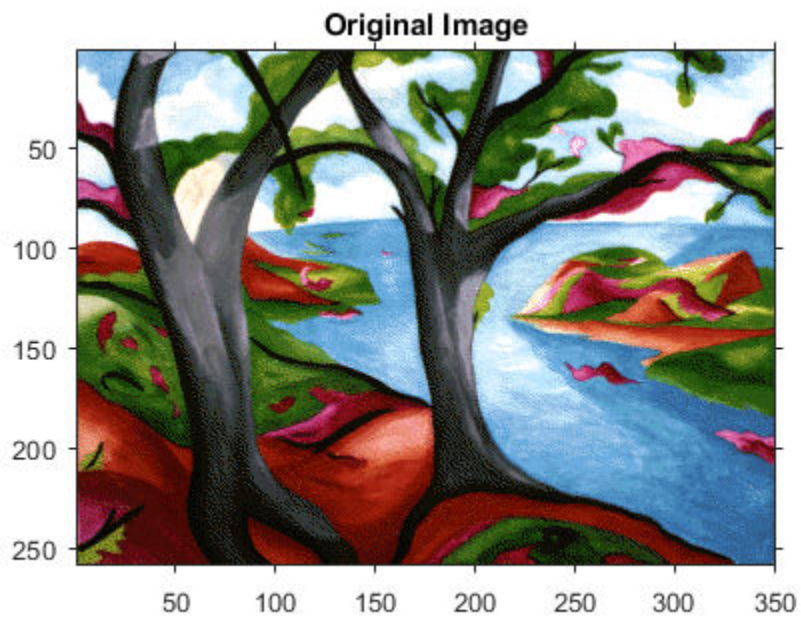
```
[X, map] = imread('trees.tif');
```

Resize the image, specifying a scale factor. By default, `imresize` returns an optimized color map with the resized indexed image.

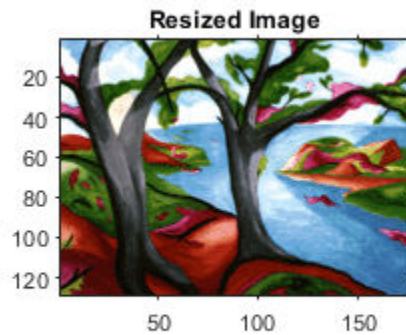
```
[Y, newmap] = imresize(X, map, 0.5);
```

Display the original image and the resized image.

```
figure  
imshow(X, map)  
title('Original Image')
```



```
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

### I — Image to be resized

numeric array | logical array | `gpuArray`

Image to be resized, specified as a numeric or logical array of any dimension.

To resize images using a GPU, specify I as a `gpuArray` that contains a numeric array of data type `double` or `single`.

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

### **scale — Resize factor**

positive number

Resize factor, specified as a positive number.

- If `scale` is less than 1, then the output image (J or Y) is smaller than the input image (I or X).
- If `scale` is greater than 1, then the output image is larger than the input image.

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

### **[numrows numcols] — Row and column dimensions of output image**

2-element vector of positive numbers

Row and column dimensions of output image, specified as a 2-element vector of positive numbers. 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**

numeric array

Indexed image to be resized, specified as a numeric array.

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

### **map — Colormap**

*c*-by-3 numeric matrix

Colormap associated with indexed image X, specified as a *c*-by-3 numeric matrix.

Data Types: `double`

### **method — Interpolation method**

'`bicubic`' (default) | string scalar | character vector | 2-element cell array

Interpolation method, specified as a string scalar, character vector or 2-element cell array.

When `method` is a string scalar or character vector, it 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.
'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
	<b>Note</b> 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

When `method` is a 2-element cell array, it defines a custom interpolation kernel. The cell array has the form  $\{f,w\}$ , where  $f$  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` | `string` | `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 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 `true` or `false`.

- If `method` is `nearest-neighbor` (`'nearest'`), then the default value of `'Antialiasing'` is `false`.
- For all other interpolation methods, the default is `true`.

Data Types: `logical`

### **Colormap** — Return optimized colormap

`'optimized'` (default) | `'original'`

Return optimized colormap for indexed image `X`, specified as the comma-separated pair consisting of `'Colormap'` and one of the following.

- `'original'` — The output colormap `newmap` is the same as the input colormap `map`.
- `'optimized'` — `imresize` returns a new optimized colormap.

This argument is valid only when resizing indexed images.

Data Types: `char` | `string`

### **Dither** — Perform color dithering

`true` (default) | `false`

Perform color dithering, specified as the comma-separated pair consisting of `'Dither'` and `true` or `false`. In dithering, you apply a form of noise to the image to randomize quantization error and prevent large-scale patterns.

This argument is valid only when resizing indexed images, `X`.

Data Types: `logical`

### **Method** — Interpolation method

`'bicubic'` (default) | `string scalar` | `character vector` | `2-element cell array`

Interpolation method, specified as the comma-separated pair consisting of 'Method' and a scalar string, character vector, or 2-element cell array. For details, see method.

Data Types: char | string | cell

## **OutputSize — Size of output image**

2-element numeric vector

Size of the output image, specified as the comma-separated pair consisting of 'OutputSize' and a 2-element numeric vector of the form [numrows numcols].

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

## **Scale — Resize scale factor**

positive number | 2-element vector of positive numbers

Resize scale factor, specified as the comma-separated pair consisting of 'Scale' and a positive number or 2-element vector of positive numbers.

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

# Output Arguments

## **J — Resized image**

numeric array | logical array | gpuArray

Resized image, returned as a numeric or logical array of the same class as the input image, I.

If the image is resized using a GPU, then J is returned as a gpuArray containing a numeric array.

## **Y — Resized indexed image**

numeric array

Resized indexed image, returned as a numeric array of the same class as the input indexed image, X.

## **newmap — Optimized colormap**

c-by-3 numeric array

Optimized colormap, returned as an *c*-by-3 numeric array.

## 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 are barely noticeable to the naked eye.
- If the size of the output image is not an integer, then `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 name-value pairs must be compile-time constants.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `gpuArray` input must be of type `single` or `double`.
- Only bicubic interpolation is supported on GPU and the function always performs antialiasing.

For more information, see “Image Processing on a GPU”.

### See Also

`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

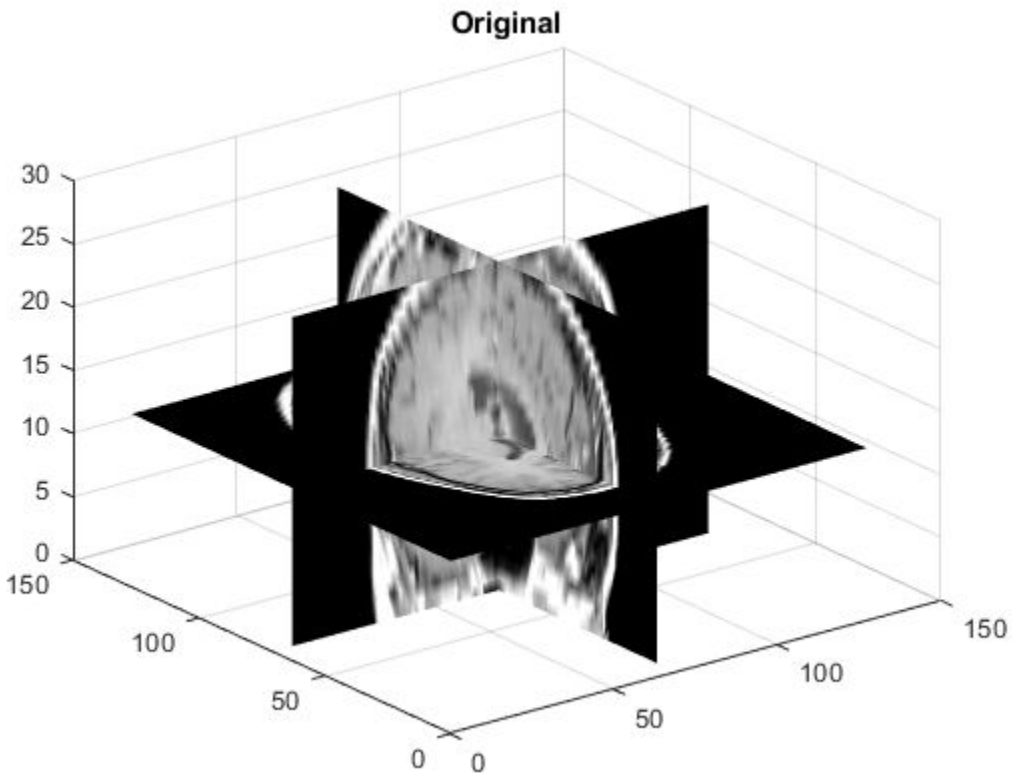
### Resize 3-D Volumetric Image

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');
```

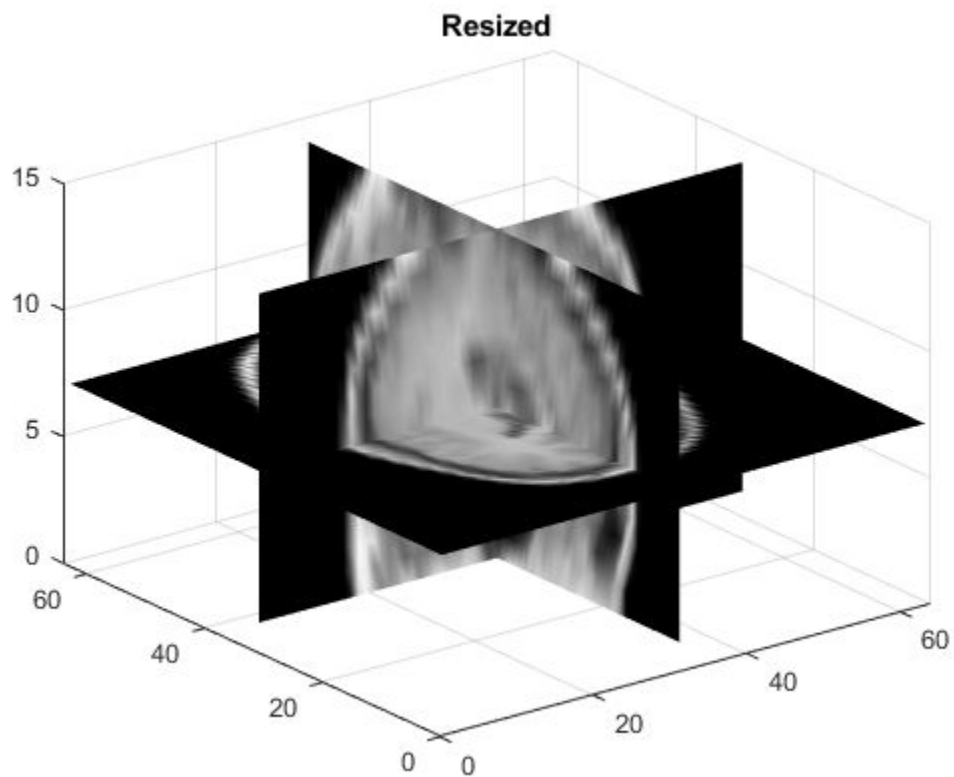


Resize the volume, reducing the size all all dimensions by one-half. This example uses the default interpolation method and antialiasing.

```
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 array with three dimensions.

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

### **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 one of the values in the following table that identifies a general method or a named interpolation kernel.

Method	Description
<code>'nearest'</code>	Nearest-neighbor interpolation
<code>'linear'</code>	Linear interpolation

Method	Description
'cubic'	Cubic interpolation  <b>Note</b> Cubic interpolation can produce pixel values outside the original range.
Interpolation Kernel	Description
'box'	Box-shaped kernel
'triangle'	Triangular kernel (equivalent to 'linear')
'lanczos2'	Lanczos-2 kernel
'lanczos3'	Lanczos-3 kernel

Data Types: char | string

## 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 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 string scalar or character vector. For details, see `method`.

Data Types: `char` | `string`

### **OutputSize — Size of output volume**

three-element numeric vector of positive values

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**

`imresize` | `imrotate` | `imrotate3` | `imwarp`

**Introduced in R2017a**

# imroi class

Region-of-interest (ROI) base class

## Description

`imroi`, an abstract class, specifies the Image Processing Toolbox interface for working with regions of interest (ROIs). You can use classes that inherit from the `imroi` interface to create interactive ROIs over an image.

## Construction

The `imroi` class is abstract, and creating an instance of the `imroi` class is not allowed. See `imellipse`, `imfreehand`, `imline`, `impoint`, `impoly`, or `imrect` to learn how to create a specific ROI object.

## Properties

### Deletable — ROI can be deleted

`true` (default) | `false`

ROI can be deleted, specified as `true` or `false`.

Data Types: `logical`

## Methods

`imroi` supports the following methods. Type `methods imroi` to see a complete list.

<code>addNewPositionCallback</code>	Add new-position callback to ROI object
<code>createMask</code>	Create mask within image
<code>delete</code>	Delete handle object
<code>getColor</code>	Get color used to draw ROI object
<code>getPosition</code>	Return current position of ROI object

<code>getPositionConstraintFcn</code>	Return function handle to current position constraint function
<code>removeNewPositionCallback</code>	Remove new-position callback from ROI object
<code>resume</code>	Resume execution of MATLAB command line
<code>setColor</code>	Set color used to draw ROI object
<code>setConstrainedPosition</code>	Set ROI object to new position
<code>setPositionConstraintFcn</code>	Set position constraint function of ROI object
<code>wait</code>	Block MATLAB command line until ROI creation is finished

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## See Also

`imellipse` | `imfreehand` | `imline` | `impoint` | `impoly` | `imrect` | `makeConstrainToRectFcn`

**Introduced in R2008a**



# addNewPositionCallback

Add new-position callback to ROI object

---

**Note** `addNewPositionCallback` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
id = addNewPositionCallback(h,fcn)
```

## Description

`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)
```

`pos` is of the form returned by the object's `getPosition` method. The return value, `id`, is used only with `removeNewPositionCallback`.

## Examples

### Display Updated Position in Title

Create a rectangle ROI object. Display the position of the rectangle in the title. The title updates when you move the rectangle.

```
imshow('cameraman.tif')  
h = imrect(gca, [10 10 100 100]);  
addNewPositionCallback(h,@(p) title(mat2str(p,3)));
```

Now drag the rectangle using the mouse to observe the callback behavior.

## Input Arguments

### **h — ROI object**

`imellipse` | `imfreehand` | `imline` | `impoint` | `impoly` | `imrect`

ROI object, specified as an `imellipse`, `imfreehand`, `imline`, `impoint`, `impoly`, or `imrect` object.

### **fcn — Function handle**

`handle`

Function handle, specified as a `handle`. The function must accept a numeric array as input. The array must have the same form as returned when calling `getPosition` on the object. For more information, see “Create Function Handle” (MATLAB).

## Output Arguments

### **id — Identifier of new-position callback function**

`struct`

Identifier of new-position callback function, returned as a `struct`.

## See Also

`getPosition` | `getPositionConstraintFcn` | `imroi` | `makeConstrainToRectFcn` | `removeNewPositionCallback` | `setPositionConstraintFcn`

## Topics

“Create Function Handle” (MATLAB)

“Anonymous Functions” (MATLAB)

“Parameterizing Functions” (MATLAB)

**Introduced in R2008a**

# createMask

Create mask within image

---

**Note** createMask is not recommended. Use the createMask method associated with the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
BW = createMask(h)
BW = createMask(h,himage)
```

## Description

BW = createMask(h) returns a mask, or binary 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 object.

“Create Binary Mask from Ellipse” on page 1-1757

BW = createMask(h,himage) returns a mask the same size as the image himage, with 1s inside the ROI object h and 0s everywhere else. This syntax is required when the axes that contains the ROI holds more than one image.

## Examples

### Create Binary Mask from Ellipse

Create an ellipse ROI.

```
imshow('coins.png');
e = imellipse;
```

Use the mouse to reshape and reposition the ellipse. Then, create a binary mask from the ROI. Pixels inside the ROI have the value 1, and pixels outside the ROI have the value 0. Display the mask in a new figure.

```
BW = createMask(e);  
figure; imshow(BW)
```

## Input Arguments

### **h** — ROI object

`imellipse` | `imline` | `impoint` | `impoly` | `imrect`

ROI object, specified as an `imellipse`, `imline`, `impoint`, `impoly`, or `imrect` object.

### **himage** — Handle to image

`handle`

Handle to one image, specified as a handle.

## Output Arguments

### **BW** — Mask

binary matrix

Mask, returned as a binary matrix. The mask is the same size as the input image contained in the same axes as `h`, or the image `himage`.

## See Also

`imroi` | `regionfill` | `roifilt2`

## Topics

“Create a Binary Mask”

**Introduced in R2008a**

# getAngleFromHorizontal

Return angle between Distance tool and horizontal axis

---

**Note** `getAngleFromHorizontal` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
angle = getAngleFromHorizontal(h)
```

## Description

`angle = getAngleFromHorizontal(h)` returns the angle, in degrees, between the line defined by the Distance tool, `h`, and the horizontal axis.

## Input Arguments

**h — Distance tool**

`imdistline`

Distance tool, specified as an `imdistline` object.

## Output Arguments

**angle — Angle**

numeric scalar

Angle, returned as a numeric scalar in the range [0, 180] degrees.

Data Types: `double`

## Algorithms

To understand how `indistline` 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

`getDistance`

**Introduced before R2006a**

# getColor

Get color used to draw ROI object

---

**Note** `getColor` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
color = getColor(h)
```

## Description

`color = getColor(h)` gets the color used to draw the ROI object `h`.

## Input Arguments

**h — ROI object**

`imellipse` | `imfreehand` | `imline` | `impoint` | `impoly` | `imrect`

ROI object, specified as an `imellipse`, `imfreehand`, `imline`, `impoint`, `impoly`, or `imrect` object.

## Output Arguments

**color — RGB color value**

3-element numeric vector

RGB color value, returned as a 3-element numeric vector.

## See Also

`imroi` | `setColor`

**Introduced before R2006a**



# getDistance

Return distance between endpoints of Distance tool

---

**Note** `getDistance` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
dist = getDistance(h)
```

## Description

`dist = getDistance(h)` returns the distance between the endpoints of the Distance tool, `h`.

## Input Arguments

**h** — Distance tool

`imdistline`

Distance tool, specified as an `imdistline` object.

## Output Arguments

**dist** — Distance between endpoints

numeric scalar

Distance between endpoints, returned as a numeric scalar. The data units of the distance are determined by the `XData` and `YData` properties of the underlying image. By default, the distance is measured in pixels.

## **See Also**

`getAngleFromHorizontal` | `getPosition`

**Introduced before R2006a**

## getLabelHandle

Return handle to text label of Distance tool

---

**Note** `getLabelHandle` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

### Syntax

```
hlabel = getLabelHandle(h)
```

### Description

`hlabel = getLabelHandle(h)` returns a handle to the text label of the Distance tool, `h`.

### Input Arguments

**h — Distance tool**

`imdistline`

Distance tool, specified as an `imdistline` object.

### Output Arguments

**hlabel — Handle to text label**

`handle`

Handle to text label, returned as a handle to a `Text` object.

### See Also

`Text`

**Introduced before R2006a**

# getLabelTextFormatter

Return format of text label of Distance tool

---

**Note** `getLabelTextFormatter` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
str = getLabelTextFormatter(h)
```

## Description

`str = getLabelTextFormatter(h)` returns a character array specifying the format used to display the label text of the Distance tool, `h`.

## Input Arguments

**h — Distance tool**

`imdistline`

Distance tool, specified as an `imdistline` object.

## Output Arguments

**str — Text format**

character array

Text format of Distance tool label, returned as a character array in a format expected by `sprintf`.

## **See Also**

`getLabelVisible` | `setLabelTextFormatter` | `setLabelVisible`

**Introduced before R2006a**

# getLabelVisible

Return visibility of text label of Distance tool

---

**Note** `getLabelVisible` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
visible = getLabelVisible(h)
```

## Description

`visible = getLabelVisible(h)` returns the visibility of the text label of the Distance tool, `h`.

## Input Arguments

**h — Distance tool**

`imdistline`

Distance tool, specified as an `imdistline` object.

## Output Arguments

**visible — Label visibility**

'on' | 'off'

Label visibility, returned as 'on' or 'off'.

## See Also

`getLabelTextFormatter` | `setLabelTextFormatter` | `setLabelVisible`

**Introduced before R2006a**



# getPosition

Return current position of ROI object

---

**Note** `getPosition` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
pos = getPosition(h)
```

## Description

`pos = getPosition(h)` returns the current position of the ROI object, `h`.

## Input Arguments

### **h** — ROI object

`imellipse` | `imfreehand` | `imline` | `impoint` | `impoly` | `imrect`

ROI object, specified as an `imellipse`, `imfreehand`, `imline`, `impoint`, `impoly`, or `imrect` object.

## Output Arguments

### **pos** — Position of ROI object

numeric array

Position of the ROI object, returned as a numeric array. The shape of the array depends on the type of ROI object.

<b>ROI Object</b>	<b>Returned position</b>
<code>imellipse</code>	4-element vector of the form <code>[xmin ymin width height]</code> , representing the size and position of a bounding box around the ellipse. The initial size of the bounding box is <code>width-by-height</code> pixels. The upper-left corner of the box is at the <code>(x,y)</code> coordinate <code>(xmin,ymin)</code> .
<code>imfreehand</code>	$n$ -by-2 matrix. The two columns define the $x$ - and $y$ -coordinates, respectively, of the $n$ points along the boundary of the freehand region.
<code>imline</code>	2-by-2 matrix of the form <code>[x1 y1; x2 y2]</code> , representing the position of the two endpoints of the line.
<code>impoint</code>	1-by-2 vector of the form <code>[x y]</code> .
<code>impoly</code>	$n$ -by-2 matrix. The two columns define the $x$ - and $y$ -coordinates, respectively, of each of the $n$ vertices.
<code>imrect</code>	4-element vector of the form <code>[xmin ymin width height]</code> . The initial size of the rectangle is <code>width-by-height</code> pixels. The upper-left corner of the rectangle is at the <code>(x,y)</code> coordinate <code>(xmin,ymin)</code> .

## See Also

`getPositionConstraintFcn` | `imroi` | `setPosition`

**Introduced in R2008a**

# getPositionConstraintFcn

Return function handle to current position constraint function

---

**Note** `getPositionConstraintFcn` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
fcn = getPositionConstraintFcn(h)
```

## Description

`fcn = getPositionConstraintFcn(h)` returns a function handle `fcn` to the current position constraint function of the ROI object `h`.

## Input Arguments

**h** — ROI object

`imellipse` | `imline` | `impoint` | `impoly` | `imrect`

ROI object, specified as an `imellipse`, `imline`, `impoint`, `impoly`, or `imrect` object.

## Output Arguments

**fcn** — Function handle

handle

Function handle, returned as a handle. For more information, see “Create Function Handle” (MATLAB).

## See Also

`getPosition` | `imroi` | `makeConstrainToRectFcn` | `setConstrainedPosition` | `setPositionConstraintFcn`

**Introduced in R2008a**

# getVertices

Return vertices on perimeter of ellipse ROI object

---

**Note** `getVertices` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
v = getVertices(h)
```

## Description

`v = getVertices(h)` returns a set of vertices that lie along the perimeter of an ellipse ROI object.

## Input Arguments

**h** — ROI object

`imellipse`

ROI object, specified as an `imellipse` object.

## Output Arguments

**v** — Vertices

*n*-by-2 matrix

Vertices, returned as an *n*-by-2 matrix. The two columns define the *x*- and *y*-coordinates, respectively, of each of the *n* vertices.

## **See Also**

`getColor` | `getPosition` | `wait`

**Introduced in R2007b**

# removeNewPositionCallback

Remove new-position callback from ROI object

---

**Note** `removeNewPositionCallback` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
removeNewPositionCallback(h,id)
```

## Description

`removeNewPositionCallback(h,id)` removes the corresponding function from the new-position callback list of the ROI object, `h`.

## Examples

### Add and Remove New Position Callback

Create a line ROI object. Display the position of the line in the title. Use `addNewPositionCallback` to update the title each time you move the line.

```
imshow('pout.tif')  
h = imline(gca,[10 100],[100 100]);  
id = addNewPositionCallback(h,@(pos) title(mat2str(pos,3)));
```

Move the line to observe the callback behavior.

After observing the callback behavior, remove the callback. The title no longer changes when you move the line.

```
removeNewPositionCallback(h,id);
```

## Input Arguments

### **h — ROI object**

`imellipse` | `imline` | `impoint` | `impoly` | `imrect`

ROI object, specified as an `imellipse`, `imline`, `impoint`, `impoly`, or `imrect` object.

### **id — Identifier of new-position callback function**

`struct`

Identifier of new-position callback function, specified as a `struct`.

## See Also

`addNewPositionCallback` | `getPositionConstraintFcn` | `imroi` |  
`makeConstrainToRectFcn` | `setPositionConstraintFcn`

**Introduced in R2008a**



# resume

Resume execution of MATLAB command line

## Syntax

```
resume(h)
```

## Description

`resume(h)` resumes execution of the MATLAB command line.

The `resume` method is useful when you need to exit `wait` from a callback function. When called after a call to `wait`, `resume` causes `wait` to return the ROI position or coordinates of vertices.

## Input Arguments

### **h** — ROI object

```
imellipse | imline | impoint | impoly | imrect
```

ROI object, specified as an `imellipse`, `imline`, `impoint`, `impoly`, or `imrect` object.

## See Also

```
imroi | wait
```

**Introduced in R2008a**

## setClosed

Set closure behavior of ROI object

---

**Note** `setClosed` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

### Syntax

`setClosed(h,TF)`

### Description

`setClosed(h,TF)` sets whether the ROI object, `h`, is closed after the last point is selected.

### Input Arguments

**h — ROI object**

`imfreehand` | `impoly`

ROI object, specified as an `imfreehand` or `impoly` object.

**TF — ROI object is closed**

`true` | `false`

ROI object is closed, specified as `true` or `false`. When set to `true`, a straight line connect the endpoints of the ROI object to create a closed region. If set to `false`, the endpoints are not connected and the region is open.

Data Types: `logical`

## **See Also**

**Introduced in R2007b**

## setColor

Set color used to draw ROI object

---

**Note** `setColor` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

### Syntax

```
setColor(h,color)
```

### Description

`setColor(h,color)` sets the color used to draw the ROI object `h`.

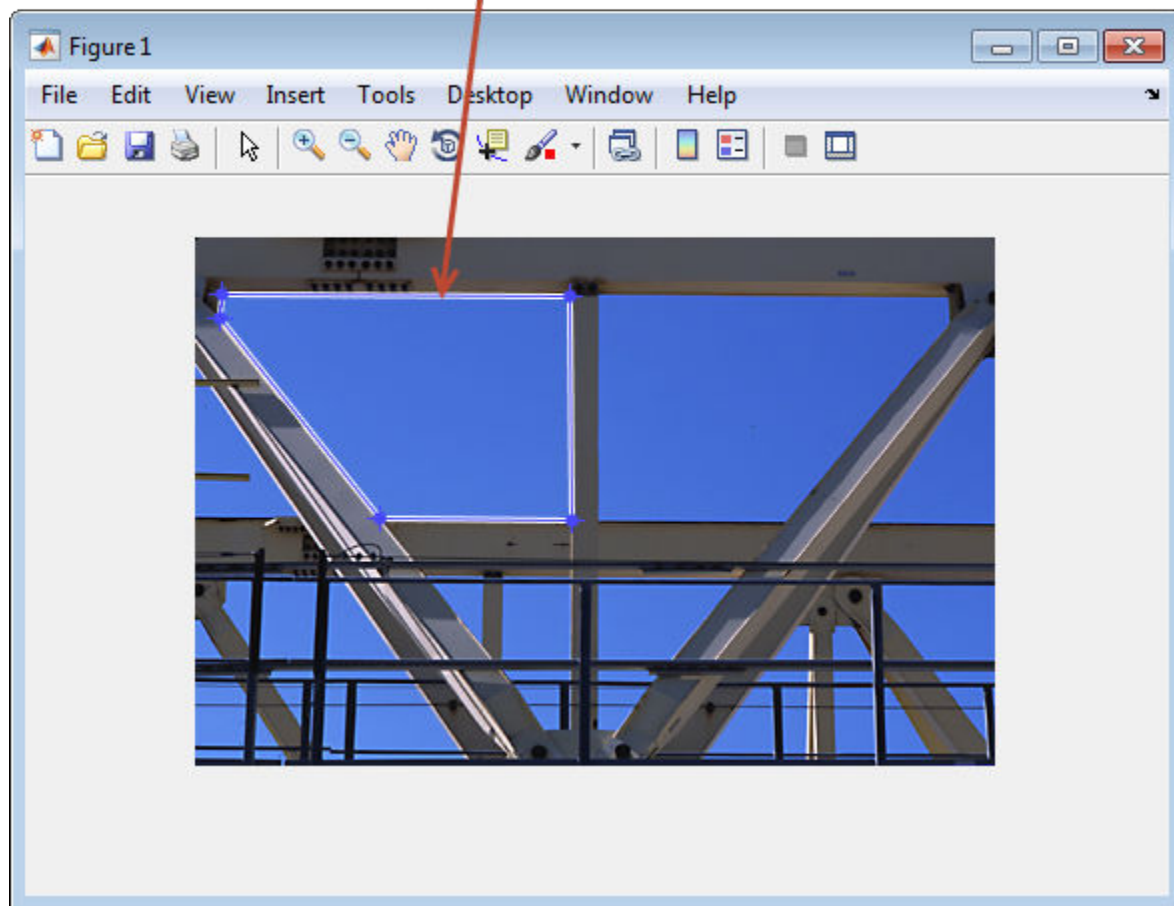
### Examples

#### Set Color of Polygon ROI

Display an image. Draw a polygon on the image, specifying the location of five vertices.

```
imshow('gantrycrane.png')  
h = impoly(gca,[188,30; 189,142; 93,141; 13,41; 14,29]);
```

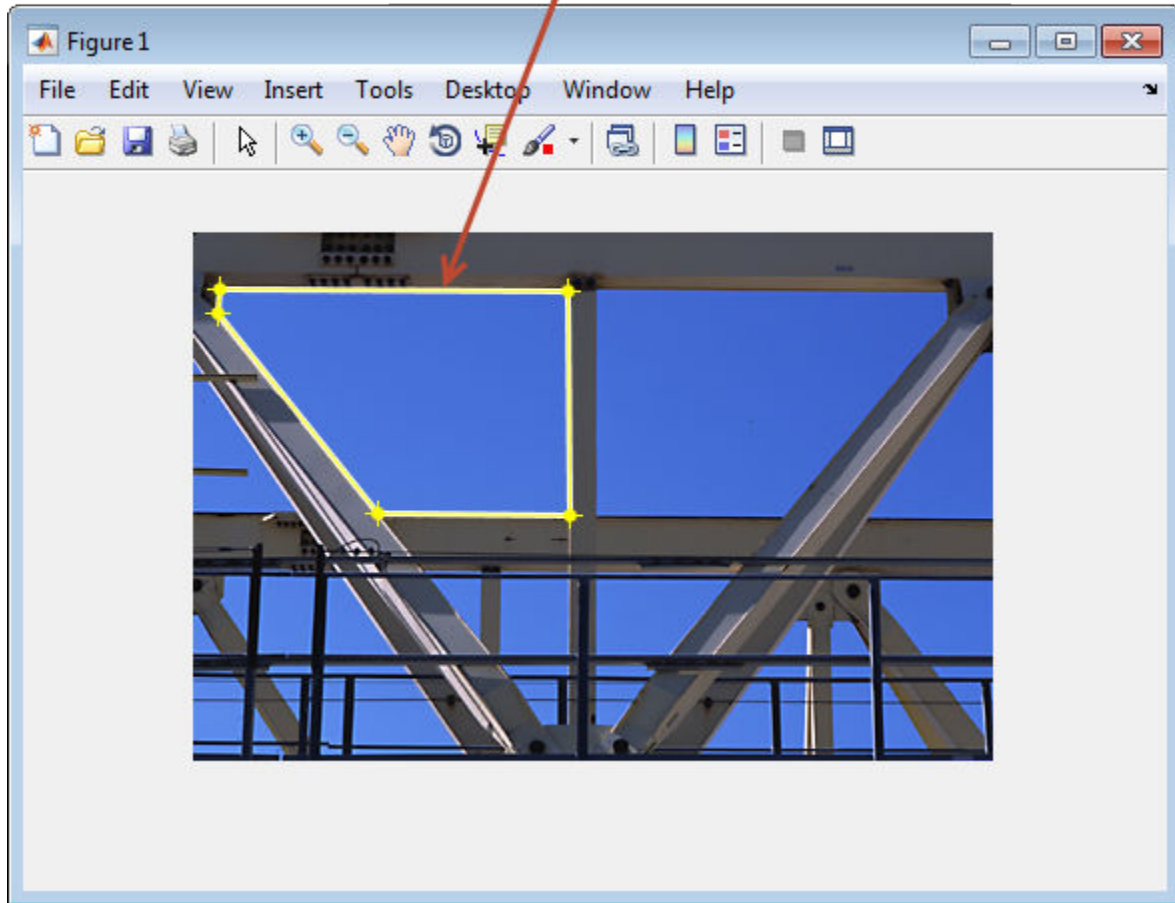
Polygon



Set the color of the polygon to yellow.

```
setColor(h, 'yellow');
```

Color of polygon changed to yellow



## Input Arguments

### **h** — ROI object

`imellipse` | `imfreehand` | `imline` | `impoint` | `impoly` | `imrect`

ROI object, specified as an `imellipse`, `imfreehand`, `imline`, `impoint`, `impoly`, or `imrect` object.

**color — RGB color value**

3-element numeric vector | predefined color name

RGB color value, specified as a 3-element numeric vector, or the long or short name of a predefined color, such as 'white' or 'w'. See `ColorSpec` for a list of predefined colors.

**See Also**

`ColorSpec` | `getColor` | `imroi`

**Introduced in R2008a**

## setConstrainedPosition

Set ROI object to new position

---

**Note** `setConstrainedPosition` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

### Syntax

```
setConstrainedPosition(h, pos)
```

### Description

`setConstrainedPosition(h, pos)` sets the ROI object `h` to a new position. The candidate position, `pos`, is subject to the position constraint function specified by `setPositionConstraintFcn`.

### Input Arguments

**h — ROI object**

`imellipse` | `imline` | `impoint` | `impoly` | `imrect`

ROI object, specified as an `imellipse`, `imline`, `impoint`, `impoly`, or `imrect` object.

**pos — Candidate position of ROI object**

numeric array

Candidate position of the ROI object, specified as a numeric array. The shape of the array depends on the type of ROI object, and is consistent with the form returned by the `setPosition` function.



ROI Object	Position
imellipse	4-element vector of the form [xmin ymin width height], representing the new size and position of a bounding box around the ellipse. The new size of the bounding box is width-by-height pixels. The upper-left corner of the box is at the new (x,y) coordinate (xmin,ymin).
imline	2-by-2 matrix of the form [x1 y1; x2 y2], representing the new position of the two endpoints of the line.
impoint	1-by-2 vector of the form [x y].
impoly	$n$ -by-2 matrix. The two columns define the new $x$ - and $y$ -coordinates, respectively, of each of the $n$ vertices.
imrect	4-element vector of the form [xmin ymin width height]. The new size of the rectangle is width-by-height pixels. The upper-left corner of the rectangle is at the new (x,y) coordinate (xmin,ymin).

## See Also

getPosition | getPositionConstraintFcn | imroi | setPosition | setPositionConstraintFcn

**Introduced in R2008a**

## setFixedAspectRatioMode

Preserve aspect ratio when resizing ROI object

---

**Note** `setFixedAspectRatioMode` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

### Syntax

```
setFixedAspectRatioMode(h,TF)
```

### Description

`setFixedAspectRatioMode(h,TF)` sets whether the aspect ratio of the ROI object is preserved during interactive resizing.

### Examples

#### Fix Aspect Ratio of Ellipse

Create an ellipse ROI object. Specify a position constraint function using `makeConstrainToRectFcn` to keep the ellipse inside the boundary of the image.

```
imshow('coins.png')
h = imellipse(gca,[10 10 100 100]);
fcn = makeConstrainToRectFcn('imellipse',get(gca,'XLim'),get(gca,'YLim'));
setPositionConstraintFcn(h,fcn);
```

Try resizing and reshaping the ellipse.

Now, fix the aspect ratio of the ellipse.

```
setFixedAspectRatioMode(h,true);
```

Try resizing the ellipse. The aspect ratio of the ellipse does not change.

## Input Arguments

**h — ROI object**

`imellipse` | `imrect`

ROI object, specified as an `imellipse` or `imrect` object.

**TF — Fix aspect ratio**

`true` | `false`

Fix the aspect ratio when resizing ROI object, specified as `true` or `false`.

Data Types: `logical`

## See Also

`setResizable`

**Introduced before R2006a**

## setLabelTextFormatter

Set format used to display text label of Distance tool

---

**Note** `setLabelTextFormatter` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

### Syntax

```
setLabelTextFormatter(h, str)
```

### Description

`setLabelTextFormatter(h, str)` sets the format used to display the label text of the Distance tool, `h`.

### Examples

#### Format Label of Distance Tool

Display an image and create a Distance tool.

```
imshow('pout.tif')  
hline = imdistline(gca,[71 171],[108 150]);
```

Modify the format of the label to indicate that distance is measured in pixels.

```
setLabelTextFormatter(hline, '%02.0f pixels');
```

### Input Arguments

**h** — Distance tool

`imdistline`

Distance tool, specified as an `imdistline` object.

**str — Text format**

character array

Text format of Distance tool label, specified as a character array in a format expected by `sprintf`.

## See Also

`setLabelTextFormatter` | `getLabelVisible` | `setLabelVisible`

**Introduced before R2006a**

## setLabelVisible

Set visibility of text label of Distance tool

---

**Note** `setLabelVisible` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

### Syntax

```
setLabelVisible(h,TF)
```

### Description

`setLabelVisible(h,TF)` sets the visibility of the text label of the Distance tool, `h`.

### Input Arguments

**h — Distance tool**

```
imdistline
```

Distance tool, specified as an `imdistline` object.

**TF — Text label is visible**

```
true | false
```

Text label is visible, specified as `true` or `false`.

Data Types: `logical`

### See Also

`getLabelTextFormatter` | `getLabelVisible` | `setLabelTextFormatter`

**Introduced before R2006a**

## setPosition

Move ROI object to new position

---

**Note** `setPosition` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

### Syntax

```
setPosition(h, pos)
setPosition(h, x, y)
```

### Description

`setPosition(h, pos)` moves the position of the ROI object, `h`, to the location specified by `pos`.

`setPosition(h, x, y)` specifies the new `x`- and `y`-coordinates of points of a line or point ROI object.

### Input Arguments

**h — ROI object**

`imellipse` | `imline` | `impoint` | `impoly` | `imrect`

ROI object, specified as an `imellipse`, `imline`, `impoint`, `impoly`, or `imrect` object.

**pos — New position of ROI object**

numeric array

New position of the ROI object, specified as a numeric array. The shape of the array depends on the type of ROI object.

<b>ROI Object</b>	<b>Position</b>
<code>imellipse</code>	4-element vector of the form <code>[xmin ymin width height]</code> , representing the new size and position of a bounding box around the ellipse. The new size of the bounding box is <code>width-by-height</code> pixels. The upper-left corner of the box is at the new <code>(x,y)</code> coordinate <code>(xmin,ymin)</code> .
<code>imline</code>	2-by-2 matrix of the form <code>[x1 y1; x2 y2]</code> , representing the new position of the two endpoints of the line.
<code>impoint</code>	1-by-2 vector of the form <code>[x y]</code> .
<code>impoly</code>	$n$ -by-2 matrix. The two columns define the new $x$ - and $y$ -coordinates, respectively, of each of the $n$ vertices.
<code>imrect</code>	4-element vector of the form <code>[xmin ymin width height]</code> . The new size of the rectangle is <code>width-by-height</code> pixels. The upper-left corner of the rectangle is at the new <code>(x,y)</code> coordinate <code>(xmin,ymin)</code> .

**x, y — New x- or y-coordinates of points**

2-element vector | numeric scalar

New  $x$ - or  $y$ -coordinates of points.

- If `h` is an `imline` object, then `x` and `y` are 2-element vectors that represent the  $x$ - and  $y$ -coordinates of the two line endpoints.
- If `h` is an `impoint` object, then `x` and `y` are numeric scalars that represent the  $x$ - and  $y$ -coordinate of the single point.

**See Also**

`getPosition` | `setPositionConstraintFcn`

**Introduced in R2008a**



# setPositionConstraintFcn

Set position constraint function of ROI object

---

**Note** `setPositionConstraintFcn` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
setPositionConstraintFcn(h,fcn)
```

## Description

`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(pos)
```

## Examples

### Update Title when Rectangle Moves

Display a rectangle ROI over an image. Display the position of the rectangle in the title. The title updates when you move the rectangle. Try dragging one side of the rectangle outside the boundary of the image.

```
imshow('cameraman.tif')  
h = imrect(gca,[10 10 100 100]);  
addNewPositionCallback(h,@(p) title(mat2str(p,3)));
```

Specify a position constraint function using `makeConstrainToRectFcn` to keep the rectangle inside the original `XLim` and `YLim` ranges.

```
fcn = makeConstrainToRectFcn('imrect',get(gca,'XLim'),get(gca,'YLim'));  
setPositionConstraintFcn(h,fcn);
```

Now drag the rectangle using the mouse. Observe that the rectangle can no longer extend past the image boundary.

## Input Arguments

### **h — ROI object**

`imellipse` | `imline` | `impoint` | `impoly` | `imrect`

ROI object, specified as an `imellipse`, `imline`, `impoint`, `impoly`, or `imrect` object.

### **fcn — Function handle**

`handle`

Function handle, specified as a handle. You can use the `makeConstrainToRectFcn` to create this function. The function must accept a numeric array as input, and it must return a numeric array as output. Both arrays must have the same form as returned when calling `getPosition` on the object. For more information, see “Create Function Handle” (MATLAB).

## See Also

`addNewPositionCallback` | `getPosition` | `getPositionConstraintFcn` | `imroi` | `makeConstrainToRectFcn` | `setConstrainedPosition`

## Topics

“Create Function Handle” (MATLAB)

“Anonymous Functions” (MATLAB)

“Parameterizing Functions” (MATLAB)

**Introduced in R2008a**

## setResizable

Set resize behavior of ROI object

---

**Note** `setResizable` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

### Syntax

```
setResizable(h,TF)
```

### Description

`setResizable(h,TF)` sets whether the ROI object may be resized interactively.

### Examples

#### Fix Size of Ellipse

Create an ellipse ROI object. Specify a position constraint function using `makeConstrainToRectFcn` to keep the ellipse inside the boundary of the image.

```
imshow('coins.png')  
h = imellipse(gca,[10 10 100 100]);  
fcn = makeConstrainToRectFcn('imellipse',get(gca,'XLim'),get(gca,'YLim'));  
setPositionConstraintFcn(h,fcn);
```

Click and drag with the mouse to try resizing, reshaping, and moving the ellipse.

Now, disable resizing the ellipse.

```
setResizable(h,false);
```

Click and drag the ellipse again. You can move it, but not change the size or shape of it.

## Input Arguments

### **h — ROI object**

`imellipse` | `imrect`

ROI object, specified as an `imellipse` or `imrect` object.

### **TF — Enable resizing of ROI object**

`true` | `false`

Enable resizing of ROI object, specified as `true` or `false`.

Data Types: `logical`

## See Also

`setFixedAspectRatioMode`

**Introduced before R2006a**

# setString

Set text label for point ROI object

---

**Note** `setString` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
setString(h,text)
```

## Description

`setString(h,text)` places a text label, `text`, to the lower right of the point ROI object, `h`.

## Examples

### Set Label of Point ROI

```
imshow('rice.png')  
h = impoint(gca,100,200);  
setString(h,'My point label');
```

## Input Arguments

**h** — Point ROI object

`impoint`

Point ROI object, specified as an `impoint` object.

**text** — Text label

character vector

Text label, specified as a character vector.

Data Types: char

## **See Also**

**Introduced before R2006a**

# setVerticesDraggable

Set vertex behavior of ROI object

---

**Note** `setVerticesDraggable` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
setVerticesDraggable(h,TF)
```

## Description

`setVerticesDraggable(h,TF)` sets whether the vertices of the ROI object, `h`, can be dragged after placement.

## Input Arguments

**h — ROI object**

`impoly`

ROI object, specified as an `impoly` object.

**TF — Polygon ROI vertices are draggable**

`true` | `false`

Polygon ROI vertices are draggable, specified as `true` or `false`.

Data Types: `logical`

## See Also

`getVertices`

**Introduced in R2007b**



## wait

Block MATLAB command line until ROI creation is finished

---

**Note** `wait` is not recommended. Use one of the ROI classes instead, described in “ROI Creation Overview”.

---

## Syntax

```
pos = wait(h)
v = wait(he)
```

## Description

`pos = wait(h)` blocks execution of the MATLAB command line until you finish positioning the ROI object `h`. Indicate completion by double-clicking on the ROI object. The function returns the position, `pos`, of the ROI object.

`v = wait(he)` blocks execution of the MATLAB command line until you finish positioning the ellipse ROI object `he`. Indicate completion by double-clicking on the ROI object. The function returns the coordinates of vertices, `v`, along the perimeter of the ellipse.

## Examples

### Click and Drag to Place Rectangle

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.

```
imshow('pout.tif')
h = imrect;
position = wait(h)
```

## Click and Drag to Place Ellipse

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.

```
imshow('coins.png')  
h = imellipse;  
position = wait(h)
```

## Input Arguments

### **h** — ROI object

`imfreehand` | `imline` | `impoint` | `impoly` | `imrect`

ROI object, specified as an `imfreehand`, `imline`, `impoint`, `impoly`, or `imrect` object.

### **he** — Ellipse ROI object

`imellipse`

Ellipse ROI object, specified as an `imellipse` object.

## Output Arguments

### **pos** — Position of ROI object

numeric array

Position of the ROI object, returned as a numeric array. The shape of the array depends on the type of ROI object, and is consistent with the output of `getPosition`.

ROI Object	Returned position
<code>imfreehand</code>	$n$ -by-2 matrix. The two columns define the $x$ - and $y$ -coordinates, respectively, of the $n$ points along the boundary of the freehand region.
<code>imline</code>	2-by-2 matrix of the form $[x1\ y1; x2\ y2]$ , representing the position of the two endpoints of the line.
<code>impoint</code>	1-by-2 vector of the form $[x\ y]$ .

ROI Object	Returned position
impoly	$n$ -by-2 matrix. The two columns define the $x$ - and $y$ -coordinates, respectively, of each of the $n$ vertices.
imrect	4-element vector of the form [xmin ymin width height]. The initial size of the rectangle is width-by-height pixels. The upper-left corner of the rectangle is at the $(x,y)$ coordinate (xmin,ymin).

### **v — Vertices of ellipse ROI object**

$n$ -by-2 matrix

Vertices of ellipse ROI object, returned as an  $n$ -by-2 matrix. The two columns define the  $x$ - and  $y$ -coordinates, respectively, of each of the  $n$  vertices. The form of the matrix is consistent with the output of `getVertices`.

## **See Also**

`getPosition` | `getVertices` | `imroi` | `resume`

**Introduced in R2008a**

## imrotate

Rotate image

### Syntax

```
J = imrotate(I,angle)
J = imrotate(I,angle,method)
J = imrotate(I,angle,method,bbox)
```

### Description

`J = imrotate(I,angle)` rotates image `I` 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 `J` large enough to contain the entire rotated image. `imrotate` uses nearest neighbor interpolation, setting the values of pixels in `J` that are outside the rotated image to 0 (zero).

You optionally can perform the rotation using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`J = imrotate(I,angle,method)` rotates image `I`, using the interpolation method specified by `method`.

---

**Note** The 'bicubic' interpolation method may give slightly different results on a GPU and CPU.

---

`J = imrotate(I,angle,method,bbox)` rotates image `I`, where `bbox` specifies the size of the output image. If you specify 'crop', then `imrotate` makes the output image the same size as the input image. If you specify 'loose', then `imrotate` makes the output image large enough to include the entirety of the rotated image.

### Examples

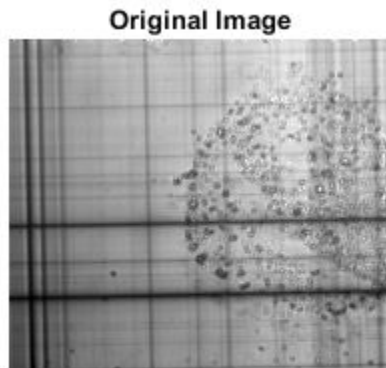
## Rotate Image Clockwise for Better Horizontal Alignment

Read an image into the workspace, and convert it to a grayscale image.

```
I = fitsread('solarspectra.fts');  
I = rescale(I);
```

Display the original image.

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

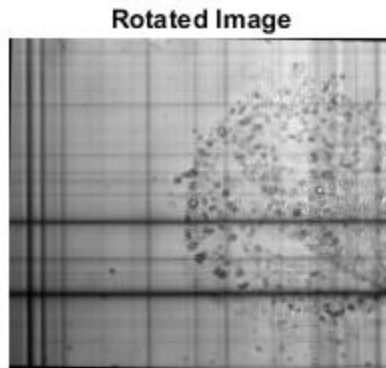


Rotate the image 1 degree clockwise to bring it into better horizontal alignment. The example specified bilinear interpolation and requests that the result be cropped to be the same size as the original image.

```
J = imrotate(I,-1,'bilinear','crop');
```

Display the rotated image.

```
figure  
imshow(J)  
title('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,'bilinear','loose');
```

Display the rotated image.

```
figure; imshow(Y)
```

## Input Arguments

### **I** — Image to be rotated

numeric array | logical array | `gpuArray`

Image to be rotated, specified as a numeric or logical array.

To perform the computation using a GPU, specify **I** as a `gpuArray` that contains a 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:

<b>Value</b>	<b>Description</b>
'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
	<b>Note</b> Bicubic interpolation can produce pixel values outside the original range.

Data Types: `char` | `string`

### **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 J the same size as the input image I, cropping the rotated image to fit.
'loose'	Make output image J large enough to contain the entire rotated image. J is larger than I.

Data Types: `char` | `string`

## Output Arguments

### **J** — Rotated image

`numeric array` | `logical array` | `gpuArray`

Rotated image, returned as a numeric or logical array.

If the image is rotated using a GPU, then J is returned as a `gpuArray` containing a numeric or logical array.

## 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 “Understand Code Generation with Image Processing Toolbox”.
- The `method` and `bbox` arguments must be compile-time constants.

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The 'bicubic' interpolation mode used in the GPU implementation of this function differs from the default (CPU) bicubic mode. The GPU and CPU versions of this function are expected to give slightly different results.

For more information, see “Image Processing on a GPU”.

## See Also

`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 trilinear 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 `'crop'`, `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

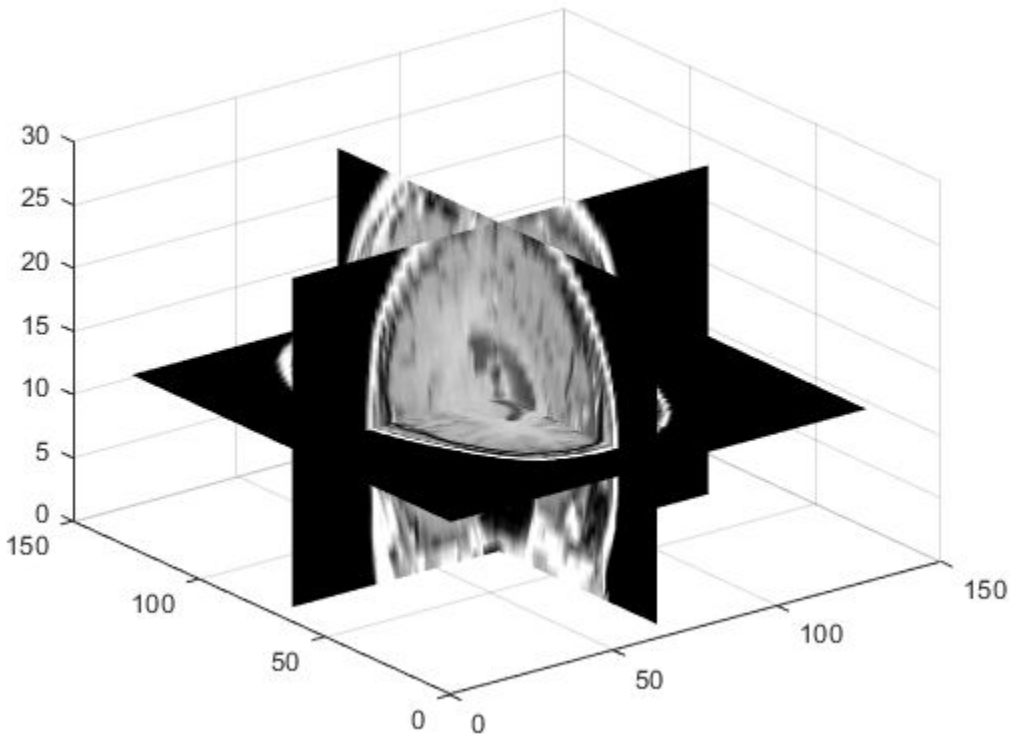
#### Rotate 3-D Volumetric Grayscale Image

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.

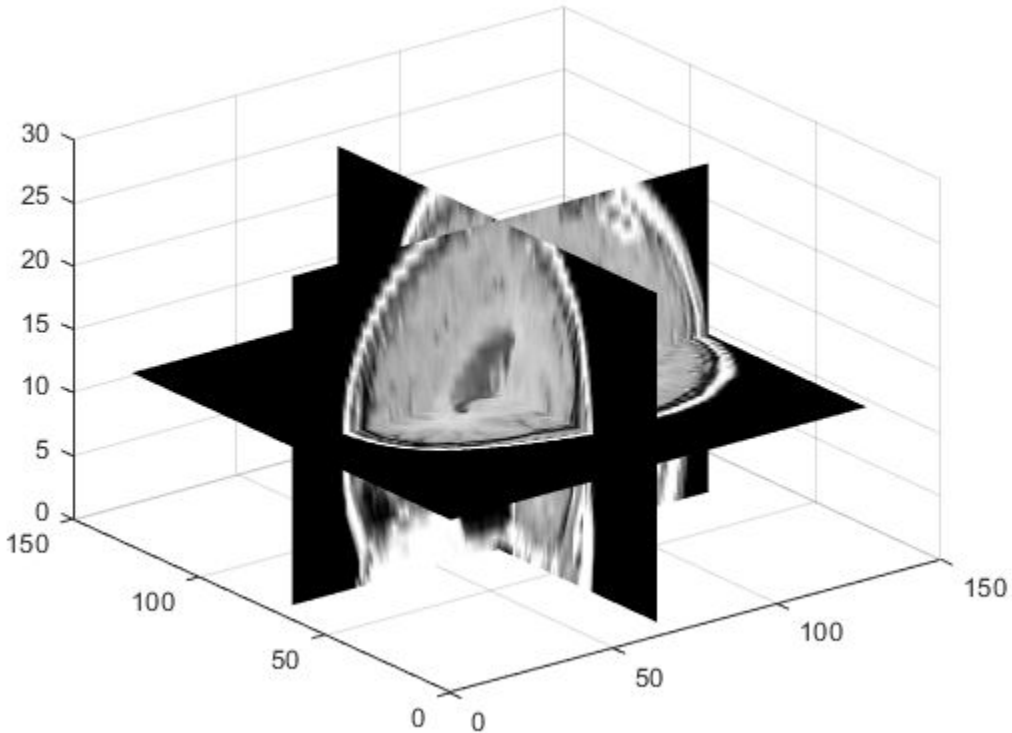
```

B = imrotate3(mriVolume,90,[0 0 1], 'nearest', 'loose', 'FillValues',0);

```

Display the rotated output. You can also display the result in the Volume Viewer app.

```
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

'linear' (default) | 'cubic' | 'nearest'

Interpolation method, specified as one of the following values.

Method	Description
'cubic'	Tricubic interpolation
	<b>Note</b> Tricubic interpolation can produce pixel values outside the original range.

Method	Description
'linear'	Trilinear interpolation
'nearest'	Nearest neighbor interpolation

Data Types: char | string

### **bbox — Size of the output volume**

'loose' (default) | 'crop'

Size of the output volume, specified as either of the following values.

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 | string

## **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 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

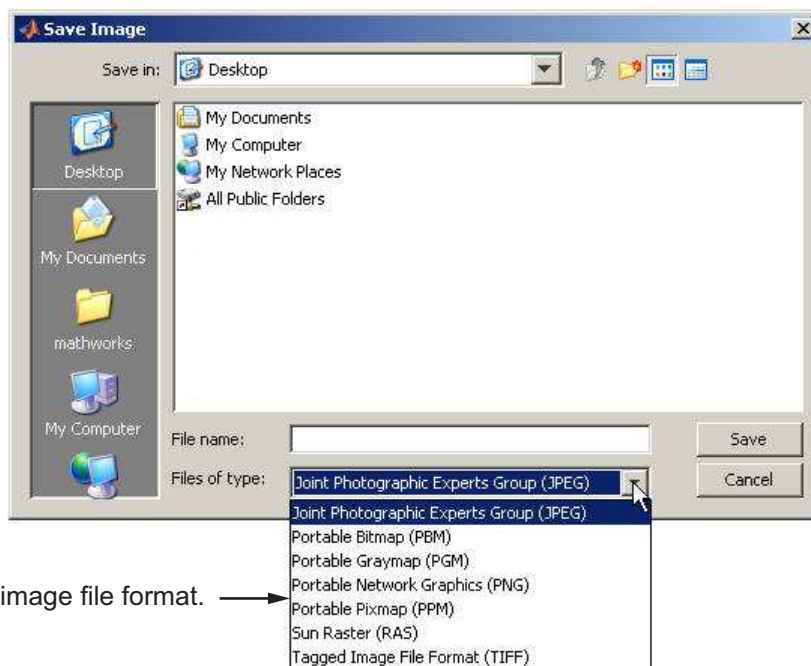
**Volume Viewer** | `imresize` | `imresize3` | `imrotate` | `imtranslate` | `imwarp`

**Introduced in R2017a**

## imsave

### Save Image Tool

Use the `imsave` function to create a Save Image tool. 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. For more information about using the tool, see “Tips” on page 1-1820.



Select image file format. →

## 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.

`imsave(h)` creates a Save Image tool associated with the image specified by the handle `h`.

`[filename,user_canceled] = imsave( ___ )` returns the full path to the file selected in `filename` and indicates whether you canceled the save operation.

## Examples

### Save Displayed Image

```
imshow peppers.png
imsave
```

## Input Arguments

**h — Handle to graphics object**

handle

Handle to a figure, axes, uipanel, or image graphics object, specified as a handle. If `h` is an axes or figure handle, `imsave` uses the first image returned by `findobj(H,'Type','image')`.

## Output Arguments

**filename — Full path to file**

character vector | ''

Full path to file, returned as a character vector. If you cancel the save operation, `filename` is returned as an empty character array, ''.

**user\_canceled — User canceled operation**

false (default) | true

User canceled operation, returned as `false` or `true`. If you press the **Cancel** button or close the save window, `imsave` sets `user_canceled` to `true`; otherwise, `false`.

### Tips

- 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.
- `imsave` uses `imwrite` to save the image, using default options.
- 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.
- The Save Image tool is modal; it blocks the MATLAB command line until you respond.

### 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.

<b>Function</b>	<b>Description</b>
setMagnification	Sets the magnification of the target image in units of screen pixels per image pixel.  <code>mag = api.setMagnification(new_mag)</code>  where <code>new_mag</code> is a scalar magnification factor.
getMagnification	Returns the current magnification factor of the target image in units of screen pixels per image pixel.  <code>mag = api.getMagnification()</code>  Multiply <code>mag</code> by 100 to convert to percentage. For example if <code>mag=2</code> , the magnification is 200%.
setMagnificationAndCenter	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.  <code>api.setMagnificationAndCenter(mag, cx, cy)</code>
findFitMag	Returns the magnification factor that would make the target image just fit in the scroll panel.  <code>mag = api.findFitMag()</code>
setVisibleLocation	Moves the target image so that the specified location is visible. Scrollbars update.  <code>api.setVisibleLocation(xmin, ymin)</code> <code>api.setVisibleLocation([xmin ymin])</code>
getVisibleLocation	Returns the location of the currently visible portion of the target image.  <code>loc = api.getVisibleLocation()</code>  where <code>loc</code> is a vector <code>[xmin ymin]</code> .
getVisibleImageRect	Returns the current visible portion of the image.  <code>r = api.getVisibleImageRect()</code>  where <code>r</code> is a rectangle <code>[xmin ymin width height]</code> .

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>

<b>Function</b>	<b>Description</b>
<code>removeNewLocationCallback</code>	<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>
<code>replaceImage</code>	<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

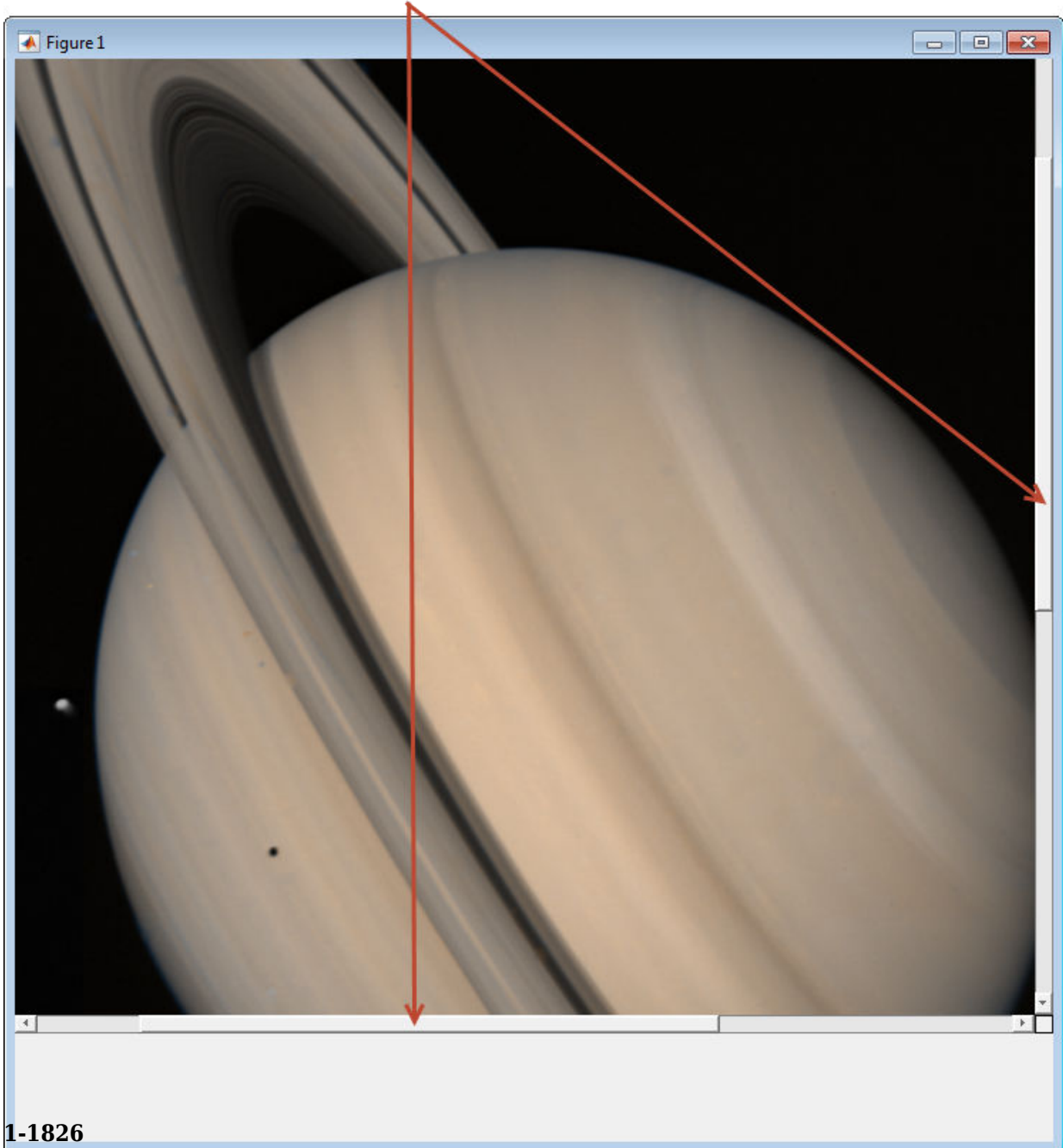
Display an image in a figure. The example suppresses the standard toolbar and menubar in the figure window because these do not work with the scroll panel.

```
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])
```

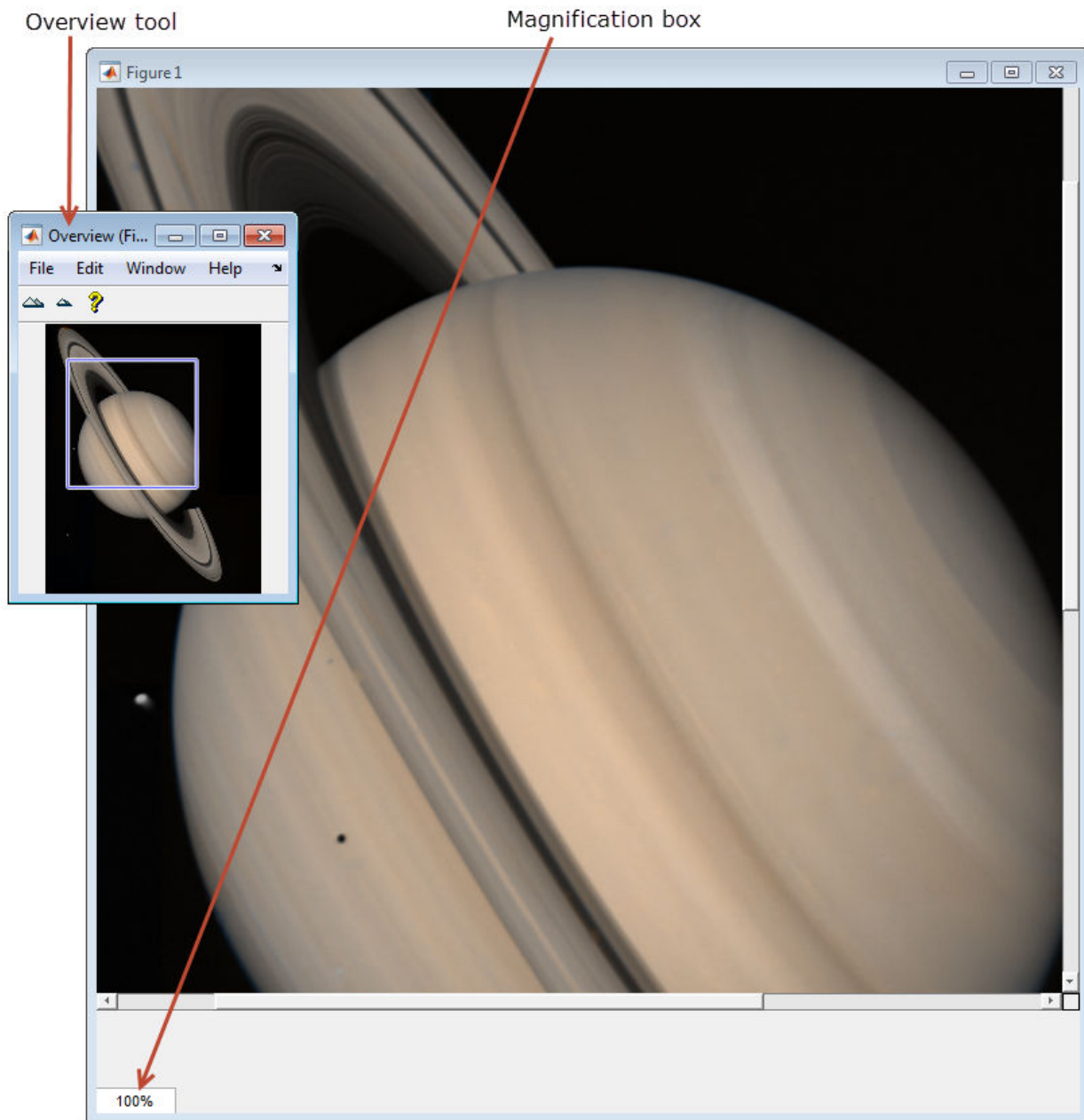
Note addition of scroll bars.





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 =
```

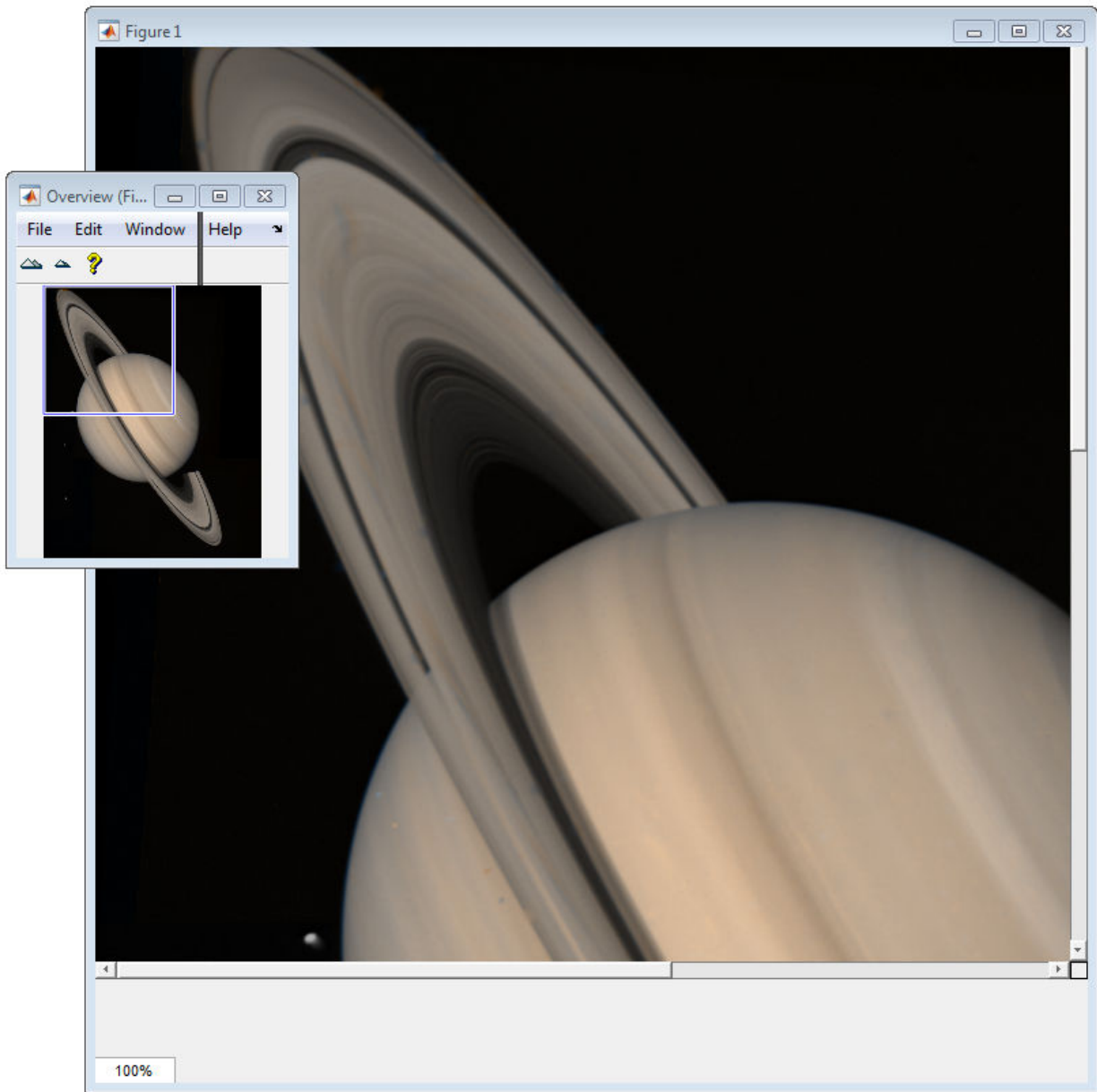
```
1
```

```
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.

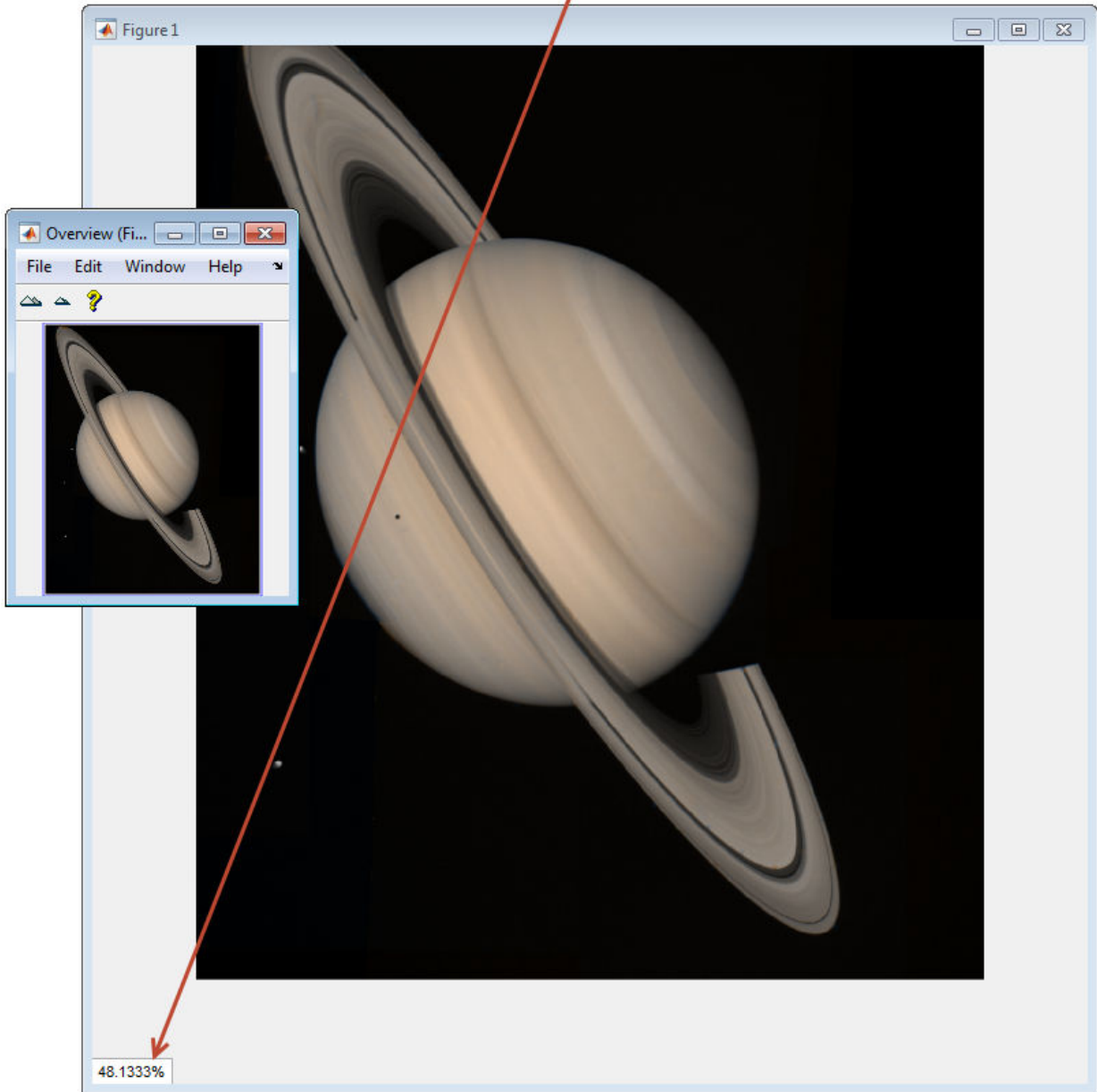
```
api.setVisibleLocation(0.5,0.5)
```



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.

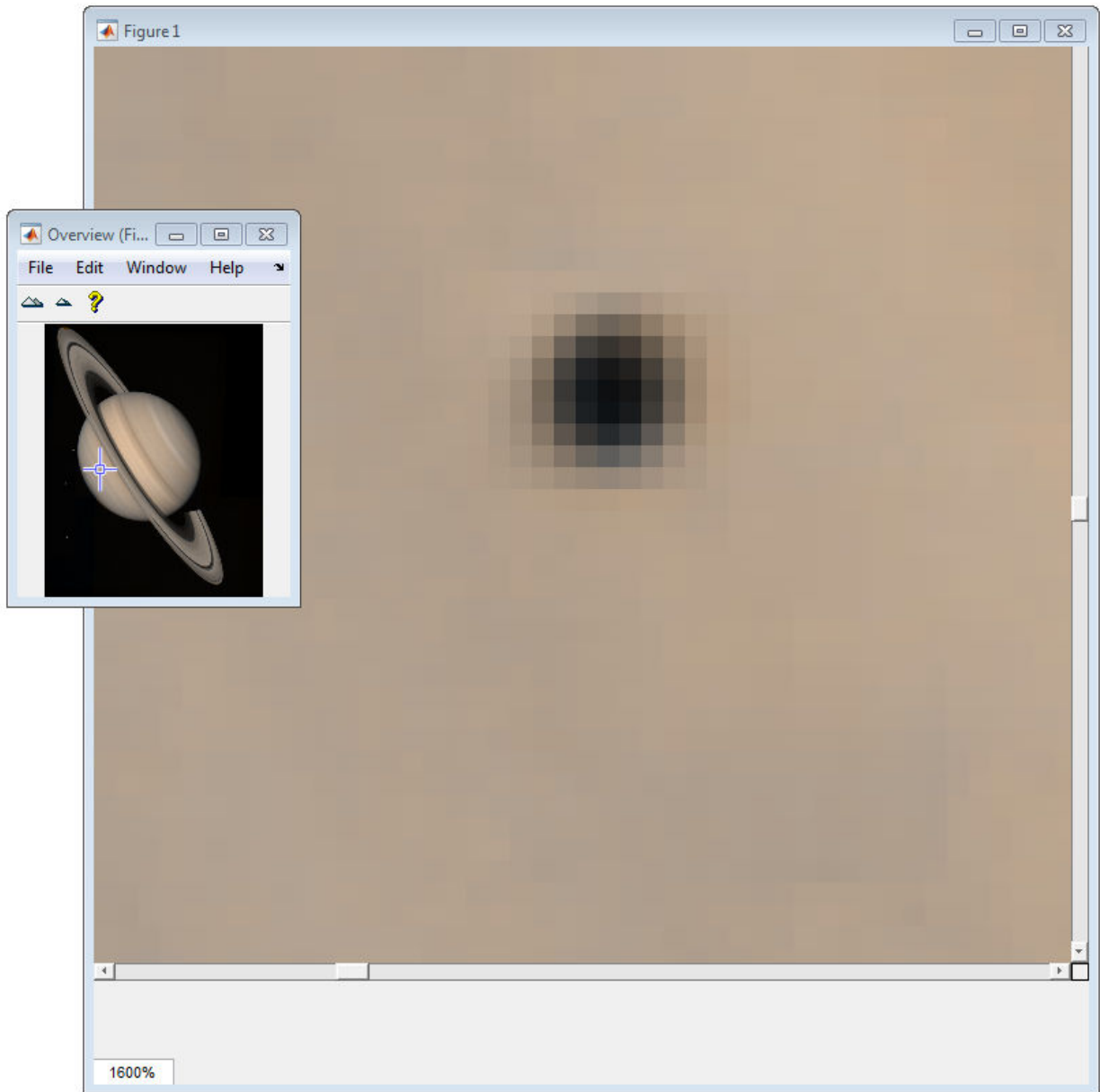
```
api.setMagnification(api.findFitMag())
```

Magnification changed to fit image completely in the figure.



Zoom in to 1600% on the dark spot.

```
api.setMagnificationAndCenter(16,306,800)
```



## See Also

1-1834 [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

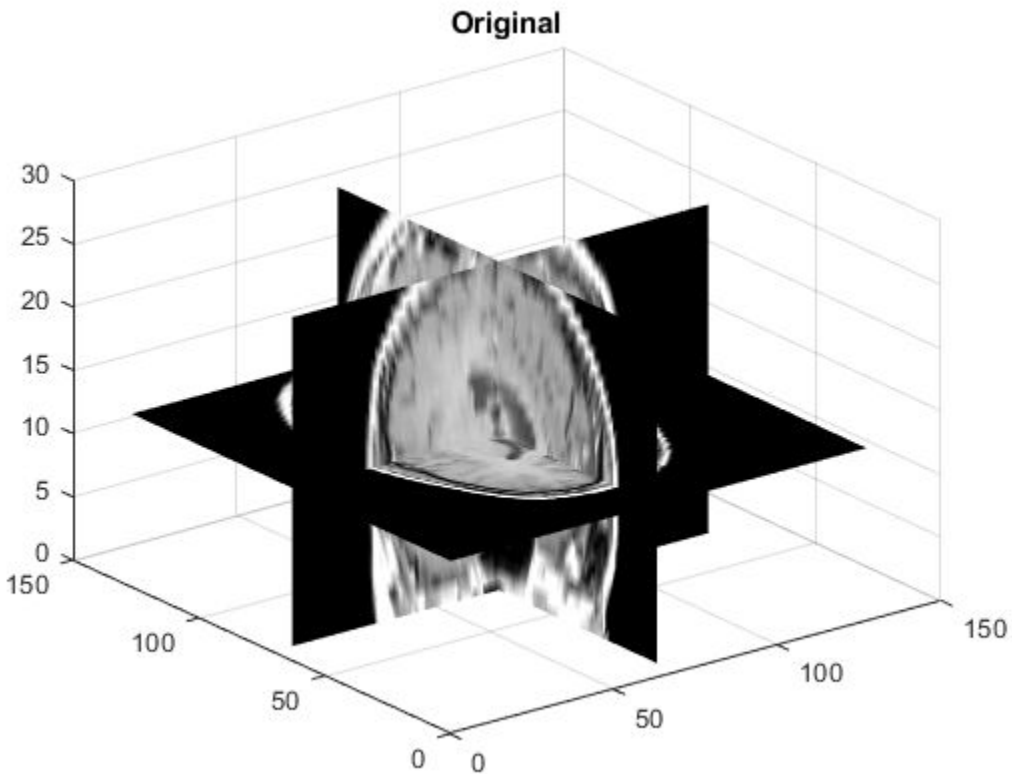
This example segments the brain from MRI data of the human head.

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.

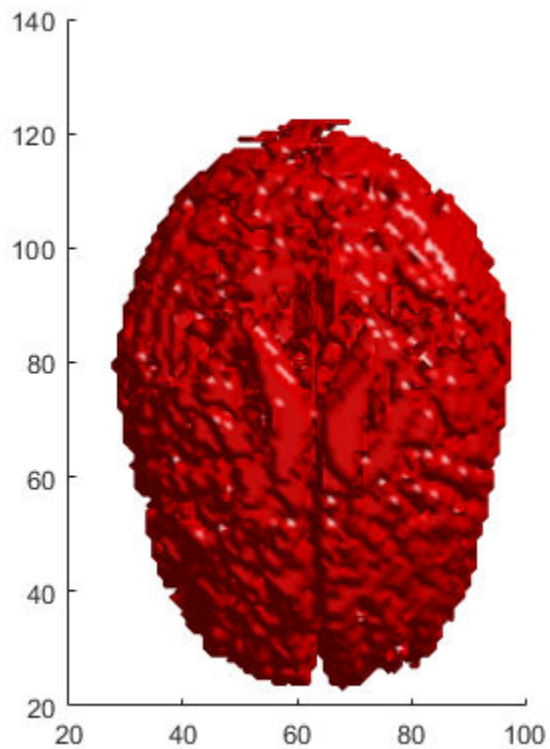
```
W = graydiffweight(V, seedC, seedR, seedP , 'GrayDifferenceCutoff', 25);
```

Segment the image using the weights.

```
thresh = 0.002;  
BW = imseghmm(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 `imsefmm` 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

**Image Segmenter** | `activecontour` | `gradientweight` | `graydiffweight` | `graydist`

**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 (trinary 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(RGB,1),size(RGB,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(RGB,1),size(RGB,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')
```

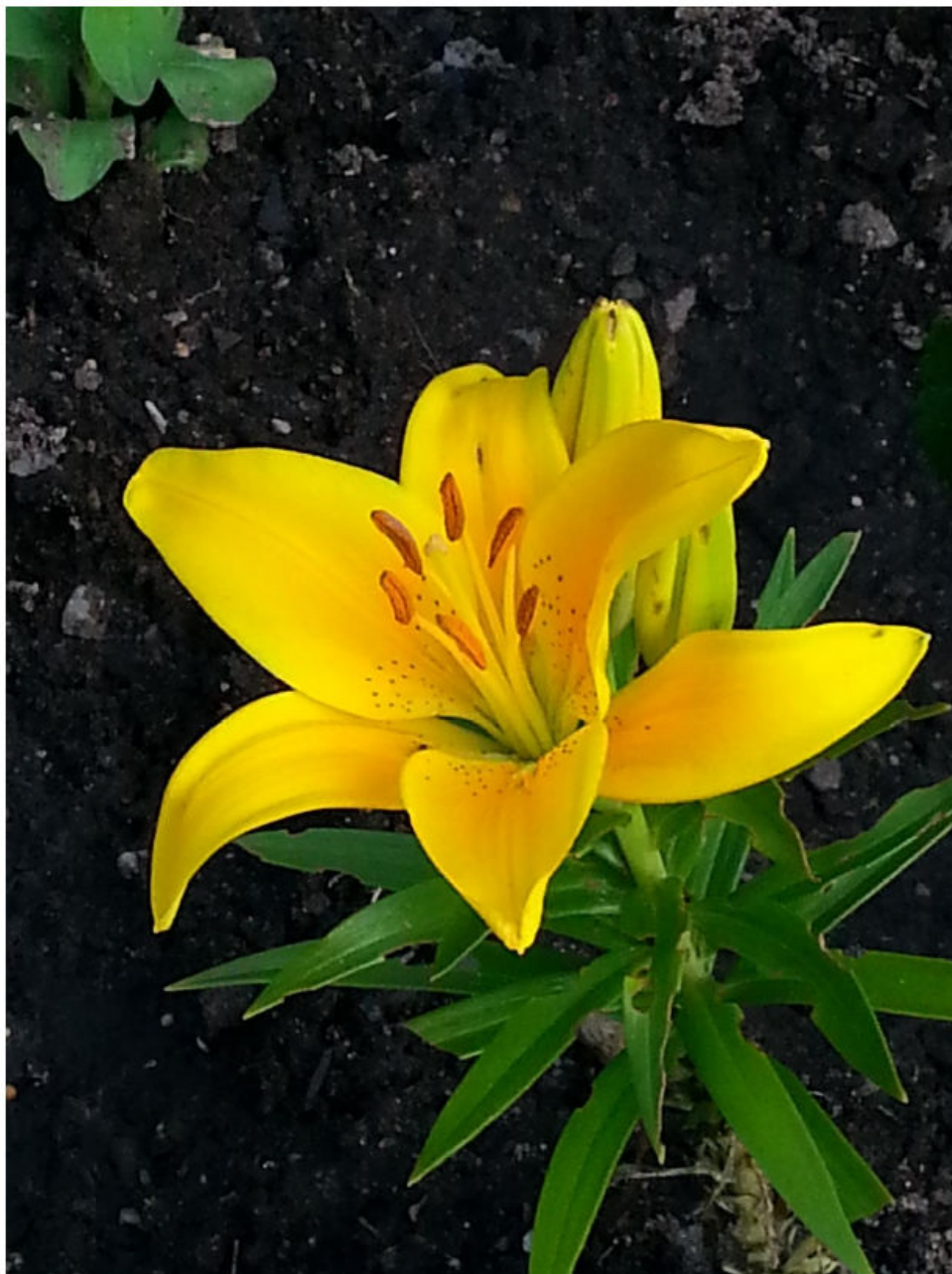
Probability that a pixel belongs to the foreground



## 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.

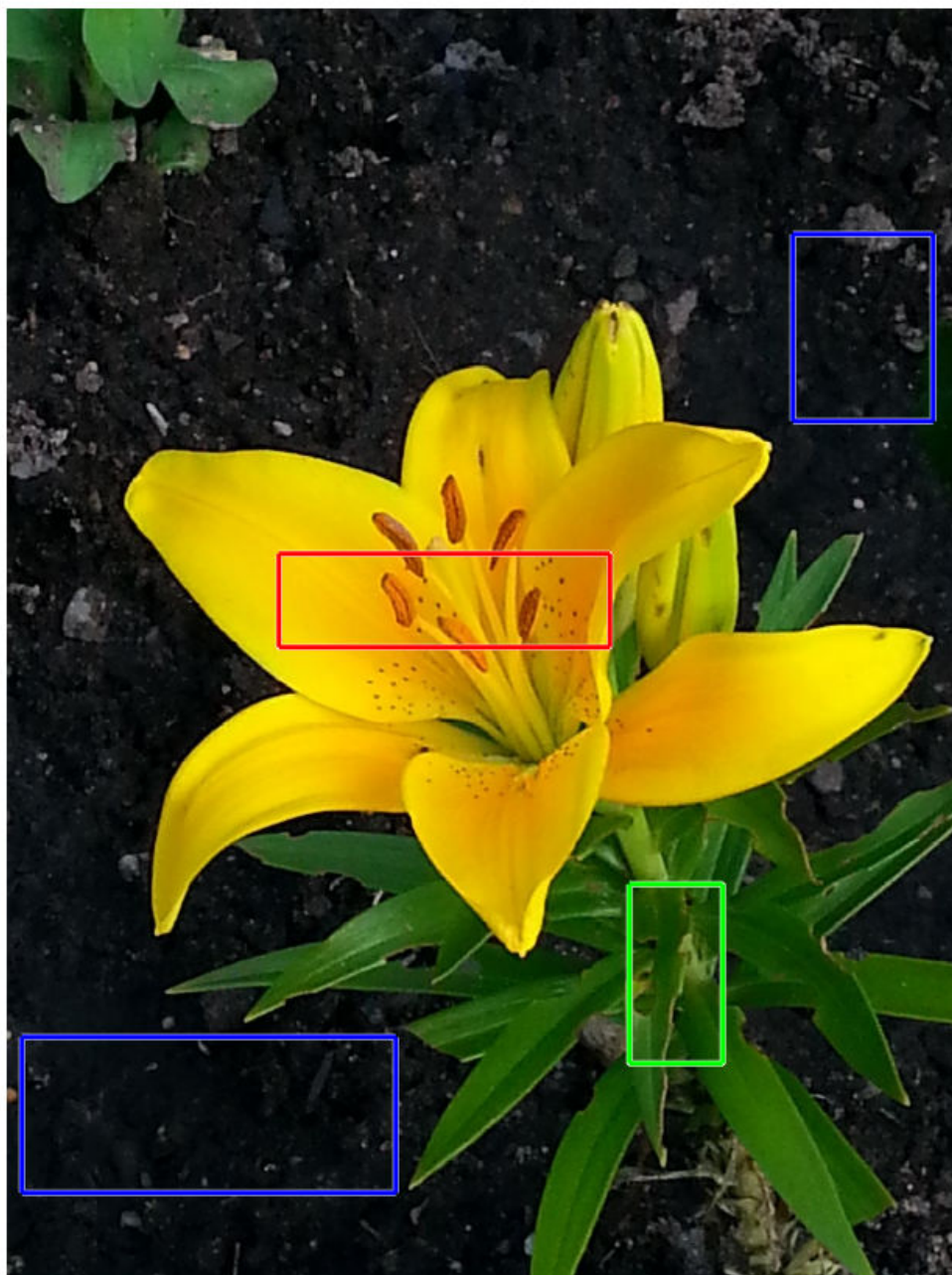
```
region1 = [350 700 425 120]; % [x y w h] format
BW1 = false(size(RGB,1),size(RGB,2));
BW1(region1(2):region1(2)+region1(4),region1(1):region1(1)+region1(3)) = true;

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 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] on page 1-1864).

## **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**

**Color Thresholder** | `activecontour` | `imseggmm` | `visboundaries`

**Introduced in R2015a**

# imsegkmeans

K-means clustering based image segmentation

## Syntax

```
L = imsegkmeans(I,k)
[L,centers] = imsegkmeans(I,k)
L = imsegkmeans(I,k,Name,Value)
```

## Description

`L = imsegkmeans(I,k)` segments image `I` into `k` clusters by performing k-means clustering and returns the segmented labeled output in `L`.

`[L,centers] = imsegkmeans(I,k)` also returns the cluster centroid locations, `centers`.

`L = imsegkmeans(I,k,Name,Value)` uses name-value arguments to control aspects of the k-means clustering algorithm.

## Examples

### Segment Grayscale Image using k-Means Clustering

Read an image into the workspace.

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

**Original Image**



Segment the image into three regions using k-means clustering.

```
[L,Centers] = imsegkmeans(I,3);  
B = labeloverlay(I,L);  
imshow(B)  
title('Labeled Image')
```



**Labeled Image**

### **Improve k-Means Segmentation Using Texture and Spatial Information**

Read an image into the workspace. Reduce the image size to make the example run more quickly.

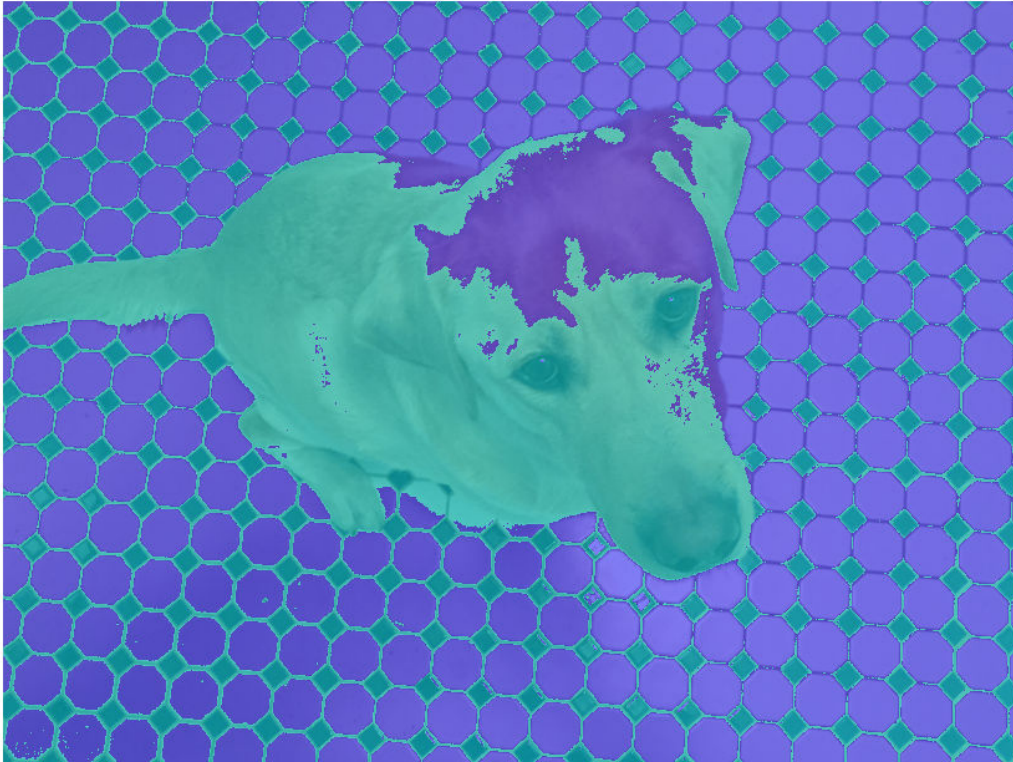
```
RGB = imread('kobi.png');  
RGB = imresize(RGB,0.5);  
imshow(RGB)
```



Segment the image into two regions using k-means clustering.

```
L = imsegkmeans(RGB,2);  
B = labeloverlay(RGB,L);  
imshow(B)  
title('Labeled Image')
```

Labeled Image



Several pixels are mislabeled. The rest of the example shows how to improve the k-means segmentation by supplementing the information about each pixel.

Supplement the image with information about the texture in the neighborhood of each pixel. To obtain the texture information, filter a grayscale version of the image with a set of Gabor filters.

Create a set of 24 Gabor filters, covering 6 wavelengths and 4 orientations.

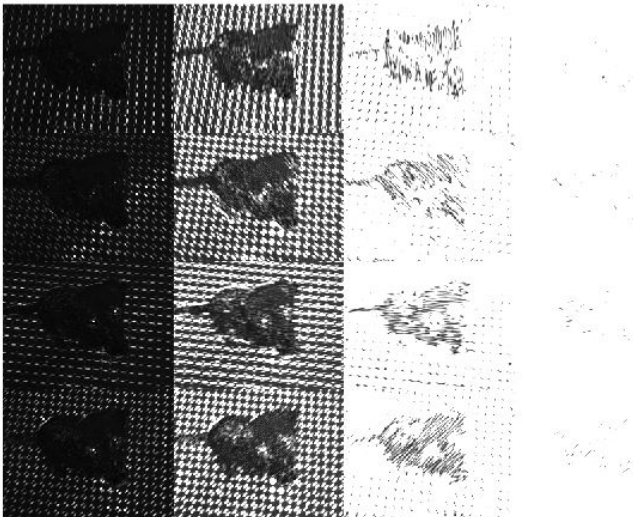
```
wavelength = 2.^(0:5) * 3;  
orientation = 0:45:135;  
g = gabor(wavelength,orientation);
```

Convert the image to grayscale.

```
I = rgb2gray(im2single(RGB));
```

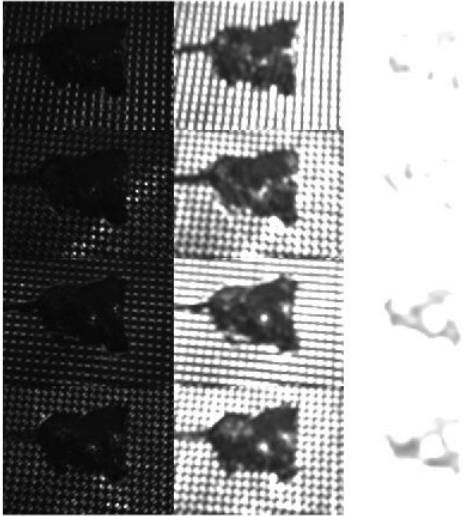
Filter the grayscale image using the Gabor filters. Display the 24 filtered images in a montage.

```
gabormag = imgaborfilt(I,g);  
montage(gabormag, 'Size', [4 6])
```



Smooth each filtered image to remove local variations. Display the smoothed images in a montage.

```
for i = 1:length(g)  
    sigma = 0.5*g(i).Wavelength;  
    gabormag(:,:,i) = imgaussfilt(gabormag(:,:,i),3*sigma);  
end  
montage(gabormag, 'Size', [4 6])
```



Supplement the information about each pixel with spatial location information. This additional information allows the k-means clustering algorithm to prefer groupings that are close together spatially.

Get the x and y coordinates of all pixels in the input image.

```
nrows = size(RGB,1);
ncols = size(RGB,2);
[X,Y] = meshgrid(1:ncols,1:nrows);
```

Concatenate the intensity information, neighborhood texture information, and spatial information about each pixel.

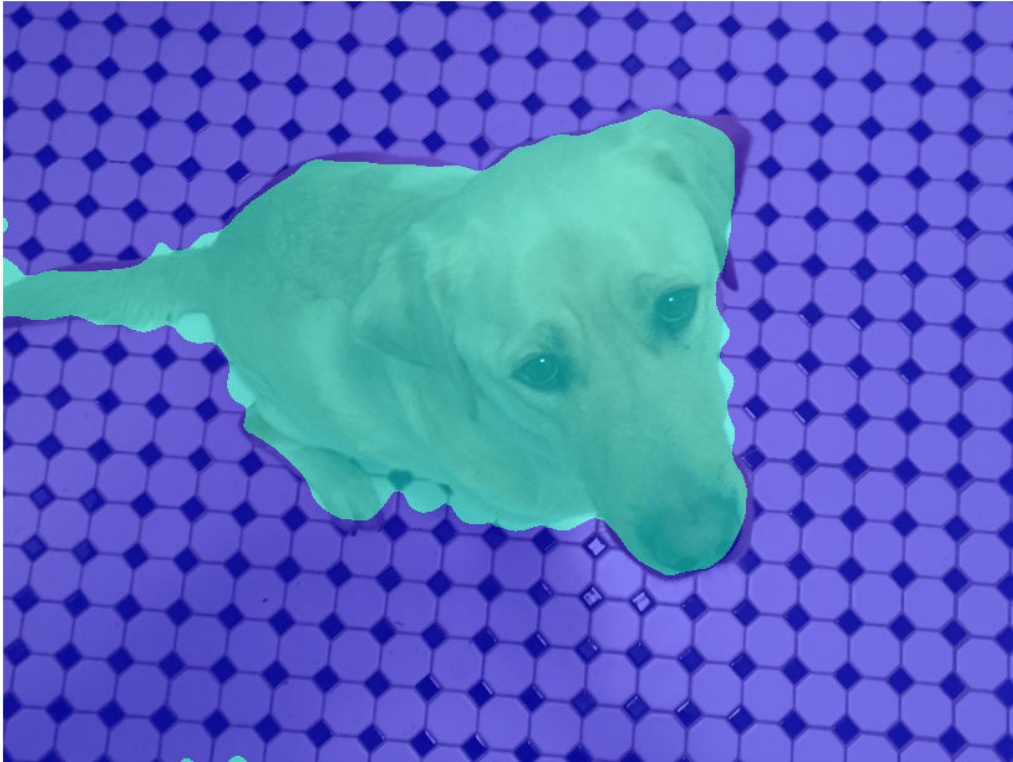
For this example, the feature set includes intensity image *I* instead of the original color image, *RGB*. The color information is omitted from the feature set because the yellow color of the dog's fur is similar to the yellow hue of the tiles. The color channels do not provide enough distinct information about the dog and the background to make a clean segmentation.

```
featureSet = cat(3,I,gabormag,X,Y);
```

Segment the image into two regions using k-means clustering with the supplemented feature set.

```
L2 = imsegkmeans(featureSet,2,'NormalizeInput',true);  
C = labeloverlay(RGB,L2);  
imshow(C)  
title('Labeled Image with Additional Pixel Information')
```

Labeled Image with Additional Pixel Information



## Compress Color Image Using k-Means Segmentation

Read an image into the workspace.

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

Original Image



Segment the image into 50 regions by using k-means clustering. Return the label matrix  $L$  and the cluster centroid locations  $C$ . The cluster centroid locations are the RGB values of each of the 50 colors.

```
[L,C] = imsegkmeans(I,50);
```

Convert the label matrix into an RGB image. Specify the cluster centroid locations,  $C$ , as the colormap for the new image.

```
J = label2rgb(L,im2double(C));
```

Display the quantized image.

```
imshow(J)  
title('Color Quantized Image')
```

**Color Quantized Image**



Write the original and compressed images to file. The quantized image file is approximate one quarter the size of the original image file.



```
imwrite(I, 'peppersOriginal.png');  
imwrite(J, 'peppersQuantized.png');
```

## Input Arguments

### **I** — Image to segment

2-D grayscale image | 2-D color image | 2-D multispectral image

Image to segment, specified as a 2-D grayscale image, 2-D color image, or 2-D multispectral image.

Data Types: `single` | `int8` | `int16` | `uint8` | `uint16`

### **k** — Number of clusters

positive integer

Number of clusters to create, specified as a positive integer.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `L = imsegkmeans(I,5, 'NumAttempts',5);`

### **NormalizeInput** — Normalize input data

`true` (default) | `false`

Normalize input data to zero mean and unit variance, specified as the comma-separated pair consisting of `'NormalizeInput'` and `true` or `false`. If you specify `true`, then `imsegkmeans` normalizes each channel of the input individually.

### **NumAttempts** — Number of times to repeat the clustering process

3 (default) | positive integer

Number of times to repeat the clustering process using new initial cluster centroid positions, specified as the comma-separated pair consisting of `'NumAttempts'` and a positive integer.

**MaxIterations — Maximum number of iterations**

100 (default) | positive integer

Maximum number of iterations, specified as the comma-separated pair consisting of 'MaxIterations' and a positive integer.

**Threshold — Accuracy threshold**

1e-4 (default) | positive number

Accuracy threshold, specified as the comma-separated pair consisting of 'Threshold' and a positive number. The algorithm stops when each of the cluster centers move less than the threshold value in consecutive iterations.

## Output Arguments

**L — Label matrix**

matrix of positive integers

Label matrix, specified as a matrix of positive integers. Pixels with label 1 belong to the first cluster, label 2 belong to the second cluster, and so on for each of the  $k$  clusters.  $L$  has the same first two dimensions as image  $I$ . The class of  $L$  depends on number of clusters.

Class of L	Number of Clusters
'uint8'	$k \leq 255$
'uint16'	$256 \leq k \leq 65535$
'uint32'	$65536 \leq k \leq 2^{32}-1$
'double'	$2^{32} \leq k$

**centers — Cluster centroid locations**

numeric matrix

Cluster centroid locations, returned as a numeric matrix of size  $k$ -by- $c$ , where  $k$  is the number of clusters and  $c$  is the number of channels. `centers` is the same class as the image  $I$ .

## Tips

- The function yields reproducible results. The output will not vary in multiple runs given the same input arguments.

## References

- [1] Arthur, D. and S. Vassilvitskii. "k-means++: The Advantages of Careful Seeding." *SODA '07: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. New Orleans, LA, January 2007, pp. 1027-1035.

## See Also

### Apps

**Image Segmenter**

### Functions

`gabor` | `imgaborfilt` | `imsegkmeans3` | `label2rgb` | `labelmatrix` | `labeloverlay` | `lazysnapping` | `superpixels` | `watershed`

## Topics

"Color-Based Segmentation Using K-Means Clustering"

**Introduced in R2018b**

## imsegkmeans3

K-means clustering based volume segmentation

### Syntax

```
L = imsegkmeans3(V,k)
[L,centers] = imsegkmeans3(V,k)
L = imsegkmeans3(V,k,Name,Value)
```

### Description

`L = imsegkmeans3(V,k)` segments volume `V` into `k` clusters by performing k-means clustering and returns the segmented labeled output in `L`.

`[L,centers] = imsegkmeans3(V,k)` also returns the cluster centroid locations, `centers`.

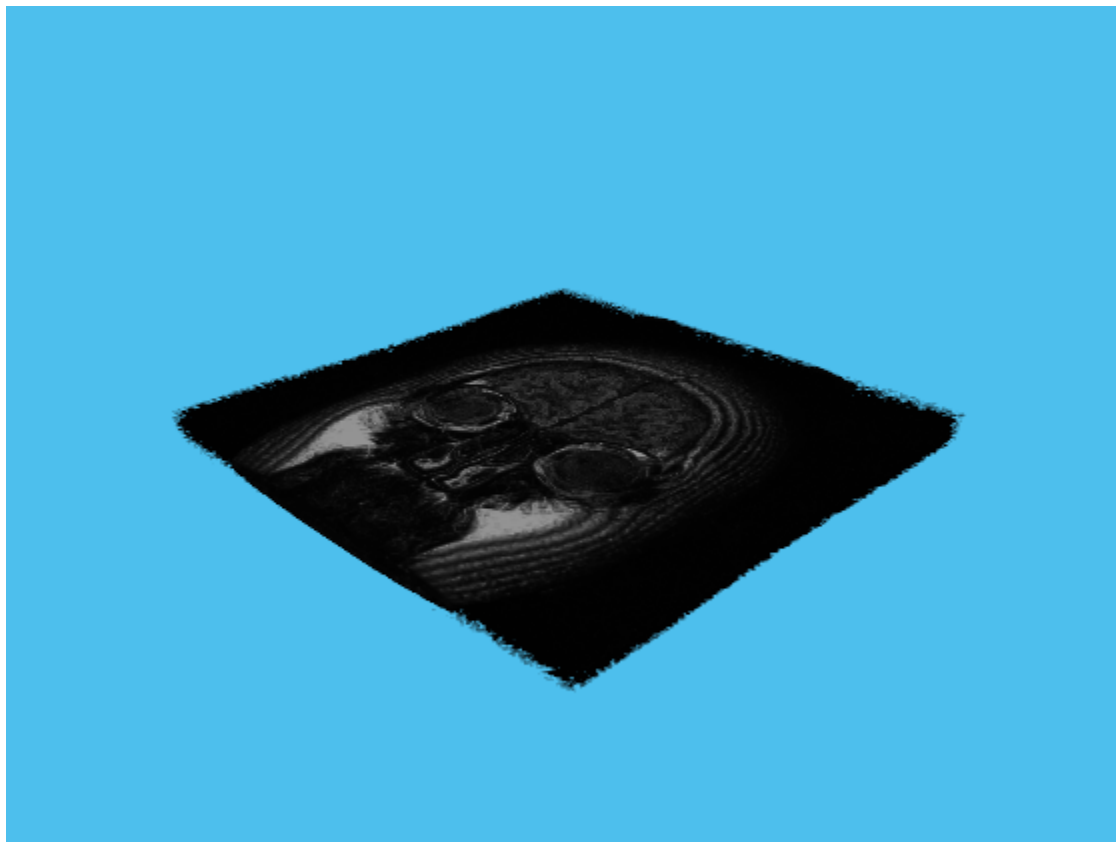
`L = imsegkmeans3(V,k,Name,Value)` uses name-value pairs to control aspects of the k-means clustering algorithm.

### Examples

#### Segment Volume Using k-Means Clustering

Load a 3-D grayscale MRI volume.

```
load mrystack
volshow(mrystack);
```

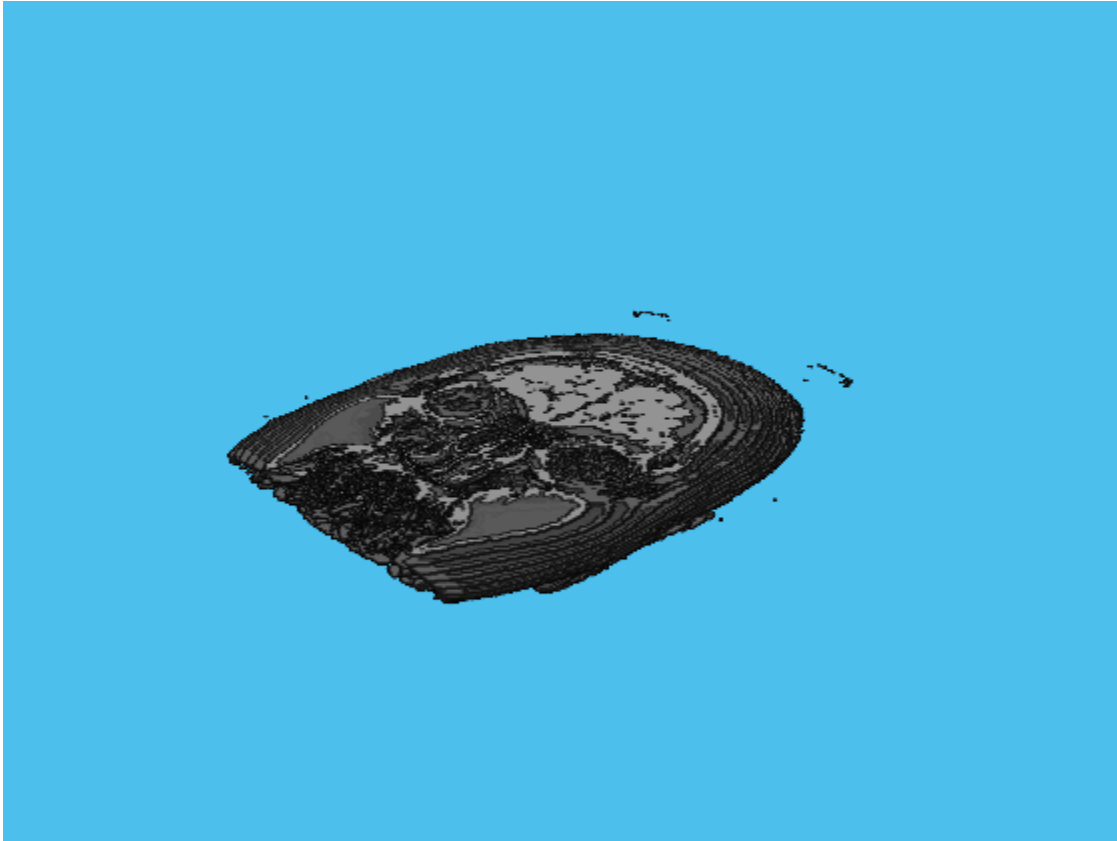


Segment the volume into three clusters.

```
L = imsegkmeans3(mristack,3);
```

Display the segmented volume in a figure window. You can explore slices of the segmented volume by using the Volume Viewer app.

```
figure  
volshow(L);
```



## Input Arguments

### **V — Volume to segment**

3-D grayscale volume | 3-D multispectral volume

Volume to segment, specified as a 3-D grayscale volume of size  $m$ -by- $n$ -by- $p$  or a 3-D multispectral volume of size  $m$ -by- $n$ -by- $p$ -by- $c$ , where  $p$  is the number of planes and  $c$  is number of channels.

---

**Note** `imsegkmeans2` treats 2-D color images like 3-D volumes of size  $m$ -by- $n$ -by-3. If you want 2-D behavior, use `imsegkmeans` instead.

---

Data Types: `single` | `int8` | `int16` | `uint8` | `uint16`

### **k — Number of clusters**

positive number

Number of clusters to create, specified as a numeric scalar.

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `L = imsegkmeans3(V,5,'NumAttempts',5);`

### **NormalizeInput — Normalize input data**

`true` (default) | `false`

Normalize input data to zero mean and unit variance, specified as the comma-separated pair consisting of `'NormalizeInput'` and `true` or `false`. If you specify `true`, then `imsegkmeans3` normalizes each channel of the input individually.

### **NumAttempts — Number of times to repeat the clustering process**

3 (default) | positive integer

Number of times to repeat the clustering process using new initial cluster centroid positions, specified as the comma-separated pair consisting of `'NumAttempts'` and a positive integer.

### **MaxIterations — Maximum number of iterations**

100 (default) | positive integer

Maximum number of iterations, specified as the comma-separated pair consisting of `'MaxIterations'` and a positive integer.

### **Threshold — Accuracy threshold**

1e-4 (default) | positive number

Accuracy threshold, specified as the comma-separated pair consisting of 'Threshold' and a positive number. The algorithm stops when each of the cluster centers move less than the threshold value in consecutive iterations.

## Output Arguments

### **L** — Label matrix

matrix of positive integers

Label matrix, specified as a matrix of positive integers. Pixels with label 1 belong to the first cluster, label 2 belong to the second cluster, and so on for each of the  $k$  clusters. **L** has the same first three dimensions as volume **V**. The class of **L** depends on number of clusters.

Class of L	Number of Clusters
'uint8'	$k \leq 255$
'uint16'	$256 \leq k \leq 65535$
'uint32'	$65536 \leq k \leq 2^{32}-1$
'double'	$2^{32} \leq k$

### **centers** — Cluster centroid locations

numeric matrix

Cluster centroid locations, returned as a numeric matrix of size  $k$ -by- $c$ , where  $k$  is the number of clusters and  $c$  is the number of channels. **centers** is the same class as the image **I**.

## Tips

- The function yields reproducible results. The output will not vary in multiple runs given the same input arguments.

## References

- [1] Arthur, D. and S. Vassilvitskii. "k-means++: The Advantages of Careful Seeding." *SODA '07: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. New Orleans, LA, January 2007, pp. 1027-1035.



## See Also

### Apps

Volume Viewer

### Functions

imsegkmeans | lazysnapping | superpixels3 | watershed

**Introduced in R2018b**

## imsharpen

Sharpen image using unsharp masking

### Syntax

```
B = imsharpen(A)
B = imsharpen(A,Name,Value)
```

### Description

`B = imsharpen(A)` sharpens the grayscale or truecolor (RGB) input image `A` by using the unsharp masking on page 1-1890 method.

`B = imsharpen(A,Name,Value)` uses name-value pairs to control aspects of the unsharp masking.

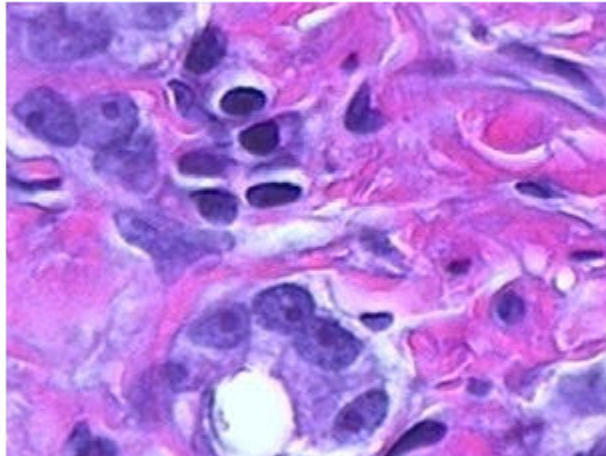
### Examples

#### Sharpen Image

Read an image into the workspace and display it.

```
a = imread('hestain.png');
imshow(a)
title('Original Image');
```

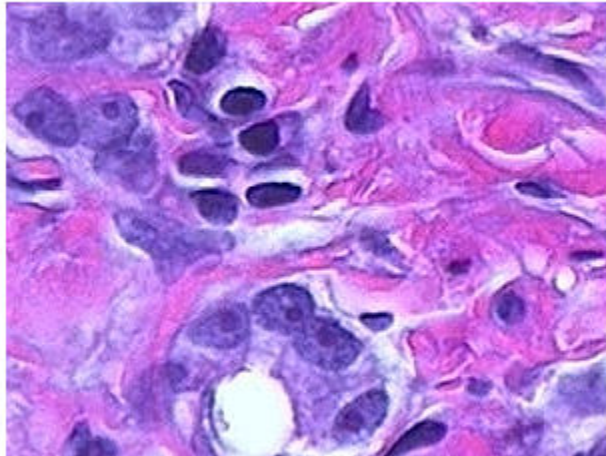
**Original Image**



Sharpen the image using the `imsharpen` function and display it.

```
b = imsharpen(a);  
figure, imshow(b)  
title('Sharpened Image');
```

**Sharpened Image**

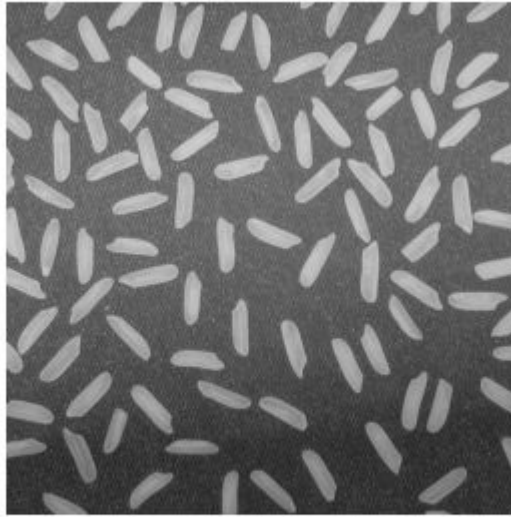


### **Control the Amount of Sharpening at the Edges**

Read an image into the workspace and display it.

```
a = imread('rice.png');  
imshow(a), title('Original Image');
```

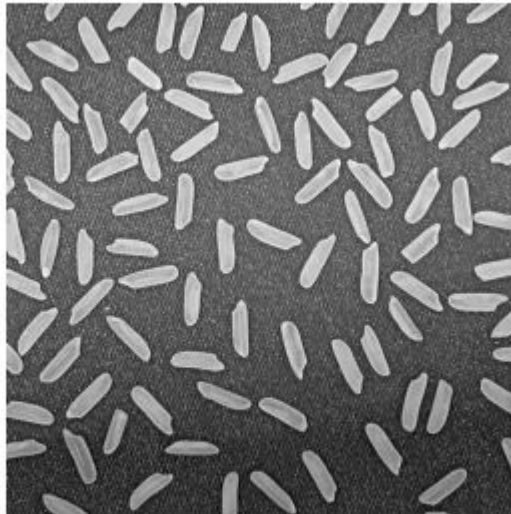
Original Image



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 — Image to be sharpened**

grayscale image | RGB image

Image to be sharpened, specified as a grayscale or RGB image.

If A is a truecolor (RGB) image, then `imsharpen` converts the image to the  $L^*a^*b^*$  color space, applies sharpening to the  $L^*$  channel only, and then converts the image back to the RGB color space 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 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) | positive number

Standard deviation of the Gaussian lowpass filter, specified as a positive number. 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 scalar

Strength of the sharpening effect, specified as a numeric scalar. 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**

numeric array

Sharpened image, returned as a numeric array of the same size and class as the input image A.

## 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

fspecial | imadjust | imcontrast

**Introduced in R2013a**



# imshow

Display image

## Syntax

```
imshow(I)
imshow(I,[low high])
imshow(I,[])
imshow(RGB)
imshow(BW)
imshow(X,map)
imshow(filename)
imshow(___,Name,Value)

himage = imshow( ___ )

imshow(I,RI)
imshow(X,RX,map)
imshow(gpuarrayIM, ___ )
```

## Description

`imshow(I)` displays the grayscale image `I` in a figure. `imshow` optimizes figure, axes, and image object properties for image display.

`imshow(I,[low high])` displays the grayscale image `I`, specifying the display range as a two-element vector, `[low high]`. For more information, see the `DisplayRange` parameter.

`imshow(I,[])` displays the grayscale image `I`, scaling the display based on the range of pixel values in `I`. `imshow` uses `[min(I(:)) max(I(:))]` as the display range. `imshow` displays the minimum value in `I` as black and the maximum value as white. For more information, see the `DisplayRange` parameter.

`imshow(RGB)` displays the truecolor image `RGB` in a figure.

`imshow(BW)` displays the binary image `BW` in a figure. For binary images, `imshow` displays pixels with the value 0 (zero) as black and 1 as white.

`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, 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(____,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, RGB, Indexed, or Binary Image

Display a grayscale, RGB (truecolor), indexed or binary image using `imshow`. MATLAB® includes a TIF file, named `corn.tif`, that contains three images: a grayscale image, an indexed image, and an RGB image. This example creates a binary image from the grayscale image.

### Display a Grayscale Image

Read the grayscale image from the `corn.tif` file into the MATLAB workspace. The grayscale version of the image is the third image in the file.

```
corn_gray = imread('corn.tif',3);
```

Display the grayscale image using `imshow`.

```
imshow(corn_gray)
```



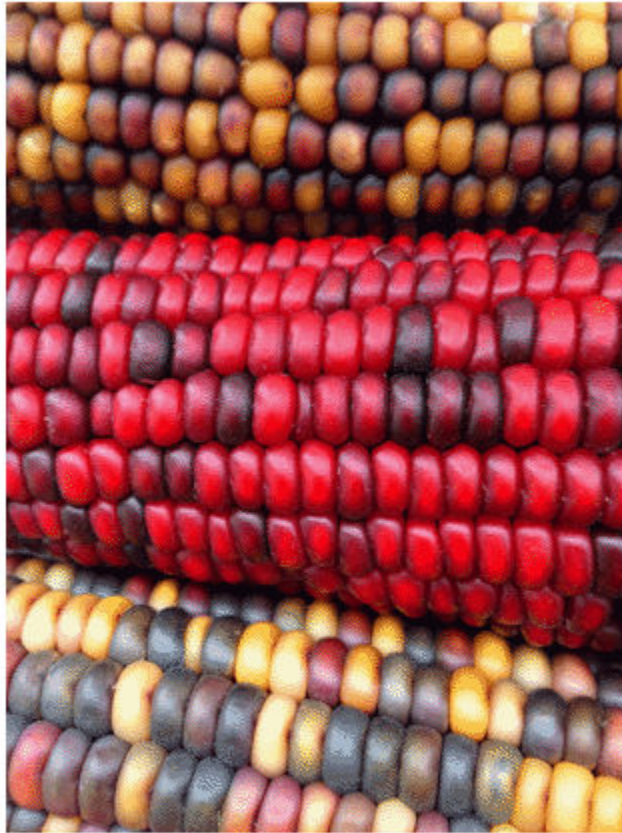
### Display an Indexed Image

Read the indexed image from the `corn.tif` file into the MATLAB workspace. The indexed version of the image is the first image in the file.

```
[corn_indexed,map] = imread('corn.tif',1);
```

Display the indexed image using `imshow`.

```
imshow(corn_indexed,map)
```



### **Display an RGB Image**

Read the RGB image from the `corn.tif` file into the MATLAB workspace. The RGB version of the image is the second image in the file.

```
[corn_rgb] = imread('corn.tif',2);  
Display the RGB image using imshow.  
imshow(corn_rgb)
```



## Display a Binary Image

Read the grayscale image from the `corn.tif` file into the MATLAB workspace and use thresholding to convert it into a binary image. The grayscale version of the image is the third image in the file.

```
[corn_gray] = imread('corn.tif',3);
```

Determine the mean value of pixels in the grayscale image.

```
meanIntensity = mean(corn_gray(:));
```

Create a binary image by thresholding, using the mean intensity value as the threshold.

```
corn_binary = corn_gray > meanIntensity;
```

Display the binary image using `imshow`.

```
imshow(corn_binary)
```



### Display Image from File

Display an image stored in a file.

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

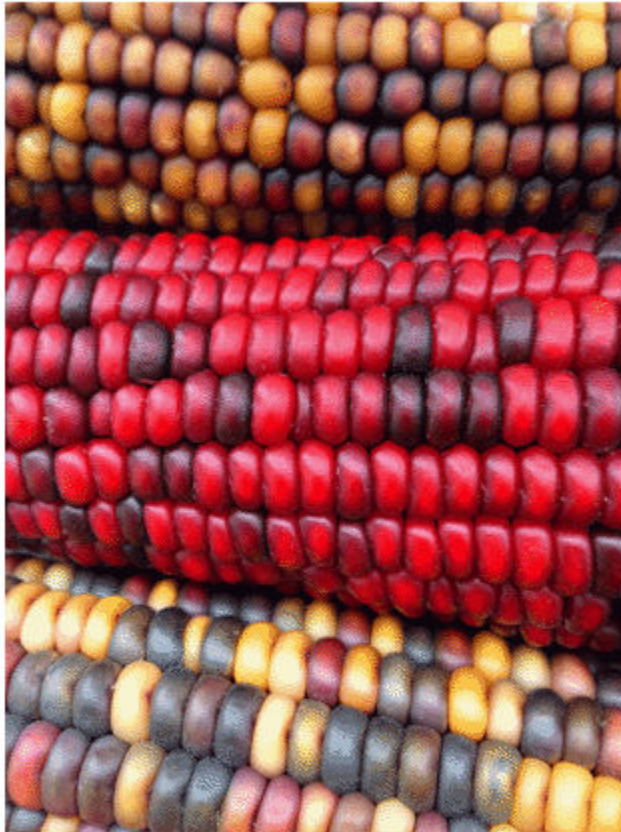


### **Change Colormap of Displayed Image**

Read a sample indexed image, `corn.tif`, into the workspace, and then display it.

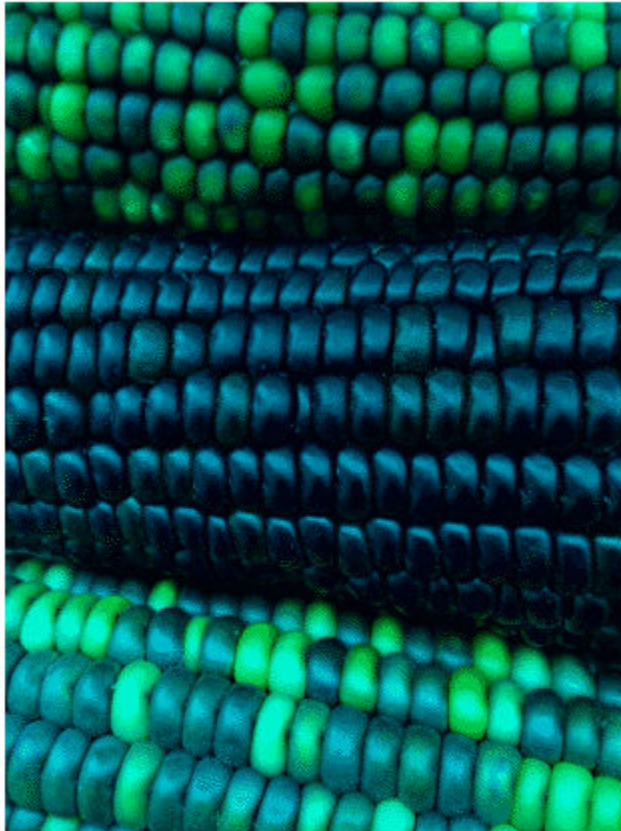
```
[X,map] = imread('corn.tif');  
imshow(X,map)
```





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)
```



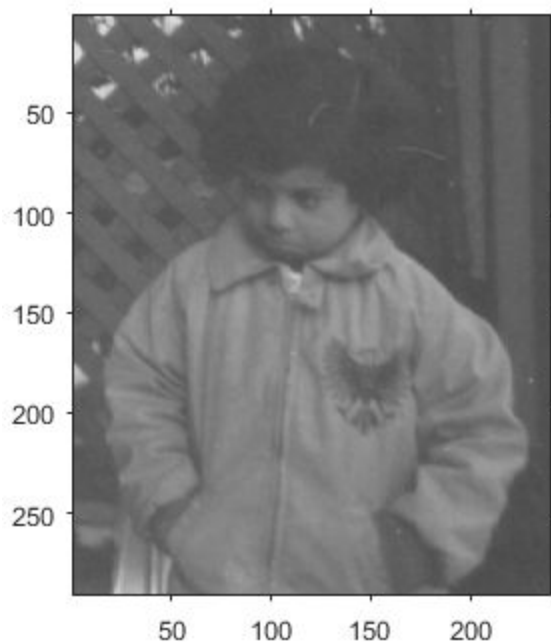
### **Display Image Using Associated Spatial Referencing Object**

Read image into the workspace.

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

Display the image. Note the axes limits reflect the size of the image.

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

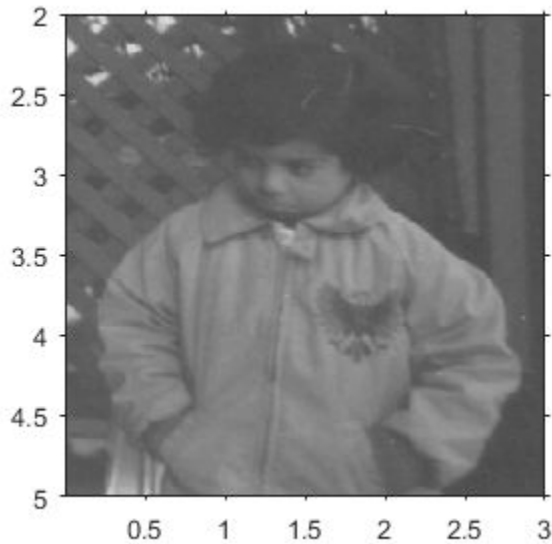


Create a spatial referencing object associated with the image. Use the referencing object to set the x- and y-axis limits in the world coordinate system.

```
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-axis limits.

```
figure; 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 grayscale image

matrix

Input grayscale image, specified as a matrix. A grayscale image can be any numeric data type.

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

### **RGB — Input truecolor image**

*m*-by-*n*-by-3 array

Input truecolor image, specified as an *m*-by-*n*-by-3 array.

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

### **BW — Input binary image**

matrix

Input binary image, specified as a matrix.

Data Types: `logical`

### **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 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'`.

In MATLAB Online™, `'InitialMagnification'` is set to `'fit'` and cannot be changed.

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 1-1898.
- 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.

## Extended Capabilities

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

### Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

## **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 composite RGB image showing A and B overlaid in different color bands. To choose another type of visualization of the two images, use the `method` argument. 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. By default, `imshowpair` scales the intensity values of A and B independently from each other. `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.

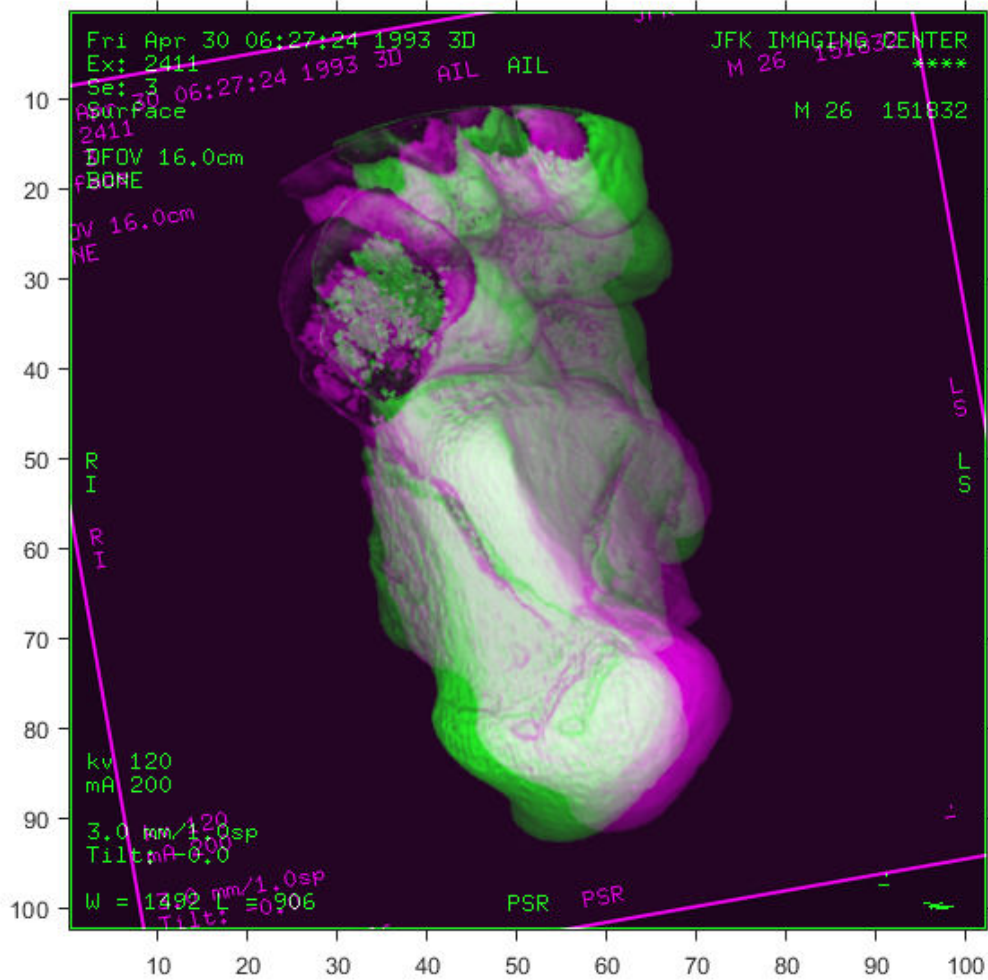
```
A = dicomread('CT-MONO2-16-ankle.dcm');  
B = imrotate(A,10,'bicubic','crop');  
B = B * 0.2;
```

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.

Data Types: char | string

## 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 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.

Data Types: `char` | `string`

## 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

`imfuse` | `imregister` | `imshow` | `imtransform`

**Introduced in R2012a**

# imsplit

Split multichannel image into its individual channels

## Syntax

```
[c1,c2,c3,...,ck] = imsplit(I)
```

## Description

`[c1,c2,c3,...,ck] = imsplit(I)` returns a set of  $k$  images representing the individual channels in the  $k$ -channel image `I`.

## Examples

### Split RGB Image into Its Component Channels

Read an RGB image into the workspace and then split the image into its component channels.

```
I = imread('peppers.png');  
[r,g,b] = imsplit(I);  
montage({r,g,b})
```



### **Split Image in HSV Colorspace into Its Component Channels**

Read an RGB image into the workspace.

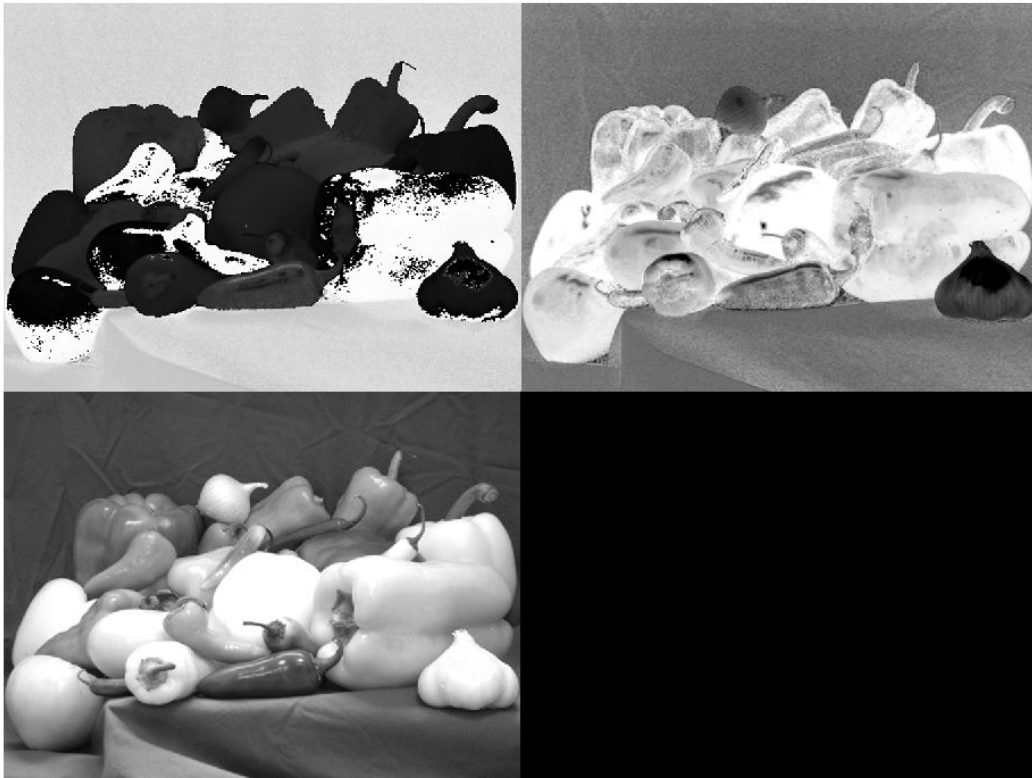
```
rgbImage = imread('peppers.png');
```

Convert the RGB image to the HSV colorspace.

```
hsvImage = rgb2hsv(rgbImage);
```

Split the HSV image into three images, each representing one of its component channels. Display a montage of all the channels.

```
[h,s,v] = imsplit(hsvImage);  
montage({h,s,v})
```



## Input Arguments

**I** — Input image  
*m*-by-*n*-by-*k* array

Input image, specified as an  $m$ -by- $n$ -by- $k$  numeric array.

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

## Output Arguments

**`c1, c2, c3, . . . , ck` — Output images**

numeric matrix for each channel

Output images, returned as  $k$  individual numeric matrices, where  $k$  is the number of channels in the input image. The output images are the same class as the input image.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`cat` | `rgb2gray`

### Topics

“Display Separated Color Channels of an RGB Image”

“Convert from HSV to RGB Color Space”

**Introduced in R2018b**

# 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`.

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 = 2x3 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.

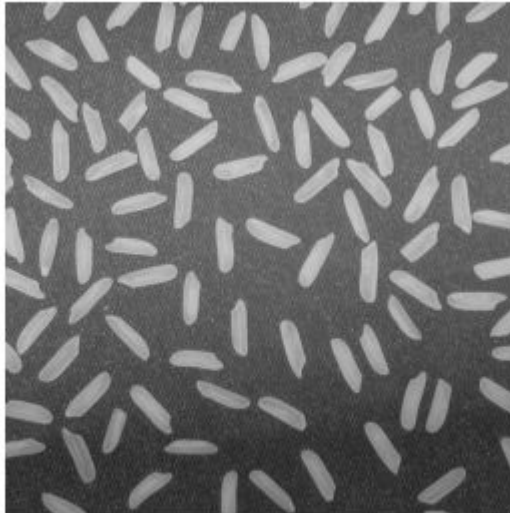
```
background = imopen(I, strel('disk', 15));
```

Subtract the background from the image.

```
J = imsubtract(I, background);
```

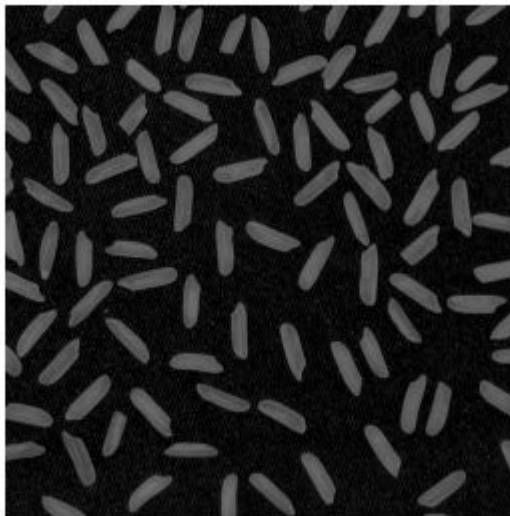
Display the original image and the processed image.

```
imshow(I)
```



```
figure  
imshow(J)
```





### **Subtract a Constant from an Image**

Read an image into the workspace.

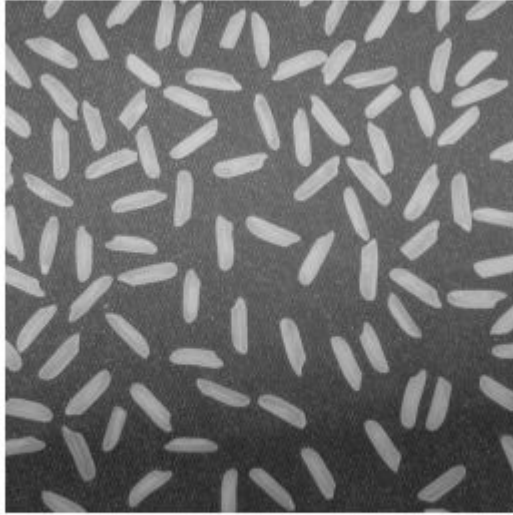
```
I = imread('rice.png');
```

Subtract a constant value from the image.

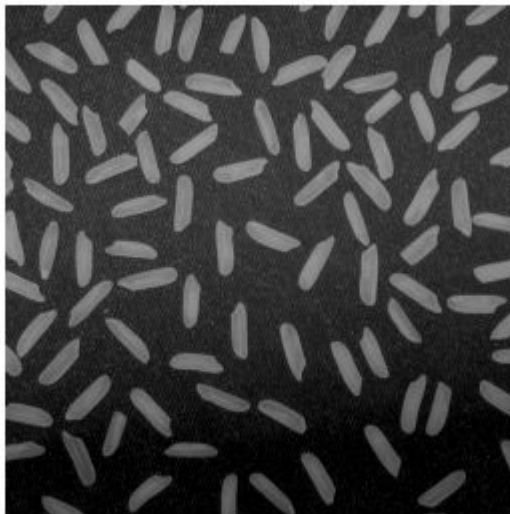
```
J = imsubtract(I,50);
```

Display the original image and the result.

```
imshow(I)
```



```
figure  
imshow(J)
```



## Input Arguments

### **X — First array**

real, nonsparse, numeric or logical array

First array (minuend), specified as a real, nonsparse, numeric or logical array of any dimension.

### **Y — Second array**

real, nonsparse, numeric or logical array

Second array (subtrahend) to be subtracted from X, specified as a real, nonsparse, numeric or logical array. Y either has the same size and class as X, or Y is a scalar of type double.

## Output Arguments

### **Z — Difference**

numeric array

Difference, returned as a numeric array of the same size as *X*. *Z* is the same class as *X* unless *X* is logical, in which case *Z* is data type `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.

### **See Also**

`imabsdiff` | `imadd` | `imcomplement` | `imdivide` | `imlincomb` | `immultiply`

**Introduced before R2006a**

# imtool

Open Image Viewer app

The Image Viewer presents an integrated environment for displaying images and performing common image processing tasks.

The Image Viewer provides all the image display capabilities of `imshow`, which optimizes figure, axes, and image object property settings for image display. The Image Viewer also provides access to several tools for navigating and exploring images, such as the Pixel Region tool, Image Information tool, and the Adjust Contrast tool.

## Syntax

```
imtool
imtool(I)
imtool(I,range)
imtool(RGB)
imtool(BW)
imtool(X,cmap)
imtool(filename)
imtool( ____,Name,Value)
hfigure = imtool( ____ )
imtool close all
```

## 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, using the default display range of the image data type.

`imtool(I,range)` displays the grayscale image `I` in the Image Viewer, specifying the display range as `range`. Pixel values outside the display range are clipped to black or white. Pixel values within the display range are displayed as intermediate shades of gray. The Image Viewer uses the default number of gray levels.

`imshow( RGB )` displays the truecolor image `RGB` in the Image Viewer.

`imshow( BW )` displays the binary image `BW` in the Image Viewer. Pixel values of 0 display as black; pixel values of 1 display as white.

`imshow( X, cmap )` displays the indexed image `X` with colormap `cmap` in the Image Viewer.

`imshow( filename )` displays the image contained in the graphics file `filename` in the Image Viewer.

`imshow( ___, Name, Value )` displays the image, specifying parameters and corresponding values that control various aspects of the image display. Parameter names can be abbreviated, and case does not matter.

`hfigure = imshow( ___ )` returns `hfigure`, a handle to the figure created by the Image Viewer.

`imshow close all` closes all open Image Viewers.

## Examples

### Display Different Types of Images

Display a color image from a file.

```
imshow('board.tif')
```

Display an indexed image.

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

Display a grayscale image.

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

Display a grayscale image, adjusting the display range.

```
h = imshow(I,[0 80]);  
close(h)
```

## Input Arguments

### **I — 2-D grayscale image**

*m*-by-*n* numeric matrix

2-D grayscale image, specified as an *m*-by-*n* numeric matrix.

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

### **range — Display range**

2-element numeric vector | []

Display range of the grayscale image *I*, specified as one of these values.

Value	Description
[low high]	Image Viewer displays pixels with the value <code>low</code> (and any value less than <code>low</code> ) as black. Image Viewer displays pixels with the value <code>high</code> (and any value greater than <code>high</code> ) as white.
[]	Image Viewer automatically sets the display range to <code>[min(I(:)) max(I(:))]</code> . The minimum value in <i>I</i> is displayed as black, and the maximum value is displayed as white.

### **RGB — 2-D RGB image**

*m*-by-*n*-by-3 numeric matrix

2-D RGB image, specified as an *m*-by-*n*-by-3 numeric matrix.

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

### **BW — 2-D binary image**

*m*-by-*n* logical matrix

2-D binary image, specified as an *m*-by-*n* logical matrix.

Data Types: `logical`

### **X — 2-D indexed image**

*m*-by-*n* numeric matrix

2-D indexed image, specified as an *m*-by-*n* numeric matrix.

Data Types: `single` | `double` | `uint8` | `logical`

## **cmap** — Color map

*c*-by-3 numeric matrix

Color map associated with indexed image *X*, specified as a *c*-by-3 numeric matrix containing the RGB values of *c* colors.

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

## **filename** — File name

string scalar | character vector

File name of the graphics file containing the image, specified as a character vector. 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.

Data Types: `char` | `string`

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `imshow(I, 'Colormap', jet)` displays the grayscale image *I* using the `jet` colormap.

## **Colormap** — Color map

*v*-by-3 numeric matrix

Color map, specified as the comma-separated pair consisting of `'Colormap'` and a *v*-by-3 numeric matrix. You can also create a colormap matrix using a predefined colormap function, such as `parula` or `jet`.

Example: `'Colormap', repmat([1:256]', 1, 3);`

Example: `'Colormap', jet`

Example: `'Colormap', parula(128)`

## **DisplayRange** — Display range

2-element vector



Display range, specified as the comma-separated pair consisting of 'DisplayRange' and a 2-element vector of the form [low high].

---

**Note** Including the parameter name is optional, except when the image is specified by a filename. The syntax `imtool(I,[low high])` is equivalent to `imtool(I,'DisplayRange',[low high])`. However, the 'DisplayRange' parameter must be specified when calling `imtool` with a filename, as in the syntax `imtool(filename,'DisplayRange',[low high])`.

---

### InitialMagnification – Initial magnification

'adaptive' | 'fit' | numeric scalar

Initial magnification, specified as the comma-separated pair consisting of 'InitialMagnification' and one of these values.

Parameter	Value
'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.
'fit'	The Image Viewer scales the entire image to fit in the window.
numeric scalar	The Image Viewer scales the entire image as a percentage of the original image size. For example, if you specify 100, the Image Viewer displays the image at 100% magnification (one screen pixel for each image pixel).  <b>Note</b> 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%.

By default, the initial magnification parameter is set to the value returned by `iptgetpref('ImtoolInitialMagnification')`.

## Output Arguments

### **hfigure — Handle to Image Viewer**

handle

Handle to Image Viewer figure, returned as a handle.

## Definitions

### 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 RGB 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. You can also set preferences programmatically with `iptsetpref`:

Preference	Description
'ImtoolInitialMagnification'	Controls the initial magnification for image display. To override this toolbox preference, specify the 'InitialMagnification' parameter when you call <code>imtool</code> , as follows:  <code>imtool(...,'InitialMagnification',initial_mag)</code>

Preference	Description
'ImtoolStartWithOverview'	<p>Controls whether the Overview tool opens automatically when you open an image using the Image Viewer. Possible values:</p> <ul style="list-style-type: none"> <li>• <code>true</code> — Overview tool opens when you open an image.</li> <li>• <code>false</code> — Overview tool does not open when you open an image. This is the default behavior.</li> </ul>

For more information about these preferences, see `iptprefs`.

## Tips

- 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.
- For 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.

- You can access the Image Viewer through the Apps tab. Navigate to the Image Processing and Computer Vision group and select Image Viewer.
- You can close a specific Image Viewer specified by the handle `hfigure`, by using the command `close(hfigure)`.

## See Also

`imageinfo` | `imcontrast` | `imoverview` | `impixelregion` | `imshow` | `iptprefs`

**Introduced before R2006a**

# imtophat

Top-hat filtering

## Syntax

```
J = imtophat(I,SE)
J = imtophat(I,nhood)
```

## Description

`J = imtophat(I,SE)` performs morphological top-hat filtering on the grayscale or binary image `I`, returning the filtered image, `J`. Top-hat filtering computes the morphological opening of the image (using `imopen`) and then subtracts the result from the original image. `SE` is a single structuring element object returned by the `strel` or `offsetstrel` functions.

You optionally can perform the top-hat filtering using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`J = imtophat(I,nhood)` top-hat filters the image `I`, where `nhood` is a matrix of 0s and 1s that specifies the structuring element neighborhood. This is equivalent to the syntax `imtophat(I,strel(nhood))`. The `imtophat` function determines the center element of the neighborhood by `floor((size(nhood)+1)/2)`.

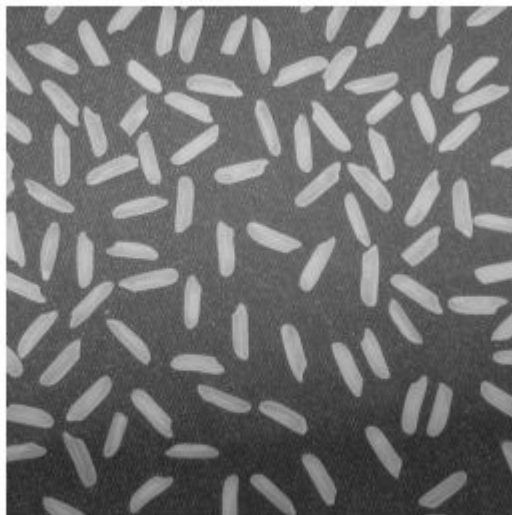
## Examples

### 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)
```

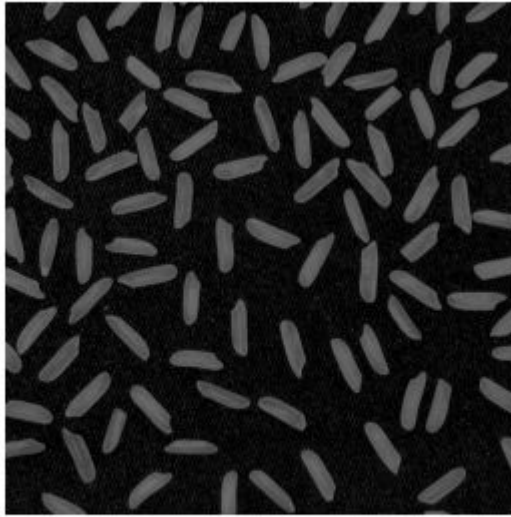


Create the structuring element.

```
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)
```

## Input Arguments

### **I** — Input image

grayscale image | binary image | `gpuArray`

Input image, specified as a grayscale image or binary image. `I` can have any dimension, and must be real and nonsparse.

To perform the top-hat filtering using a GPU, specify `I` as a `gpuArray` that contains an image of type `uint8` or `logical`.

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

### **SE** — Structuring element

`strel` object | `offsetstrel` object

Structuring element, specified as a single `strel` object or `offsetstrel` object. If the image `I` is data type `logical`, the structuring element must be flat.

If you perform the top-hat filtering using a GPU, then `SE` must be flat and two-dimensional.

### **nhood** — Structuring element neighborhood

matrix of 0s and 1s

Structuring element neighborhood, specified as a matrix of 0s and 1s.

Example: `[0 1 0; 1 1 1; 0 1 0]`



## Output Arguments

### **J** — Top-hat filtered image

grayscale image | binary image | gpuArray

Top-hat filtered image, returned as a grayscale image or binary image. J has the same class as input image I.

If the filtering is performed using a GPU, then J is returned as a `gpuArray` that contains a grayscale or binary image of the same class 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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, the image input argument, `I`, must be 2-D or 3-D and the structuring element input argument, `SE`, must be a compile-time constant.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `gpuArray` input must be of type `uint8` or `logical` and the structuring element must be flat and two-dimensional.

For more information, see “Image Processing on a GPU”.

## **See Also**

### **Functions**

`imbothat` | `imclose` | `imdilate` | `imerode` | `imopen`

### **Objects**

`offsetstrel` | `strel`

**Introduced before R2006a**

# imtransform

Apply 2-D spatial transformation to image

---

**Note** `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**

## 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` | `makeresampler` | `maketform` | `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.

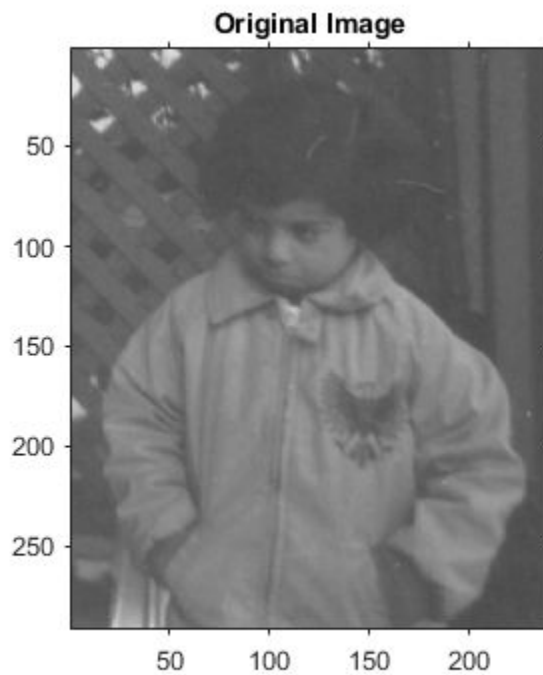
```
I = imread('pout.tif');
```

Translate the image.

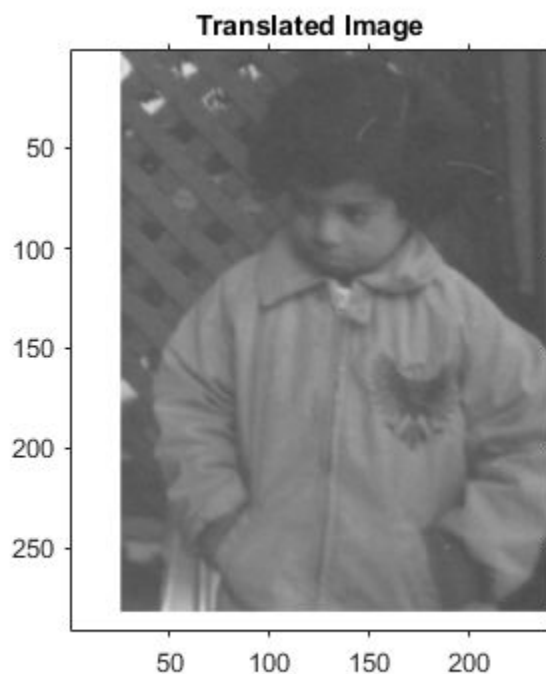
```
J = imtranslate(I,[25.3, -10.1], 'FillValues',255);
```

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');
```

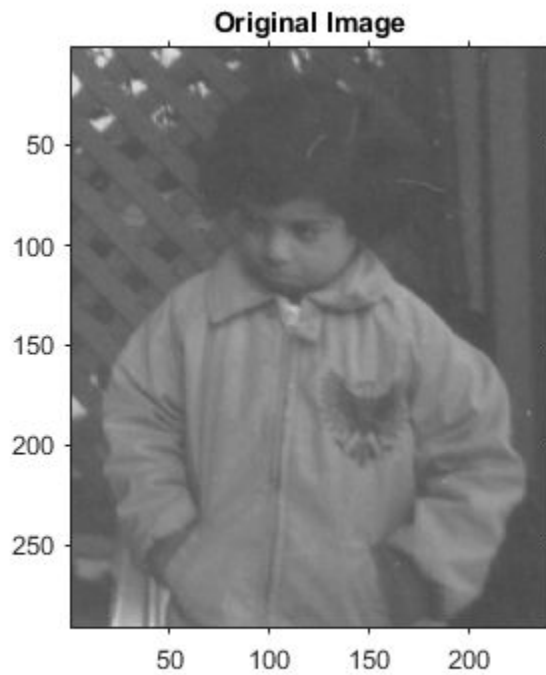
Translate the image. Use the `OutputView` parameter to specify that you want the entire translated image to be visible.

```
J = imtranslate(I,[25.3, -10.1],'FillValues',255,'OutputView','full');
```

Display the original image and the translated image.

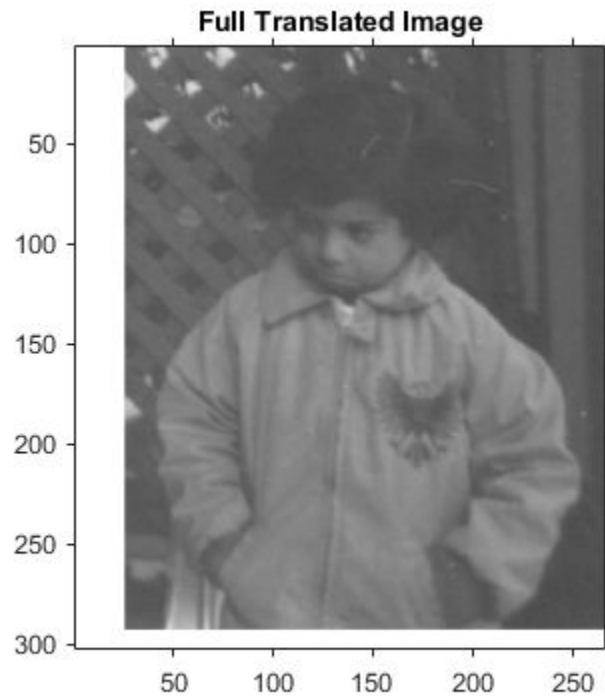
```
figure  
imshow(I);
```

```
title('Original Image');  
set(gca, 'Visible', 'on');
```



```
figure  
imshow(J);  
title('Full Translated Image');  
set(gca, 'Visible', 'on');
```

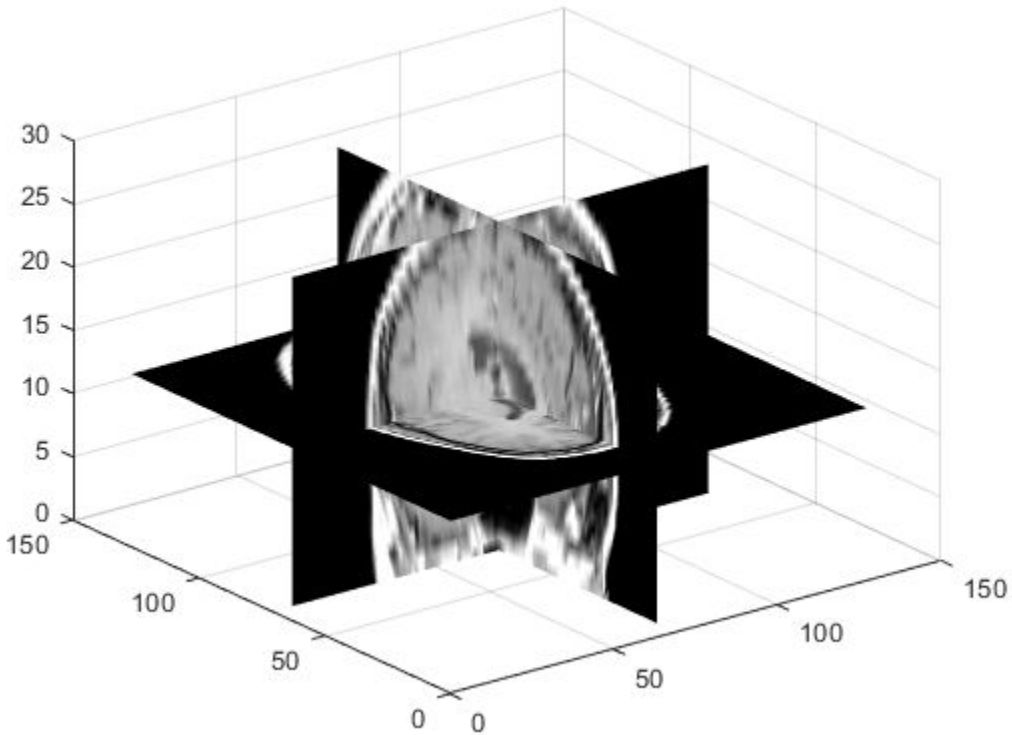




### Translate 3-D MRI Dataset

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
```

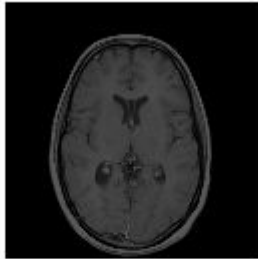


Apply a translation in the X,Y direction.

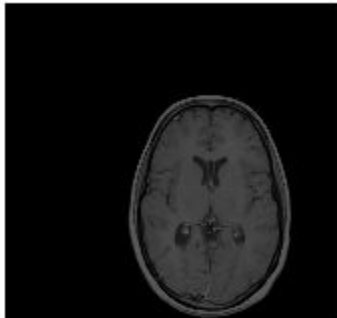
```
mriVolumeTranslated = imtranslate(mriVolume,[40,30,0],'OutputView','full');
```

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);
```



## 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**

`imref2d` or `imref3d` spatial referencing object

Spatial referencing information associated with the input image A, specified as an `imref2d` or `imref3d` spatial referencing object.

**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
<code>'cubic'</code>	Cubic interpolation.  <b>Note</b> Cubic interpolation can produce pixel values outside the original range.
<code>'linear'</code>	Linear interpolation
<code>'nearest'</code>	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` | `string`

## 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 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 | string

### 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

`imref2d` or `imref3d` spatial referencing object

Spatial referencing information associated with the output image, returned as an `imref2d` or `imref3d` spatial referencing object.

## 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 “Understand Code Generation with Image Processing Toolbox”.
- The function supports only 2-D translation vectors, `translation`. 3-D translations are not supported.

### See Also

`imref2d` | `imref3d` | `imresize` | `imrotate` | `imwarp`

### Topics

“Translate an Image using `imtranslate` Function”

**Introduced in R2014a**

## 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.

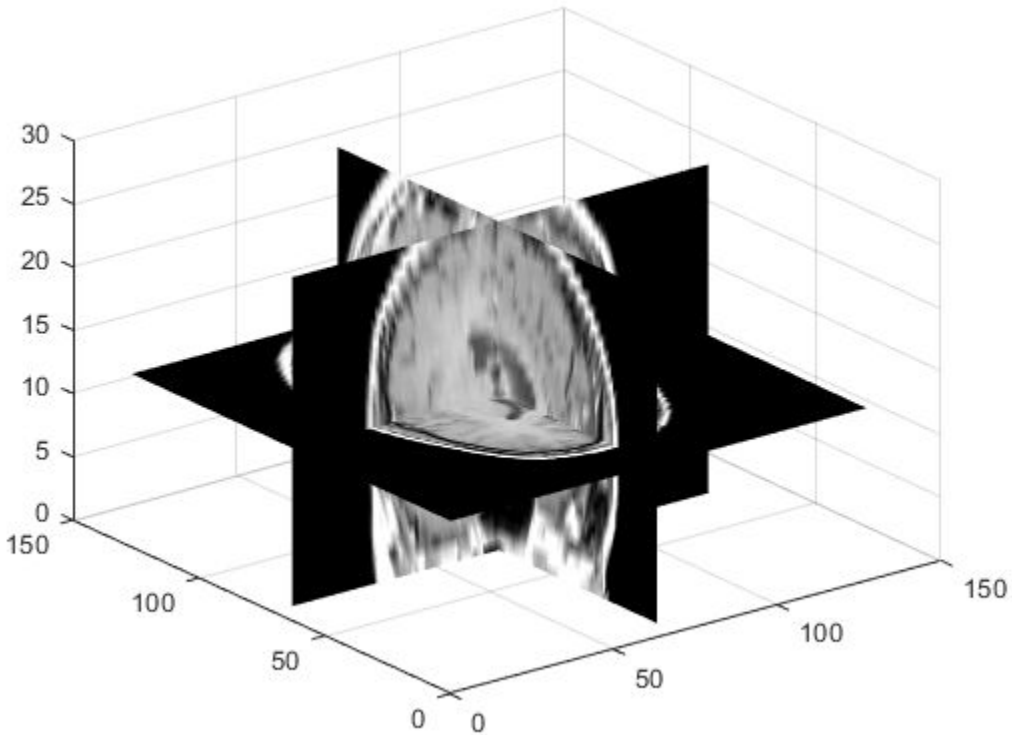
```
J = imwarp(I,tform);  
figure  
imshow(J)
```



## 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 = 4x4

    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:
```

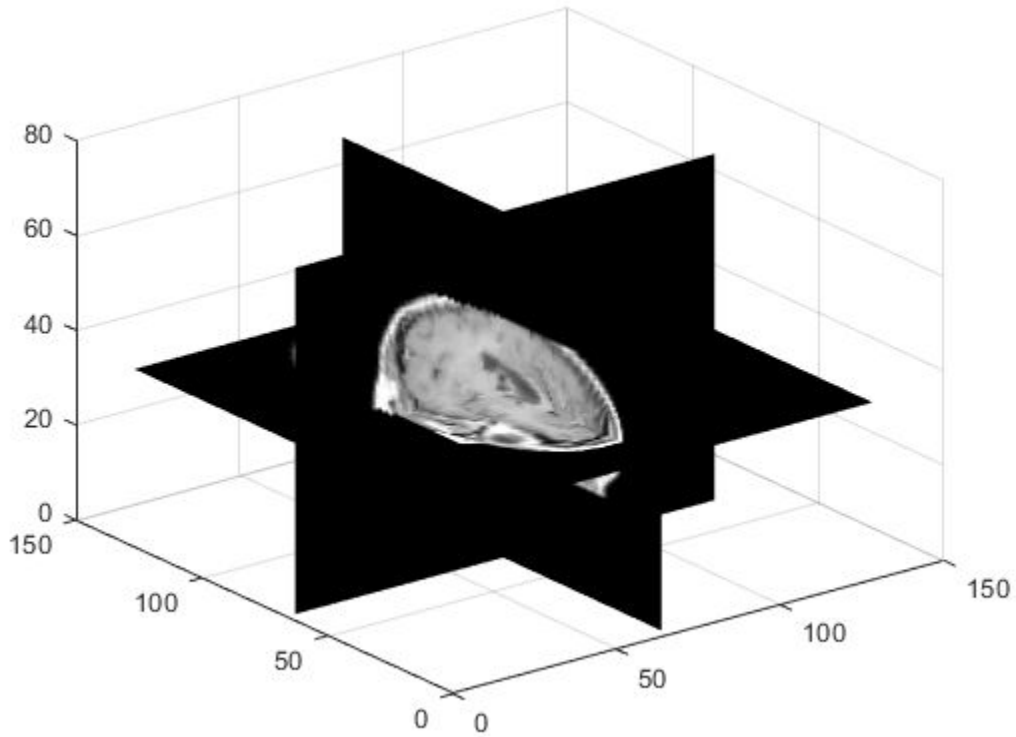
```
          T: [4x4 double]  
  Dimensionality: 3
```

Apply the transformation to the image.

```
mriVolumeRotated = imwarp(mriVolume,tform);
```

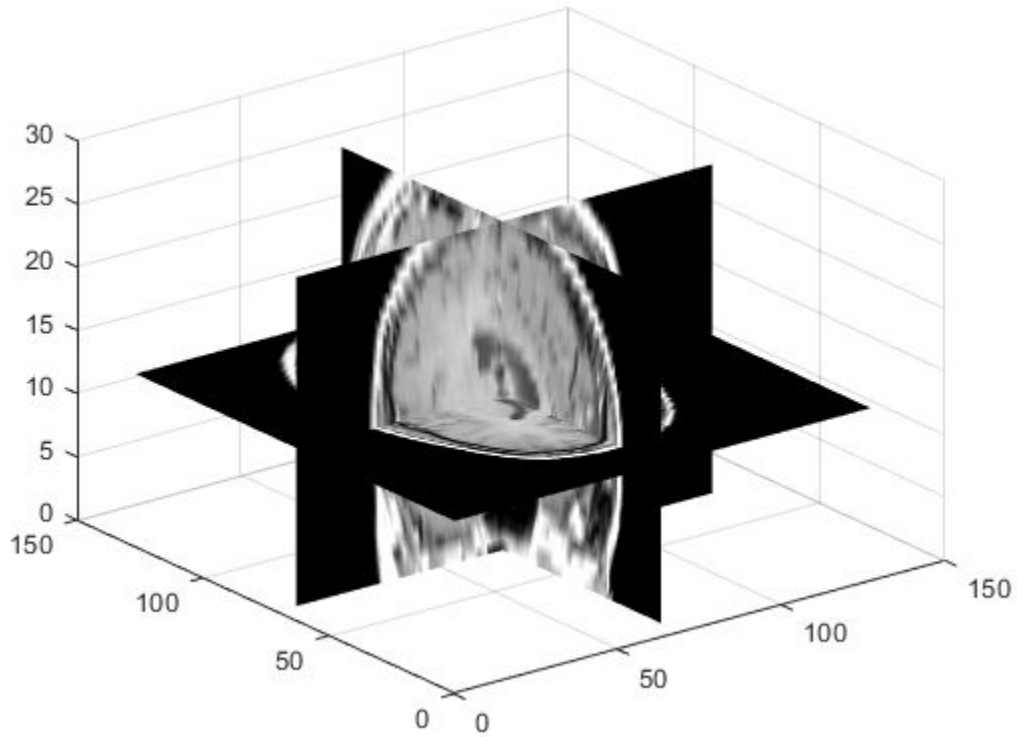
Visualize three slice planes through the center of the transformed volumes.

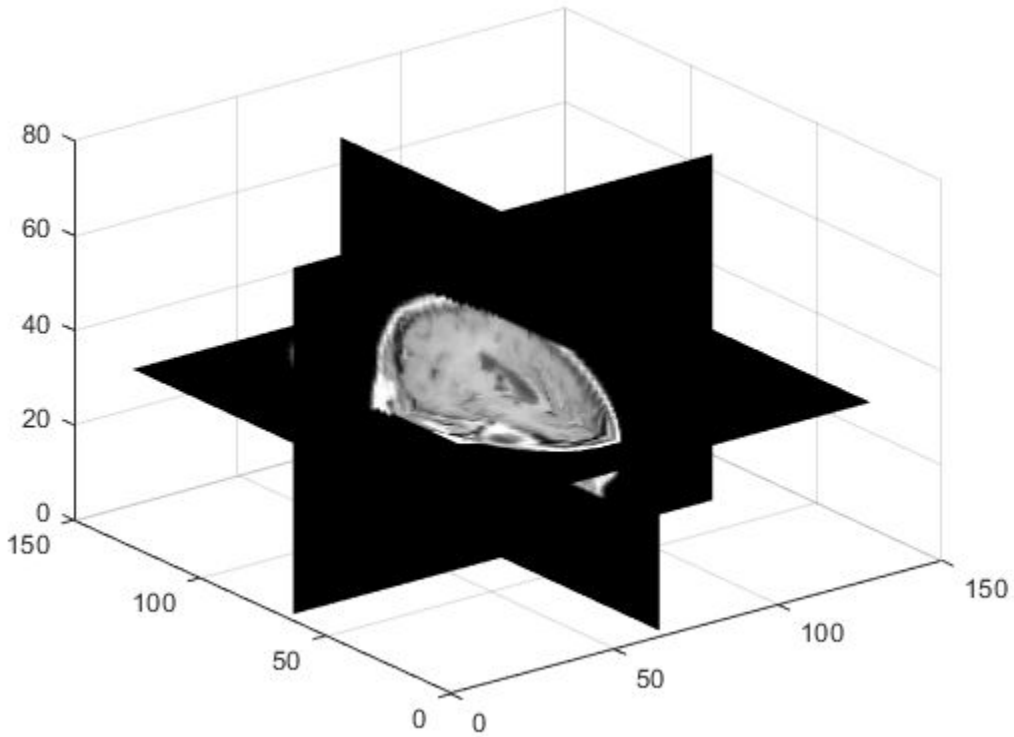
```
sizeOut = size(mriVolumeRotated);  
hFigRotated = figure;  
hAxRotated = axes;  
slice(double(mriVolumeRotated),sizeOut(2)/2,sizeOut(1)/2,sizeOut(3)/2);  
grid on, shading interp, colormap gray
```



Link views of both axes together.

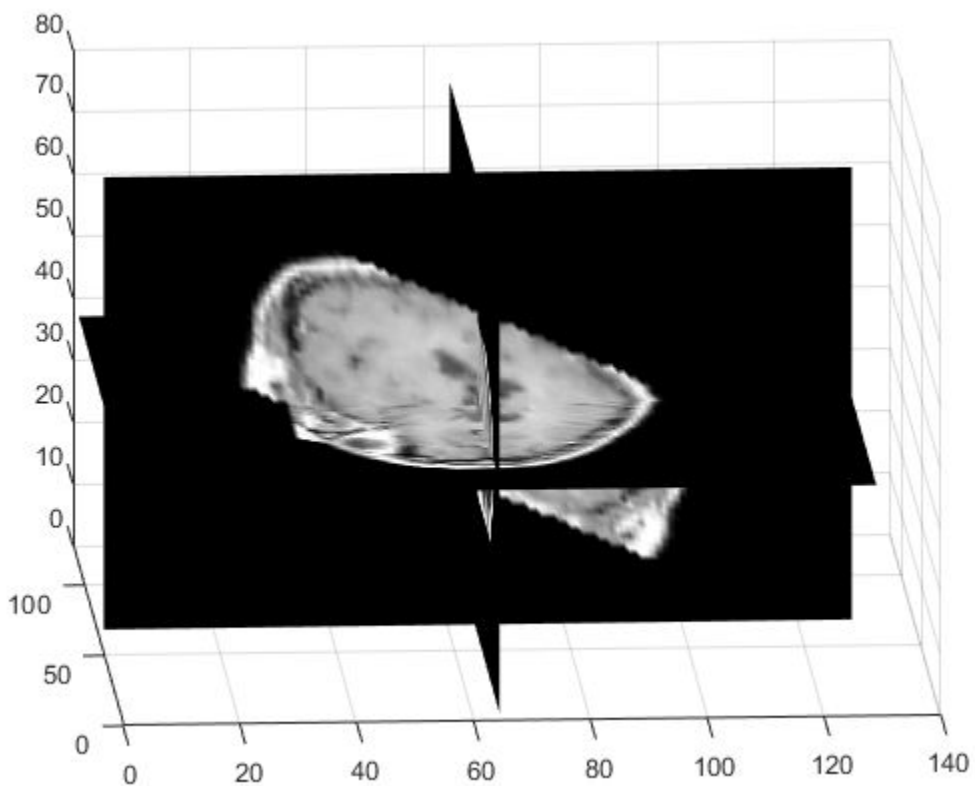
```
linkprop([hAxOriginal,hAxRotated],'View');
```



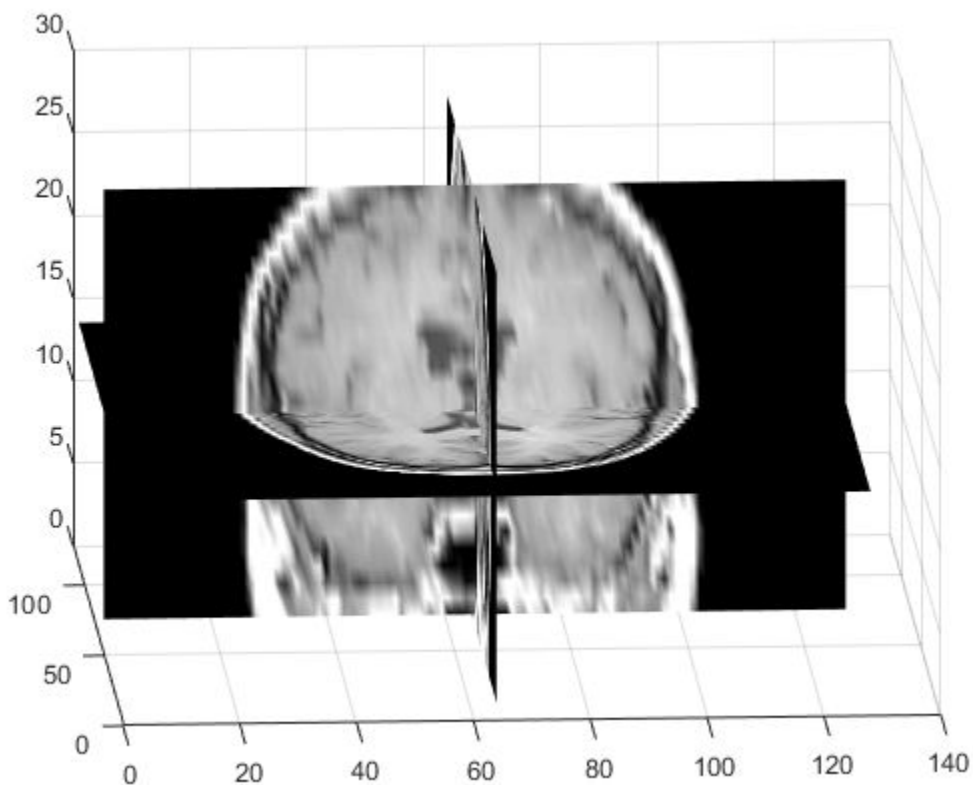


Set view to see effect of rotation.

```
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 these 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 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:

<b>FillValue</b>	<b>Effect</b>
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

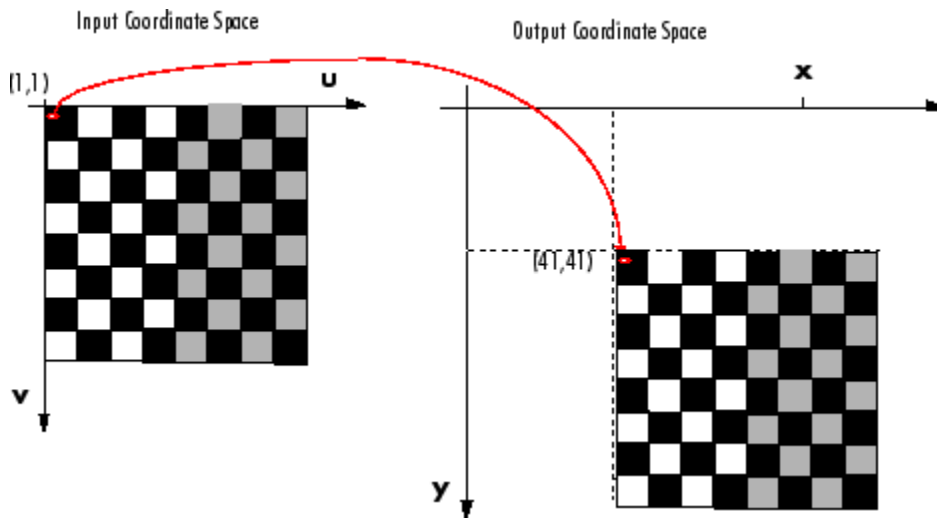
`imref2d` or `imref3d` spatial referencing object

Spatial referencing information associated with the transformed image, returned as an `imref2d` or `imref3d` spatial referencing object.

## Algorithms

`imwarp` determines the value of pixels in the output image by mapping locations in the output image to the corresponding locations in the input image (inverse mapping). `imwarp` interpolates within the input image to compute the output pixel value.

The following figure illustrates a translation transformation. By convention, the axes in input space are labeled  $u$  and  $v$  and the axes in output space are labelled  $x$  and  $y$ . In the figure, note how `imwarp` modifies the spatial coordinates that define the locations of pixels in the input image. The pixel at  $(1,1)$  is now positioned at  $(41,41)$ . In the checkerboard image, each black, white, and gray square is 10 pixels high and 10 pixels wide. For more information about the distinction between spatial coordinates and pixel coordinates, see “Image Coordinate Systems”.



**Input Image Translated**

## 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 “Understand 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

#### Apps

**Registration Estimator**

#### Functions

`imregdemons` | `imregister` | `imregtform` | `imtranslate`

#### Objects

`affine2d` | `affine3d` | `geometricTransform2d` | `geometricTransform3d` | `projective2d`

#### Topics

“2-D and 3-D Geometric Transformation Process Overview”

**Introduced in R2013a**

# ind2gray

Convert indexed image to grayscale image

## Syntax

```
I = ind2gray(X,cmap)
```

## Description

`I = ind2gray(X,cmap)` converts the indexed image `X` with colormap `cmap` to a grayscale image, `I`. The `ind2gray` function removes the hue and saturation information from the input image while retaining the luminance.

## 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**

## Input Arguments

### **X — Indexed image**

numeric array

Indexed image, specified as a numeric array of any size and dimensionality.

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

### **cmap — Color map**

c-by-3 numeric array

Color map associated with indexed image *X*, specified as a c-by-3 numeric array, where *c* is the number of colors.

Data Types: `double`

## Output Arguments

### **I** — Grayscale image

numeric array

Grayscale image, specified as a numeric array. **I** has the same size, dimensionality and class as **X**.

## 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,cmap)
```

## Description

`RGB = ind2rgb(X,cmap)` converts the indexed image `X` and corresponding colormap `cmap` to RGB (truecolor) format.

## Input Arguments

### **X** — Indexed image

*m*-by-*n* numeric matrix

Indexed image, specified as an *m*-by-*n* numeric matrix.

### **cmap** — Color map

*c*-by-3 numeric array

Color map associated with indexed image `X`, specified as a *c*-by-3 numeric array, where *c* is the number of colors.

Data Types: `double`

## Output Arguments

### **RGB** — RGB image

*m*-by-*n*-by-3 numeric array

RGB image, returned as an *m*-by-*n*-by-3 numeric array.

## **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 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/filterSize.^2`, if scalar, and `1/prod(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

`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 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

`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 1-1997 , J, from the intensity image, I.

`J = integralImage(I,orientation)` calculates the integral image with the orientation specified by `orientation`.

### Examples

#### Create Integral Image

Create a simple sample matrix.

```
I = magic(5)
```

```
I = 5×5
```

```
    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
```

In an integral image, each pixel is the cumulative sum of the pixel directly above it and to its left. For example, in this trivial matrix, the pixel in the original matrix at row 1, column



1 (value 17) would be unchanged in an integral image because you are adding 0s to the value. (The `integrallImage` function adds a row of 0s above and to the left of the original matrix.) Calculating the integral image value for pixel (1,2) in the original matrix, you add the pixel above it (0) and the pixel to its left (17):  $24 + 17 + 0 = 41$ . Calculating the integral image value for Pixel (1,3) in the original matrix, you add the pixel above it (0) and the pixel to its left, which has already been summed, 41. Thus the value at pixel (2,4) in the integral image is  $1 + 41 + 0 = 42$ . This process continues for each pixel in the original image, as you can see in the integral image of the original matrix calculated by the `integrallImage` function.

```
J = integrallImage(I)
```

```
J = 6×6
```

```

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
```

### Compute Integral Image with Rotated Orientation

Create sample image.

```
I = magic(5)
```

```
I = 5×5
```

```

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` | `string`

## 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.

## Extended Capabilities

### C/C++ Code Generation

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

## **See Also**

`cumsum` | `integralBoxFilter`

## **Topics**

“Integral Image”

**Introduced in R2015b**

# integrallImage3

Calculate 3-D integral image

## Syntax

```
J = integrallImage3(I)
```

## Description

`J = integrallImage3(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 = integrallImage3(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)`.

## 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.

## See Also

`integralBoxFilter3` | `integralImage`

## Topics

“Integral Image”

**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 string scalar or 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`.

### Examples

Read metadata from an Interfile file.

```
info = interfileinfo('MyFile.hdr');
```

### References

- [1] Todd-Pokropek, A, Craddock, T.D., and Deconinck, F., *A File Format for the Exchange of Nuclear Medicine Image Data: a specification of Interfile Version 3.3*. Nucl Med Commun 13(9): 673-99, 1992.



## **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`.

### Examples

Read image data from an Interfile file.

```
img = interfileread('MyFile.hdr');
```

## References

- [1] Todd-Pokropek, A, Craddock, T.D., and Deconinck, F., *A File Format for the Exchange of Nuclear Medicine Image Data: a specification of Interfile Version 3.3*. Nucl Med Commun 13(9): 673-99, 1992.

## 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`.

### 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 = 3x4 uint8 matrix
```

```
     4     8    16     0
    150   200   255     0
    150   200     2     4
```

## Input Arguments

### **A** — Input matrix

array of integers

Input matrix, specified as an array of integers.

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

### **lut** — Lookup table

vector of integers

Lookup table, specified as a vector of integers.

- If **A** has data type `uint8`, then **lut** must be a `uint8` vector with 256 elements.
- If **A** has data type `uint16` or `int16`, then **lut** must be a vector with 65536 elements of the same class as **A**.

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

## Output Arguments

### **B** — Converted matrix

array of integers

Converted matrix, returned as an array of integers. **B** has the same size and data type as **A**.

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

## Algorithms

- When A has data type `uint8` or `uint16`, an offset of 1 is applied when indexing into the lookup table. For example, if an element of A has the value *alpha*, then the corresponding element in B has the value `lut(alpha+1)`.
- When A has data type `int16`, an additional offset of 32768 is applied to the lookup table index. For example, if an element of A has the value *alpha*, then the corresponding element in B has the value `lut(alpha+32768+1)`.

## 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 “Understand Code Generation with Image Processing Toolbox”.

### See Also

`ind2gray` | `rgb2ind`

**Introduced before R2006a**

# intrinsicToWorld

Convert from intrinsic to world coordinates

## Syntax

```
[xWorld, yWorld] = intrinsicToWorld(R,xIntrinsic,yIntrinsic)  
[xWorld, yWorld, zWorld] = intrinsicToWorld(R,xIntrinsic,yIntrinsic,  
zIntrinsic)
```

## Description

`[xWorld, yWorld] = intrinsicToWorld(R,xIntrinsic,yIntrinsic)` maps points from the 2-D intrinsic system (`xIntrinsic,yIntrinsic`) to the 2-D world system (`xWorld,yWorld`) based on the relationship defined by 2-D spatial referencing object `R`.

If the  $k$ th input coordinates (`xIntrinsic(k),yIntrinsic(k)`) fall outside the image bounds in the intrinsic coordinate system, `intrinsicToWorld` extrapolates `xWorld(k)` and `yWorld(k)` outside the image bounds in the world coordinate system.

`[xWorld, yWorld, zWorld] = intrinsicToWorld(R,xIntrinsic,yIntrinsic, zIntrinsic)` maps points from the intrinsic coordinate system to the world coordinate system using 3-D spatial referencing object `R`.

## Examples

### Convert 2-D Intrinsic Coordinates to World Coordinates

Read a 2-D grayscale image into the workspace.

```
m = dicominfo('knee1.dcm');  
A = dicomread(m);
```

Create an `imref2d` object, specifying the size and the resolution of the pixels. The DICOM file contains a metadata field `PixelSpacing` that specifies the image resolution in each dimension in millimeters per pixel.

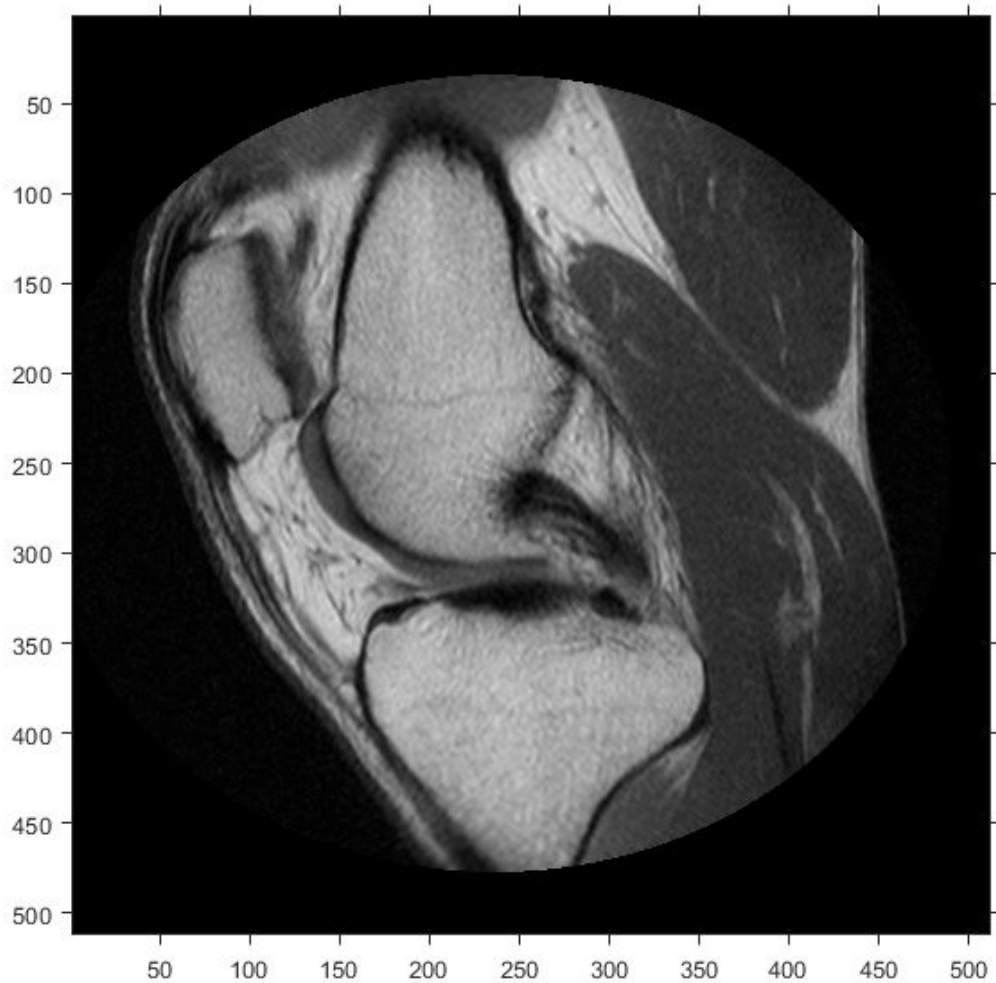
```
RA = imref2d(size(A),m.PixelSpacing(2),m.PixelSpacing(1))
```

```
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]
```

Display the image, omitting the spatial referencing object. The axes coordinates reflect the intrinsic coordinates. Notice that the coordinate (0,0) is in the upper left corner.

```
figure  
imshow(A, 'DisplayRange', [0 512])  
axis on
```





Suppose you want to calculate the approximate position and width of the knee in millimeters. Select the endpoints of a line segment that runs horizontally across the knee at the level of the kneecap. For example, use the  $(x,y)$  points  $(34,172)$  and  $(442,172)$ .

```
xIntrinsic = [34 442];  
yIntrinsic = [172 172];
```

Convert these points from intrinsic coordinates to world coordinates.

```
[xWorld,yWorld] = intrinsicToWorld(RA,xIntrinsic,yIntrinsic)
```

```
xWorld = 1×2
```

```
10.6250 138.1250
```

```
yWorld = 1×2
```

```
53.7500 53.7500
```

The world coordinates of the two points are (10.625,53.75) and (138.125,53.75), in units of millimeters. The approximate width of the knee in millimeters is:

```
width = xWorld(2) - xWorld(1)
```

```
width = 127.5000
```

## Convert 3-D Intrinsic Coordinates to World Coordinates

Read a 3-D volume into the workspace. This image consists of 27 frames of 128-by-128 pixel images.

```
load mri;  
D = squeeze(D);  
D = ind2gray(D,map);
```

Create an `imref3d` spatial referencing object associated with the volume. For illustrative purposes, provide a pixel resolution in each dimension. The resolution is in millimeters per pixel.

```
R = imref3d(size(D),2,2,4)
```

```
R =  
imref3d with properties:
```

```
    XWorldLimits: [1 257]
```

```

YWorldLimits: [1 257]
ZWorldLimits: [2 110]
  ImageSize: [128 128 27]
PixelExtentInWorldX: 2
PixelExtentInWorldY: 2
PixelExtentInWorldZ: 4
ImageExtentInWorldX: 256
ImageExtentInWorldY: 256
ImageExtentInWorldZ: 108
  XIntrinsicLimits: [0.5000 128.5000]
  YIntrinsicLimits: [0.5000 128.5000]
  ZIntrinsicLimits: [0.5000 27.5000]

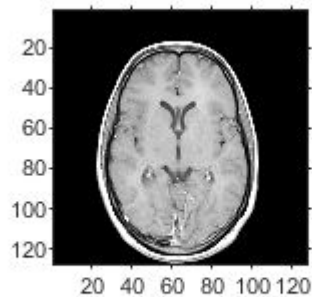
```

Display the middle slice of the volume, omitting the spatial referencing object. The axes coordinates reflect the intrinsic coordinates. Notice that the coordinate (0,0) is in the upper left corner of this plane.  $z=0$  is right below the first slice, and the  $z$ -axis is positive in the upward direction, towards the crown of the head.

```

figure
imshow(D(:,:,13))
axis on

```



Suppose you want to determine the position, in millimeters, of features within this slice. Select four sample points, and store their intrinsic coordinates in vectors. For example, the first point has intrinsic coordinates (54,46,13). The intrinsic  $z$ -coordinate is the same for all points within this slice.

```
xI = [54 71 57 70];  
yI = [46 48 79 80];  
zI = [13 13 13 13];
```

Convert the intrinsic coordinates to world coordinates using `intrinsicToWorld`.

```
[xW,yW,zW] = intrinsicToWorld(R,xI,yI,zI)
```

```
xW = 1×4
```

```
    108    142    114    140
```

```
yW = 1×4
```

```
    92    96    158    160
```

```
zW = 1×4
```

```
    52    52    52    52
```

The resulting vectors are the world *x*-, *y*-, and *z*-coordinates, in millimeters, of the selected points. The first point, for example, is offset from the origin by 108mm in the *x*-direction, 92 mm in the *y*-direction, and 52 mm in the *z*-direction.

## Input Arguments

### **R** — Spatial referencing object

`imref2d` or `imref3d` object

Spatial referencing object, specified as an `imref2d` or `imref3d` object.

### **xIntrinsic** — Coordinates along the *x*-dimension in the intrinsic coordinate system

numeric scalar or vector

Coordinates along the *x*-dimension in the intrinsic coordinate system, specified as a numeric scalar or vector.

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

**yIntrinsic — Coordinates along the y-dimension in the intrinsic coordinate system**

numeric scalar or vector

Coordinates along the y-dimension in the intrinsic coordinate system, specified as a numeric scalar or vector. `yIntrinsic` is the same length as `xIntrinsic`.

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

**zIntrinsic — Coordinates along the z-dimension in the intrinsic coordinate system**

numeric scalar or vector

Coordinates along the z-dimension in the intrinsic coordinate system, specified as a numeric scalar or vector. `zIntrinsic` is the same length as `xIntrinsic`.

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

## Output Arguments

**xWorld — Coordinates along the x-dimension in the world coordinate system**

numeric scalar or vector

Coordinates along the x-dimension in the world coordinate system, returned as a numeric scalar or vector. `xWorld` is the same length as `xIntrinsic`.

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

**yWorld — Coordinates along the y-dimension in the world coordinate system**

numeric scalar or vector

Coordinates along the y-dimension in the world coordinate system, returned as a numeric scalar or vector. `yWorld` is the same length as `xIntrinsic`.

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

**zWorld — Coordinates along the z-dimension in the world coordinate system**

numeric scalar or vector

Coordinates along the z-dimension in the world coordinate system, returned as a numeric scalar or vector. `zWorld` is the same length as `xIntrinsic`.

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

## See Also

`imref2d` | `imref3d` | `worldToIntrinsic`

**Introduced in R2013a**

# invert

Invert geometric transformation

## Syntax

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

## Description

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

## Examples

### Invert a 2-D Rotation

Create an `affine2d` object that defines a 30 degree rotation in the counterclockwise direction around the origin. View the transformation matrix stored in the `T` property.

```
theta = 30;  
tform = affine2d([cosd(theta) sind(theta) 0; -sind(theta) cosd(theta) 0; 0 0 1]);  
tform.T
```

```
ans = 3×3
```

```
    0.8660    0.5000         0  
   -0.5000    0.8660         0  
         0         0    1.0000
```

Invert the geometric transformation. The result is a new `affine2d` object.

```
invtform = invert(tform);  
invtform.T
```

```
ans = 3x3
    0.8660    -0.5000         0
    0.5000     0.8660         0
         0         0     1.0000
```

This matrix represents a 30 degree rotation in the clockwise direction.

## Test the Inverse Geometric Transformation

Read an image, and display it.

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





Apply the forward geometric transformation, `tform`, to the image. Display the rotated image.

```
J = imwarp(I,tform);  
figure;  
imshow(J)
```



Apply the inverse geometric transformation, `invtfom`, to the rotated image `J`.

```
K = imwarp(J,invtfom);  
imshow(K)
```



The final image, `K`, has the correct orientation. The two transformations introduced padding that surrounds the image, but the size, shape, and orientation of the image data have not changed.

## Input Arguments

### **tform** — Geometric transformation

`affine2d`, `affine3d`, or `projective2d` geometric transformation object

Geometric transformation, specified as an `affine2d`, `affine3d`, or `projective2d` geometric transformation object.

## Output Arguments

### **invtform** — Inverse geometric transformation

geometric transformation object

Inverse geometric transformation, returned as a geometric transformation object. `invtform` is the same type of object as `tform`.

## See Also

`transformPointsForward` | `transformPointsInverse`

**Introduced in R2013a**

## 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 pixel connectivity argument.

- If the connectivity argument is valid, then `iptcheckconn` returns nothing. Valid connectivities are one of these scalar values: 1, 4, 6, 8, 18, or 26. A connectivity argument can also be a 3-by-3-by- ... -by-3 array of 0s and 1s. The central element of a connectivity array must be nonzero and the array must be symmetric about its center.
- If the connectivity argument is invalid, then the function issues a formatted error message that includes information about the function name (`func_name`), the variable name (`var_name`), and the argument position (`arg_pos`). These values are used only to create the error message, not to check whether the pixel connectivity argument is valid.

### Examples

#### Check Validity of 4-by-4 Matrix

Create a 4-by-4 array and pass it as the connectivity argument.

```
iptcheckconn(eye(4), 'myfun', 'myvar', 2)
```

`eye(4)` is not a valid pixel connectivity so `iptcheckconn` returns an error message:

```
Function MYFUN expected input number 2, myvar, to be a valid connectivity specifier. A connectivity specifier must be 3-by-3-by- ... -by-3.
```

## Input Arguments

### **conn** — Pixel connectivity

numeric scalar | numeric array

Pixel connectivity to check, specified as a numeric scalar or array.

Data Types: `double` | `logical`

### **func\_name** — Function name

character vector | string scalar

Function name to include in an error message when `conn` is an invalid pixel connectivity, specified as a character vector or string scalar.

Data Types: `char` | `string`

### **var\_name** — Variable name

character vector | string scalar

Variable name to include in an error message when `conn` is an invalid pixel connectivity, specified as a character vector or string scalar.

Data Types: `char` | `string`

### **arg\_pos** — Argument position

numeric scalar

Argument position to include in an error message when `conn` is an invalid pixel connectivity, specified as a numeric scalar.

## 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.

## **See Also**

conndef

## **Topics**

“Pixel Connectivity”

**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

`iptcheckmap` | `iptnum2ordinal` | `narginchk` | `validateattributes` | `validatestring`

**Introduced before R2006a**

# iptcheckinput

Check validity of array

---

**Note** `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` | `iptnum2ordinal` | `narginchk` |  
`validateattributes` | `validatestring`

**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](#) | [narginchk](#) | [validateattributes](#) | [validatestring](#)

**Introduced before R2006a**

# iptchecknargin

Check number of input arguments

---

**Note** `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` | `iptcheckmap` | `iptnum2ordinal` | `narginchk` |  
`validateattributes` | `validatestring`

**Introduced before R2006a**



# iptcheckstrs

Check validity of option

**Note** `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` | `iptcheckmap` | `iptnum2ordinal` | `narginchk` |  
`validateattributes` | `validatestring`

**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

`imdistline` | `immagbox` | `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 `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. If the figure contains a pointer behavior structure on page 1-2050, then the pointer manager controls the pointer behavior for graphics objects in the figure. 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.

---

**Note** If the figure already contains a pointer manager, then `iptPointerManager(hFigure)` does not create a new pointer manager. The syntax has the same behavior as `iptPointerManager(hFigure, 'enable')`.

---

`iptPointerManager(hFigure, 'disable')` disables the figure's pointer manager.

`iptPointerManager(hFigure, 'enable')` enables and updates the figure's pointer manager.

## Examples

### Create Pointer Manager in Figure with Line Object

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);
```

## Input Arguments

### **hFigure** — Figure

Figure object

Figure, specified as a Figure object. For more information, see [figure](#).

## Tips

- `iptPointerManager` considers not just the object the pointer is over, but all objects in the figure. `iptPointerManager` searches the graphics objects hierarchy to find the first object that contains a pointer behavior structure. The `iptPointerManager` then executes that object's pointer behavior function. For more information, see “Graphics Object Hierarchy” (MATLAB).

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`

## **Topics**

“Graphics Object Hierarchy” (MATLAB)

**Introduced in R2006a**

## **iptprefs**

Display Image Processing Toolbox Preferences dialog box

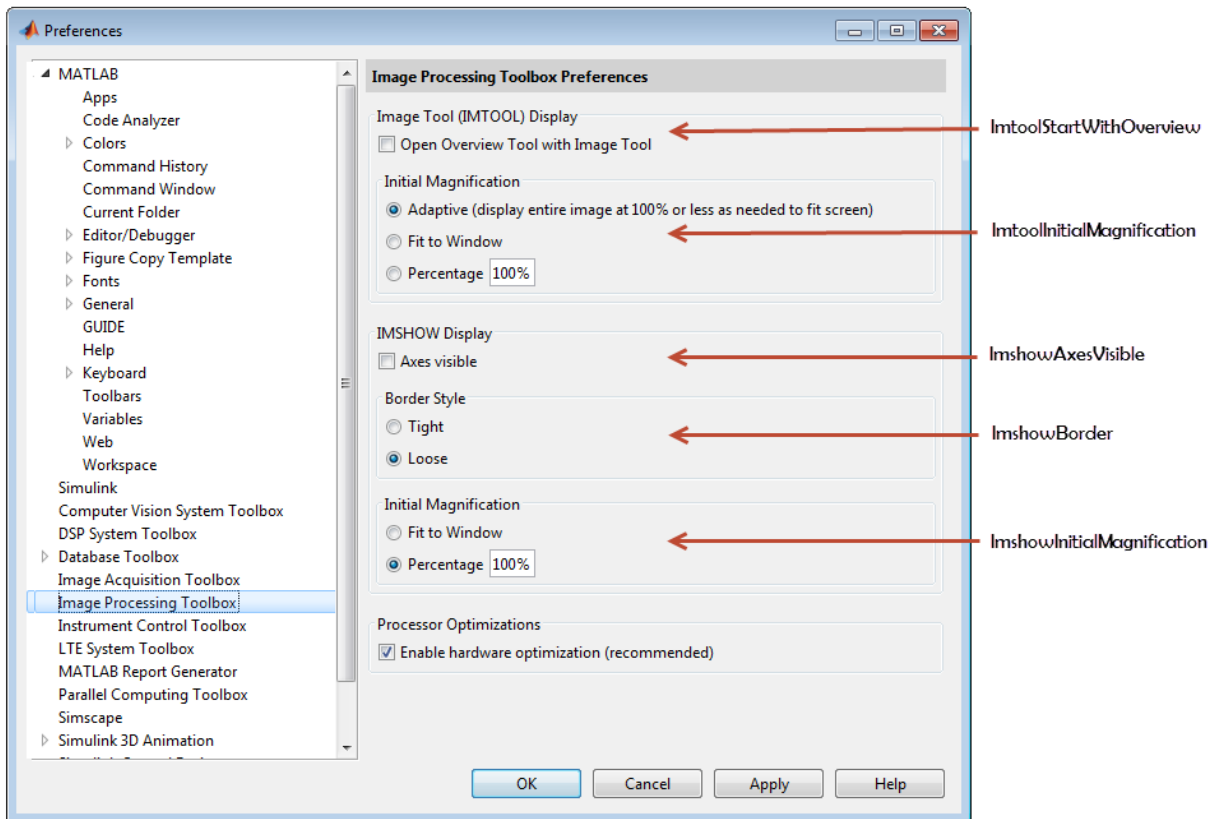
### **Syntax**

`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 `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 1-2050.

`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 1-2050 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><b>Note</b> 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>

<b>Preference Name</b>	<b>Description</b>
'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 <code>100</code>. 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>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>

Preference Name	Description
'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>
'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**

**Volume Viewer** | `imshow` | `imtool` | `iptgetpref` | `iptprefs`

**Introduced before R2006a**



# iptwindowalign

Align figure windows

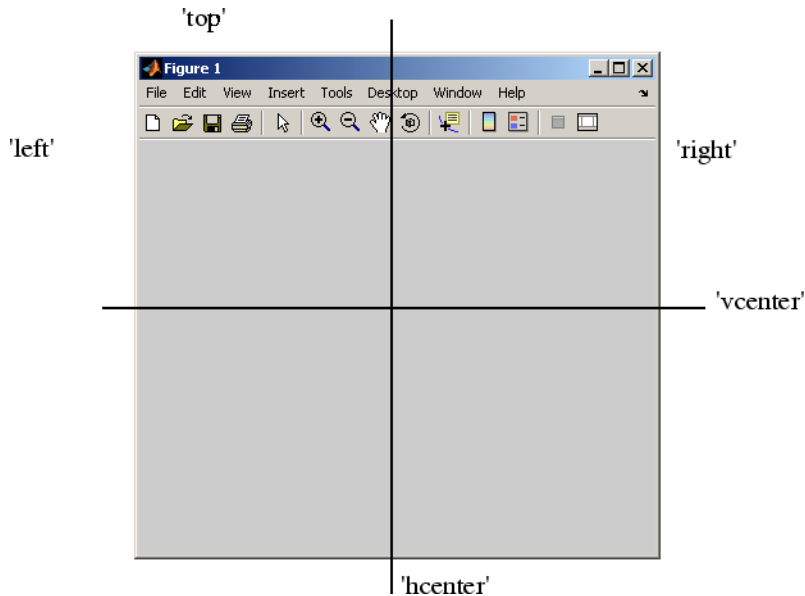
## Syntax

```
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.

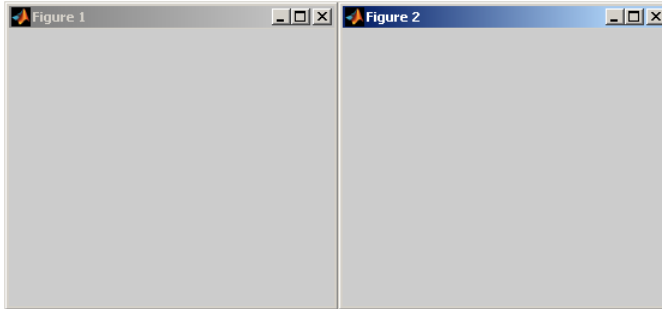
## Examples

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;
```

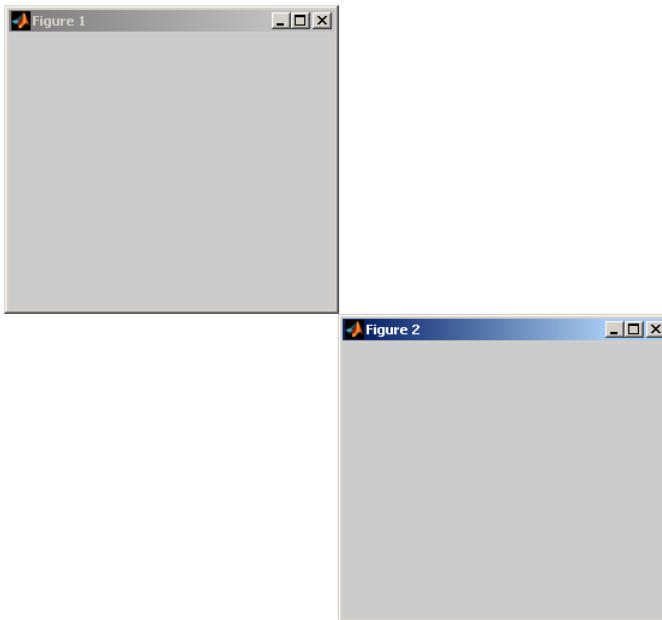
Use `iptwindowalign` to move `fig2` so its left edge is aligned with the right edge of `fig1`.

```
iptwindowalign(fig1,'right',fig2,'left');
```



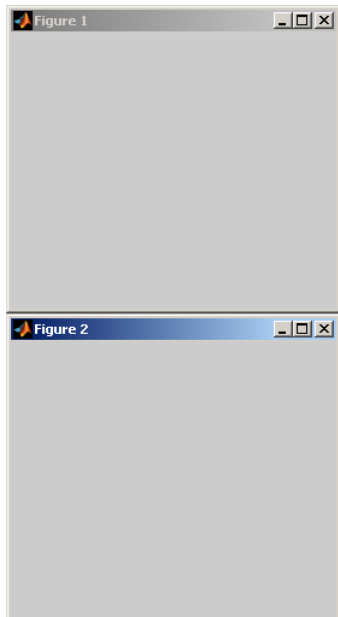
Now move `fig2` so its top edge is aligned with the bottom edge of `fig1`.

```
iptwindowalign(fig1, 'bottom', fig2, 'top');
```



Now move `fig2` so the two figures are centered horizontally.

```
iptwindowalign(fig1, 'hcenter', fig2, 'hcenter');
```



## See Also

`imtool`

**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( ___ )
```

## Description

`I = iradon(R,theta)` reconstructs the image `I` from projection data in `R`. `theta` describes the angles (in degrees) at which the projections were taken.

You optionally can compute the inverse Radon transform using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`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 arguments that you omit.

`[I,H] = iradon( ___ )` also returns the frequency response of the filter, `H`.

## Examples

### Compare Filtered and Unfiltered Backprojection

Create an image of the phantom. Display the image.

```
P = phantom(128);
imshow(P)
title('Original image')
```



Perform a Radon transform of the image.

```
R = radon(P,0:179);
```

Perform filtered backprojection.

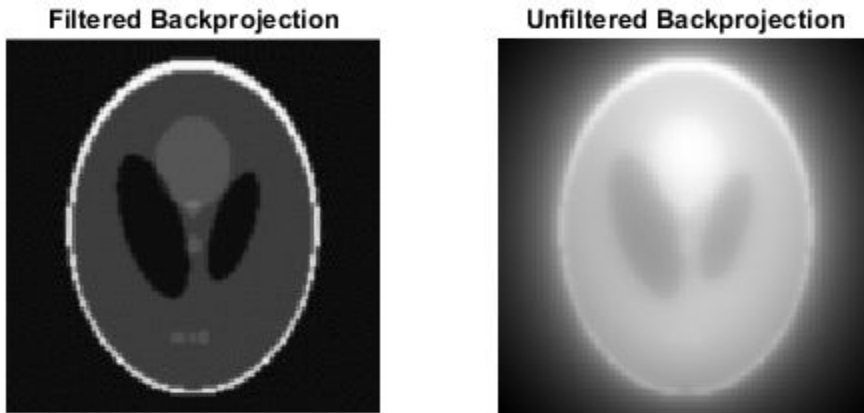
```
I1 = iradon(R,0:179);
```

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')
```



### 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);
```

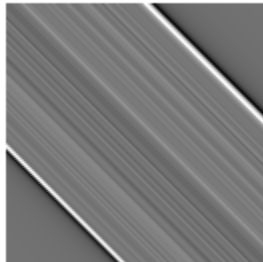
Perform the inverse Radon transform of this single projection vector. The `iradon` syntax does not allow you to do this directly, because if `theta` is a scalar it is treated as an increment. You can accomplish the task by passing in two copies of the projection vector and then dividing the result by 2.

```
I = iradon([r45 r45], [45 45])/2;
```

Display the result.

```
imshow(I, [])  
title('Backprojection from 45 degrees')
```

**Backprojection from 45 degrees**



## Calculate 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')
```

## Input Arguments

### R — Parallel beam projection data

numeric column vector | numeric matrix | gpuArray

Parallel beam projection data, specified as one of the following.

- If **theta** is a scalar, then specify R as a numeric column vector containing the Radon transform for **theta** degrees.
- If **theta** is a vector, then specify R as a 2-D matrix in which each column is the Radon transform for one of the angles in **theta**.

To calculate the inverse Radon transform using a GPU, specify R as a `gpuArray` that contains a numeric column vector or numeric matrix.

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

### theta — Projection angles

numeric vector | numeric scalar | [] | gpuArray

Projection angles (in degrees), specified as one of the following.

Value	Description
numeric vector	Projection angles. There must be equal spacing between the angles.
numeric scalar	Incremental angle between projections. Projections are taken at angles $m \cdot \text{theta}$ , where $m = 0, 1, 2, \dots, \text{size}(R, 2) - 1$ .
[]	Automatically set the incremental angle between projections to $180 / \text{size}(R, 2)$

To calculate the inverse Radon transform using a GPU, you optionally can specify **theta** as a `gpuArray` that contains a numeric vector or scalar.

Data Types: `double`

**interp — Type of interpolation**`'linear'` (default) | `'nearest'` | `'spline'` | `'pchip'`

Type of interpolation to use in the back projection, specified as one of these values, listed in order of increasing accuracy and computational complexity.

Value	Description
<code>'nearest'</code>	Nearest-neighbor interpolation
<code>'linear'</code>	Linear interpolation (the default)
<code>'spline'</code>	Spline interpolation
<code>'pchip'</code>	Shape-preserving piecewise cubic interpolation

If you calculate the inverse Radon transform using a GPU, then `interp` must be `'nearest'` or `'linear'`.

Data Types: `char` | `string`

**filter — Filter**`'Ram-Lak'` (default) | `'Shepp-Logan'` | `'Cosine'` | `'Hamming'` | `'Hann'` | `'None'`

Filter to use for frequency domain filtering, specified as one of these values.

Value	Description
<code>'Ram-Lak'</code>	Cropped Ram-Lak or ramp filter. The frequency response of this filter is $ f $ . Because this filter is sensitive to noise in the projections, one of the filters listed below might be preferable. These filters multiply the Ram-Lak filter by a window that deemphasizes high frequencies.
<code>'Shepp-Logan'</code>	Multiplies the Ram-Lak filter by a <code>sinc</code> function
<code>'Cosine'</code>	Multiplies the Ram-Lak filter by a <code>cosine</code> function
<code>'Hamming'</code>	Multiplies the Ram-Lak filter by a Hamming window
<code>'Hann'</code>	Multiplies the Ram-Lak filter by a Hann window
<code>'None'</code>	No filtering. <code>iradon</code> returns unfiltered backprojection data.

Data Types: `char` | `string`

**frequency\_scaling — Scale factor**`1` (default) | positive number in the range (0, 1]

Scale factor for rescaling the frequency axis, specified as a positive number in the range (0, 1]. If `frequency_scaling` is less than 1, then 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** — Number of rows and columns in the reconstructed image

positive integer

Number of rows and columns in the reconstructed image, specified as a positive integer. If `output_size` is not specified, the size is determined from the length of the projections according to:

```
output_size = 2*floor(size(R,1)/(2*sqrt(2)))
```

If you specify `output_size`, then `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, then the reconstructed image might not be the same size as the original image.

## Output Arguments

### **I** — Grayscale image

numeric matrix | `gpuArray`

Grayscale image, returned as a numeric matrix. If input projection data `R` is data type `single`, then `I` is `single`; otherwise `I` is `double`.

If the inverse Radon transform is computed using a GPU, then `I` is returned as a `gpuArray` containing a numeric matrix.

Data Types: `single` | `double`

### **H** — Frequency response

numeric vector

Frequency response of the filter, returned as a numeric vector.

If the inverse Radon transform is computed using a GPU, then `I` is returned as a `gpuArray` containing a numeric vector.

Data Types: `double`

## Algorithms

`i radon` assumes that the center of rotation is the center point of the projections, which is defined as `ceil(size(R,1)/2)`.

`i radon` 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.

## Extended Capabilities

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The GPU implementation of this function supports only nearest-neighbor and linear interpolation methods.

For more information, see “Image Processing on a GPU”.

### See Also

`fan2para` | `fanbeam` | `ifanbeam` | `para2fan` | `phantom` | `radon`

**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**

## 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](http://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**

## isnitr

Check if file is National Imagery Transmission Format (NITF) file

### Syntax

```
[tf, NITF_version] = isnitr(filename)
```

### Description

`[tf, NITF_version] = isnitr(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, `isnitr` 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**



## isRigid

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.

### Examples

#### Check If 2-D 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
```

## Check If 3-D 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
```

## Input Arguments

### **tform** — Geometric transformation

`affine2d` or `affine3d` geometric transformation object

Geometric transformation, specified as an `affine2d` or `affine3d` geometric transformation object.

## Output Arguments

### **TF** — Flag indicating rigid transformation

scalar

Flag indicating rigid transformation, returned as a logical scalar. TF is True when `tform` is a rigid transformation.

Data Types: `logical`

## Definitions

### Rigid Transformation

A rigid transformation includes only rotation and translation. It does not include reflection, and it does not modify the size or shape of an input object.

### See Also

`isSimilarity` | `isTranslation`

**Introduced in R2013a**

## isrset

Check if file is R-Set

## Syntax

```
[tf, supported] = isrset(filename)
```

## Description

`[tf, supported] = isrset(filename)` returns `true` if the file specified by `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**

# isSimilarity

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.

## Examples

### Check if 2-D 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
```

## Check if 3-D 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
```

## Input Arguments

### **tform** — Geometric transformation

`affine2d` or `affine3d` geometric transformation object

Geometric transformation, specified as an `affine2d` or `affine3d` geometric transformation object.

## Output Arguments

### **TF** — Flag indicating similarity transformation

scalar

Flag indicating similarity transformation, returned as a logical scalar. TF is True when `tform` is a similarity transformation.

Data Types: `logical`

## Definitions

### Similarity Transformation

A similarity transformation includes only rotation, translation, isotropic scaling, and reflection. A similarity transformation does not modify the shape of an input object. Straight lines remain straight, and parallel lines remain parallel.

---

**Note** `isSimilarity` returns True if the transformation includes reflection. Some toolbox functions, such as `imregister`, support only non-reflective similarity. Other functions, such as `fitgeotrans`, support reflection.

---

### See Also

`isRigid` | `isTranslation`

**Introduced in R2013a**

## isTranslation

Determine if transformation is pure translation

### Syntax

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

### Description

`TF = isTranslation(tform)` determines whether or not the affine transformation specified by `tform` is a pure translation.

### Examples

#### Check If 2-D 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 the transformation is a pure translation.

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



```
tf =
```

```
    1
```

### Check If 3-D Transformation Is a Pure 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 the transformation is a pure translation. Since `tform` scales the object,

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

```
tf =
```

```
    0
```

As expected, the transformation is not a pure translation since scaling changes the size and shape of an input volume.

## Input Arguments

### **tform** — Geometric transformation

`affine2d` or `affine3d` geometric transformation object

Geometric transformation, specified as an `affine2d` or `affine3d` geometric transformation object.

## Output Arguments

### TF — Flag indicating pure translation transformation

scalar

Flag indicating pure translation transformation, returned as a logical scalar. TF is True when `tform` represents a pure translation.

Data Types: `logical`

## Definitions

### Translation Transformation

A translation transformation shifts an image without modifying the image size, shape, or orientation. A 2-D translation is represented by a matrix T of the form:

```
[1 0 0;  
 0 1 0;  
 e f 1];
```

A 3-D translation is represented by a matrix of the form:

```
[1 0 0 0;  
 0 1 0 0;  
 0 0 1 0;  
 j k l 1];
```

## See Also

`isRigid` | `isSimilarity`

**Introduced in R2013a**

# jaccard

Jaccard similarity coefficient for image segmentation

## Syntax

```
similarity = jaccard(BW1,BW2)
similarity = jaccard(L1,L2)
similarity = jaccard(C1,C2)
```

## Description

`similarity = jaccard(BW1,BW2)` computes the intersection of binary images `BW1` and `BW2` divided by the union of `BW1` and `BW2`, also known as the Jaccard index. The images can be binary images, label images, or categorical images.

`similarity = jaccard(L1,L2)` computes the Jaccard index for each label in label images `L1` and `L2`.

`similarity = jaccard(C1,C2)` computes the Jaccard index for each category in categorical images `C1` and `C2`.

## Examples

### Compute Jaccard Similarity Coefficient for Binary Segmentation

Read an image containing an object to segment. Convert the image to grayscale, and display the result.

```
A = imread('hands1.jpg');
I = rgb2gray(A);
figure
imshow(I)
title('Original Image')
```

Original Image



Use the active contours (snakes) method to segment the hand.

```
mask = false(size(I));  
mask(25:end-25,25:end-25) = true;  
BW = activecontour(I, mask, 300);
```

Read in the ground truth against which to compare the segmentation.

```
BW_groundTruth = imread('hands1-mask.png');
```

Compute the Jaccard index of this segmentation.

```
similarity = jaccard(BW, BW_groundTruth);
```

Display the masks on top of each other. Colors indicate differences in the masks.

```
figure  
imshowpair(BW, BW_groundTruth)  
title(['Jaccard Index = ' num2str(similarity)])
```

Jaccard Index = 0.72158



### Compute Jaccard Similarity Coefficient for Multi-Region Segmentation

This example shows how to segment an image into multiple regions. The example then computes the Jaccard similarity coefficient for each region.

Read in an image with several regions to segment.

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

Create scribbles for three regions that distinguish their typical color characteristics. The first region classifies the yellow flower. The second region classifies the green stem and leaves. The last region classifies the brown dirt in two separate patches of the image. Regions are specified by a 4-element vector, whose elements indicate the x- and y-coordinate of the upper left corner of the ROI, the width of the ROI, and the height of the ROI.

```
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;

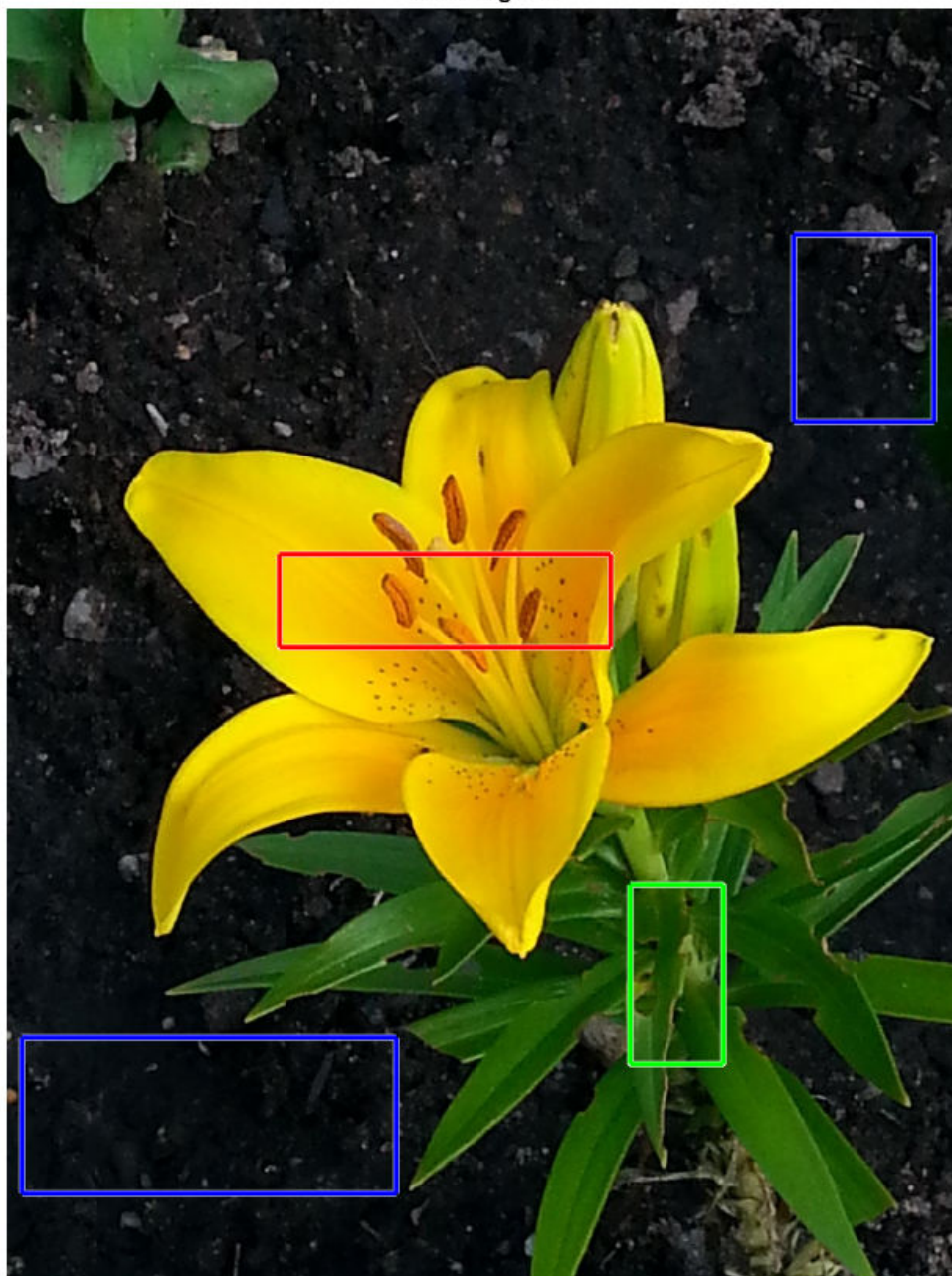
region2 = [800 1124 120 230];
BW2 = false(size(RGB,1),size(RGB,2));
BW2(region2(2):region2(2)+region2(4),region2(1):region2(1)+region2(3)) = true;

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)) = t
BW3(region3(2,2):region3(2,2)+region3(2,4),region3(2,1):region3(2,1)+region3(2,3)) = t
```

Display the seed regions on top of the image.

```
figure
imshow(RGB)
hold on
visboundaries(BW1, 'Color', 'r');
visboundaries(BW2, 'Color', 'g');
visboundaries(BW3, 'Color', 'b');
title('Seed Regions')
```

Seed Regions



Segment the image into three regions using geodesic distance-based color segmentation.

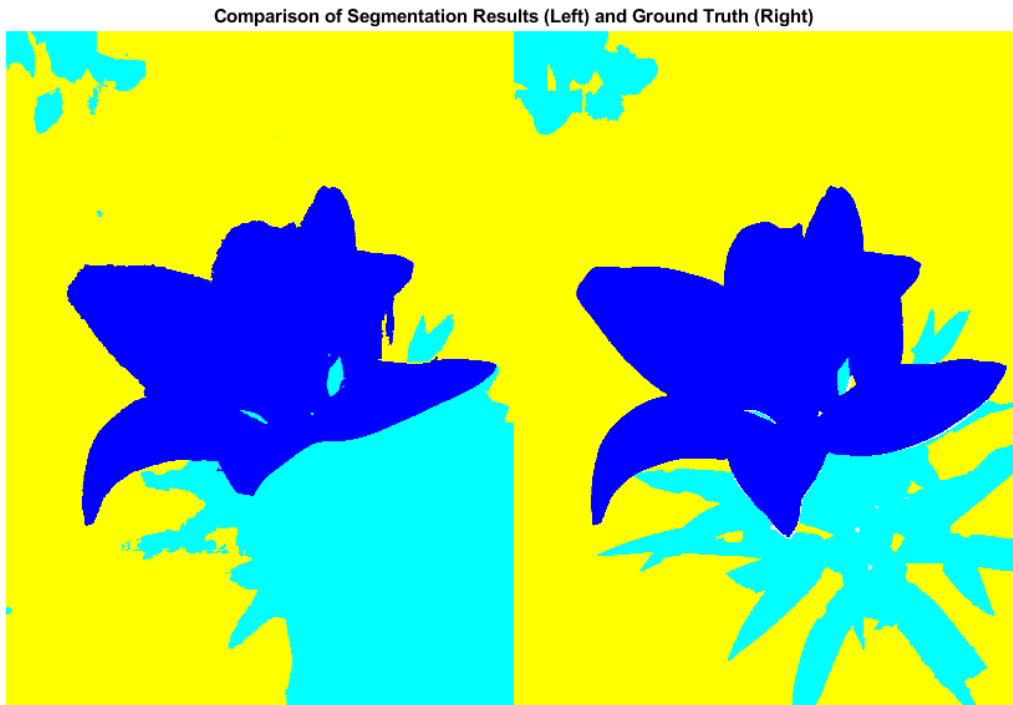
```
L = imseggeodesic(RGB,BW1,BW2,BW3,'AdaptiveChannelWeighting',true);
```

Load a ground truth segmentation of the image.

```
L_groundTruth = double(imread('yellowlily-segmented.png'));
```

Visually compare the segmentation results with the ground truth.

```
figure  
imshowpair(label2rgb(L),label2rgb(L_groundTruth),'montage')  
title('Comparison of Segmentation Results (Left) and Ground Truth (Right)')
```



Compute the Jaccard similarity index (IoU) for each segmented region.

```
similarity = jaccard(L, L_groundTruth)
```



```
similarity = 3×1
    0.8861
    0.5683
    0.8414
```

The Jaccard similarity index is noticeably smaller for the second region. This result is consistent with the visual comparison of the segmentation results, which erroneously classifies the dirt in the lower right corner of the image as leaves.

## Input Arguments

### **BW1 — First binary image**

logical array

First binary image, specified as a logical array of any dimension.

Data Types: logical

### **BW2 — Second binary image**

logical array

Second binary image, specified as a logical array of the same size as BW1.

Data Types: logical

### **L1 — First label image**

array of nonnegative integers

First label image, specified as an array of nonnegative integers, of any dimension.

Data Types: double

### **L2 — Second label image**

array of nonnegative integers

Second label image, specified as an array of nonnegative integers, of the same size as L1.

Data Types: double

### **C1 — First categorical image**

categorical array

First categorical image, specified as a `categorical` array of any dimension.

Data Types: `category`

## **C2 — Second categorical image**

`categorical` array

Second categorical image, specified as a `categorical` array of the same size as C1.

Data Types: `category`

## **Output Arguments**

### **similarity — Jaccard similarity coefficient**

numeric scalar | numeric vector

Jaccard similarity coefficient, returned as a numeric scalar or numeric vector with values in the range [0, 1]. A `similarity` of 1 means that the segmentations in the two images are a perfect match. If the input arrays are:

- binary images, `similarity` is a scalar.
- label images, `similarity` is a vector, where the first coefficient is the Jaccard index for label 1, the second coefficient is the Jaccard index for label 2, and so on.
- categorical images, `similarity` is a vector, where the first coefficient is the Jaccard index for the first category, the second coefficient is the Jaccard index for the second category, and so on.

Data Types: `double`

## **Definitions**

### **Jaccard Similarity Coefficient**

The Jaccard similarity coefficient of two sets  $A$  and  $B$  (also known as intersection over union or IoU) is expressed as:

$$\text{jaccard}(A,B) = \frac{|\text{intersection}(A,B)|}{|\text{union}(A,B)|}$$

where  $|A|$  represents the cardinal of set  $A$ . The Jaccard index can also be expressed in terms of true positives ( $TP$ ), false positives ( $FP$ ) and false negatives ( $FN$ ) as:

$$\text{jaccard}(A,B) = TP / (TP + FP + FN)$$

The Jaccard index is related to the Dice index according to:

$$\text{jaccard}(A,B) = \text{dice}(A,B) / (2 - \text{dice}(A,B))$$

## See Also

[bfscore](#) | [dice](#)

**Introduced in R2017b**

## 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](http://www.color.org)) for representing  $L^*a^*b^*$  values as unsigned 8-bit or 16-bit integers. The ICC encoding convention is illustrated by these tables.

Value ( $L^*$ )	uint8 Value	uint16 Value
0.0	0	0
100.0	255	65280
$100.0 + (25500/65280)$	None	65535

Value ( $a^*$ or $b^*$ )	uint8 Value	uint16 Value
-128.0	0	0
0.0	128	32768
127.0	255	65280
$127.0 + (255/256)$	None	65535

## Class Support

`lab` is a `uint8`, `uint16`, or `double` array that must be real and nonsparse. `labd` is `double`.

## Examples

### Convert 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 = 1×3
```

```
    100     0     0
```

## See Also

[applycform](#) | [lab2uint16](#) | [lab2uint8](#) | [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 = 1×3
    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 = 1×3
    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 = 1×3
    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 = 1x3 uint8 row vector
    188    167    153
```

## Input Arguments

### lab — Color values to convert

*c*-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 *c*-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 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, [1.000, 1.000, 1.000]. Useful as a theoretical reference.
'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 horizon light.
'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.



Value	White Point
'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 [0.9642, 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

numeric array

Converted color values, returned as a numeric array of the same shape as the input. The output type is the same as the input type unless you specify 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**

lab2xyz | rgb2lab | 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](http://www.color.org)) for representing  $L^*a^*b^*$  values as unsigned 8-bit or 16-bit integers. The ICC encoding convention is illustrated by these tables.

Value ( $L^*$ )	uint8 Value	uint16 Value
0.0	0	0
100.0	255	65280
$100.0 + (25500/65280)$	None	65535

Value ( $a^*$ or $b^*$ )	uint8 Value	uint16 Value
-128.0	0	0
0.0	128	32768
127.0	255	65280
$127.0 + (255/256)$	None	65535

## Class Support

`lab` can be a `uint8`, `uint16`, or `double` array that must be real and nonsparse. `lab16` is of class `uint16`.

## Examples

### Convert 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 = 1x3 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](http://www.color.org)) for representing  $L^*a^*b^*$  values as unsigned 8-bit or 16-bit integers. The ICC encoding convention is illustrated by these tables.

Value ( $L^*$ )	uint8 Value	uint16 Value
0.0	0	0
100.0	255	65280
$100.0 + (25500/65280)$	None	65535

Value ( $a^*$ or $b^*$ )	uint8 Value	uint16 Value
-128.0	0	0
0.0	128	32768
127.0	255	65280
$127.0 + (255/256)$	None	65535

## Class Support

`lab` is a `uint8`, `uint16`, or `double` array that must be real and nonsparse. `lab8` is `uint8`.

## Examples

### Convert 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 = 1x3 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 (2° observer).

`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 = 1×3
```

```
    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 = 1×3  
    0.1970    0.1842    0.1729
```

## Input Arguments

### lab — Color values to convert

*c*-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 *c*-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 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.



Value	White Point
'e'	Equal-energy radiator, [1.000, 1.000, 1.000]. Useful as a theoretical reference.
'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 horizon light.
'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 [0.9642, 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

numeric array

Converted color values, returned as a numeric array of the same shape and type as the input.

## See Also

lab2rgb | rgb2xyz | xyz2lab

**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 = 8×8
```

```
    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 = 1x3 cell array
    {24x1 double}    {4x1 double}    {4x1 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 = 4x1
```

```

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` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

## 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

`label2rgb` | `labelmatrix` | `superpixels`

**Introduced in R2016a**

# label2rgb

Convert label matrix into RGB image

## Syntax

```
RGB = label2rgb(L)
RGB = label2rgb(L, cmap)
RGB = label2rgb(L, cmap, zerocolor)
RGB = label2rgb(L, cmap, 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. The `label2rgb` function picks colors from the entire range of the color map.

`RGB = label2rgb(L, cmap)` specifies the colormap `cmap` to be used in the RGB image.

`RGB = label2rgb(L, cmap, zerocolor)` specifies the RGB color of the background elements (pixels labeled 0).

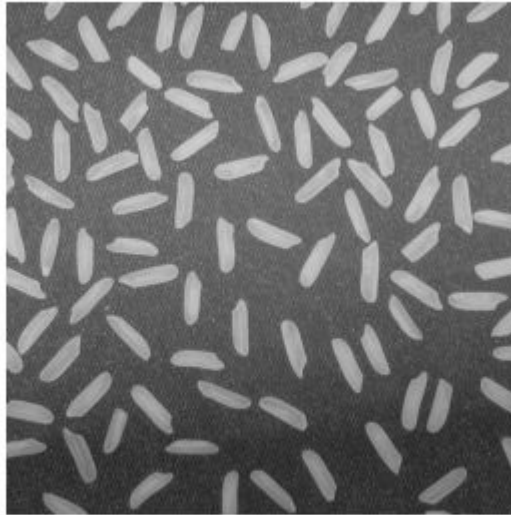
`RGB = label2rgb(L, cmap, zerocolor, order)` controls how `label2rgb` assigns colors to regions in the label matrix.

## Examples

### Use Color to Highlight Elements in Label Matrix

Read an image and display it.

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



Create a label matrix from the image.

```
BW = imbinarize(I);  
CC = bwconncomp(BW);  
L = labelmatrix(CC);
```

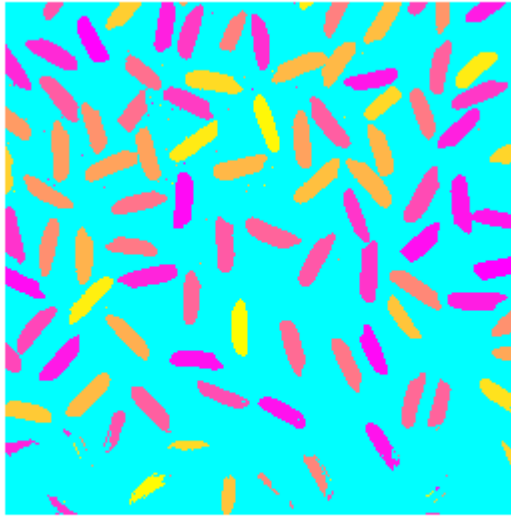
Convert the label matrix into RGB image, using default settings.

```
RGB = label2rgb(L);  
figure  
imshow(RGB)
```



Convert the label matrix into an RGB image, specifying optional parameters. This example uses the 'spring' colormap, sets background pixels to the color cyan, and randomizes how colors are assigned to the labels.

```
RGB2 = label2rgb(L, 'spring', 'c', 'shuffle');  
figure  
imshow(RGB2)
```



## Input Arguments

### **L** — Label matrix

*m*-by-*n* matrix of nonnegative integers

Label matrix of contiguous regions, specified as an *m*-by-*n* matrix of nonnegative integers. Pixels labeled 0 are the background. Pixels labeled 1 make up one object; pixels labeled 2 make up a second object; and so on.

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

### **cmap** — Color map

'jet' (default) | *c*-by-3 matrix | colormap function | handle

Color map, specified as one of the following.



Value	Description
<i>c</i> -by-3 matrix of data type double	Color map matrix specifying <i>c</i> colors, each as an RGB triple. <i>c</i> must be greater than or equal to the number of labels, <i>numlabels</i> , in label matrix <i>L</i> . You can compute the number of labels as <code>numlabels = max(L(:))</code> .  If <i>c</i> is greater than <i>numlabels</i> , then <code>label2rgb</code> creates the RGB image using only the first <i>numlabels</i> rows in the matrix.
colormap function	Name of a MATLAB colormap function, such as 'jet' or 'gray'. See <code>colormap</code> for a list of supported colormaps.
colormap handle	Handle of a colormap function, such as <code>@jet</code> or <code>@gray</code> .

### zerocolor — Color map

[1 1 1] (white) (default) | 3-element vector | 'b' | 'c' | 'g'

Label matrix of contiguous regions, specified as a 3-element vector representing an RGB triple, or one of the following color abbreviations.

Value	Color
'b'	Blue
'c'	Cyan
'g'	Green
'k'	Black
'm'	Magenta
'r'	Red
'w'	White
'y'	Yellow

### order — Color order

'noshuffle' (default) | 'shuffle'

Color order, specified as 'noshuffle' or 'shuffle'. The 'noshuffle' order arranges colormap colors to label matrix regions in numerical order. The 'shuffle' order assigns colormap colors pseudorandomly.

## Output Arguments

### RGB — RGB image

*m*-by-*n*-by-3 numeric matrix

RGB image, returned as an *m*-by-*n*-by-3 numeric matrix.

Data Types: `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, for best results when using the standard syntax `RGB = label2rgb(L, cmap, zerocolor, order)`:
  - Submit at least two input arguments: the label matrix, `L`, and the colormap matrix, `cmap`.
  - `cmap` must be a *c*-by-3 matrix of data type `double`. You cannot specify the name of a MATLAB colormap function or a function handle of a colormap function.
  - If you set the background color `zerocolor` to the same color as one of the regions, then `label2rgb` will not issue a warning.
  - If you supply a value for `order`, then it must be 'noshuffle'.

## See Also

[bwconncomp](#) | [bwlabel](#) | [bwlabeln](#) | [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, `L`, from the connected components structure `CC` returned by `bwconncomp`.

`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.

### 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.**

## Input Arguments

### CC — Connected components

struct

Connected components, specified as a structure with four fields.

Field	Description
Connectivity	Connectivity of the connected components (objects)
ImageSize	Size of the binary image
NumObjects	Number of connected components (objects) in the binary image.
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.

## Output Arguments

### L — Label matrix

matrix of nonnegative integers

Label matrix of contiguous regions, returned as matrix of nonnegative integers. 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 size of L is CC.ImageSize. The class of L depends on CC.NumObjects, as shown in the 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'	$CC.NumObjects \geq 2^{32}$

Data Types: double | uint8 | uint16 | uint32

## See Also

[bwconncomp](#) | [bwlabel](#) | [bwlabeln](#) | [label2rgb](#) | [regionprops](#)

**Introduced in R2009a**

# labeloverlay

Overlay label matrix regions on 2-D image

## Syntax

```
B = labeloverlay(A,L)
B = labeloverlay(A,BW)
B = labeloverlay(A,C)
B = labeloverlay( ____,Name,Value)
```

## Description

`B = labeloverlay(A,L)` fuses the input image, `A`, with a different color for each label in the label matrix, `L`.

`B = labeloverlay(A,BW)` fuses the input image with a color where the mask, `BW`, is true.

`B = labeloverlay(A,C)` fuses the input image with a different color for each label in the categorical matrix, `C`.

`B = labeloverlay( ____,Name,Value)` computes the fused overlay image, `B`, using `Name,Value` pairs to control aspects of the computation.

## Examples

### Visualize Segmentation over Color Image

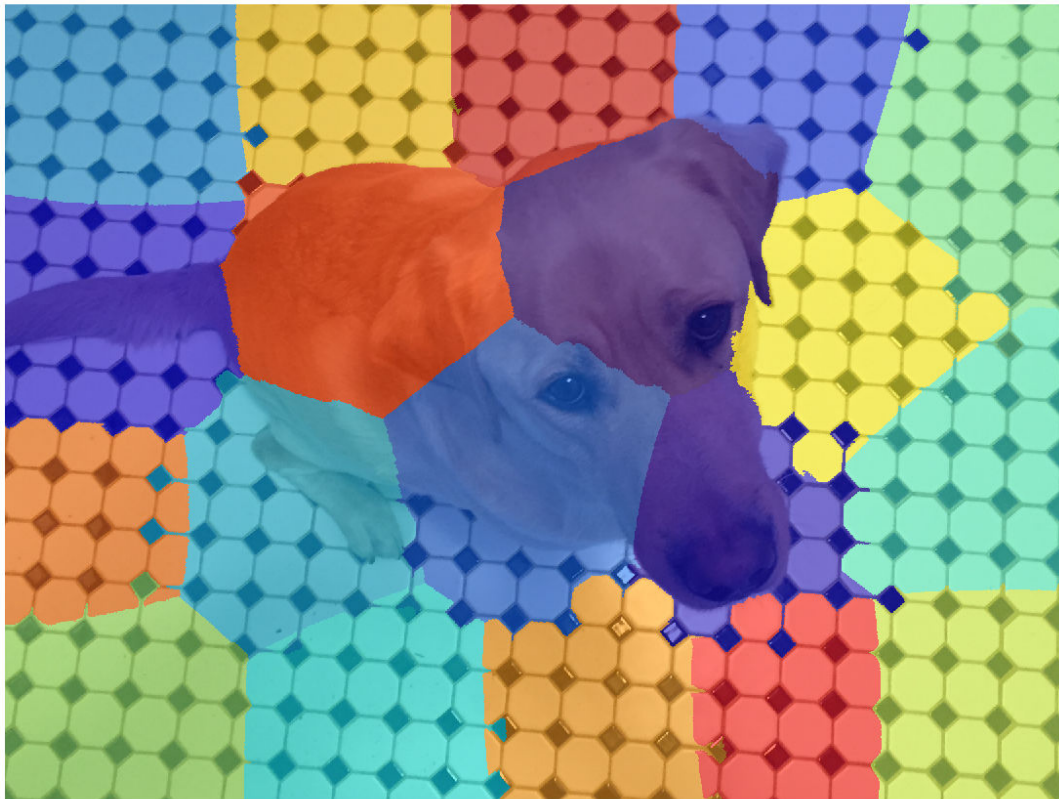
Read an image, then segment it using superpixels.

```
A = imread('kobi.png');
[L,N] = superpixels(A,20);
```

Fuse the segmentation results with the original image. Display the fused image.



```
B = labeloverlay(A,L);  
imshow(B)
```



### Visualize Binary Mask over Grayscale Image

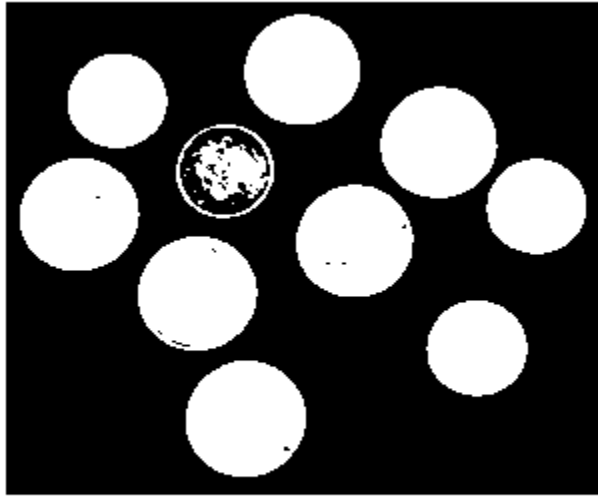
Read a grayscale image and display it.

```
A = imread('coins.png');  
imshow(A)
```



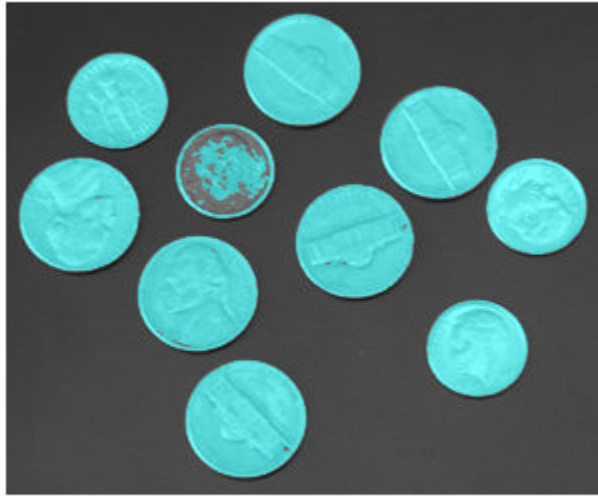
Create a mask using binary thresholding.

```
t = graythresh(A);  
BW = imbinarize(A,t);  
imshow(BW)
```



Fuse the mask with the original image. Display the fused image.

```
B = labeloverlay(A,BW);  
imshow(B)
```



## Visualize Categorical Labels over Image

Read a grayscale image and create a mask using binary thresholding.

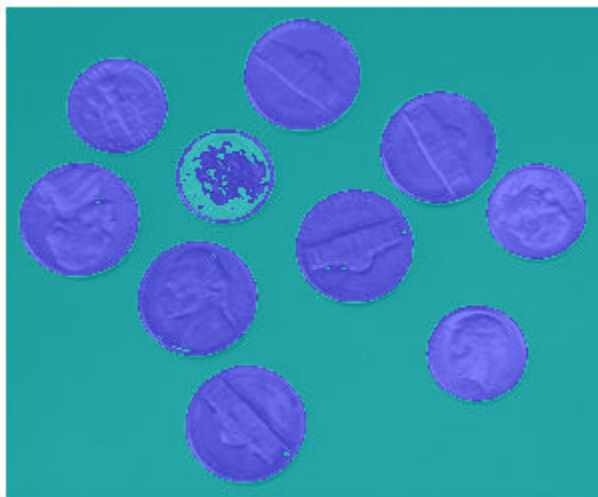
```
A = imread('coins.png');  
t = graythresh(A);  
BW = imbinarize(A,t);
```

Create categorical labels based on the image contents.

```
stringArray = repmat("table",size(BW));  
stringArray(BW) = "coin";  
categoricalSegmentation = categorical(stringArray);
```

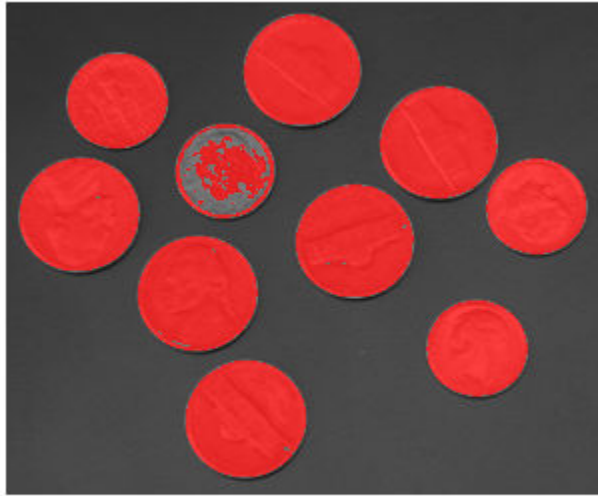
Fuse the categorical segmentation with the original image. Display the fused image.

```
B = labeloverlay(A,categoricalSegmentation);  
imshow(B)
```



Fuse the original image with only one label from the categorical segmentation. Change the colormap and make the labels more opaque, and display the result.

```
C = labeloverlay(A,categoricalSegmentation,'IncludedLabels',"coin",...  
                'Colormap','autumn','Transparency',0.25);  
imshow(C)
```



## Input Arguments

### **A — Input image**

2-D grayscale image | 2-D color image

Input image, specified as a 2-D grayscale or color image.

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

### **L — Labels**

numeric matrix

Labels, specified as a numeric matrix.

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

### **BW — Mask**

logical matrix

Mask, specified as a logical matrix.

Data Types: `logical`

### **C — Category labels**

categorical matrix

Category labels, specified as a `categorical` matrix.

Data Types: `categorical`

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `labeloverlay(myImage, myLabels, 'Colormap', 'hot')` displays labels in colors from the 'hot' color map.

### **Colormap — Color map**

'jet' (default) | *l*-by-3 color map | string | character vector

Color map, specified as the comma-separated pair consisting of 'Colormap' and one of these values:

- An *l*-by-3 color map. RGB triplets in each row of the color map must be normalized to the range [0, 1]. *l* is the number of labels in label matrix *L*, binary mask *BW*, or categorical matrix *C*.
- A string or character vector corresponding to one of the valid inputs to the `colormap` function. `labeloverlay` permutes the specified color map so that adjacent labels are more distinct.

Example: `[0.2, 0.1, 0.5; 0.1, 0.5, 0.8]`

Example: `'hot'`

Data Types: `single` | `double` | `char` | `string`

### **IncludedLabels — Labels to display**

integer | vector of integers | string | vector of strings

Labels to display in the fused image, specified as the comma-separated pair consisting of 'IncludedLabels' and one of the following:

- An integer, or vector of integers, in the range  $[1, \max(L(:))]$ .
- A string, or vector of strings, corresponding to labels in categorical matrix C.

By default, all labels are displayed.

Example: `[1,3,4]`

Example: `["flower","stem"]`

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

### Transparency — Transparency

`0.5` (default) | number in the range  $[0, 1]$

Transparency of displayed labels, specified as the comma-separated pair consisting of 'Transparency' and a number in the range  $[0, 1]$ .

- A value of `0` makes the colored labels completely opaque.
- A value of `1` makes the colored labels completely transparent.

Data Types: `single` | `double`

## Output Arguments

### B — Fused image

numeric matrix

Fused image, returned as a numeric matrix with the same size as A.

Data Types: `uint8`

## See Also

`imoverlay` | `imshowpair` | `superpixels`

## Topics

“Semantic Segmentation Basics” (Computer Vision System Toolbox)



**Introduced in R2017b**

## 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

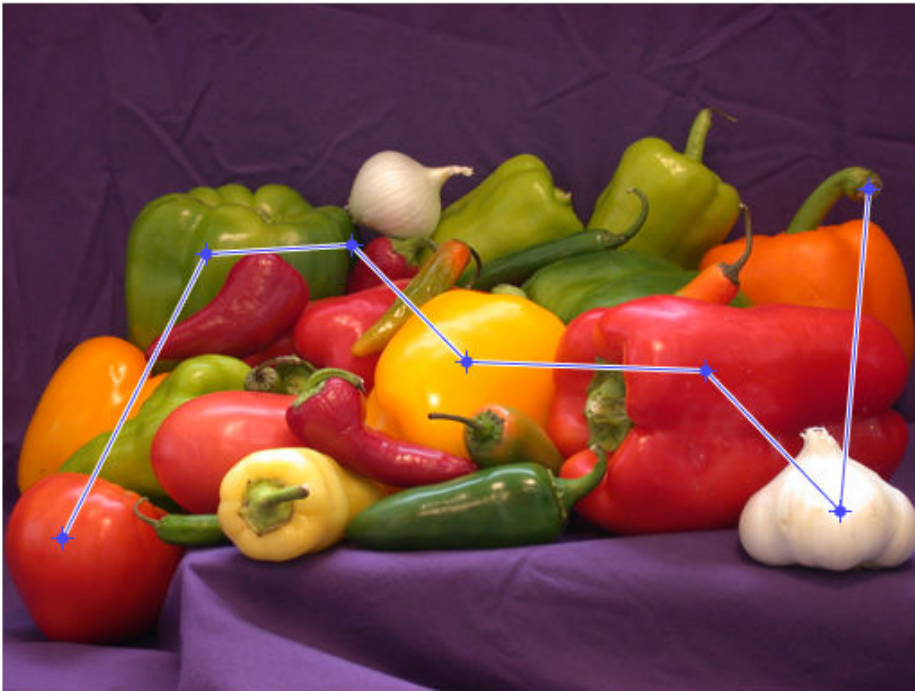
#### Segment Image into Foreground and Background

Read an image into the workspace.

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

Mark locations on image as foreground.

```
figure;  
imshow(IMG)  
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(IMG),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)
```



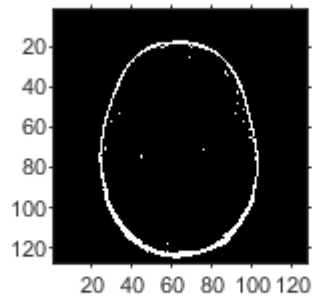
## Segment Volume in Foreground and Background

Load 3-D volumetric image into the workspace.

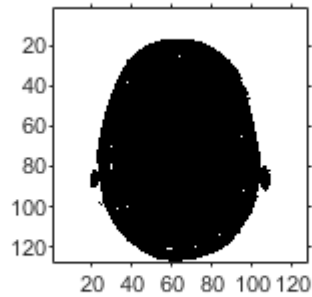
```
D = load('mri.mat');  
V = squeeze(D.D);
```

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.

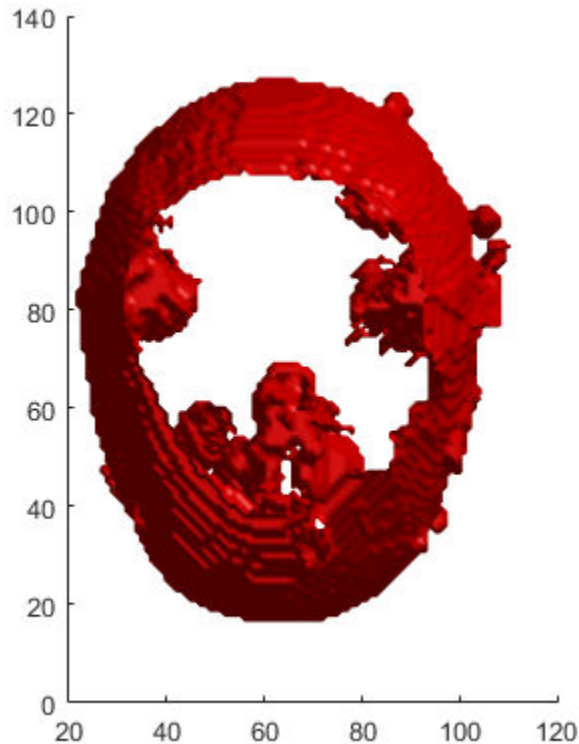
```
L = superpixels3(V,500);
```

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
```



## 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.

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 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 the comma-separated pair consisting of `'Connectivity'` and one of the following: 4 or 8, for 2-D images, and 6, 18, or 26 for 3-D images (volumes).

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 the comma-separated pair consisting of `'EdgeWeightScaleFactor'` and 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

**Image Segmenter**

## Topics

“Segmentation Using Graph Cut in Image Segmenter”

**Introduced in R2017a**

## lin2rgb

Apply gamma correction to linear RGB values

### Syntax

```
B = lin2rgb(A)
B = lin2rgb(A,Name,Value)
```

### Description

`B = lin2rgb(A)` applies a gamma correction to the linear RGB values in image `A` so that `B` is in the sRGB color space, which is suitable for display.

`B = lin2rgb(A,Name,Value)` applies gamma correction using name-value pairs to control additional options.

### Examples

#### Plot Gamma Curve of sRGB and Adobe RGB

Define a range of linear values. This vector defines 257 equally spaced points between 0 and 1.

```
lin = linspace(0,1,257);
```

Apply gamma correction to the linear values based on the sRGB standard. Then apply gamma correction to the linear values based on the Adobe RGB (1998) standard.

```
sRGB = lin2rgb(lin);
adobeRGB = lin2rgb(lin, 'ColorSpace', 'adobe-rgb-1998');
```

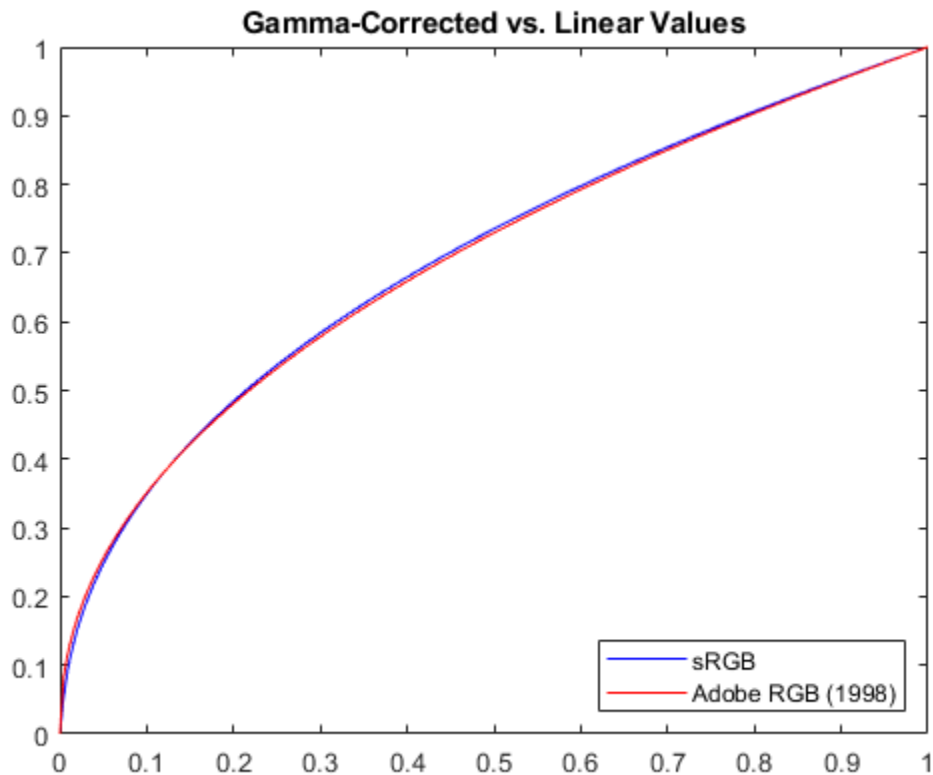
Plot the gamma-corrected curves.

```
figure
plot(lin,sRGB,'b',lin,adobeRGB,'r')
```

```

title('Gamma-Corrected vs. Linear Values')
legend('sRGB', 'Adobe RGB (1998)', 'Location', 'southeast')

```



For an alternative visualization, plot color bars representing each color space.

```

cb_lin = ones(30,257) .* lin;
cb_sRGB = ones(30,257) .* sRGB;
cb_adobeRGB = ones(30,257) .* adobeRGB;

figure
subplot(3,1,1); imshow(cb_lin); title('Linear RGB')
subplot(3,1,2); imshow(cb_sRGB); title('sRGB');
subplot(3,1,3); imshow(cb_adobeRGB); title('Adobe RGB (1998)');

```

**Linear RGB**



**sRGB**



**Adobe RGB (1998)**



The gamma-corrected color spaces get brighter more quickly than the linear color space, as expected.

### **Apply sRGB Gamma Correction to Linear RGB Image**

Open an image file containing minimally processed linear RGB intensities.

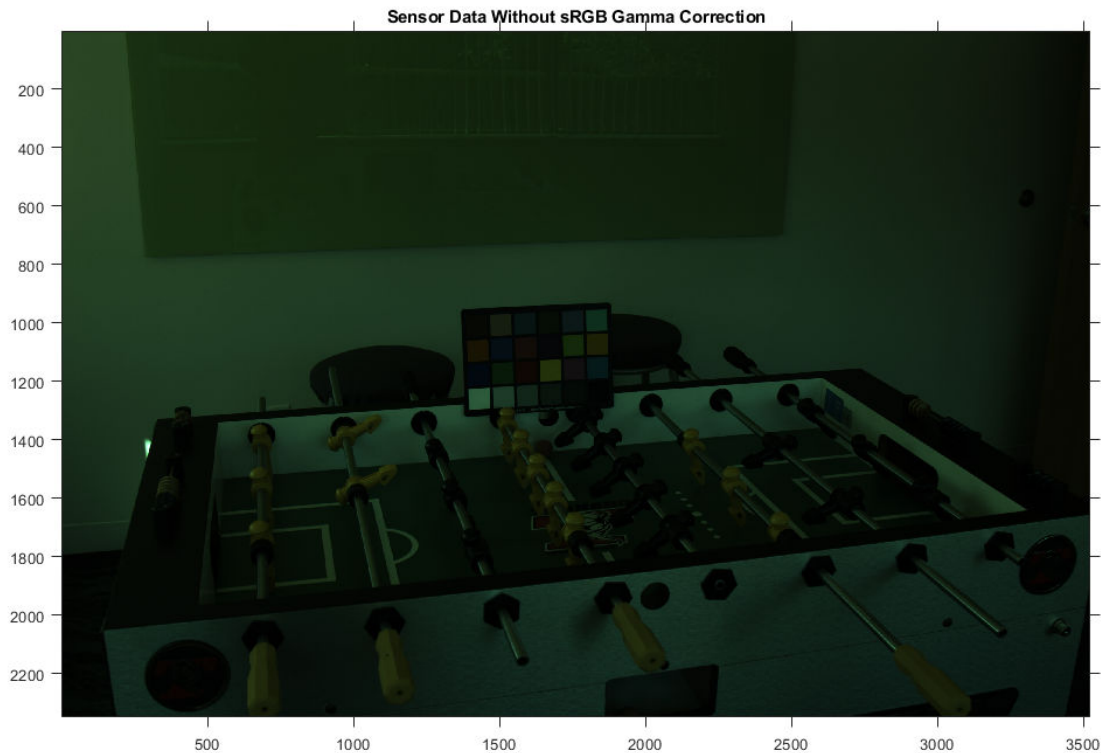
```
A = imread('foosballraw.tiff');
```

The image data is the raw sensor data after correcting the black level and scaling to 16 bits per pixel. Interpolate the intensities to reconstruct color by using the `demosaic` function. The color filter array pattern is RGGB.

```
A_demosaiced = demosaic(A, 'rggb');
```

Display the image. To shrink the image so that it appears fully on the screen, set the optional initial magnification to a value less than 100.

```
figure
imshow(A_demosaiced, 'InitialMagnification', 25)
title('Sensor Data Without sRGB Gamma Correction')
```

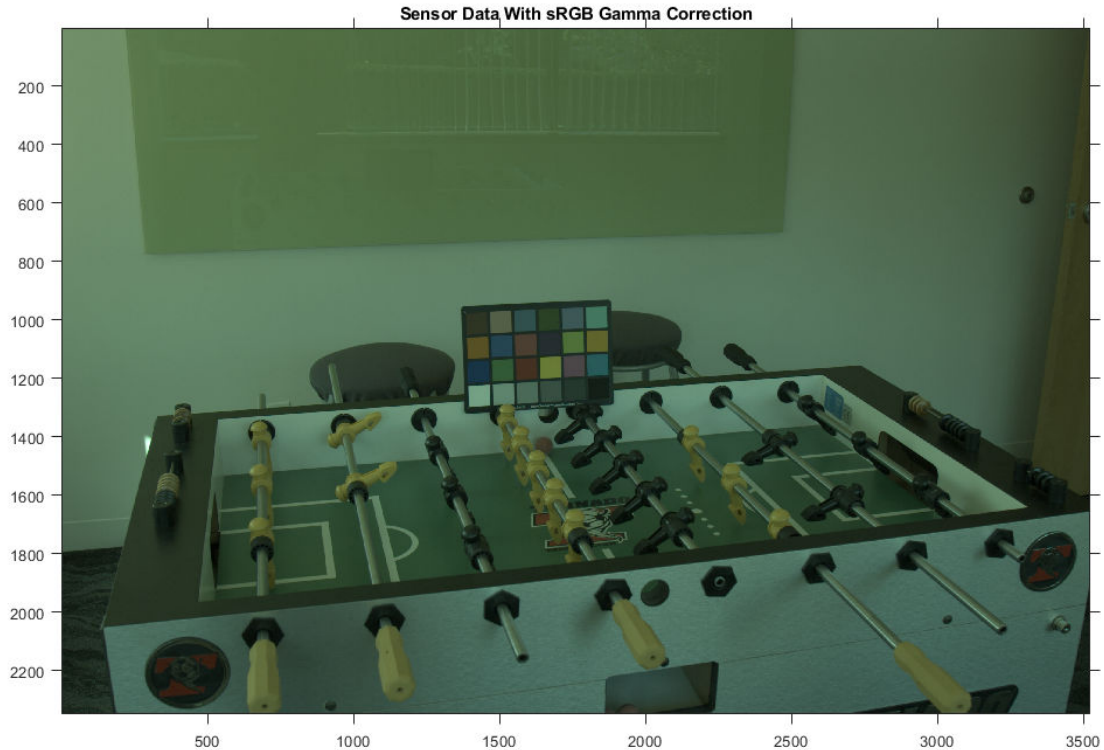


The image appears dark because it is in the linear RGB color space. Apply gamma correction to the image according to the sRGB standard, storing the values in double precision.

```
A_sRGB = lin2rgb(A_demosaiced, 'OutputType', 'double');
```

Display the gamma-corrected image, setting the optional magnification.

```
figure
imshow(A_sRGB,'InitialMagnification',25)
title('Sensor Data With sRGB Gamma Correction');
```



The gamma-corrected image looks brighter than the linear image, as expected.

## Input Arguments

### A — Linear RGB image

real, nonsparse,  $m$ -by- $n$ -by-3 array

Linear RGB image, specified as a real, nonsparse,  $m$ -by- $n$ -by-3 array.

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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `B = lin2rgb(I, 'ColorSpace', 'adobe-rgb-1998')` applies gamma correction to an image, `I`, according to the Adobe RGB (1998) standard.

### **ColorSpace** — Color space of the output image

`'srgb'` (default) | `'adobe-rgb-1998'`

Color space of the output image, specified as the comma-separated pair consisting of `'ColorSpace'` and `'srgb'` or `'adobe-rgb-1998'`.

Data Types: `char` | `string`

### **OutputType** — Data type of output RGB values

`'double'` | `'single'` | `'uint8'` | `'uint16'`

Data type of the output RGB values, specified as the comma-separated pair consisting of `'OutputType'` and `'double'`, `'single'`, `'uint8'`, or `'uint16'`. By default, the output data type is the same as the data type of `A`.

Data Types: `char` | `string`

## Output Arguments

### **B** — Gamma-corrected RGB image

real, nonsparse,  $m$ -by- $n$ -by-3 array

Gamma-corrected RGB image, returned as a real, nonsparse,  $m$ -by- $n$ -by-3 array.

## Algorithms

### **Gamma Correction Using the sRGB Standard**

The gamma correction to transform linear RGB tristimulus values into sRGB tristimulus values is defined by the following parametric curve:

$$f(u) = -f(-u), \quad u < 0$$

$$f(u) = c \cdot u, \quad 0 \leq u < d$$

$$f(u) = a \cdot u^\gamma + b, \quad u \geq d,$$

where  $u$  represents a color value with these parameters:

$$a = 1.055$$

$$b = -0.055$$

$$c = 12.92$$

$$d = 0.0031308$$

$$\gamma = 1/2.4$$

## Gamma Correction Using the Adobe RGB (1998) Standard

The gamma correction to transform linear RGB tristimulus values into Adobe RGB (1998) tristimulus values uses a simple power function:

$$v = u^\gamma, \quad u \geq 0$$

$$v = -(-u)^\gamma, \quad u < 0,$$

with

$$\gamma = 1/2.19921875$$

## References

- [1] Ebner, Marc. "Gamma Correction." *Color Constancy*. Chichester, West Sussex: John Wiley & Sons, 2007.
- [2] Adobe Systems Incorporated. "Inverting the color component transfer function." *Adobe RGB (1998) Color Image Encoding*. Section 4.3.5.2, May 2005, p.12.

## See Also

rgb2lin

**Introduced in R2017b**

## 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;  
B2 = localcontrast(A, edgeThreshold, amount);
```

Display the new results again, compared to the original image.

```
imshowpair(A, B2, '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**

`imadjust` | `imcontrast` | `imsharpen` | `locallapfilt`

**Introduced in R2016b**

## locallapfilt

Fast local Laplacian filtering of images

### Syntax

```
B = locallapfilt(I,sigma,alpha)
B = locallapfilt(I,sigma,alpha,beta)
B = locallapfilt( ____,Name,Value)
```

### Description

`B = locallapfilt(I,sigma,alpha)` filters the grayscale or RGB image `I` with an edge-aware, fast local Laplacian filter. `sigma` characterizes the amplitude of edges in `I`. `alpha` controls smoothing of details.

`B = locallapfilt(I,sigma,alpha,beta)` filters the image using `beta` to control the dynamic range of `A`.

`B = locallapfilt( ____,Name,Value)` uses name-value pairs to control advanced aspects of the filter. Argument 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')
```



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(@() localapfilt(A, sigma, alpha, 'NumIntensityLevels', 20))  
t_speed = 0.1769
```

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'])
```

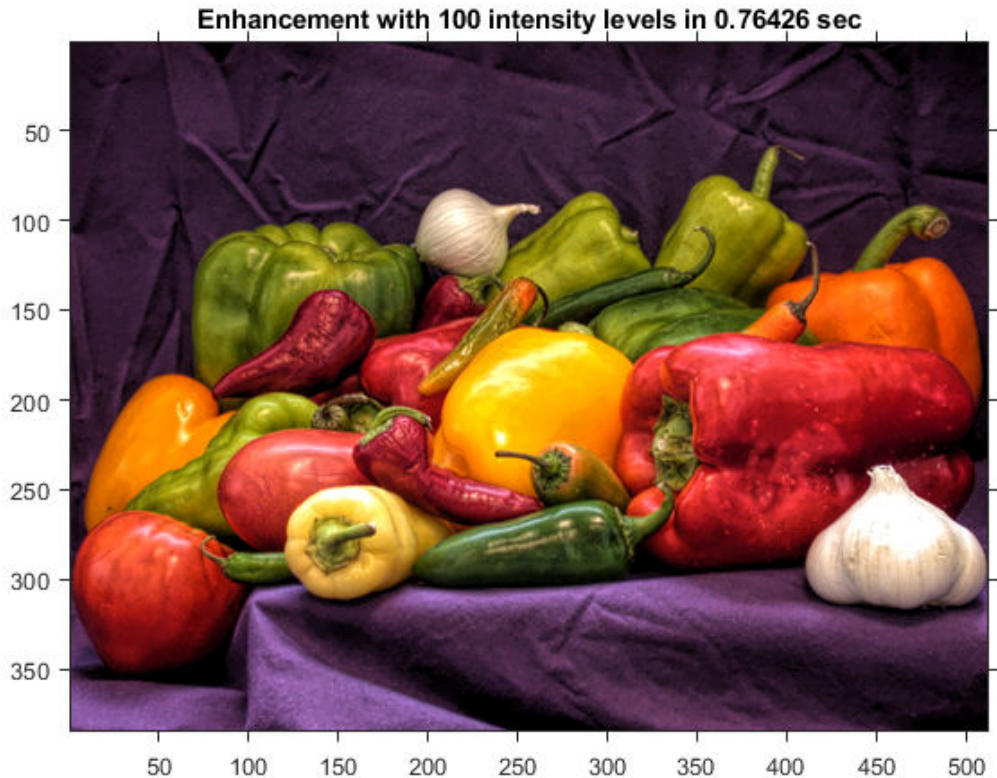


A larger number of samples yields better looking results at the expense of more processing time. Time the function using 100 intensity levels.

```
t_quality = timeit(@() locallapfilt(A, sigma, alpha, 'NumIntensityLevels', 100))  
t_quality = 0.7643
```

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'])
```



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  $\alpha$ . By default, `locallapfilt` 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')
```



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);
```

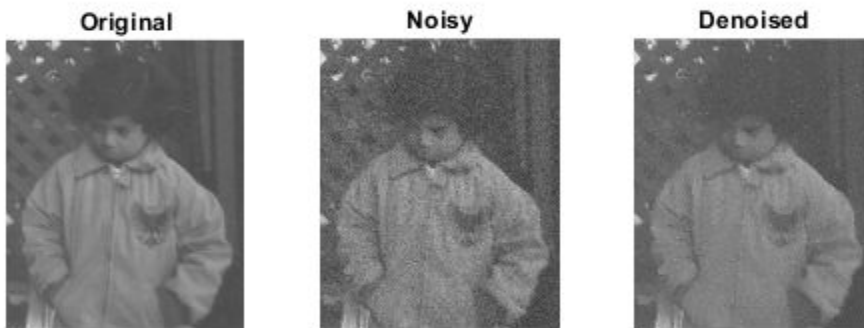
The peak signal-to-noise ratio of the denoised image is 32.3065

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

### **I** — Image to filter

2-D grayscale image | 2-D color image

Image to filter, specified as a 2-D grayscale image of size  $m$ -by- $n$  or a 2-D color image of size  $m$ -by- $n$ -by-3.

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
<code>alpha</code> less than 1	Increases the details of the input image, effectively enhancing the local contrast of the image without affecting edges or introducing halos.
<code>alpha</code> greater than 1	Smooths details in the input image while preserving crisp edges
<code>alpha</code> 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.
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 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 | string

### 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 'auto' 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` | `string`

## 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

`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 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

`locallapfilt` | `tonemap`

**Introduced in R2016b**

# images.geotrans.LocalWeightedMeanTransformation2D

2-D local weighted mean geometric transformation

## Description

A `LocalWeightedMeanTransformation2D` object encapsulates a 2-D local weighted mean geometric transformation.

## Creation

You can create a `LocalWeightedMeanTransformation2D` object using the following methods:

- The `fitgeotrans` function, which estimates a geometric transformation that maps pairs of control points between two images.
- The `images.geotrans.LocalWeightedMeanTransformation2D` described here. This function creates a `LocalWeightedMeanTransformation2D` object using coordinates of fixed points and moving points, and a specified number of points to use in the local weighted mean calculation.

## Syntax

```
tform = images.geotrans.LocalWeightedMeanTransformation2D(  
movingPoints, fixedPoints, n)
```

## Description

`tform = images.geotrans.LocalWeightedMeanTransformation2D(movingPoints, fixedPoints, n)` creates a `LocalWeightedMeanTransformation2D` object given control point coordinates in `movingPoints` and `fixedPoints`, which define matched control points in the moving and fixed images, respectively. The `n` closest points are used to infer a second degree polynomial transformation for each control point pair.

## 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

## Properties

### **Dimensionality** — Dimensionality of the geometric transformation

2

Dimensionality of the geometric transformation for both input and output points, specified as the value 2.

## Object Functions

outputLimits Find output spatial limits given input spatial limits  
transformPointsInverse Apply inverse geometric transformation

## 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 the `LocalWeightedMeanTransformation2D` object at the control points.

```
movingPointsComputed = transformPointsInverse(tformLocalWeightedMean, fixedPoints);
```

```
errorInFit = hypot(movingPointsComputed(:,1)-movingPoints(:,1), ...  
                  movingPointsComputed(:,2)-movingPoints(:,2))
```

## Algorithms

The local weighted mean transformation infers 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.

## See Also

### Functions

`cpselect` | `fitgeotrans` | `imwarp`

### Objects

`PiecewiseLinearTransformation2D` | `PolynomialTransformation2D` | `affine2d`  
| `projective2d`

**Introduced in R2013b**

## makecform

Create color transformation structure

The `makecform` function supports conversions between members of the family of device-independent color spaces defined by the *Commission Internationale de l'Éclairage* (International Commission on Illumination, or CIE). `makecform` also supports conversions to and from the *sRGB* and *CMYK* color spaces. To perform a color space transformation, pass the color transformation structure created by `makecform` as an argument to the `applycform` function.

### Syntax

```
C = makecform(type)
C = makecform(type, 'WhitePoint', WP)
C = makecform(type, 'AdaptedWhitePoint', WP)
C = makecform('adapt', 'WhiteStart', WPS, 'WhiteEnd', WPE, 'AdaptModel',
model)

C = makecform('srgb2cmyk', 'RenderingIntent', intent)
C = makecform('cmyk2srgb', 'RenderingIntent', intent)

C = makecform('icc', src_profile, dest_profile)
C = makecform('icc', src_profile,
dest_profile, 'SourceRenderingIntent',
src_intent, 'DestRenderingIntent', dest_intent)

C = makecform('mattrc', MatTRC, 'Direction', direction)
C = makecform('mattrc', profile, 'Direction', direction)
C = makecform('mattrc', profile, 'Direction',
direction, 'RenderingIntent', trc_intent)
C = makecform('graytrc', profile, 'Direction', direction)
C = makecform('graytrc', profile, 'Direction',
direction, 'RenderingIntent', trc_intent)

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



## Description

`C = makecform(type)` creates a color transformation structure `C` that defines the color space conversion specified by `type`.

`C = makecform(type, 'WhitePoint', WP)` specifies the value of the reference white point, `WP`, for 'xyz2lab' or 'lab2xyz' conversions.

`C = makecform(type, 'AdaptedWhitePoint', WP)` specifies the adapted white point, `WP`, for 'srgb2lab', 'lab2srgb', 'srgb2xyz', or 'xyz2srgb' conversions.

`C = makecform('adapt', 'WhiteStart', WPS, 'WhiteEnd', WPE, 'AdaptModel', model)` creates a linear chromatic-adaptation color transformation using the chromatic-adaptation model, `model`, starting with whitepoint `WPS` and ending with whitepoint `WPE`.

`C = makecform('srgb2cmyk', 'RenderingIntent', intent)` and

`C = makecform('cmyk2srgb', 'RenderingIntent', intent)` specify the rendering intent for color transformations between sRGB IEC61966-2.1 and "Specifications for Web Offset Publications" (SWOP) CMYK.

`C = makecform('icc', src_profile, dest_profile)` creates a color transformation based on two ICC profiles, `src_profile` and `dest_profile`.

`C = makecform('icc', src_profile, dest_profile, 'SourceRenderingIntent', src_intent, 'DestRenderingIntent', dest_intent)` creates a color transformation based on two ICC color profiles, `src_profile` and `dest_profile`, specifying the rendering intent for the source and destination profiles.

`C = makecform('mattrc', MatTRC, 'Direction', direction)` creates a color transformation based on a Matrix/Tone Reproduction Curve (MatTRC) model, in either the forward or inverse direction.

`C = makecform('mattrc', profile, 'Direction', direction)` creates a color transformation based on the 'MatTRC' field of the ICC color profile `profile`, in either the forward or inverse direction.

`C = makecform('mattrc', profile, 'Direction', direction, 'RenderingIntent', trc_intent)` adds the option of specifying the rendering intent.

`C = makecform('graytrc',profile,'Direction',direction)` creates a monochrome transformation based on a single-channel Tone Reproduction Curve (GrayTRC) contained in an ICC color profile.

`C = makecform('graytrc',profile,'Direction',direction,'RenderingIntent',trc_intent)` adds the option of specifying the rendering intent.

`C = makecform('clut',profile,LUTtype)` creates a color transformation based on a color lookup table of the type `LUTtype`, contained in an ICC color profile, `profile`.

`C = makecform('named',profile,space)` creates a color transformation from a named color profile (with a `NamedColor2` field) to coordinates in the color space space.

## Examples

### Convert sRGB Image to L\*a\*b\*

Convert RGB image to  $L^*a^*b^*$ , assuming input image is *sRGB*.

```
rgb = imread('peppers.png');  
cform = makecform('srgb2lab');  
lab = applycform(rgb,cform);
```

### Convert RGB to XYZ

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');
```

## Input Arguments

**type** — Color space conversion type

'cmyk2srgb' | 'srgb2cmyk' | 'lab2xyz' | 'xyz2lab' | ...

Color space conversion type, specified as one of the following character vectors. For a list of the abbreviations used by the Image Processing Toolbox software for each color space, see “Definitions” on page 1-2186.

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.
'upvpl2xyz'	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.
'xyl2xyz'	Convert from the $xyY$ to the <i>XYZ</i> color space.
'xyz2lab'	Use xyz2lab instead.
'xyz2srgb'	Use xyz2rgb instead.
'xyz2upvpl'	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.
'xyz2xyl'	Convert from the <i>XYZ</i> to the $xyY$ color space.

Data Types: char | string

### WP — White point

[0.9642 1.0000 0.8249] (default) | 1-by-3 numeric vector

Reference or adapted white point, specified as a 1-by-3 numeric vector of *XYZ* values, scaled so that  $Y = 1$ . Use the `whitepoint` function to create the **WP** vector. The default white point is the vector returned by `whitepoint('ICC')`.

To get an adaptive whitepoint value that is consistent with some published *sRGB* equations, set the value of **WP** to [0.9504, 1.0000, 1.0888], which is the vector returned by `whitepoint('D65')`.

**WPS, WPE — Starting or ending white point**

1-by-3 numeric vector

Starting or ending white point used for a linear chromatic-adaptation transform, specified as a 1-by-3 numeric vector of XYZ values, scaled so that  $Y = 1$ . Use the `whitepoint` function to create the WPS or WPE vector.

**intent — Rendering intent**

'Perceptual' (default) | 'AbsoluteColorimetric' | 'RelativeColorimetric' | 'Saturation'

Rendering intent, specified as 'Perceptual', 'AbsoluteColorimetric', 'RelativeColorimetric', or 'Saturation'.

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. Each rendering intent has distinct aesthetic and color-accuracy trade-offs.

Value	Description
'AbsoluteColorimetric'	Maps all out-of-gamut colors to the nearest gamut surface while maintaining the relationship of all in-gamut colors. This absolute rendering contains color data that is relative to a perfectly reflecting diffuser.
'Perceptual' (default)	Employs vendor-specific gamut mapping techniques for optimizing the range of producible colors of a given device. The objective is to provide the most aesthetically pleasing result even though the relationship of the in-gamut colors might not be maintained. This media-relative rendering contains color data that is relative to the device's white point.
'RelativeColorimetric'	Maps all out-of-gamut colors to the nearest gamut surface while maintaining the relationship of all in-gamut colors. This media-relative rendering contains color data that is relative to the device's white point.

Value	Description
'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.

**src\_intent, dest\_intent — Source or destination rendering intent**

'Perceptual' (default) | 'AbsoluteColorimetric' | 'RelativeColorimetric' | 'Saturation'

Source or destination rendering intent for a color transformation between two ICC profiles, specified as 'Perceptual', 'AbsoluteColorimetric', 'RelativeColorimetric', or 'Saturation'. For more information, see `intent`.

**trc\_intent — Rendering intent for tone reproduction curve**

'RelativeColorimetric' (default) | 'AbsoluteColorimetric'

Rendering intent for tone reproduction curve (MatTRC or grayTRC), specified as 'RelativeColorimetric' or 'AbsoluteColorimetric'. When 'AbsoluteColorimetric' is specified, the colorimetry is referenced to a perfect diffuser, rather than to the media white point of the ICC color profile, `profile`. For more information, see `intent`.

**model — Chromatic-adaptation model**

'Bradford' (default) | 'vonKries'

Chromatic-adaptation model used to create a linear chromatic-adaptation transform, specified as 'Bradford' or 'vonKries'.

**profile — ICC color profile**

struct

ICC color profile, specified as a structure as returned by `iccread`. If `profile` is a named color profile, it must have a `NamedColor2` field.

**src\_profile, dest\_profile — Source or destination ICC color profile**

struct

Source or destination ICC color profile, specified as a structure as returned by `iccread`.

### MatTRC — Matrix/tone reproduction curve model

struct

Matrix/tone reproduction curve model, specified as a structure. `MatTRC` is typically obtained from the `'MatTRC'` field of an ICC profile structure returned by `iccread`, based on tags contained in an ICC color profile. The `MatTRC` model contains an *RGB*-to-*XYZ* matrix and *RGB* tone reproduction curves.

### direction — Direction to apply tone reproduction curve model

'forward' | 'inverse'

Direction to apply the tone reproduction curve model, specified as `'forward'` or `'inverse'`.

- For a multi-channel tone reproduction curve (`'mattrc'`), `'forward'` applies the model in the *RGB* to *XYZ* direction, and `'inverse'` applies the model in the *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](http://www.color.org).
- For a single-channel tone reproduction curve (`'graytrc'`), `'forward'` applies the model in the device to PCS direction, and `'inverse'` applies the model in the 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`.

### LUTtype — Lookup table type

'AToB0' (default) | 'AToB1' | 'BToA0' | 'Gamut' | 'Preview0' | ...

Lookup table type, specified as one of the following values. `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 <http://www.color.org>.

LUT Type	Description
'AToB0' (default)	Device to PCS: perceptual rendering intent
'AToB1'	Device to PCS: media-relative colorimetric rendering intent

LUT Type	Description
'AToB2'	Device to PCS: saturation rendering intent
'AToB3'	Device to PCS: ICC-absolute rendering intent
'BToA0'	PCS to device: perceptual rendering intent
'BToA1'	PCS to device: media-relative colorimetric rendering intent
'BToA2'	PCS to device: saturation rendering intent
'BToA3'	PCS to device: ICC-absolute rendering intent
'Gamut'	Determines which PCS colors are out of gamut for a given device
'Preview0'	PCS colors to the PCS colors available for soft proofing using the perceptual rendering
'Preview1'	PCS colors available for soft proofing using the media-relative colorimetric rendering.
'Preview2'	PCS colors to the PCS colors available for soft proofing using the saturation rendering.

### space — Color space

'PCS' | 'Device'

Color space, specified as '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. The 'Device' option, when active, returns device coordinates, the dimension depending on the 'ColorSpace' field in profile.Header. Coordinates are always returned in 'double' format.

## Output Arguments

### C — Color transformation

struct

Color transformation structure, returned as a struct.

## Definitions

### Color Space Abbreviations

The Image Processing Toolbox software uses the following abbreviations to represent color spaces.

Abbreviation	Description
xyz	1931 CIE XYZ tristimulus values (2° observer)
xy $\ell$	1931 CIE xyY chromaticity values (2° observer), where x and y refer to the xy-coordinates of the associated CIE chromaticity diagram, and $\ell$ refers to Y (luminance).
uv $\ell$	1960 CIE uvY values, where u and v refer to the uv-coordinates, and $\ell$ refers to Y (luminance).
upv $\ell$	1976 CIE u'v'Y values, where up and vp refer to the u'v'-coordinates and $\ell$ refers to Y (luminance).
lab	1976 CIE L*a*b* values. Note that $\ell$ 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

[applycform](#) | [iccread](#) | [iccwrite](#) | [isicc](#) | [lab2rgb](#) | [lab2xyz](#) | [rgb2lab](#) | [rgb2xyz](#) | [whitepoint](#) | [xyz2lab](#) | [xyz2rgb](#)

### External Websites

<http://www.color.org>

**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

`imdistribline` | `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 “Algorithms” on page 1-2192.)

`HDR = makehdr(files,Name,Value)` uses name-value pairs to control various aspects of the image creation. Argument names can be abbreviated and case does not matter.

---

**Note** You can use only one of the `BaseFile`, `ExposureValues`, and `RelativeExposure` arguments at a time.

---

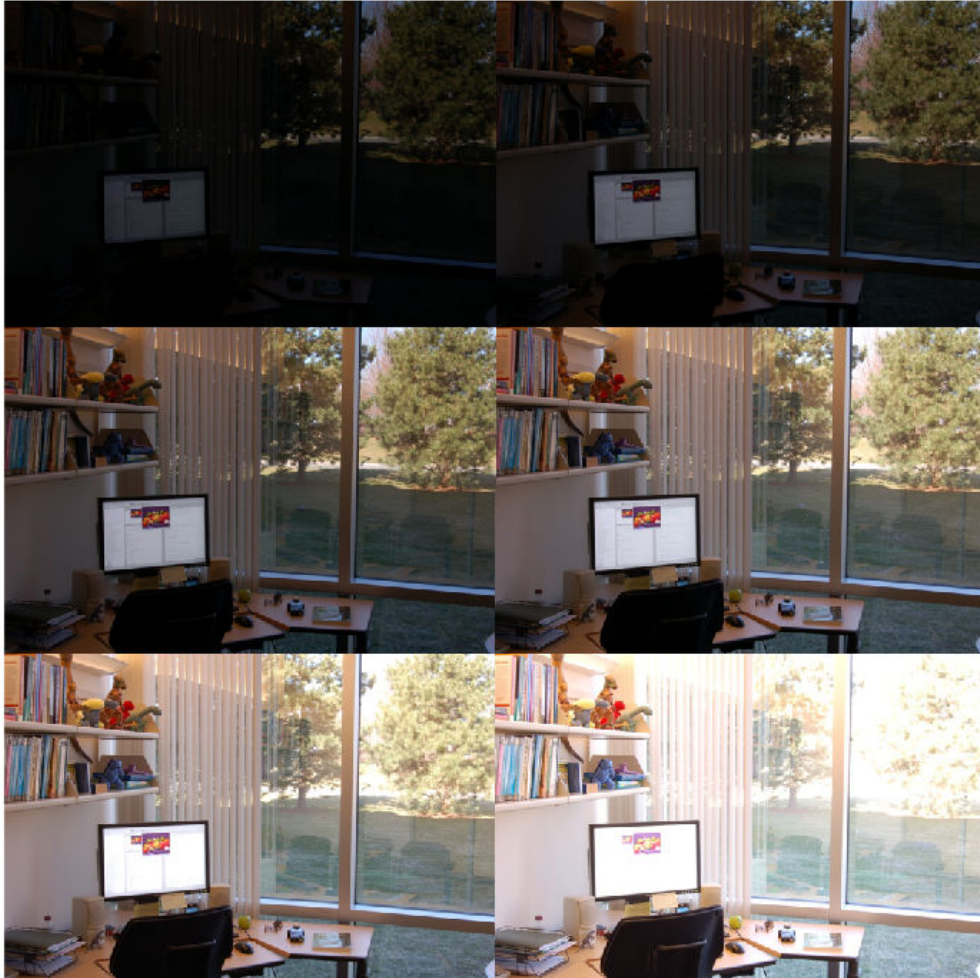
### Examples

#### 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];  
montage(files)
```



Combine the images into an HDR image.

```
hdr = makehdr(files, 'RelativeExposure', expTimes./expTimes(1));
```

Visualize the HDR image.

```
rgb = tonemap(hdr);  
imshow(rgb)
```



## Input Arguments

**files** — Set of spatially registered, low dynamic-range images  
string array | cell array of character vectors

Set of spatially registered, low dynamic-range images, specified as a string array or 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` | `string` | `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 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

`string scalar` | `character vector`

Name of the file to use as the base exposure, specified as a string scalar or character vector.

Data Types: `char` | `string`

### **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

`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

*m*-by-*n*-by-3 numeric array

High dynamic-range image, returned as an *m*-by-*n*-by-3 numeric array.

Data Types: `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

hdrread | tonemap

**Introduced in R2008a**

## makelut

Create lookup table for use with `bwlookup`

### Syntax

```
lut = makelut(fun,n)
```

### Description

`lut = makelut(fun,n)` creates a lookup table. `fun` is a function that creates a numeric output from a 2-by-2 or 3-by-3 binary neighborhood, where `n` determines the neighborhood size. `makelut` creates a lookup table by passing all possible neighborhoods to `fun`, one at a time, and storing the outputs in vector `lut`.

Use the lookup table with `bwlookup` to perform nonlinear neighborhood filtering.

### Examples

#### Make Lookup Table for 2-by-2 Neighborhood

Construct a lookup table for 2-by-2 neighborhoods. In this example, the function passed to `makelut` returns `TRUE` if the number of 1s 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
```

## Input Arguments

### **fun** — Function handle

handle

Function handle, specified as a handle. The function must accept an  $n$ -by- $n$  binary matrix of 1s and 0s as input and return a scalar.

For more information about function handles, see “Create Function Handle” (MATLAB).

### **n** — Neighborhood size

2 | 3

Neighborhood size for the lookup table, specified as 2 or 3.

## Output Arguments

### **lut** — Lookup table

16-element numeric vector | 512-element numeric vector

Lookup table, returned as a 16-element numeric vector when  $n$  is 2, or a 512-element numeric vector when  $n$  is 3.

Data Types: double

## See Also

bwlookup

**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 = makeresampler({'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

'cubic' | 'linear' | 'nearest' | cell array

Interpolating kernel, specified as 'nearest', 'linear', 'cubic', or a cell array. The following table lists the named interpolants:

Interpolant	Description
'cubic'	Cubic interpolation
'linear'	Linear interpolation
'nearest'	Nearest-neighbor interpolation

If you want to use a custom interpolating kernel, specify 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 | string | cell

**padmethod** — Method used to assign values to output elements that map outside the input array

'bound' | 'circular' | 'replicate' | 'symmetric' | 'fill'

Method used to assign values to output elements that map outside the input array, specified as one of the following values.

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` | `string`



## 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 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 values.

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` | `string`

### PadMethod — Method used to assign values to output elements that map close to or outside the edge of the input array

character vector | string scalar

See the `padmethod` argument for more information.

Data Types: `char` | `string`

### Interpolant — Interpolating kernel

character vector | string scalar | cell array

See the `interpolant` argument for more information.

Data Types: `char` | `string` | `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 `tfmarray`. 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 | string scalar | character vector

User-defined data, specified using a string scalar, character vector, or numeric array.

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

## Output Arguments

### **R — Resampler**

structure

Resampler, returned as a structure.

## See Also

tformarray

**Introduced before R2006a**

## maketform

Create spatial transformation structure (TFORM)

---

**Note** 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.

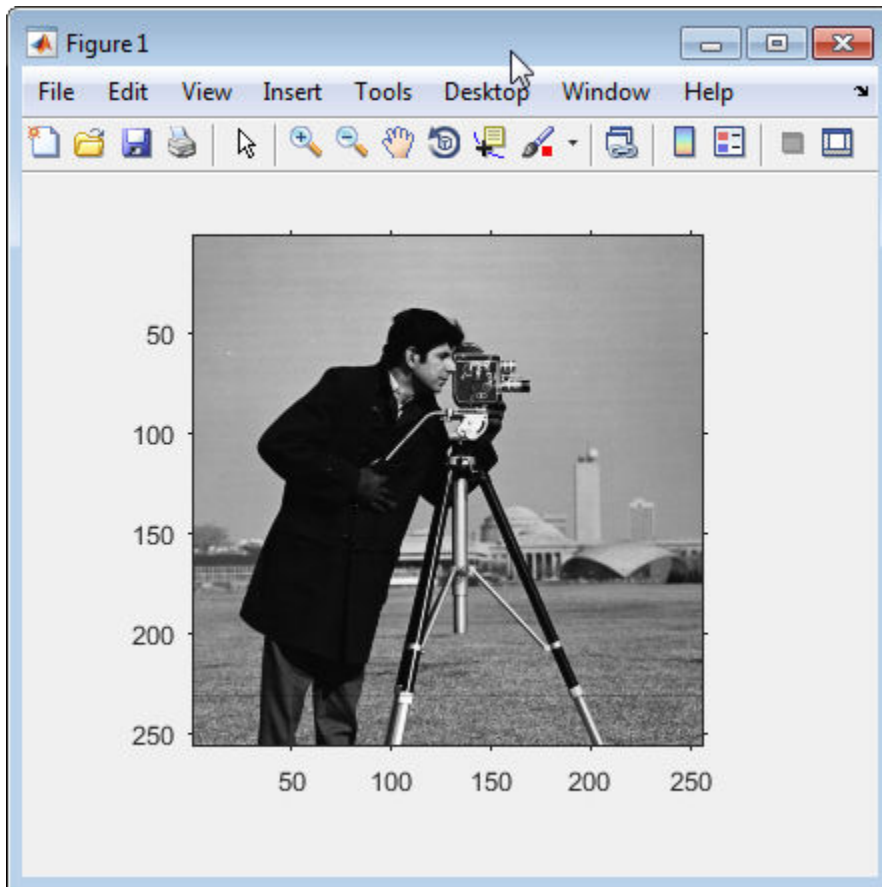
```
tformfwd([10 20],T)
```

```
ans =
```

```
    15    40
```

Read an image into the workspace and display it.

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

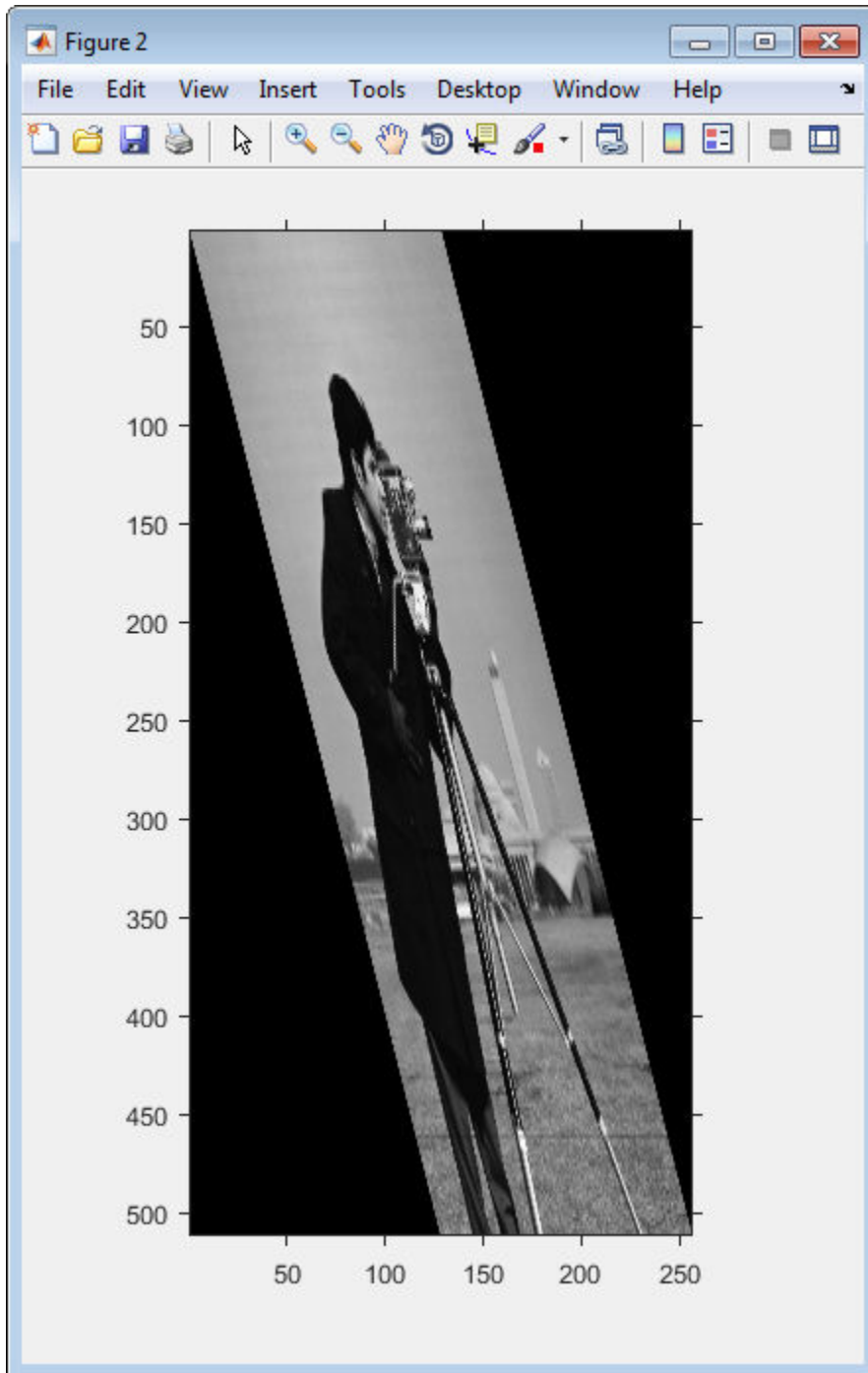


Apply the transformation to the image.

```
I2 = imtransform(I,T);
```

Display the original image and the transformed image.

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



1-2210

**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.

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.

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.

Data Types: double

### **LOW, HIGH — Corners of output box**

n-element vectors.

Corners of output box, specified as an n-element vector.

Data Types: double

## **T1, T2, . . . , TL — Forward and inverse functions**

function handles

Forward and inverse functions, specified as function handles.

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**

[fliptform](#) | [imtransform](#) | [tformarray](#) | [tformfwd](#) | [tforminv](#)

**Introduced before R2006a**

# mat2gray

Convert matrix to grayscale image

## Syntax

```
I = mat2gray(A,[amin amax])  
I = mat2gray(A)
```

## Description

`I = mat2gray(A,[amin amax])` converts the matrix `A` to an intensity image `I` that contains values in the range 0 (black) to 1 (white). `amin` and `amax` are the values in `A` that correspond to 0 and 1 in `I`. Values less than `amin` become 0, and values greater than `amax` become 1.

You optionally can perform the operation using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

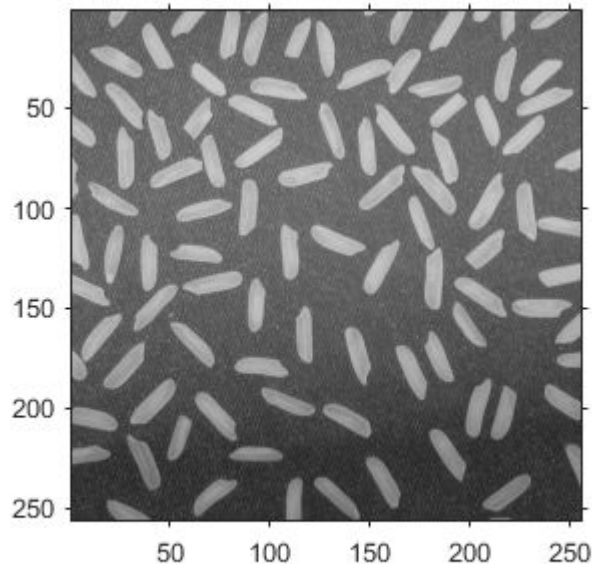
`I = mat2gray(A)` sets the values of `amin` and `amax` to the minimum and maximum values in `A`.

## Examples

### Convert a Matrix into an Image

Read an image and display it.

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



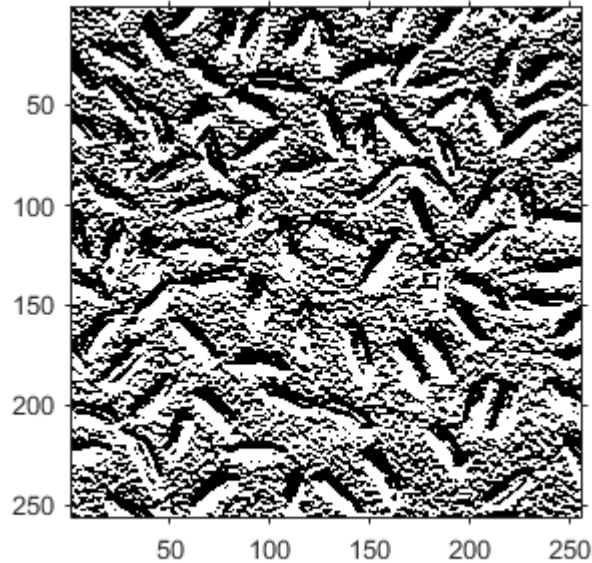
Perform an operation that returns a numeric matrix. This operation looks for edges.

```
J = filter2(fspecial('sobel'),I);  
min_matrix = min(J(:))  
  
min_matrix = -779  
  
max_matrix = max(J(:))  
  
max_matrix = 560
```

Note that the matrix has data type `double` with values outside of the range `[0,1]`, including negative values.

Display the result of the operation. Because the data range of the matrix is outside the default display range of `imshow`, every pixel with a positive value displays as white, and every pixel with a negative or zero value displays as black. It is challenging to see the edges of the grains of rice.

```
figure  
imshow(J)
```



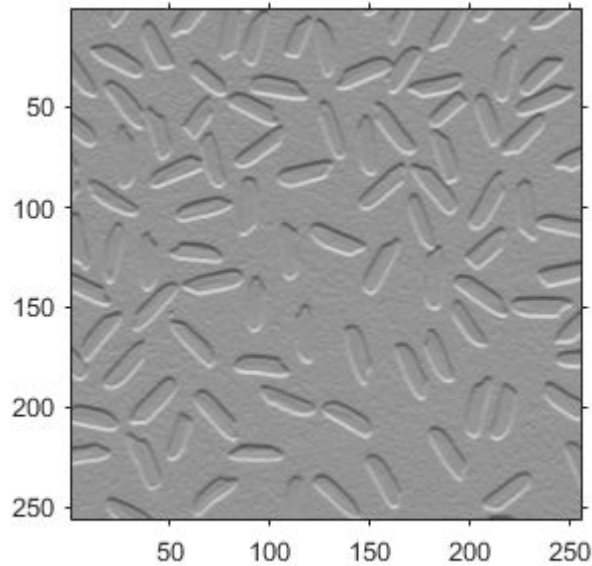
Convert the matrix into an image. Display the maximum and minimum values of the image.

```
K = mat2gray(J);  
min_image = min(K(:))  
  
min_image = 0  
  
max_image = max(K(:))  
  
max_image = 1
```

Note that values are still data type `double`, but that all values are in the range `[0, 1]`.

Display the result of the conversion. Pixels show a range of grayscale colors, which makes the location of the edges more apparent.

```
figure  
imshow(K)
```



## Input Arguments

### **A** — Input image

numeric matrix | gpuArray

Input image, specified as a numeric matrix.

To perform the computation using a GPU, specify **A** as a `gpuArray` that contains a numeric matrix.

### **[amin amax]** — Input black and white values

2-element numeric vector

Input black and white values, specified as a 2-element numeric vector.

- Values in input image A that are less than or equal to `amin` are mapped to the value 0 in the intensity image, I.
- Values in A that are greater than or equal to `amax` are mapped to the value 1 in I.

## Output Arguments

### **I** — Output intensity image

numeric matrix | gpuArray

Output intensity image, returned as a numeric matrix with values in the range [0, 1].

If the output intensity image is computed using a GPU, then B is returned as a `gpuArray` containing a numeric matrix.

Data Types: `double`

## Extended Capabilities

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

### See Also

`gray2ind` | `ind2gray` | `rescale` | `rgb2gray`

**Introduced before R2006a**

# MattesMutualInformation

Mattes mutual information metric configuration

## Description

A `MattesMutualInformation` object describes a mutual information metric configuration that you pass to the function `imregister` to solve image registration problems.

## Creation

You can create a `MattesMutualInformation` object using the following methods:

- `imregconfig` — Returns a `MattesMutualInformation` object paired with an appropriate optimizer for registering multimodal images
- Entering

```
metric = registration.metric.MattesMutualInformation;
```

on the command line creates a `MattesMutualInformation` object with default settings

## 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.

## **Examples**

### **Register Images with Mattes Mutual Information Metric**

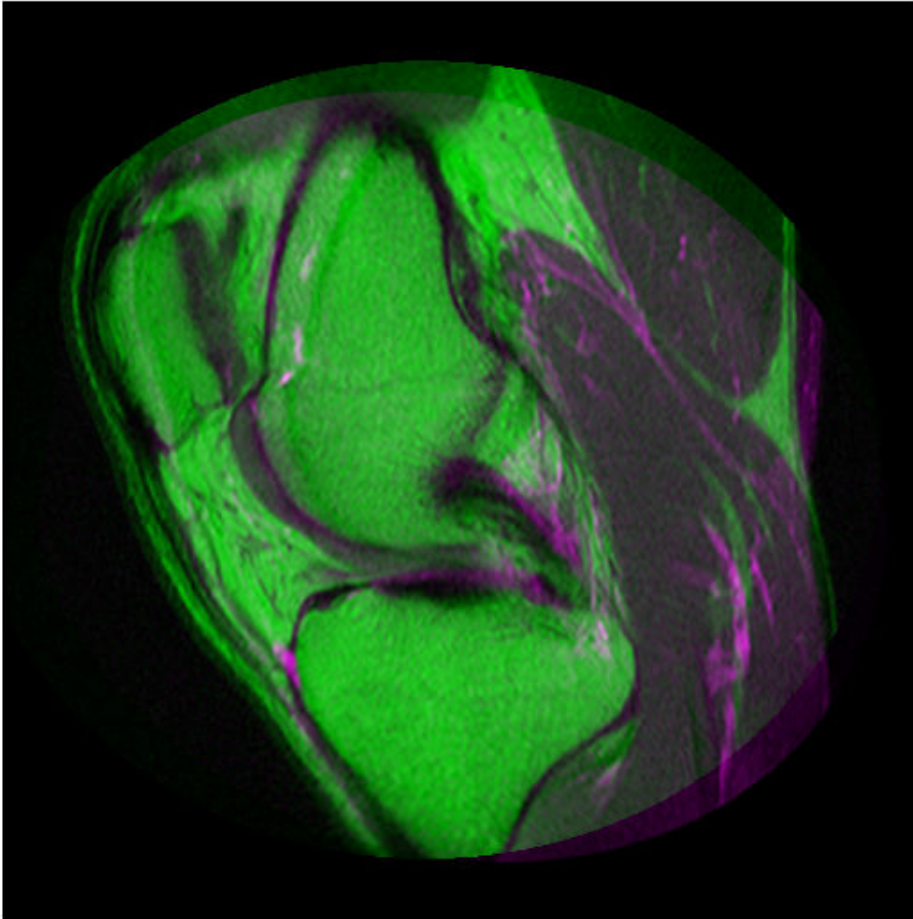
Create a `MattesMutualInformation` object and use it to register two MRI images of a knee that were obtained using different protocols.

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.

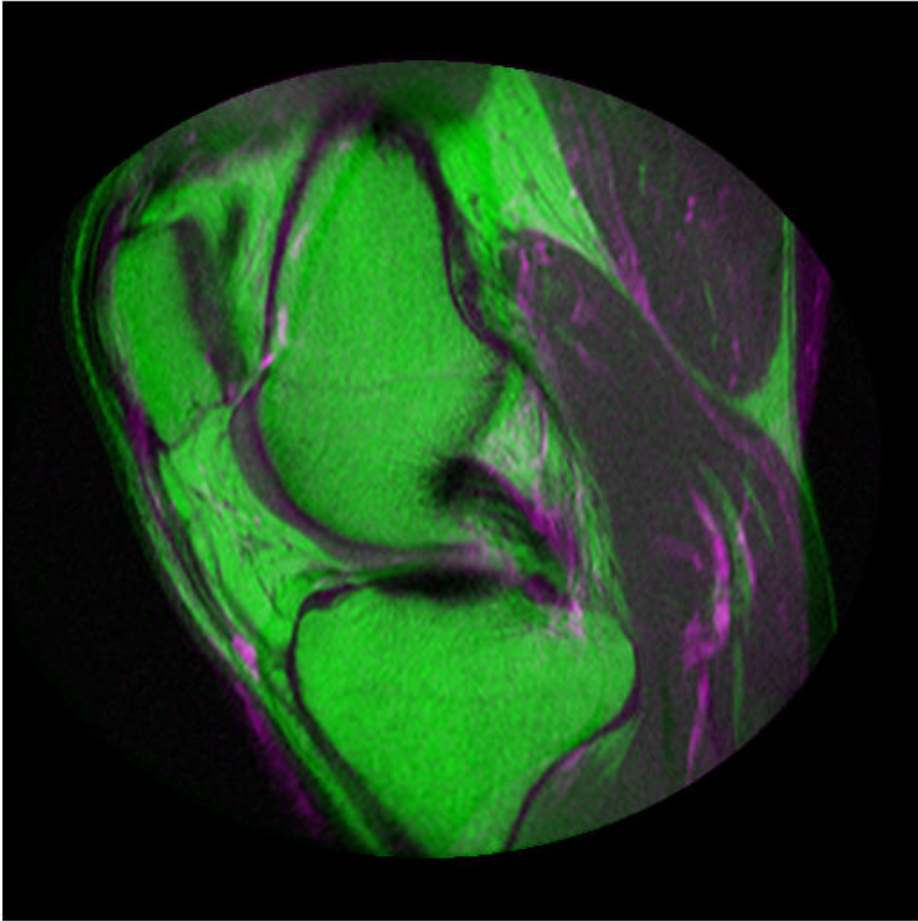
```
optimizer.InitialRadius = 0.009;  
optimizer.Epsilon = 1.5e-4;  
optimizer.GrowthFactor = 1.01;  
optimizer.MaximumIterations = 300;
```

Perform the registration.

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

View the registered images.

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



## 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

### Functions

`imregconfig` | `imregister`

### Objects

`MeanSquares` | `OnePlusOneEvolutionary` | `RegularStepGradientDescent`

### Topics

“Create an Optimizer and Metric for Intensity-Based Image Registration”

### Introduced in R2012a

# mean2

Average or mean of matrix elements

## Syntax

```
B = mean2(A)
```

## Description

`B = mean2(A)` computes the mean of all values in array `A`.

You optionally can compute the mean using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

## 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
```

### Compute 2-D Mean on a GPU

Read a grayscale image into the workspace as a `gpuArray` object, then calculate the mean of the pixel intensity values using a GPU.

```
I = gpuArray(imread('liftingbody.png'));  
meanval = mean2(I)
```

## Input Arguments

### A — Input data

numeric array | logical array | gpuArray

Input data, specified as a numerical or logical array.

To perform the computation using a GPU, specify A as a `gpuArray` that contains a numeric or logical array.

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

## Output Arguments

### B — Mean

numeric scalar | gpuArray

Mean of input data, returned as a numeric scalar. If the data type of A is `single`, then the data type of B is also `single`. Otherwise, the data type of B is `double`.

If the image mean is computed using a GPU, then B is returned as a `gpuArray` containing a numeric scalar.

Data Types: `single` | `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”.

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

## See Also

corr2 | mean | std | std2

**Introduced before R2006a**

# MeanSquares

Mean square error metric configuration

## Description

A `MeanSquares` object describes a mean square error metric configuration that you pass to the function `imregister` to solve image registration problems.

## Creation

You can create a `MeanSquares` object using the following methods:

- `imregconfig` — Returns a `MeanSquares` object paired with an appropriate optimizer for registering monomodal images
- Entering

```
metric = registration.metric.MeanSquares;  
on the command line creates a MeanSquares object
```

## 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, 'DisplayOptimizat  
ion', true);
```

## 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 squared differences.

## See Also

### Functions

`imregconfig` | `imregister`

### Objects

`MattesMutualInformation` | `OnePlusOneEvolutionary` |  
`RegularStepGradientDescent`

## Topics

“Create an Optimizer and Metric for Intensity-Based Image Registration”

**Introduced in R2012a**

## measureChromaticAberration

Measure chromatic aberration at slanted edges using Imatest® eSFR chart

### Syntax

```
aberrationTable = measureChromaticAberration(chart)
aberrationTable = measureChromaticAberration(chart,Name,Value)
```

### Description

`aberrationTable = measureChromaticAberration(chart)` measures the chromatic aberration at all slanted edge regions of interest (ROIs) of an Imatest® eSFR chart.

`aberrationTable = measureChromaticAberration(chart,Name,Value)` measures the chromatic aberration with additional parameters to specify a subset of ROIs to measure.

### Examples

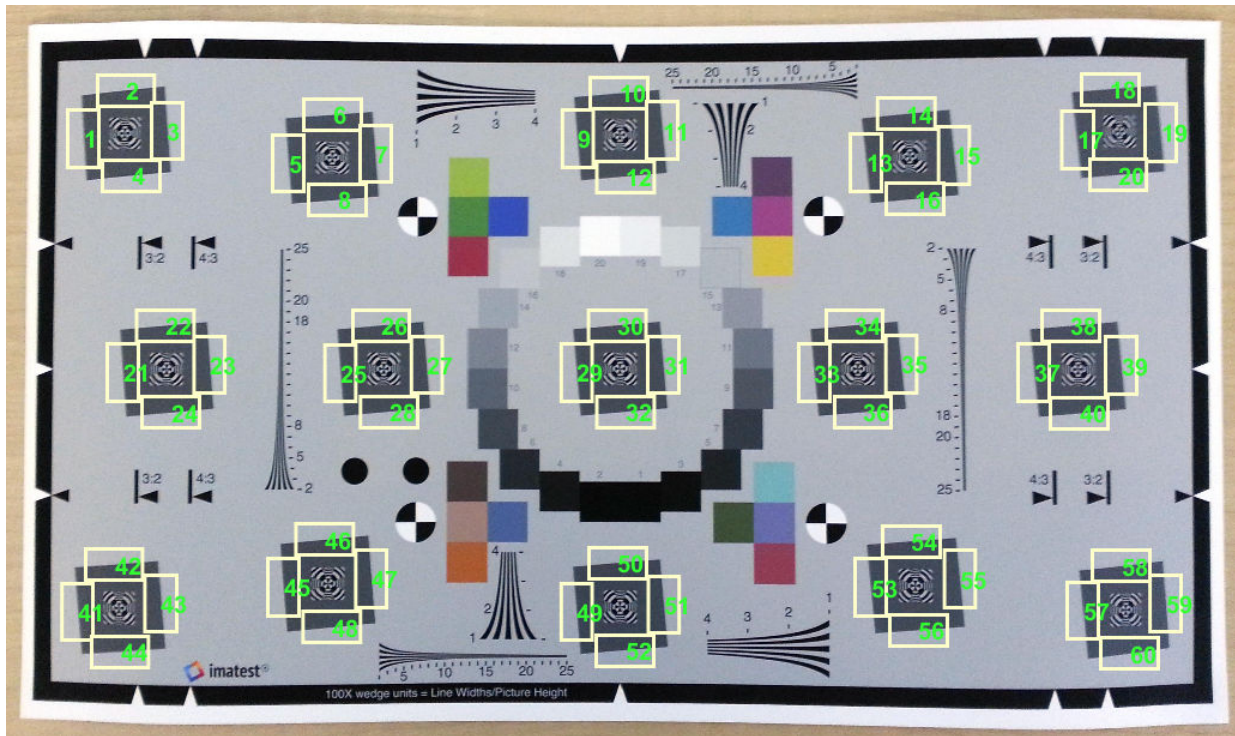
#### Measure Chromatic Aberration of Slanted Edges on eSFR Chart

Read an image of an eSFR chart into the workspace.

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

Create an `esfrChart` object, then display the chart with ROI annotations. The 60 slanted edge ROIs are labeled with green numbers.

```
chart = esfrChart(I);
displayChart(chart,'displayColorROIs',false,...
    'displayGrayROIs',false,'displayRegistrationPoints',false)
```



Measure the chromatic aberration in all slanted edge ROIs. Examine the contents of the returned table, `chTable`, for a single ROI.

```
chTable = measureChromaticAberration(chart);
ROIIndex = 3;
chTable(3,:)
```

```
ans=1x5 table
```

ROI	aberration	percentAberration	edgeProfile	normalizedEdgeProfile
3	1.9164	0.14883	[348x4 table]	[348x4 table]

Store the normalized edge profile in a separate variable, `edgeProfile`, for clarity. Examine the normalized color intensity of the first and last pixel of `edgeProfile`.

```
edgeProfile = chTable.normalizedEdgeProfile{ROIIndex};  
edgeProfile([1 end],:)
```

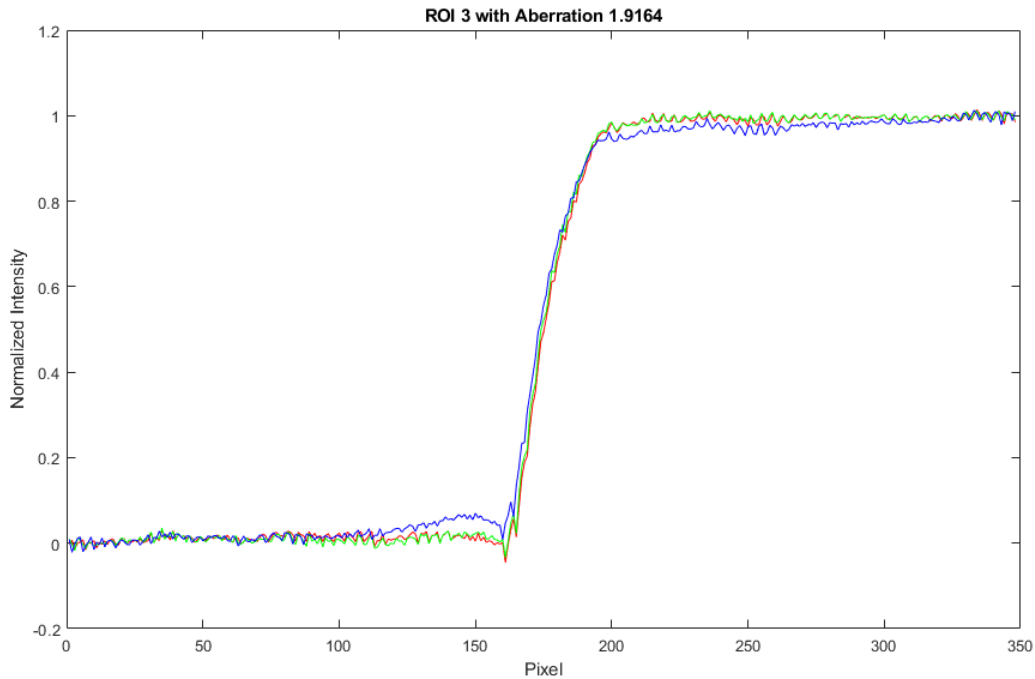
```
ans=2x4 table
```

normalizedEdgeProfile_R	normalizedEdgeProfile_G	normalizedEdgeProfile_B	n
-0.0022049	0.0094112	0.0088172	
0.98511	0.98953	1.01	

Plot the normalized intensity for the ROI.

```
npix = length(edgeProfile.normalizedEdgeProfile_R);  
plot(1:npix,edgeProfile.normalizedEdgeProfile_R,'r', ...  
     1:npix,edgeProfile.normalizedEdgeProfile_G,'g', ...  
     1:npix,edgeProfile.normalizedEdgeProfile_B,'b')  
xlabel('Pixel')  
ylabel('Normalized Intensity')  
title(['ROI ' num2str(ROIIndex) ' with Aberration ' num2str(chTable.aberration(ROIIndex))])
```





The blue channel has a higher intensity than the red and green channels immediately before the edge, and a lower intensity than the red and green channels immediately after the edge. This difference in intensity contributes to the measured value of chromatic aberration.

The measured values of `aberration` and `percentAberration` for this edge are relatively small. Visual inspection of the image confirms that the sides of the edge do not have a strong color tint.

## Input Arguments

### **chart** — eSFR chart

`esfrChart` object

eSFR chart, specified as an `esfrChart` object.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `measureChromaticAberration(myChart, 'ROIIndex', 2)` measures the chromatic aberration only of ROI 2.

### **ROIIndex** — ROI indices

1:60 (default) | scalar | vector

ROI indices to include in measurements, specified as the comma-separated pair consisting of `'ROIIndex'` and a scalar or vector of integers in the range [1, 60]. The indices match the ROI numbers displayed by `displayChart`.

---

**Note** `measureChromaticAberration` uses the intersection of ROIs specified by `'ROIIndex'` and `'ROIOrientation'`.

---

Example: 29:32

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

### **ROIorientation** — ROI orientation

'both' (default) | 'vertical' | 'horizontal'

ROI orientation, specified as the comma-separated pair consisting of `'ROIorientation'` and `'both'`, `'vertical'`, or `'horizontal'`. The `measureChromaticAberration` function performs measurements only on ROIs with the specified orientation.

---

**Note** `measureChromaticAberration` uses the intersection of ROIs specified by `'ROIIndex'` and `'ROIorientation'`.

---

Example: 'vertical'

Data Types: char | string

## Output Arguments

### aberrationTable — Chromatic aberration measurements

*m*-by-5 table

Chromatic aberration measurements, returned as an *m*-by-5 table. *m* is the number of sampled ROIs.

The five columns represent these variables:

Variable	Description
ROI	Index of the sampled ROI. The value of ROI is an integer in the range [1, 60].
aberration	Chromatic aberration, measured as the area between the maximum and the minimum red, green, and blue edge intensity profiles. The measured chromatic aberration indicates perceptual chromatic aberration. <b>aberration</b> is a scalar of type <b>double</b> .
percentAberration	Aberration, expressed as a percentage of the distance in pixels between the center of the image and the center of the ROI.
edgeProfile	Intensity profile of each color channel across the edge in the ROI. <b>edgeProfile</b> is an <i>s</i> -by-4 table, where <i>s</i> is the number of samples across the edge. The four columns represent the red, green, blue, and luminance values, averaged along the edge.  Luminance ( <i>Y</i> ) is a linear combination of the red ( <i>R</i> ), green ( <i>G</i> ), and blue ( <i>B</i> ) channels according to: $Y = 0.213R + 0.715G + 0.072B$
	<b>Note</b> The sampling rate for the chromatic aberration measurement is about four times the sampling rate of the image.
normalizedEdgeProfile	Intensity profile, normalized between [0, 1] using 5% of the front end and tail end of data. <b>normalizedEdgeProfile</b> is an <i>s</i> -by-4 table with a similar structure to <b>edgeProfile</b> .

## Tips

- Chromatic aberration is best measured at slanted edges that are:
  - Roughly orthogonal to the line connecting the center of the image and the center of the ROI
  - Farthest from the center of the image

Because chromatic aberration increases radially from the center of the image, measurements at slanted edges near the center of the image can be ignored.

- The absolute chromatic aberration reported in the `aberration` field is measured in the horizontal or vertical direction. However, chromatic aberration is a radial phenomenon, and radial measurements are more accurate.

## See Also

`displayChart` | `measureSharpness`

## Topics

“Anatomy of an eSFR Chart”

“Evaluate Quality Metrics on eSFR Test Chart”

**Introduced in R2017b**

# measureColor

Measure color reproduction using Imatest® eSFR chart

## Syntax

```
colorTable = measureColor(chart)
[colorTable,colorCorrectionMatrix] = measureColor(chart)
```

## Description

`colorTable = measureColor(chart)` measures the color values at all color regions of interest (ROIs) of an Imatest® eSFR chart.

`[colorTable,colorCorrectionMatrix] = measureColor(chart)` also returns a color correction matrix computed using a linear least squares fit.

## Examples

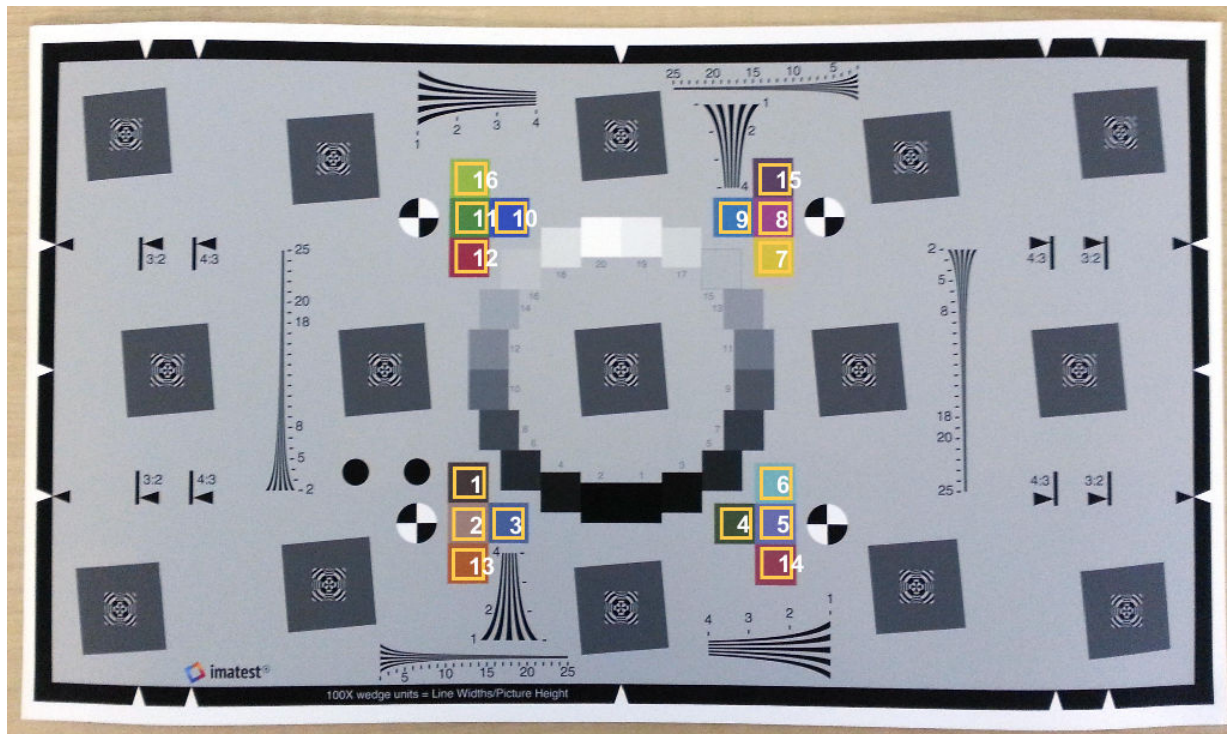
### Measure Color Accuracy of eSFR Chart

Read an image of an eSFR chart into the workspace.

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

Create an `esfrChart` object, then display the chart with ROI annotations. The 16 color patch ROIs are labeled with white numbers.

```
chart = esfrChart(I);
displayChart(chart,'displayEdgeROIs',false, ...
    'displayGrayROIs',false,'displayRegistrationPoints',false)
```



Measure the color in all color patch ROIs.

```
colorTable = measureColor(chart)
```

colorTable=16x8 table

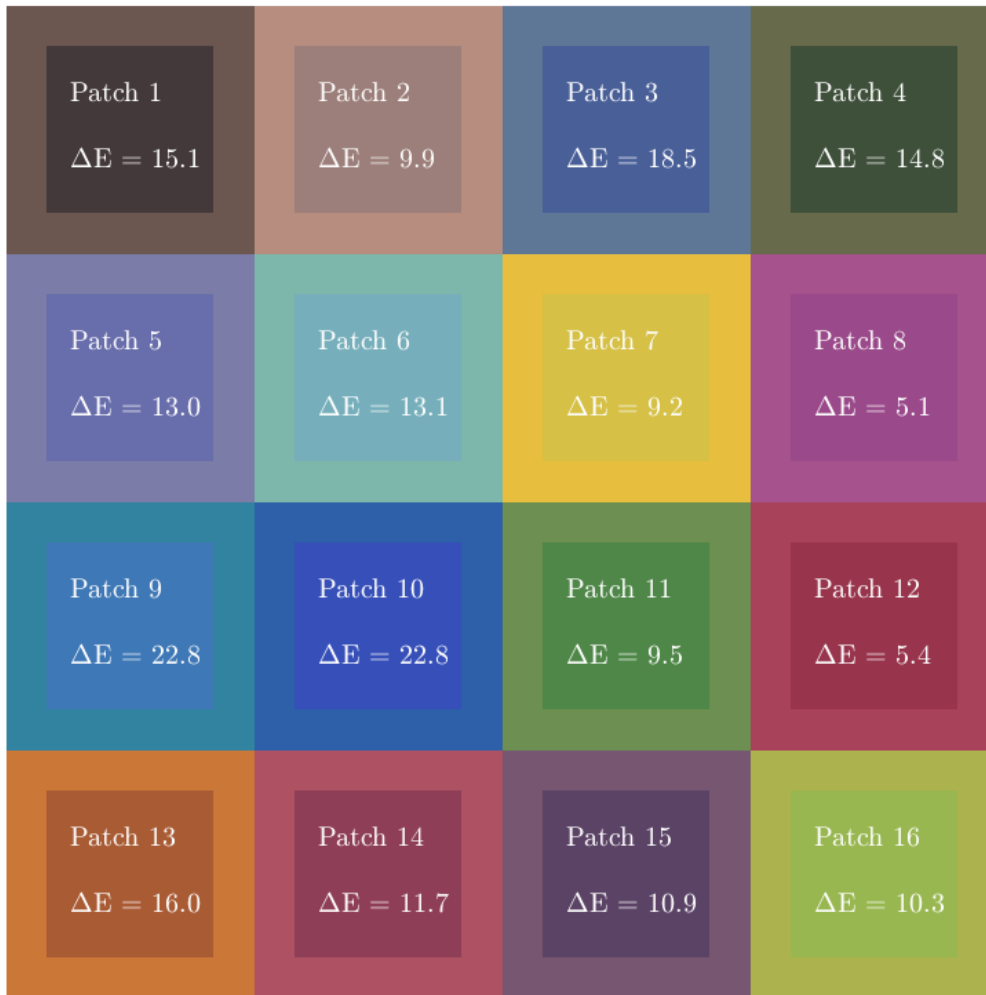
ROI	Measured_R	Measured_G	Measured_B	Reference_L	Reference_a	Ref
1	67	57	58	38.586	7.541	7
2	156	127	122	62.182	13.225	7
3	73	95	152	49.369	-0.51463	-2
4	62	79	58	43.926	-6.8587	-2
5	104	109	171	53.415	9.457	-2
6	118	175	187	69.95	-20.889	-0
7	214	192	69	78.643	1.8052	0
8	154	73	138	46.853	41.998	-2
9	62	120	182	51.05	-15.166	-2
10	55	80	185	40.811	8.7346	-2

---

11	79	135	72	55.716	-23.419
12	152	53	77	42.759	44.167
13	169	91	52	58.211	27.58
14	142	63	87	47.012	39.15
15	91	67	102	40.591	17.951
16	152	183	80	70.505	-16.318

Display the color accuracy measurements. Each square color patch is the measured color, and the thick surrounding border is the reference color for that ROI. Each color accuracy measurement is displayed as `Delta_E`, the Euclidean distance between measured and reference colors in the CIE 1976 L\*a\*b\* color space. More accurate colors have a smaller `Delta_E`.

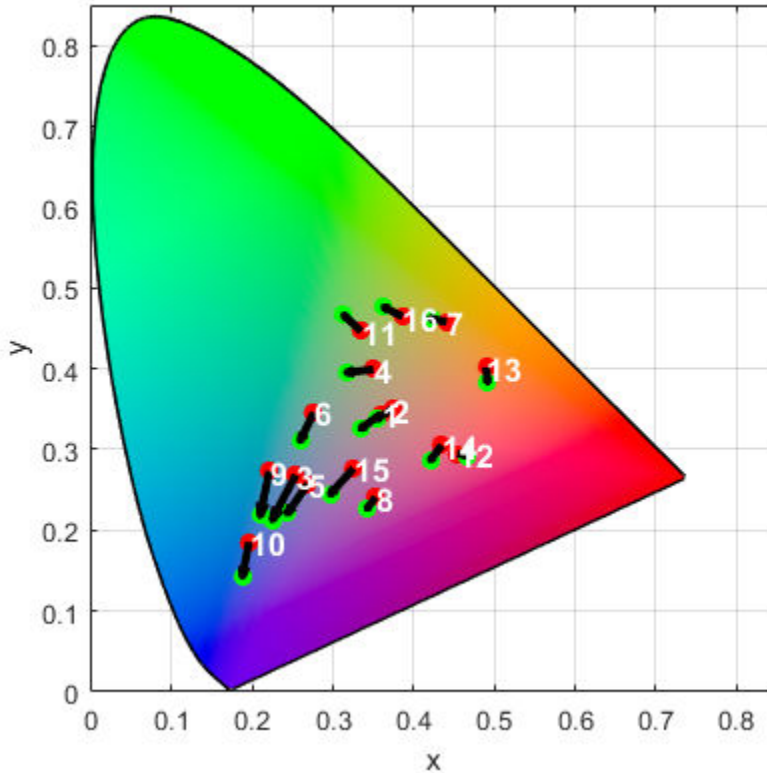
```
figure  
displayColorPatch(colorTable)
```



For an alternative representation of the color accuracy measurements, plot the measured and reference colors in the CIE 1976 L\*a\*b\* color space on a chromaticity diagram. Red circles indicate the reference color. Green circles indicate the measured color of each color patch. The chromaticity diagram does not portray the brightness of color.



```
figure  
plotChromaticity(colorTable)
```



ROIs with a shorter distance between the reference and measurement points have smaller differences in chromaticity, which can contribute to a smaller value of  $\Delta E$ . However, brightness also contributes to the value of  $\Delta E$ . For example, even though the reference and measurement points for ROI 13 are near each other on the chromaticity diagram, they have a large  $\Delta E$  because of their large difference in brightness.

## Input Arguments

### **chart** — eSFR chart

esfrChart object

eSFR chart, specified as an `esfrChart` object.

## Output Arguments

### **colorTable** — Color values

16-by-8 table

Color values in each color patch, returned as a 16-by-8 table. The 16 rows correspond to the 16 color patches on the eSFR chart.

The eight columns represent these variables:

Variable	Description
ROI	Index of the sampled ROI. The value of ROI is an integer in the range [1, 16]. The indices match the ROI numbers displayed by <code>displayChart</code> .
Measured_R	Mean value of red channel pixels in an ROI. <code>Measured_R</code> is a scalar of the same data type as <code>chart.Image</code> , which can be of type <code>single</code> , <code>double</code> , <code>uint8</code> , or <code>uint16</code> .
Measured_G	Mean value of green channel pixels in an ROI. <code>Measured_G</code> is a scalar of the same data type as <code>chart.Image</code> .
Measured_B	Mean value of blue channel pixels in an ROI. <code>Measured_B</code> is a scalar of the same data type as <code>chart.Image</code> .
Reference_L	Reference L* value corresponding to the ROI. <code>Reference_L</code> is a scalar of type <code>double</code> .
Reference_a	Reference a* value corresponding to the ROI. <code>Reference_a</code> is a scalar of type <code>double</code> .
Reference_b	Reference b* value corresponding to the ROI. <code>Reference_b</code> is a scalar of type <code>double</code> .

Variable	Description
Delta_E	Euclidean color distance between the measured and reference color values, as outlined in CIE 1976.

### **colorCorrectionMatrix — Color correction coefficients**

4-by-3 matrix

Color correction coefficients, returned as a 4-by-3 matrix. `colorCorrectionMatrix` represents an affine transformation, used to color-correct images that are captured under similar lighting conditions as the test chart image.

Data Types: `double`

## **See Also**

`displayColorPatch` | `measureIlluminant` | `plotChromaticity`

## **Topics**

“Correct Colors Using Color Correction Matrix”

“Anatomy of an eSFR Chart”

“Evaluate Quality Metrics on eSFR Test Chart”

**Introduced in R2017b**

## measureIlluminant

Measure scene illuminant using Imatest® eSFR chart

### Syntax

```
illuminant = measureIlluminant(chart)
```

### Description

`illuminant = measureIlluminant(chart)` measures the scene illuminant using the gray regions of interest (ROIs) of an Imatest® eSFR chart.

### Examples

#### Measure Illuminant of eSFR Chart

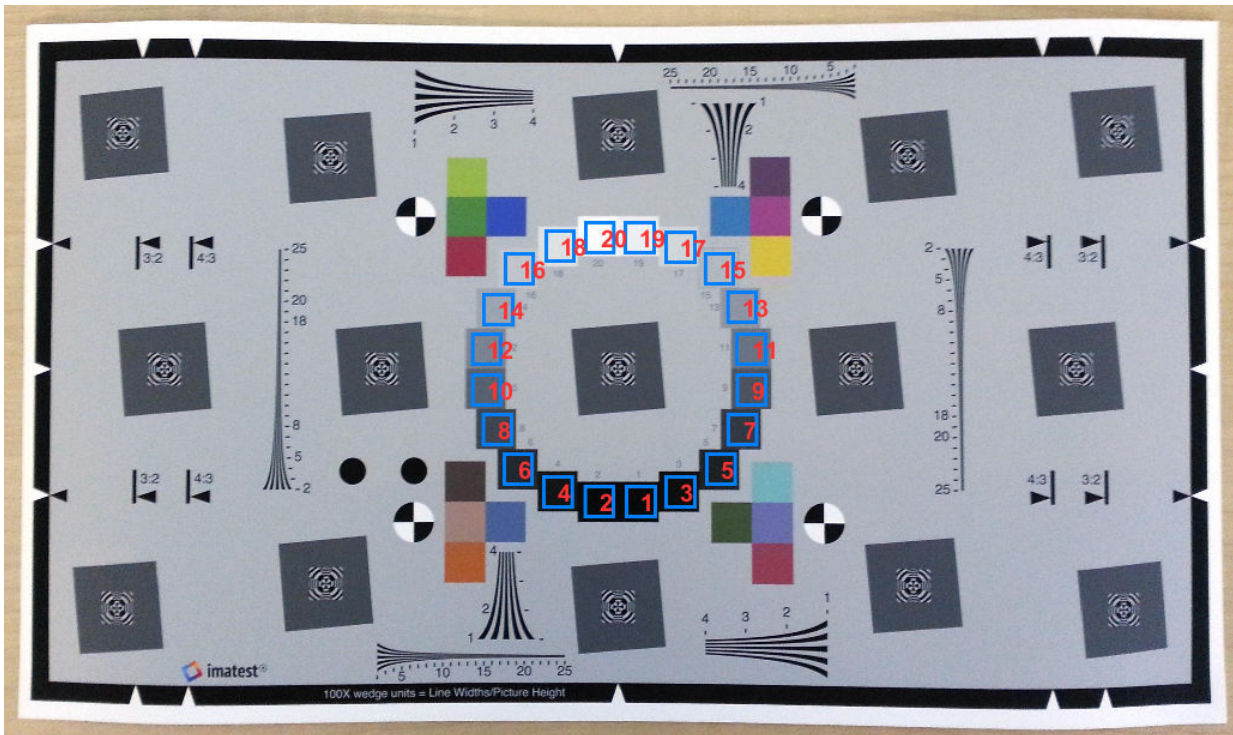
This example shows how to measure the illuminant of an eSFR chart using the gray patch ROIs. The example then white balances the image of the eSFR chart.

Read an image of an eSFR chart into the workspace.

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

Create an `esfrChart` object. Display the chart, highlighting the 20 gray patches.

```
chart = esfrChart(I);  
displayChart(chart, 'displayEdgeROIs', false, ...  
    'displayColorROIs', false, 'displayRegistrationPoints', false)
```



Estimate the illuminant using the gray patch ROIs. The illuminant has a stronger blue component than the red and green. This result is consistent with the image of the test chart, which has a blue tint.

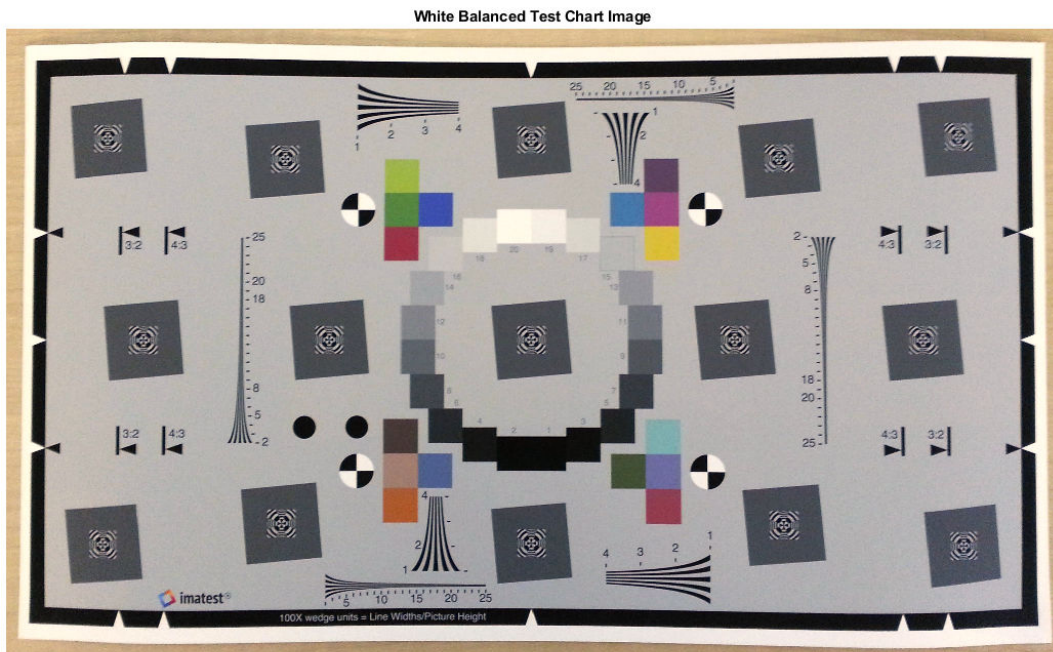
```
illum = measureIlluminant(chart)
```

```
illum = 1×3
```

```
110.9147 116.0008 123.2339
```

White balance the chart image and display the result. The white balanced image has less of a blue tint, especially in the middle gray patches and over the background of the image.

```
J = chromadapt(I,illum);
imshow(J)
title('White Balanced Test Chart Image')
```



You can use the estimated illuminant to white balance other images acquired under similar lighting conditions.

## Input Arguments

**chart** — eSFR chart

esfrChart object

eSFR chart, specified as an esfrChart object.

## Output Arguments

**illuminant** — Scene illuminant

3-element row vector

Scene illuminant, returned as a 3-element row vector.

Data Types: double

## Tips

- To white-balance an image, use the `chromadapt` function.

## See Also

`chromadapt` | `measureColor`

## Topics

[“Anatomy of an eSFR Chart”](#)

[“Evaluate Quality Metrics on eSFR Test Chart”](#)

**Introduced in R2017b**

## measureNoise

Measure noise using Imatest® eSFR chart

### Syntax

```
noiseTable = measureNoise(chart)
```

### Description

`noiseTable = measureNoise(chart)` measures the noise levels using the gray regions of interest (ROIs) of an Imatest® eSFR chart.

### Examples

#### Measure Noise of eSFR Chart

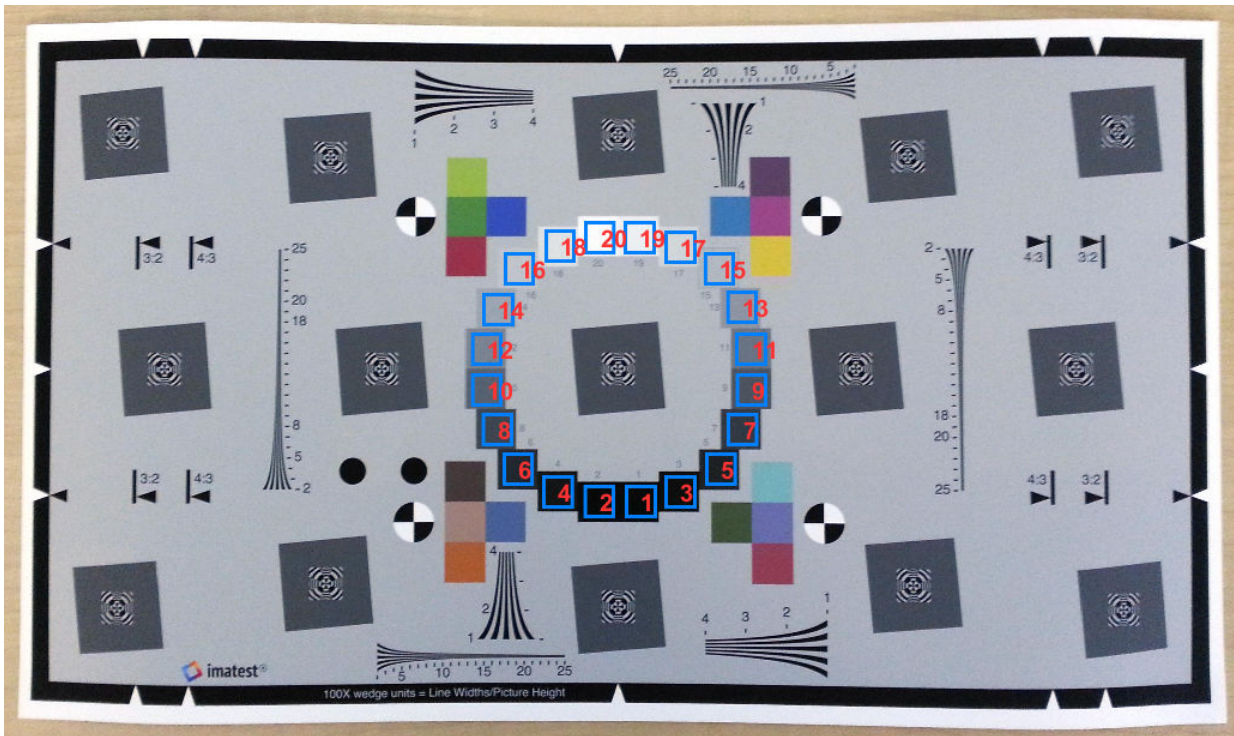
Read an image of an eSFR chart into the workspace.

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

Create an `esfrChart` object, then display the chart with ROI annotations. The 20 gray patch ROIs are labeled with red numbers.

```
chart = esfrChart(I);  
displayChart(chart, 'displayColorROIs', false, ...  
    'displayEdgeROIs', false, 'displayRegistrationPoints', false)
```





Measure the noise in all gray patch ROIs.

```
noiseTable = measureNoise(chart)
```

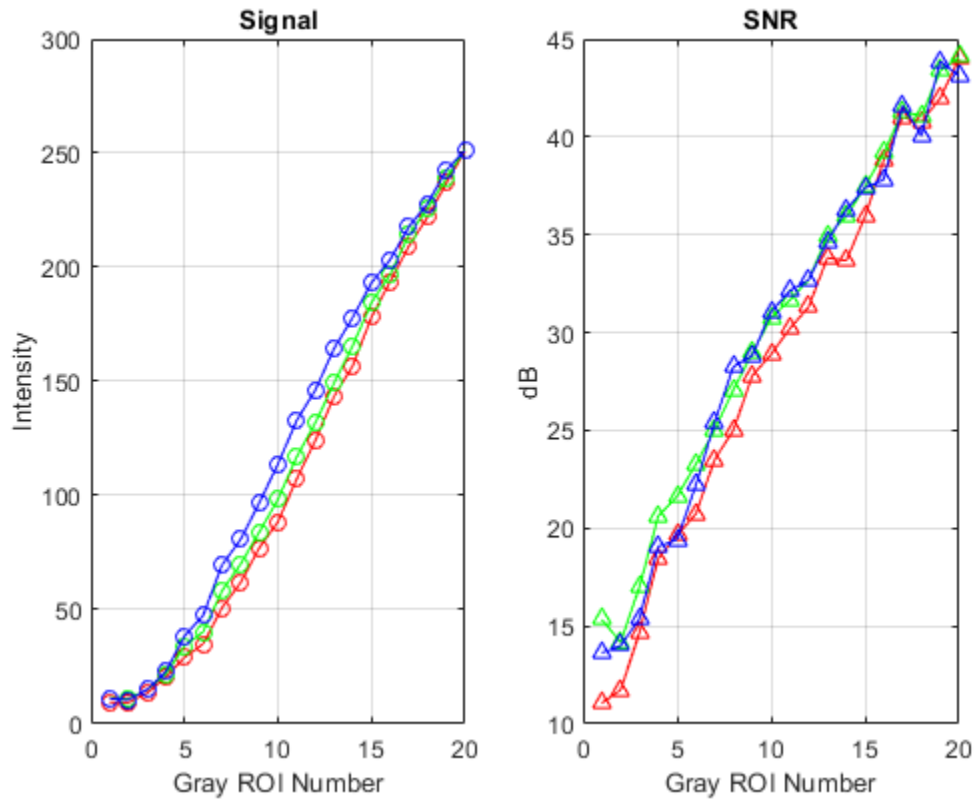
```
noiseTable=20x22 table
```

ROI	MeanIntensity_R	MeanIntensity_G	MeanIntensity_B	RMSNoise_R	RMSNoise_G	RMSNoise_B
1	9.4147	11.349	11.099	2.6335	2.6335	2.6335
2	9.2873	10.896	10.503	2.405	2.405	2.405
3	13.488	14.95	15.017	2.4966	2.4966	2.4966
4	20.411	21.689	22.946	2.4395	2.4395	2.4395
5	29.189	34.144	38.442	3.0436	3.0436	3.0436
6	35.009	40.337	47.544	3.2201	3.2201	3.2201
7	50.768	58.206	69.539	3.3931	3.3931	3.3931
8	61.871	69.98	80.779	3.4734	3.4734	3.4734
9	77.115	83.999	96.869	3.1467	3.1467	3.1467
10	88.552	98.426	113.87	3.1846	3.1846	3.1846

11	107.25	116.97	132.94	3.3128	3.0
12	124.23	131.96	146.27	3.3817	3.0
13	143.52	149.3	164.52	2.922	2.6
14	156.87	165.76	178.05	3.2507	2.6
15	178.25	184.59	193.3	2.8498	2.7
16	193.81	196.97	203.42	2.2181	2.7
:					

Display a graph of the mean signal and the signal to noise ratio (SNR) of the three color channels over the 20 gray patch ROIs.

```
figure
subplot(1,2,1)
plot(noiseTable.ROI,noiseTable.MeanIntensity_R,'r-o', ...
     noiseTable.ROI,noiseTable.MeanIntensity_G,'g-o', ...
     noiseTable.ROI,noiseTable.MeanIntensity_B,'b-o')
title('Signal')
ylabel('Intensity')
xlabel('Gray ROI Number')
grid on
subplot(1,2,2)
plot(noiseTable.ROI,noiseTable.SNR_R,'r-^', ...
     noiseTable.ROI,noiseTable.SNR_G,'g-^', ...
     noiseTable.ROI,noiseTable.SNR_B,'b-^')
title('SNR')
ylabel('dB')
xlabel('Gray ROI Number')
grid on
```



## Input Arguments

### chart — eSFR chart

esfrChart object

eSFR chart, specified as an esfrChart object.

## Output Arguments

### **noiseTable** — Noise values

20-by-22 table

Noise values of each gray patch, returned as a 20-by-22 table. The 20 rows correspond to the 20 gray patches on the eSFR chart. The 22 columns represent the variables shown in the table. Each variable is a scalar of type `double`.

<b>Variable</b>	<b>Description</b>
<code>ROI</code>	Index of the sampled ROI. The value of <code>ROI</code> is an integer in the range [1, 20]. The indices match the ROI numbers displayed by <code>displayChart</code> .
<code>MeanIntensity_R</code>	Mean value of red channel pixels in the ROI.
<code>MeanIntensity_G</code>	Mean value of green channel pixels in the ROI.
<code>MeanIntensity_B</code>	Mean value of blue channel pixels in the ROI.
<code>RMSNoise_R</code>	Root mean square (RMS) noise of red channel pixels in the ROI.
<code>RMSNoise_G</code>	RMS noise of green channel pixels in the ROI.
<code>RMSNoise_B</code>	RMS noise of blue channel pixels in the ROI.
<code>PercentNoise_R</code>	RMS noise of red pixels, expressed as a percentage of the maximum of the original chart image data type.
<code>PercentNoise_G</code>	RMS noise of green pixels, expressed as a percentage of the maximum of the original chart image data type.
<code>PercentNoise_B</code>	RMS noise of blue pixels, expressed as a percentage of the maximum of the original chart image data type.
<code>SignalToNoiseRatio_R</code>	Ratio of signal ( <code>MeanIntensity_R</code> ) to noise ( <code>RMSNoise_R</code> ) in the red channel.
<code>SignalToNoiseRatio_G</code>	Ratio of signal ( <code>MeanIntensity_G</code> ) to noise ( <code>RMSNoise_G</code> ) in the green channel.
<code>SignalToNoiseRatio_B</code>	Ratio of signal ( <code>MeanIntensity_B</code> ) to noise ( <code>RMSNoise_B</code> ) in the blue channel.
<code>SNR_R</code>	Signal-to-noise ratio (SNR) of the red channel, in dB.  $SNR_R = 20 * \log(\text{MeanIntensity}_R / \text{RMSNoise}_R)$ .

Variable	Description
SNR_G	SNR of the green channel, in dB. $SNR_G = 20 \cdot \log(\text{MeanIntensity}_G / \text{RMSNoise}_G)$ .
SNR_B	SNR of the blue channel, in dB. $SNR_B = 20 \cdot \log(\text{MeanIntensity}_B / \text{RMSNoise}_B)$ .
PSNR_R	Peak SNR of the red channel, in dB.
PSNR_G	Peak SNR of the green channel, in dB.
PSNR_B	Peak SNR of the blue channel, in dB.
RMSNoise_Y	RMS noise of luminance (Y) channel pixels in the ROI.
RMSNoise_Cb	RMS noise of chrominance (Cb) channel pixels in the ROI.
RMSNoise_Cr	RMS noise of chrominance (Cr) channel pixels in the ROI.

## Tips

- To linearize data for noise measurements, first undo the gamma correction of an sRGB test chart image by using the `rgb2lin` function. Then, create an `esfrChart` object from the linear image, and input the `esfrChart` object to the `measureNoise` function.

## See Also

`displayChart` | `measureIlluminant`

## Topics

“Anatomy of an eSFR Chart”

“Evaluate Quality Metrics on eSFR Test Chart”

**Introduced in R2017b**

## measureSharpness

Measure spatial frequency response using Imatest® eSFR chart

### Syntax

```
sharpnessTable = measureSharpness(chart)
sharpnessTable = measureSharpness(chart,Name,Value)
[sharpnessTable,aggregateSharpnessTable] = measureSharpness( ___ )
```

### Description

`sharpnessTable = measureSharpness(chart)` measures the spatial frequency response (SFR) at all slanted edge regions of interest (ROIs) of an Imatest® eSFR chart. `sharpnessTable` includes the frequency for each ROI at which the response drops to 50% of the initial and peak values.

`sharpnessTable = measureSharpness(chart,Name,Value)` measures the SFR at all specified slanted edge ROIs, specifying additional parameters.

`[sharpnessTable,aggregateSharpnessTable] = measureSharpness( ___ )` also returns the average SFR of vertical and horizontal ROIs, using the input arguments of either of the previous syntaxes.

### Examples

#### Measure Sharpness of Slanted Edges on eSFR Chart

Read an image of an eSFR chart into the workspace.

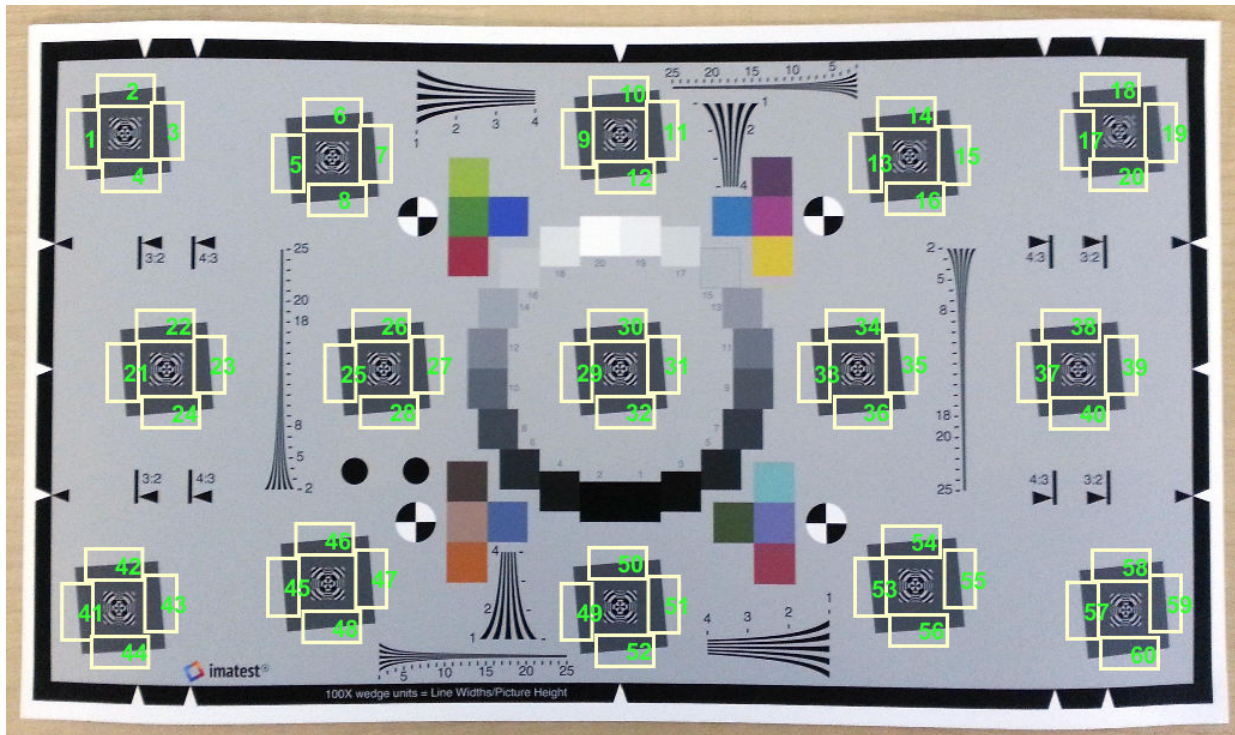
```
I = imread('eSFRTestImage.jpg');
```

Create an `esfrChart` object, then display the chart with ROI annotations. The 60 slanted edge ROIs are labeled with green numbers.

```

chart = esfrChart(I);
displayChart(chart, 'displayColorROIs', false, ...
    'displayGrayROIs', false, 'displayRegistrationPoints', false)

```



Measure the edge sharpness in ROIs 25-28, and return the measurements in sharpnessTable. Include measurements of the MTF70 and MTF30 by specifying the 'percentResponse' name-value pair argument.

```

sharpnessTable = measureSharpness(chart, 'ROIIndex', 25:28, 'PercentResponse', [70 30])

```

sharpnessTable=4x9 table

ROI	slopeAngle	confidenceFlag	SFR	comment
25	4.2391	true	[88x5 table]	[ ] 0.060224 0.05
26	5.0254	true	[88x5 table]	[ ] 0.18561 0.1
27	4.7131	true	[88x5 table]	[ ] 0.070042 0.06

28

4.8996

true

[88x5 table]

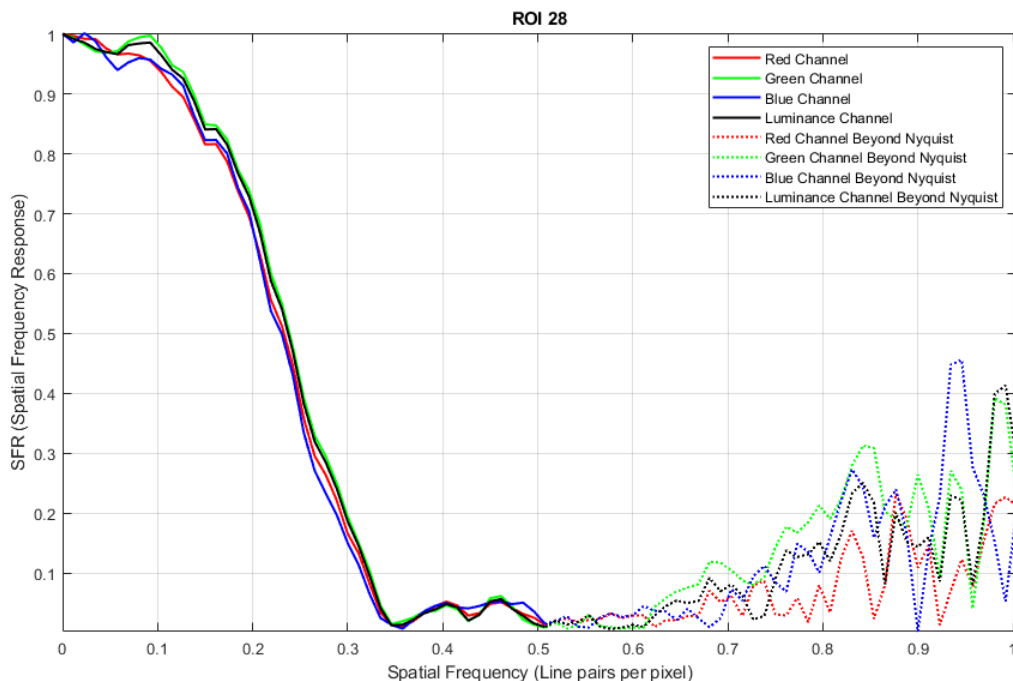
[]

0.19485

0.2

Select the fourth row in the sharpness table, which corresponds to ROI 28. Display the SFR plot of the ROI.

```
idx = 4;
plotSFR(sharpnessTable(idx,:))
```



Print the MTF70 and MTF30 measurements of the ROI. Compare the measurements against the plot.

The MTF70 measurement of the red and blue color channels are slightly smaller than 0.2, while the MTF70 measurement of the green and luminance channels are slightly larger than 0.2. These measurements agree with a visual inspection of the SFR plot, on which an SFR value of 0.7 occurs at spatial frequencies around 0.2 line pairs per pixel.

```
mtf70 = sharpnessTable.MTF70(idx,:)
```



```
mtf70 = 1×4
    0.1949    0.2042    0.1967    0.2017
```

The MTF30 measurement of the blue color channel is noticeably smaller than the MTF30 measurement of the other color channels. This measurement agrees with a visual inspection of the SFR plot, on which the SFR curve of the blue channel drops off more quickly than the other channels.

```
mtf30 = sharpnessTable.MTF30(idx,:)
mtf30 = 1×4
    0.2645    0.2753    0.2600    0.2719
```

## Input Arguments

### chart — eSFR chart

esfrChart object

eSFR chart, specified as an esfrChart object.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `measureSharpness(myChart, 'ROIIndex', 2)` measures the sharpness only of ROI 2.

### ROIIndex — ROI indices

1:60 (default) | scalar | vector

ROI indices to include in measurements, specified as the comma-separated pair consisting of 'ROIIndex' and a scalar or vector of integers in the range [1, 60]. The indices match the ROI numbers displayed by `displayChart`.

---

**Note** `measureSharpness` uses the intersection of ROIs specified by `'ROIIndex'` and `'ROIOrientation'`.

---

Example: 29:32

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

### **ROIorientation — ROI orientation**

`'both'` (default) | `'vertical'` | `'horizontal'`

ROI orientation, specified as the comma-separated pair consisting of `'ROIorientation'` and `'both'`, `'vertical'`, or `'horizontal'`. The `measureSharpness` function performs measurements only on ROIs with the specified orientation.

---

**Note** `measureSharpness` uses the intersection of ROIs specified by `'ROIIndex'` and `'ROIOrientation'`.

---

Example: `'vertical'`

Data Types: `char` | `string`

### **PercentResponse — Value of frequency response**

50 (default) | `scalar` | `vector`

Value of frequency response at which to report the corresponding spatial frequency, specified as the comma-separated pair consisting of `'PercentResponse'` and a scalar or vector of integers in the range [1, 100].

Each value of `PercentResponse` adds two columns to the `sharpnesTable` and `aggregateSharpnesTable` output arguments. The columns indicate the frequency at which the SFR drops to the specified percent of the initial and peak values. For example, when `PercentResponse` has the value 50, both output tables have the columns `MTF50` and `MTF50P`. These columns indicate the frequency at which the SFR drops to 50% of the initial value and peak value, respectively.

Example: 30

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

## Output Arguments

### sharpnessTable — SFR measurements of edge

*m*-by-*n* table

SFR measurements of edge, returned as an *m*-by-*n* table. *m* is the number of sampled ROIs. *n* changes values depending on PercentResponse. The first five columns are always present and represent these variables:

Variable	Description
ROI	Index of the sampled ROI. The value of ROI is an integer in the range [1, 60].
slopeAngle	Angle between the slanted edge and pure vertical or horizontal, depending on the ROI orientation. The angle is measured in degrees, and it is returned as a scalar of type double.
confidenceFlag	<p>Boolean flag that indicates whether the sharpness measurement is reliable. confidenceFlag is true when the measurement is reliable. confidenceFlag is false when the measurement is unreliable due to the following conditions:</p> <ul style="list-style-type: none"> <li>• slopeAngle is less than 3.5 degrees or more than 15 degrees.</li> <li>• The contrast within the ROI is less than 20%.</li> </ul> <p>The contrast of a slanted edge ROI is defined as <math>100 * (I_{High} - I_{Low}) / (I_{High} + I_{Low})</math>, where IHigh and ILow are the estimated average intensities of the high and low intensity regions across the edge. The contrast is computed for only the red channel.</p>
SFR	<p>Spatial frequency response of the edge in the ROI. SFR is an <i>f</i>-by-5 table. The five columns represent the frequency value and the red, green, blue, and luminance values corresponding to that frequency. <i>f</i> is the number of frequency samples of the MTF.</p> <p>Luminance (<i>Y</i>) is a linear combination of the red (<i>R</i>), green (<i>G</i>), and blue (<i>B</i>) channels according to:</p> $Y = 0.213R + 0.715G + 0.072B$

Variable	Description
comment	When confidenceFlag is false, then comment describes the reason the measurement is unreliable. When confidenceFlag is true, then comment is the empty vector, [].

Each value of PercentResponse adds two columns that indicate the frequency at which the SFR drops to the specified percent of the initial and peak value. The format of each entry in the column is a 1-by-4 vector. The four elements correspond to the red, green, blue, and luminance channels, respectively.

### **aggregateSharpnessTable — Average SFR measurements of vertical and horizontal edges**

table with one or two rows

Average SFR measurements of vertical and horizontal edges, returned as a table with one or two rows. aggregateSharpnessTable has one row when all sampled ROIs have the same orientation. It has two rows when the sampled ROIs have mixed orientation. aggregateSharpnessTable has three fewer columns than sharpnessTable.

The first two columns of aggregateSharpnessTable are always present and represent these variables:

Variable	Description
Orientation	Orientation of the averaged SFRs. The value of Orientation is either 'horizontal' or 'vertical'.
SFR	<p>Averaged spatial frequency response of all edges in included ROIs with the orientation specified by Orientation.</p> <p>SFR is an <i>s</i>-by-5 table. The five columns represent the frequency value, and the averaged red, green, blue, and luminance values corresponding to that frequency. <i>s</i> is the number of frequency samples of the MTF.</p> <p>Luminance (<i>Y</i>) is computed as a linear combination of the red (<i>R</i>), green (<i>G</i>), and blue (<i>B</i>) channels according to:</p> $Y = 0.213R + 0.715G + 0.072B$

Each value of PercentResponse adds two columns that indicate the frequency at which the SFR drops to the specified percent of the initial and peak value. The format of each entry in the column is a 1-by-4 vector. The four elements correspond to the red, green,

blue, and luminance channels, averaged among all sampled ROIs with the same orientation.

## Tips

- Slanted edges on a properly oriented chart are at an angle of 5 degrees from the horizontal or vertical. Sharpness measurements are not accurate when the edge orientation deviates significantly from 5 degrees.
- Sharpness is higher toward the center of the imaged region and decreases toward the periphery. Horizontal sharpness is usually higher than vertical sharpness.

## Algorithms

The SFR measurement algorithm is based on work by Peter Burns [1] [2]. First, `measureSharpness` determines the edge position with sub-pixel resolution for each *scan line*, or row or column of pixels perpendicular to the edge, in the ROI. For example, each row of pixels is a scan line for a near-vertical edge. Next, `measureSharpness` aligns and averages the scan lines to create an oversampled edge intensity profile. The function takes the derivative of the intensity profile and applies a windowing function. The returned SFR measurement is the absolute value of the Fourier transform of the windowed derivative.

## References

[1] Burns, Peter. "Slanted-Edge MTF for Digital Camera and Scanner Analysis." *Society for Imaging Science and Technology; Proceedings of the Image Processing, Image Quality, Image Capture Systems Conference*. Portland, Oregon, March 2000. pp 135-138.

[2] Burns, Peter. "sfrmat3: SFR evaluation for digital cameras and scanners." URL: [http://losburns.com/imaging/software/SFRedge/sfrmat3\\_post/index.html](http://losburns.com/imaging/software/SFRedge/sfrmat3_post/index.html).

## See Also

`displayChart` | `plotSFR`

## **Topics**

*“Anatomy of an eSFR Chart”*

*“Evaluate Quality Metrics on eSFR Test Chart”*

*“Fourier Transform”*

**Introduced in R2017b**

# medfilt2

2-D median filtering

## Syntax

```
J = medfilt2(I)
J = medfilt2(I,[m n])
J = medfilt2( ____,padopt)
```

## Description

`J = medfilt2(I)` performs median filtering of the image `I` in two dimensions. Each output pixel contains the median value in a 3-by-3 neighborhood around the corresponding pixel in the input image.

You optionally can compute the normalized cross-correlation using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`J = medfilt2(I,[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.

`J = medfilt2( ____,padopt)` controls how `medfilt2` pads the image boundaries.

This syntax is not supported on a GPU.

## 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'));
```

Add noise to the image, then perform median filtering and display the result.

```
J = imnoise(I,'salt & pepper',0.02);  
K = medfilt2(J);  
figure, montage({J,K})
```

## Input Arguments

### **I** — Input image

2-D grayscale image | 2-D binary image | `gpuArray`

Input image, specified as a 2-D grayscale or binary image.

To perform the median filtering using a GPU, specify `I` as a `gpuArray` that contains a 2-D grayscale or binary image.

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

**[m n] — Neighborhood size**

3-by-3 (default) | 2-element vector

Neighborhood size, specified as a 2-element vector of real positive integers.

If you perform the median filtering using a GPU, then the neighborhood must be square with odd-length sides between 3 and 15.

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

**padopt — Padding option**

'zeros' (default) | 'symmetric' | 'indexed'

Padding option, specified as one of the following values.

Value	Description
'zeros' (default)	Pad the image with 0s.
'symmetric'	Symmetrically extend the image at the boundaries.
'indexed'	If the class of I is <code>double</code> , then pad the image with 1s; otherwise, pad with 0s.

---

**Note** The `padopt` argument is not supported on a GPU.

---

Data Types: `char` | `string`

## Output Arguments

**J — Output image**

numeric matrix | `gpuArray`

Output image, returned as a numeric matrix of the same class as the input image I.

If the median filtering is performed using a GPU, then J is returned as a `gpuArray` containing a numeric matrix.

## 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 `I` is of an integer class, then all the output values are returned as integers. If the number of pixels in the neighborhood (i.e.,  $m*n$ ) is even, then 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
```

- If you specify `padopt` as 'zeros' or 'indexed', then the padding can skew the median near the image boundary. Pixels within one-half the width of the neighborhood ( $[m\ n]/2$ ) of the edges can appear distorted.

## 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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, the `padopt` argument must be a compile-time constant.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Padding options are not supported on the GPU.

For more information, see “Image Processing on a GPU”.

### See Also

`filter2` | `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
<code>'symmetric'</code>	Pad array with mirror reflections of itself
<code>'replicate'</code>	Pad array by repeating border elements
<code>'zeros'</code>	Pad array with 0s

Data Types: `char` | `string`

## 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

`medfilt2`**Introduced in R2016b**



# montage

Display multiple image frames as rectangular montage

## Syntax

```
montage(filenamees)
montage(I)
montage(imagelist)
montage(imds)
montage(X,map)
montage( ____,Name,Value)
img = montage( ____ )
```

## Description

`montage(filenamees)` displays a montage of the images with file names specified in `filenamees`. The images can be of different types and sizes. By default, `montage` arranges the images so that they roughly form a square.

`montage(I)` displays all frames of a multiframe image array `I`.

`montage(imagelist)` displays a montage of the images specified in the cell array `imagelist`.

`montage(imds)` displays a montage of the images specified in the image datastore `imds`.

`montage(X,map)` treats all grayscale images in `X` as indexed images and applies the specified color map `map` for all frames. If `X` is a cell array of file names, then `map` overrides any internal color map present in the image files.

`montage( ____,Name,Value)` uses name-value pair arguments to customize the display of the image montage.

`img = montage( ____ )` returns a handle to the single image object that contains all the frames displayed.

## Examples

### Create Montage from Images in Files

Create a montage from a series of images in files. Make the montage a 2-by-5 rectangle. Then, create a second montage, this time using the `DisplayRange` parameter to highlight structures in the image.

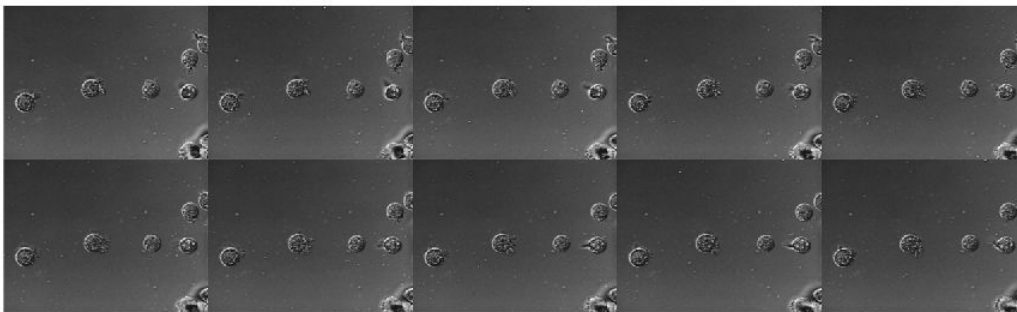
### Display the Images as a Rectangular Montage

Create a string array containing a series of file names.

```
fileFolder = fullfile(matlabroot, 'toolbox', 'images', 'imdata');  
dirOutput = dir(fullfile(fileFolder, 'AT3_lm4_*.tif'));  
fileNames = string({dirOutput.name});
```

Display the images as a montage. Specify the shape of the montage as a 2-by-5 rectangle.

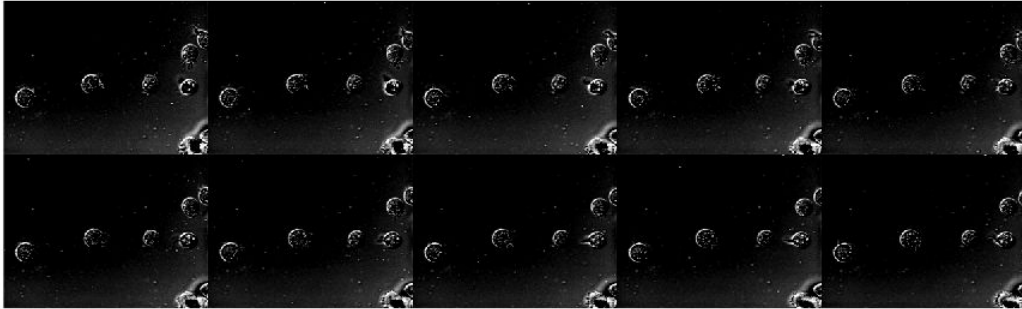
```
montage(fileNames, 'Size', [2 5]);
```



### Adjust the Contrast of the Images in the Montage

In another figure, create the same 2-by-5 montage. In addition, specify the display range to adjust the contrast of the images in the montage.

```
figure  
montage(fileNames, 'Size', [2 5], 'DisplayRange', [75 200]);
```



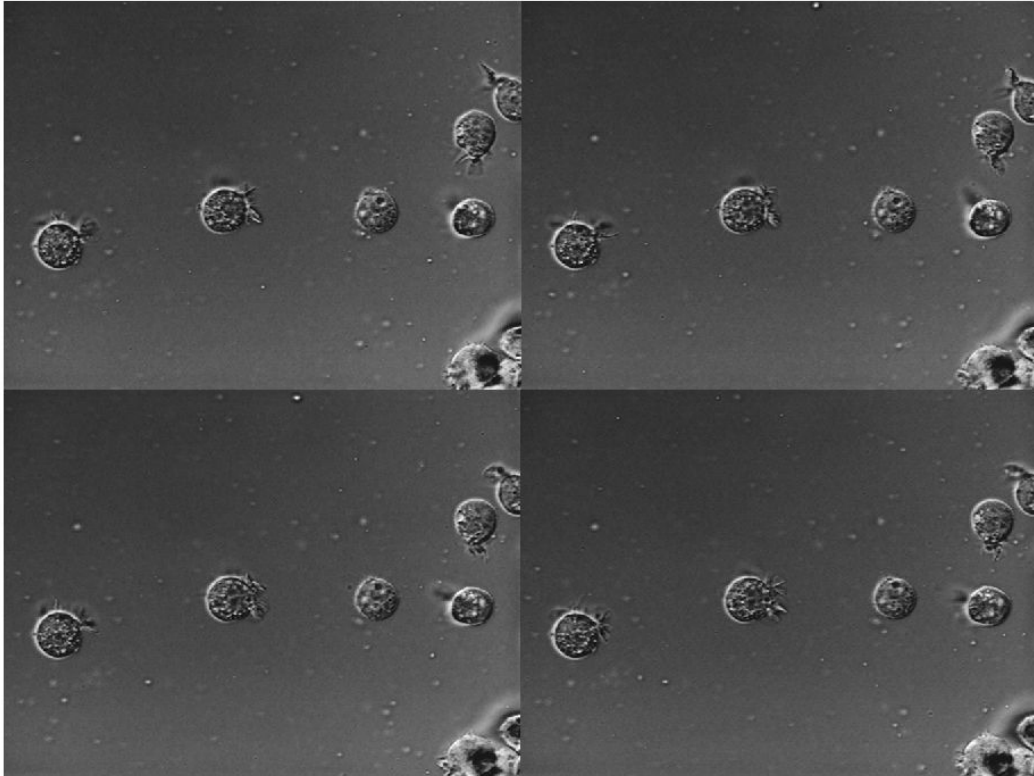
### Create Montage from Multiframe Image

Create an m-by-n-by-4 multiframe image from a series of grayscale images. The images must all be the same size.

```
img1 = imread('AT3_lm4_01.tif');  
img2 = imread('AT3_lm4_02.tif');  
img3 = imread('AT3_lm4_03.tif');  
img4 = imread('AT3_lm4_04.tif');  
multi = cat(3,img1,img2,img3,img4);
```

Display a montage of the images in the multiframe image.

```
montage(multi);
```



### Create Montage Containing Images of Different Types and Sizes

Read several images of different types and sizes into the workspace.

```
imRGB = imread('peppers.png');  
imGray = imread('coins.png');
```

Display a montage containing all of the images.

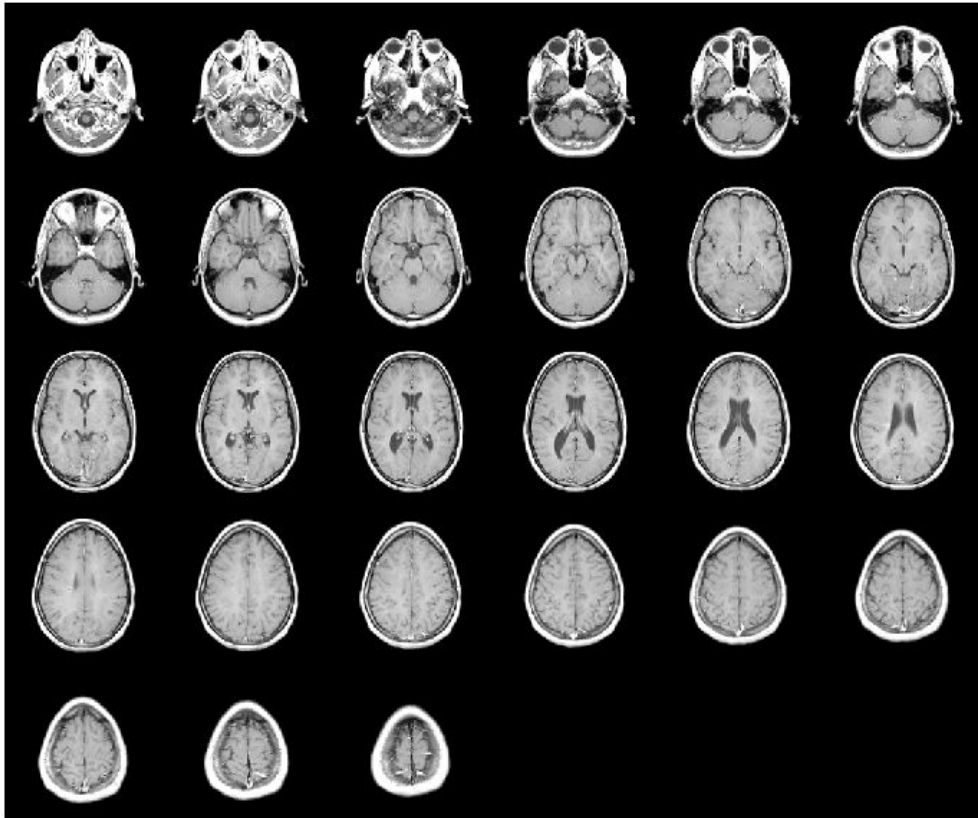
```
figure  
montage({imRGB, imGray, 'cameraman.tif'})
```



### Customize Number of Images in Montage

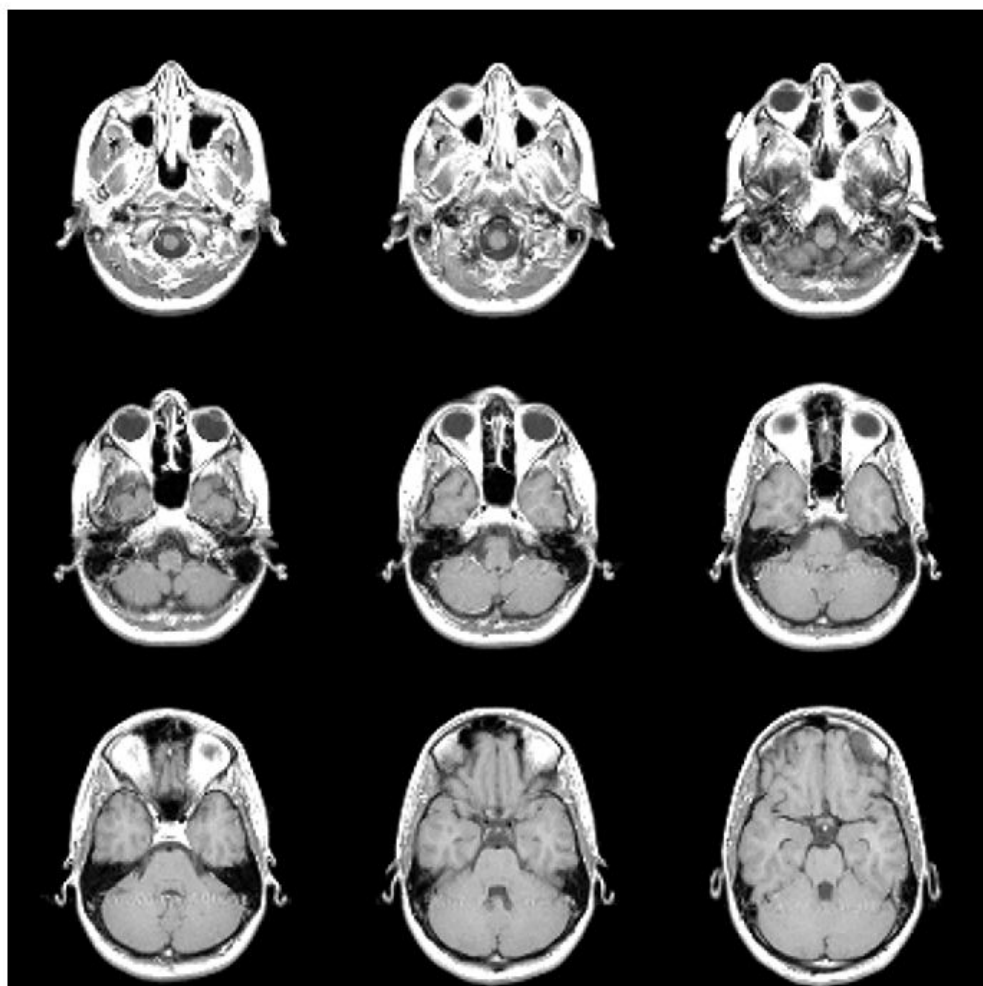
View all the images in a MRI data set using montage with default settings. There are 27 images in the set.

```
load mri
montage(D, map)
```



Create a new montage containing only the first 9 images.

```
figure  
montage(D, map, 'Indices', 1:9);
```



## Create Montage from ImageDatastore

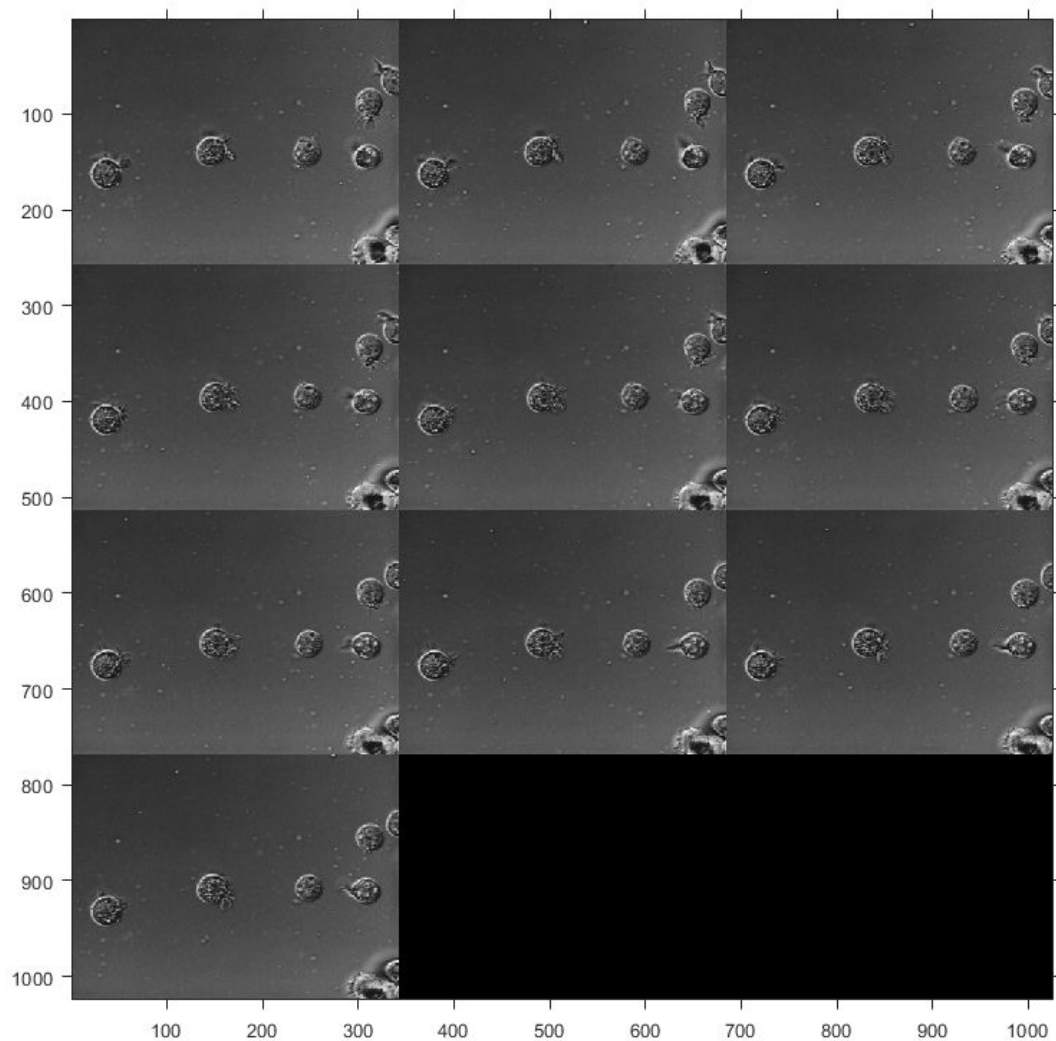
Create an ImageDatastore containing a series of ten images from the Image Processing Toolbox sample image folder.

```
fileFolder = fullfile(matlabroot, 'toolbox', 'images', 'imdata');  
imds = imageDatastore(fullfile(fileFolder, 'AT3*'));
```

Display the contents of the imageDatastore as a montage.

```
img = montage(imds)
```





```
img =  
  Image with properties:  
      CData: [1024x1026 uint8]
```

CDataMapping: 'scaled'

Show all properties

## Input Arguments

### **imds** — Image data store

ImageDatastore

Image data store, specified as an ImageDatastore.

### **filenames** — Names of files containing images

array of strings |  $n$ -by-1 or 1-by- $n$  cell array

Names of files containing image, specified as an array of strings or an  $n$ -by-1 or 1-by- $n$  cell array of string scalars or character vectors. If the files are not in the current folder or in a folder on the MATLAB path, then specify the full path name. See the `imread` command for more information.

Data Types: char | string | cell

### **imagelist** — Set of images

$n$ -by-1 or 1-by- $n$  cell array

Set of images, specified as an  $n$ -by-1 or 1-by- $n$  cell array of numeric matrices of size  $m$ -by- $n$  or  $m$ -by- $n$ -by-3.

Data Types: single | double | int16 | uint8 | uint16 | logical | cell

### **I** — Multiframe image array

numeric array

Multiframe image array, specified as one of the following:

- $m$ -by- $n$ -by- $k$  numeric array representing a sequence of  $k$  binary or grayscale images
- $m$ -by- $n$ -by-1-by- $k$  numeric array representing a sequence of  $k$  binary or grayscale images
- $m$ -by- $n$ -by-3-by- $k$  numeric array representing a sequence of  $k$  truecolor images

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

**X — Indexed image**

2-D numeric matrix

Indexed image, specified as a 2-D numeric matrix.

Data Types: `single` | `double` | `uint8` | `uint16` | `logical`**map — Color map***c*-by-3 numeric matrixColor map, specified as an *c*-by-3 numeric matrix, where *c* is the number of colors in the color map.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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `montage(D,map,'Indices',1:9)`**BackgroundColor — Background color**

'black' (default) | [R G B] | short name | long name

Background color, specified as a MATLAB `ColorSpec`. The `montage` function fills all blank spaces with this color, including the space specified by `BorderSize`. If you specify a background color, then the `montage` function renders the output as an RGB image.**BorderSize — Padding around each thumbnail image**

[0 0] (default) | nonnegative integer | 1-by-2 vector of nonnegative integers

Padding around each thumbnail image, specified as a nonnegative integer or a 1-by-2 vector of nonnegative integers. `montage` pads the image borders with the background color, `BackgroundColor`.Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`**DisplayRange — Display range**

1-by-2 vector

Display range of grayscale images in array `I`, specified as 1-by-2 vector of the form `[LOW HIGH]`. All pixel values less than or equal to `LOW` display as black. All pixel values greater than or equal to `HIGH` display as white. If you specify an empty matrix (`[]`), then `montage` uses the minimum and maximum pixel values of the images.

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

### **Indices — Frames to display**

array of positive integers

Frames to display in the montage, specified as an array of positive integers. The `montage` function interprets the values as indices into array `I` or into cell array `filenames` or `imagelist`.

By default, `montage` displays all frames or image files.

Example: `montage(I, 'Indices', 1:4)` create a montage of the first four frames in `I`

Example: `montage(I, 'Indices', 1:2:20)` displays every other frame.

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

### **Parent — Parent of the image object created by montage**

axes object

Parent of the image object created by `montage`, specified as an axes object. The `montage` function resizes the image to fit the extents available in the parent axes.

### **Size — Number of rows and columns of images**

2-element vector

Number of rows and columns of images, specified as a 2-element vector of the form `[NROWS NCOLS]`.

If you specify `NaN` or `Inf` for a particular dimension, then `montage` calculates the value of the dimension to display all images in the montage. For example, if `'Size'` is `[2 NaN]`, then the montage will have two rows and the minimum number of columns to display all images. When there is a mismatch between `'Size'` and number of images (frames) specified, `imtile` creates the tiled image based on `'Size'`.

Data Types: `single` | `double`

### **ThumbnailSize — Size of each thumbnail**

2-element vector of positive integers

Size of each thumbnail, in pixels, specified as a 2-element vector of positive integers. The aspect ratio of each image is preserved, and any blank space is filled with the background color, `BackgroundColor`.

If you specify an empty array (`[]`), `montage` uses the full size of the first image as the thumbnail size. If you specify either element as `NaN` or `Inf`, then `montage` calculates the corresponding value automatically to preserve the aspect ratio of the first image.

Data Types: `single` | `double`

## **Output Arguments**

### **img — Montage image**

handle to a single image object

Montage image, returned as a handle to a single `matlab.graphics.primitive.Image` object.

## **Tips**

- If you specify an indexed image, then `montage` converts it to RGB using the colormap present in the file.
- If there is a data type mismatch between images, then all images are rescaled to be `double` using the `im2double` function.
- When calculating the number of images to display horizontally and vertically, `montage` considers the aspect ratio of the images, so that the displayed montage is nearly square.

## **See Also**

`ImageDatastore` | `immovie` | `implay` | `imshow` | `imtile`

**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)
```

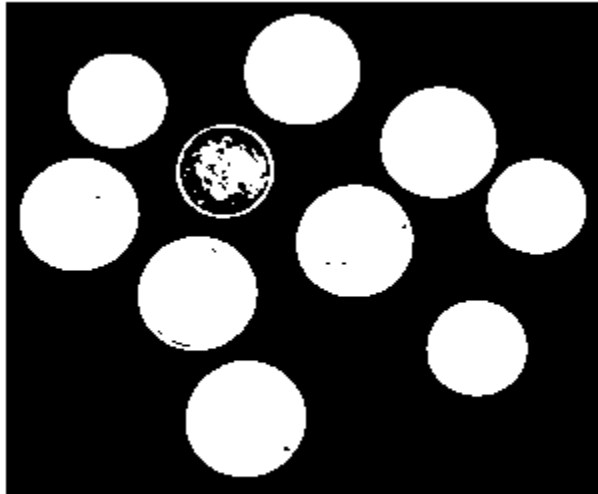


Calculate a single threshold value for the image.

```
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,[])
```



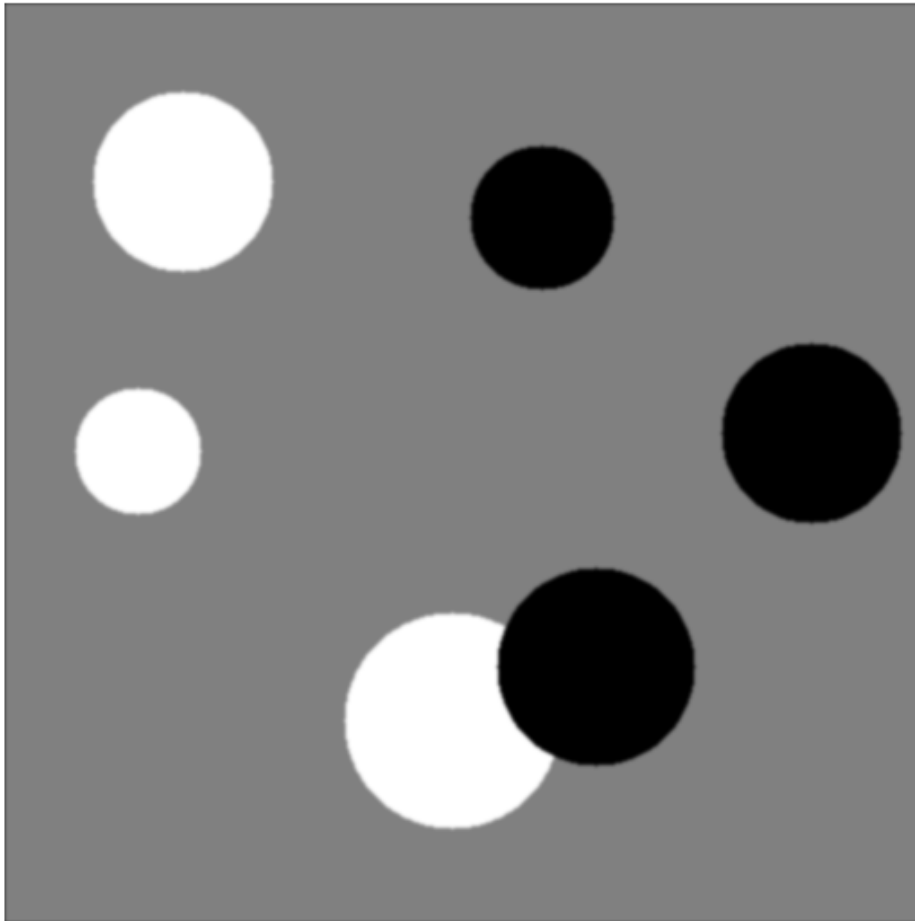
### Segment Image into Three Levels Using Two Thresholds

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);
```

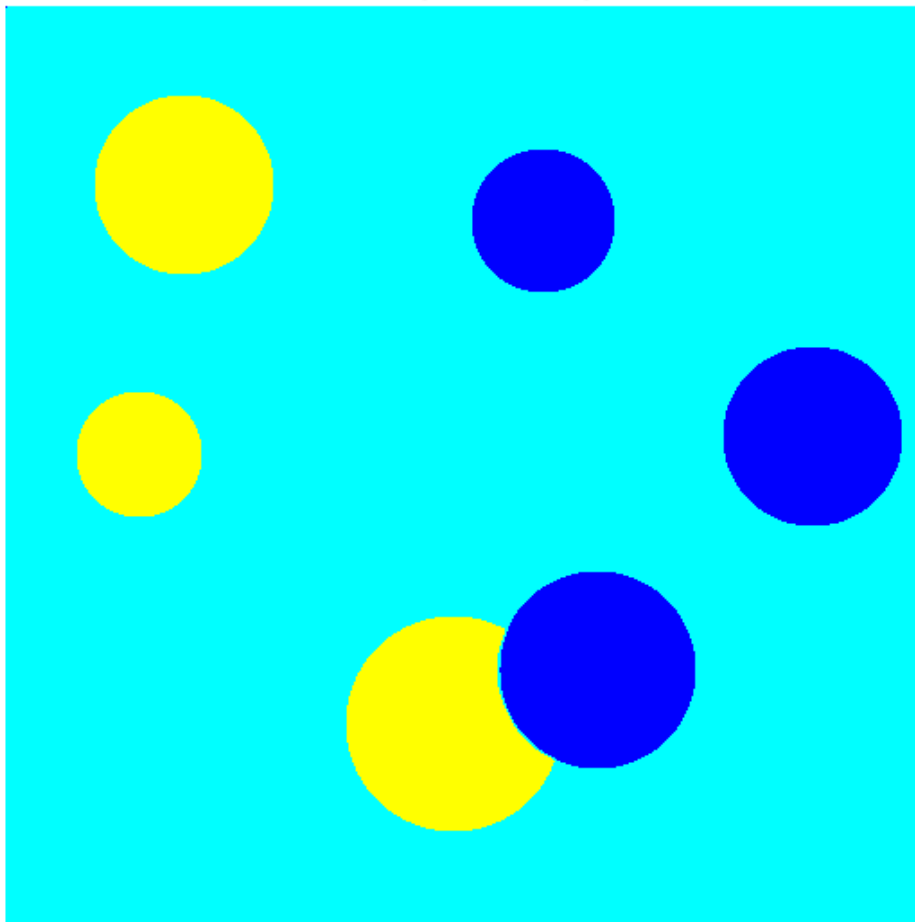
Segment the image into three levels using `imquantize`.

```
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);

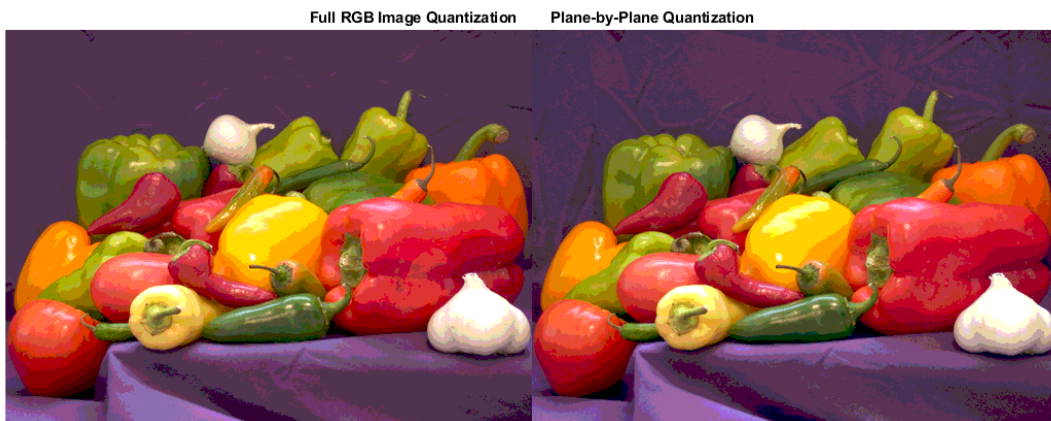
```

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 = 9x1
```

```
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 = 1x8
```

```
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 few
```

```
thresh = 1x9
```

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 “Understand Code Generation with Image Processing Toolbox”.

- The input argument `N` must be a compile-time constant.

### **See Also**

`graythresh` | `im2bw` | `imquantize` | `rgb2ind`

**Introduced in R2012b**

# niftiinfo

Read metadata from NIFTI file

## Syntax

```
info = niftiinfo(filename)
```

## Description

`info = niftiinfo(filename)` reads metadata from the file specified by `filename`. The file must use the Neuroimaging Informatics Technology Initiative (NIFTI) format. `niftiinfo` returns the metadata in the structure `info`.

## Examples

### View Metadata Fields from NIFTI Header File

Load metadata from the NIFTI file `brain.nii`.

```
info = niftiinfo('brain.nii');
```

Display the pixel dimensions of the file.

```
info.PixelDimensions
```

```
ans = 1×3
```

```
    1    1    1
```

Display the raw header content.

```
info.raw
```

```
ans = struct with fields:  
    sizeof_hdr: 348
```

```
    dim_info: ' '
      dim: [3 256 256 21 1 1 1 1]
    intent_p1: 0
    intent_p2: 0
    intent_p3: 0
  intent_code: 0
    datatype: 2
      bitpix: 8
  slice_start: 0
    pixdim: [1 1 1 1 0 0 0 0]
  vox_offset: 352
    scl_slope: 0
    scl_inter: 0
  slice_end: 0
  slice_code: 0
  xyzt_units: 0
    cal_max: 0
    cal_min: 0
  slice_duration: 0
    toffset: 0
    descrip: ''
    aux_file: ''
  qform_code: 0
  sform_code: 0
  quatern_b: 0
  quatern_c: 0
  quatern_d: 0
  qoffset_x: 0
  qoffset_y: 0
  qoffset_z: 0
    srow_x: [0 0 0 0]
    srow_y: [0 0 0 0]
    srow_z: [0 0 0 0]
  intent_name: ''
    magic: 'n+1 '
```

Display the intent code from the raw structure.

```
info.raw.intent_code
```

```
ans = 0
```

## Input Arguments

### **filename** — Name of NIfTI file

character vector | string scalar

Name of NIfTI file, specified as a string scalar or a character vector. If you do not specify a file extension, `niftiinfo` looks for a file with the extension `.nii` (or `.nii.gz` if the file is compressed). If `niftiinfo` cannot find a file with that name, it looks for a file with the file extension `.hdr` (or `.hdr.gz` if the file is compressed). In the dual-file NIfTI format, the `.hdr` file holds the metadata associated with the volume.

Data Types: `char` | `string`

## Output Arguments

### **info** — Metadata associated with a NIfTI volume

structure

Metadata associated with a NIfTI volume, returned as a structure.

`niftiinfo` returns the metadata from the header in simplified form---renaming, reordering, and packaging fields into easier to read MATLAB structures. For example, `niftiinfo` creates the `DisplayIntensityRange` field from the `cal_max` and `cal_min` fields of the file metadata. To view the metadata as it appears in the file, see the `raw` field of the structure returned.

## References

- [1] Cox, R. W., J. Ashburner, H. Breman, K. Fissell, C. Haselgrove, C.J. Holmes, J.L. Lancaster, D.E. Rex, S.M. Smith, J.B. Woodward, and S.C. Strother. 'A (sort of) new image data format standard: Nifti-1. " *Neuroimage*, Vol. 22(Suppl 1):e1440, 2004.

## See Also

`niftiread` | `niftiwrite`

**Introduced in R2017b**

## niftiread

Read NIfTI image

### Syntax

```
V = niftiread(filename)
V = niftiread(headerfile,imgfile)
V = niftiread(info)
```

### Description

`V = niftiread(filename)` reads the NIfTI image file specified by `filename`, in the current folder or on the path, and returns volumetric data in `V`.

NIfTI (Neuroimaging Informatics Technology Initiative) is an NIH-sponsored working group to promote the interoperability of functional neuroimaging software tools. NIfTI uses a single or dual file storage format. The dual format stores data in a pair of files: a header file (`.hdr`) containing the metadata and a data file (`.img`). The single file format stores the data in a single file (`.nii`), which contains header information followed by data.

`V = niftiread(headerfile,imgfile)` reads a NIfTI header file (`.hdr`) and image file (`.img`) pair.

`V = niftiread(info)` reads a NIfTI file described by the metadata structure `info`. To create an `info` structure, use the `niftiinfo` function

### Examples

#### Load Volume from NifTI File Using File Name

Load volumetric data from a NIfTI file. The file uses the NIfTI combined format--the image and metadata are in the same file. This type of NifTI file has the `.nii` file extension.

```
V = niftiread('brain.nii');
```

View the variable in the workspace.

```
whos V
```

Name	Size	Bytes	Class	Attributes
V	256x256x21	1376256	uint8	

## Load Volume from NIfTI File Using Its Header Structure

Read the metadata from a NIfTI file.

```
info = niftiinfo('brain.nii');
```

Read the volumetric image using the metadata structure returned by `niftiinfo`.

```
V = niftiread(info);
```

View the variable in the workspace.

```
whos V
```

Name	Size	Bytes	Class	Attributes
V	256x256x21	1376256	uint8	

## Input Arguments

### **filename** — Name of NIfTI file

string scalar | character vector

Name of the NIfTI file, specified as a string scalar or character vector. If you do not specify a file extension, `niftiread` looks for a file with the `.nii` extension. If `niftiread` cannot find a file with that extension, it looks for a gzipped version of the file, with extension `.nii.gz`. If `niftiread` cannot find a file with that extension, it looks for a file with the `.hdr`, `.hdr.gz`, `.img`, or `.img.gz` file extension. If `niftiread` cannot find a file that matches any of these options, it returns an error.

Data Types: char | string

## **headerfile** — Name of file containing metadata

string scalar | character vector

Name of the file containing metadata, specified as a string scalar or a character vector. The NIfTI header file (.hdr) holds the metadata associated with a NIfTI volume. If you do not specify a corresponding `imgfile`, then `niftiread` looks in the same folder for a file with the same name and extension `.img`.

Data Types: char | string

## **imgfile** — Name of file containing volume

string scalar | character vector

Name of the file containing volume, specified as a string scalar or a character vector. The NIfTI image file (.img) holds the volume data. If you do not specify a corresponding header file, `niftiread` looks in the same folder for a file with the same name and extension `.hdr`.

Data Types: char | string

## **info** — NIfTI file metadata

structure

NIfTI file metadata, specified as a structure returned by `niftiinfo`.

Data Types: struct

## Output Arguments

### **V** — Volumetric data

numeric array

Volumetric data, returned as a numeric array.

## References

- [1] Cox, R. W., J. Ashburner, H. Breman, K. Fissell, C. Haselgrove, C.J. Holmes, J.L. Lancaster, D.E. Rex, S.M. Smith, J.B. Woodward, and S.C. Strother. 'A (sort of) new image data format standard: Nifti-1. " *Neuroimage*, Vol. 22(Suppl 1):e1440, 2004.



## **See Also**

niftiinfo | niftiwrite

**Introduced in R2017b**

## niftiwrite

Write volume to file using NIFTI format

### Syntax

```
niftiwrite(V, filename)
niftiwrite(V, filename, info)
niftiwrite(V, filename, info, Name, Value)
```

### Description

`niftiwrite(V, filename)` writes the volumetric image data `V` to a file by using the Neuroimaging Informatics Technology Initiative (NIFTI) format. By default, `niftiwrite` creates a combined NIFTI file that contains both metadata and volumetric data. `niftiwrite` names the file `filename`, adding the `.nii` file extension. `niftiwrite` populates the metadata using appropriate default values and volume properties, such as size and data type.

`niftiwrite(V, filename, info)` writes the volumetric data `V` to a file, including the file metadata from `info`. If the metadata does not match the image contents and size, `niftiwrite` returns an error.

`niftiwrite(V, filename, info, Name, Value)` writes the volumetric data to a file, using options specified in `Name, Value` pairs.

### Examples

#### Write Median-Filtered Volume to NIFTI File

Load a NIFTI image by using its `.nii` file name.

```
V = niftiread('brain.nii');
```

Filter the image in 3-D by using a 3-by-3 median filter.

```
V = medfilt3(V);
```

Write the filtered image to a `.nii` file, using default header values.

```
niftiwrite(V,'outbrain.nii');
```

## Write Data to NIfTI File and Modify Header Structure

Read the metadata from a NIfTI file by using its `.nii` file name.

```
info = niftiinfo('brain.nii');
```

Read volumetric data from the file by using the file metadata.

```
V = niftiread(info);
```

Edit the Description metadata field of the file.

```
info.Description = 'Modified using MATLAB R2017b';
```

Write the volumetric data with the modified metadata to a new `.nii` file.

```
niftiwrite(V,'outbrain.nii',info);
```

## Input Arguments

### **filename** — Name of NIfTI file

character vector | string scalar

Name of NIfTI file, specified as a string scalar or character vector. By default, `niftiwrite` creates a combined format file that contains both metadata and image data and has the file extension `.nii`. If you specify the `'Compressed'` name-value pair, `niftiwrite` adds the file extension `.nii.gz`. If you set the `'Combined'` name-value pair to `false`, then `niftiwrite` creates two files with the same name and different file extensions. One file contains the metadata associated with the volume and has the file extension `.hdr`. The other file contains image data and has the file extension `.img`.

Data Types: `char` | `string`

## **V — Volumetric data**

numeric array

Volumetric data, specified as a numeric array.

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

## **info — File metadata**

structure

File metadata, specified as a structure returned by the `niftiinfo` function.

Data Types: `struct`

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `niftiwrite(V, 'outbrain.nii', 'Compressed', true)`

## **Combined — Type of NifTI file to create**

`true` (default) | `false`

Type of NifTI file to create, specified as `true` or `false`. If `true` (the default), `niftiwrite` creates a single file with the file extension `.nii`. If `false`, `niftiwrite` creates a pair of files with the same name but with different file extensions: `.hdr` for the file containing metadata, and `.img` for the file containing the volumetric data.

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

## **Compressed — Compress image data**

`false` (default) | `true`

Compress image data, specified as `true` or `false`. If `'Compressed'` is `true`, then `niftiwrite` generates compressed files, using `gzip`, with the file name extension `.gz`.

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

**Endian — Endianness of the data**`'little' (default) | 'big'`

Endianness of the data, specified as `'little'`, to indicate little-endian format (default) or `'big'`, to indicate big-endian format.

Data Types: `char` | `string`

**References**

- [1] Cox, R. W., J. Ashburner, H. Breman, K. Fissell, C. Haselgrove, C.J. Holmes, J.L. Lancaster, D.E. Rex, S.M. Smith, J.B. Woodward, and S.C. Strother. 'A (*sort of*) new image data format standard: Nifti-1. " *Neuroimage*, Vol. 22(Suppl 1):e1440, 2004.

**See Also**`niftiinfo` | `niftiread`**Introduced in R2017b**

## niqe

Naturalness Image Quality Evaluator (NIQE) no-reference image quality score

### Syntax

```
score = niqe(A)
score = niqe(A,model)
```

### Description

`score = niqe(A)` calculates the no-reference image quality score for image A using the Naturalness Image Quality Evaluator (NIQE). `niqe` compares A to a default model computed from images of natural scenes. A smaller score indicates better perceptual quality.

`score = niqe(A,model)` calculates the image quality score using a custom model.

### Examples

#### Calculate NIQE Score Using Default Feature Model

Compute the NIQE score for a natural image and its distorted versions using the default model.

Read an image into the workspace. Create copies of the image with noise and blurring distortions.

```
I = imread('lighthouse.png');
Inoise = imnoise(I,'salt & pepper',0.02);
Iblur = imgaussfilt(I,2);
```

Display the images.

```
figure
montage({I,Inoise,Iblur},'Size',[1 3])
title('Original Image | Noisy Image | Blurry Image')
```



Calculate the NIQE score for each image using the default model. Display the score.

```
niqeI = niqe(I);
fprintf('NIQE score for original image is %0.4f.\n',niqeI)

NIQE score for original image is 2.5455.

niqeInoise = niqe(Inoise);
fprintf('NIQE score for noisy image is %0.4f.\n',niqeInoise)

NIQE score for noisy image is 10.8770.

niqeIblur = niqe(Iblur);
fprintf('NIQE score for blurry image is %0.4f.\n',niqeIblur)

NIQE score for blurry image is 5.2661.
```

The original undistorted image has the best perceptual quality and therefore the lowest NIQE score.

## Calculate NIQE Score Using Custom Feature Model

Train a custom NIQE model and calculate a NIQE score for a natural image using the trained model.

Train a custom model using natural images stored in an image datastore.

```
setDir = fullfile(toolboxdir('images'),'imdata');  
imds = imageDatastore(setDir,'FileExtensions',{' .jpg'});  
model = fitniqe(imds);
```

```
Extracting features from 37 images.
```

```
..  
Completed 4 of 37 images. Time: Calculating...  
...  
Completed 13 of 37 images. Time: 00:20 of 00:49  
...  
Completed 24 of 37 images. Time: 00:32 of 00:47  
...  
Done.
```

Read an image of a natural scene. Display the image.

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





Calculate the NIQE score for the image using the custom model. Display the score.

```
niqeI = niqe(I,model);  
fprintf('NIQE score for the image is %0.4f.\n',niqeI)
```

```
NIQE score for the image is 1.8601.
```

## Input Arguments

### A — Input image

2-D grayscale image | 2-D RGB image

Input image, specified as a 2-D grayscale or RGB image.

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

## **model** — Custom model

`niqeModel` object

Custom model of image features, specified as a `niqeModel` object. `model` is derived from natural scene statistics.

## Output Arguments

### **score** — No-reference image quality score

nonnegative scalar

No-reference image quality score, returned as a nonnegative scalar. Lower values of score reflect better perceptual quality of image A with respect to the input `model`.

Data Types: `double`

## Algorithms

NIQE measures the distance between the NSS-based features calculated from image A to the features obtained from an image database used to train the model. The features are modeled as multidimensional Gaussian distributions.

## References

- [1] Mittal, A., R. Soundararajan, and A. C. Bovik. "Making a Completely Blind Image Quality Analyzer." *IEEE Signal Processing Letters*. Vol. 22, Number 3, March 2013, pp. 209-212.

## See Also

### Functions

`brisque` | `fitbrisque` | `fitniqe` | `piqe`

### Objects

`niqeModel`

## **Topics**

“Image Quality Metrics”

**Introduced in R2017b**

## **niqueModel**

Naturalness Image Quality Evaluator (NIQE) model

### **Description**

A `niqueModel` object encapsulates a model used to calculate the Naturalness Image Quality Evaluator (NIQE) perceptual quality score of an image.

### **Creation**

You can create a `niqueModel` object using the following methods:

- `fitnique` — Train a NIQE model with parameters derived from your image datastore. Use this function if you do not have a pretrained model.
- The `niqueModel` function described here. Use this function if you have a pretrained NIQE model, or if the default model is sufficient for your application.

### **Syntax**

```
m = niqueModel  
m = niqueModel(mean,covariance,blockSize,sharpnessThreshold)
```

### **Description**

`m = niqueModel` creates a NIQE model object with default property values that are derived from the pristine image database noted in [1].

`m = niqueModel(mean,covariance,blockSize,sharpnessThreshold)` creates a custom NIQE model and sets the Mean, Covariance, BlockSize, and SharpnessThreshold properties. You must provide all four arguments to create a custom model.

## Properties

### **Mean — Mean of natural scene statistics (NSS) based image feature vectors**

36-element numeric row vector

Mean of natural scene statistics (NSS) based image feature vectors, specified as a 36-element numeric row vector.

Example: `rand(1,36)`

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

### **Covariance — Covariance matrix of NSS-based image feature vectors**

36-by-36 numeric matrix

Covariance matrix of NSS-based image feature vectors, specified as a 36-by-36 numeric matrix.

Example: `rand(36,36)`

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

### **BlockSize — Block size used to partition an image**

[96 96] (default) | 2-element row vector of positive even integers

Block size used to partition an image into nonoverlapping blocks, specified as a 2-element row vector of positive even integers. The two elements specify the number of rows and columns in each partition, respectively.

Example: `[10 10]`

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

### **SharpnessThreshold — Sharpness threshold used to calculate feature vectors**

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

Sharpness threshold used to calculate feature vectors, specified as a real scalar in the range [0, 1]. The threshold determines which blocks are selected to calculate the feature vectors.

Example: `0.25`

Data Types: `single` | `double`

## Examples

### Create NIQE Model Object with Default Properties

```
model = niqeModel

model =
  niqeModel with properties:

          Mean: [1x36 double]
    Covariance: [36x36 double]
      BlockSize: [96 96]
  SharpnessThreshold: 0
```

### Create NIQE Model Object with Custom Properties

Create a `niqeModel` object using precomputed Mean, Covariance, BlockSize, and SharpnessThreshold properties. Random initializations are shown for illustrative purposes only.

```
model = niqeModel(rand(1,36),rand(36,36),[10 10],0.25);
```

You can use the custom model to calculate the NIQE score for an image.

```
I = imread('lighthouse.png');
score = niqe(I,model)
```

```
score = 3.6866
```

## References

- [1] Mittal, A., R. Soundararajan, and A. C. Bovik. "Making a Completely Blind Image Quality Analyzer." *IEEE Signal Processing Letters*. Vol. 22, Number 3, March 2013, pp. 209-212.

## See Also

### Functions

fitnriqe | nriqe

### Objects

brisqueModel

### Topics

“Image Quality Metrics”

“Train and Use a No-Reference Quality Assessment Model”

**Introduced in R2017b**

## **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 string scalar or 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 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

isnitf | nitfinfo

**Introduced in R2007b**

## nlfilter

General sliding-neighborhood operations

### Syntax

```
B = nlfilter(A,[m n],fun)
B = nlfilter(A,'indexed', ___)
```

### Description

`B = nlfilter(A,[m n],fun)` applies the function `fun` to each `m`-by-`n` sliding block of the grayscale image `A`.

`B = nlfilter(A,'indexed', ___)` processes `A` as an indexed image, padding with 0s if the class of `A` is `uint8`, `uint16`, or `logical`, and padding with 1s otherwise.

---

**Note** `nlfilter` can take a long time to process large images. In some cases, the `colfilt` function can perform the same operation much faster.

---

### Examples

#### Apply Median Filter to Image

This example shows how to apply a median filter to an image using `nlfilter`. This example produces the same result as calling `medfilt2` with a 3-by-3 neighborhood.

Read an image into the workspace.

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

Convert the image to `double`.

```
A = im2double(A);
```

Create the function you want to apply to the image--a median filter.

```
fun = @(x) median(x(:));
```

Apply the filter to the image.

```
B = nfilter(A,[3 3],fun);
```

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

```
montage({A,B})  
title('Original Image (Left) and Median Filtered Image (Right)')
```

Original Image (Left) and Median Filtered Image (Right)



## Input Arguments

**A** — Image to be filtered

numeric array

Image to be filtered, specified as a numeric array of any class supported by `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`.

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

### **[m n] — Block size**

2-element vector of positive integers

Block size, specified as a 2-element vector of positive integers. `m` is the number of rows and `n` is the number of columns in the block.

Example: `B = nlfilter(A,[3 3],fun);`

Data Types: `single` | `double` | `logical`

### **fun — Function handle**

handle

Function handle specified as a handle. The function must accept an `m`-by-`n` matrix as input and return a scalar result.

`c = fun(x)`

`c` is the output value for the center pixel in the `m`-by-`n` block `x`. `nlfilter` calls `fun` for each pixel in `A`. `nlfilter` zero-pads the `m`-by-`n` block at the edges, if necessary.

Data Types: `function_handle`

## **Output Arguments**

### **B — Filtered image**

numeric array

Filtered image, returned as numeric array. The class of `B` depends on the class of the output from `fun`.

## **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)
```

### Description

`C = normxcorr2(template,A)` computes the normalized cross-correlation of the matrices `template` and `A`. The resulting matrix `C` contains the correlation coefficients.

You optionally can compute the normalized cross-correlation using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

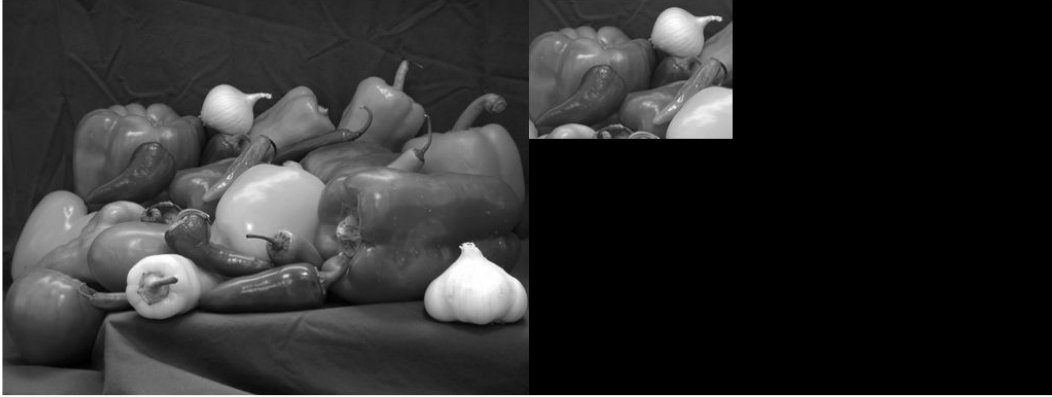
### Examples

#### Use Cross-Correlation to Find Template in Image

Read two images into the workspace, and convert them to grayscale for use with `normxcorr2`. Display the images side-by-side.

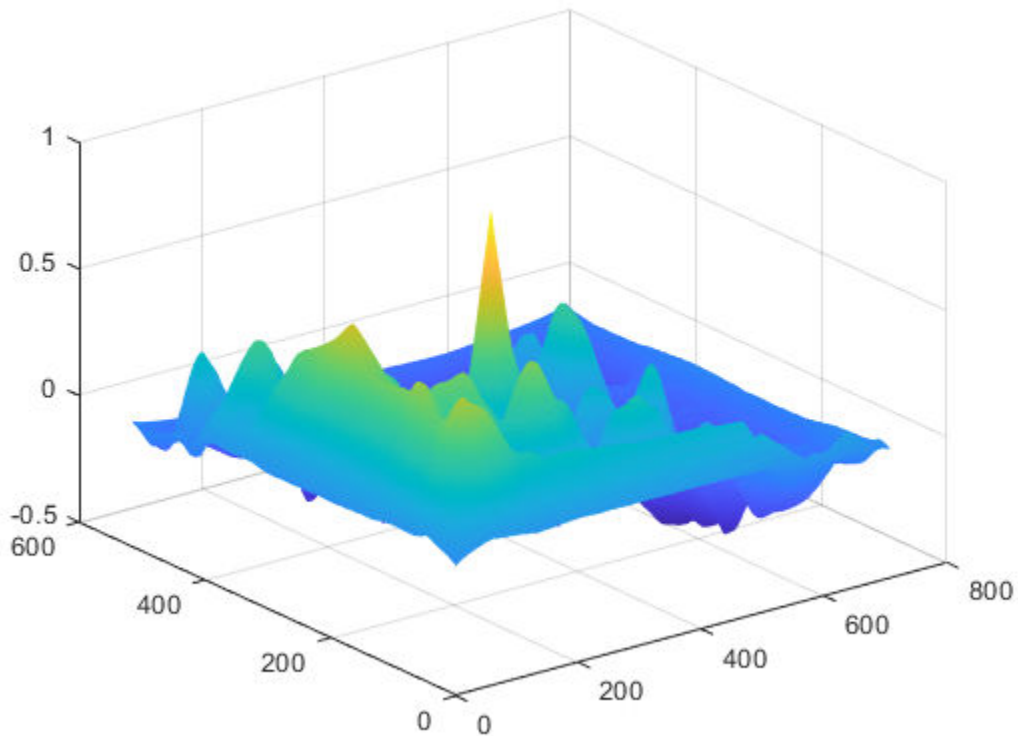
```
onion    = rgb2gray(imread('onion.png'));  
peppers = rgb2gray(imread('peppers.png'));  
imshowpair(peppers,onion,'montage')
```





Perform cross-correlation, and display the result as a surface.

```
c = normxcorr2(onion,peppers);  
figure, surf(c), shading flat
```



Find the peak in cross-correlation.

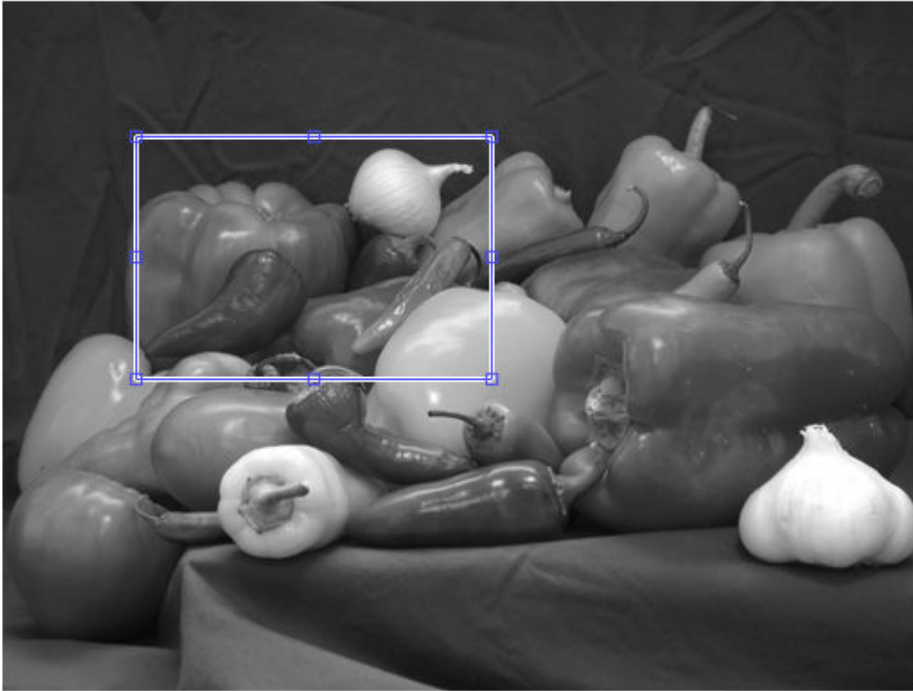
```
[ypeak, xpeak] = find(c==max(c(:)));
```

Account for the padding that `normxcorr2` adds.

```
yoffSet = ypeak-size(onion,1);  
xoffSet = xpeak-size(onion,2);
```

Display the matched area.

```
figure  
imshow(peppers);  
imrect(gca, [xoffSet+1, yoffSet+1, size(onion,2), size(onion,1)]);
```



### Use Cross-Correlation to Find Template in Image on a GPU

Read two images into `gpuArrays`.

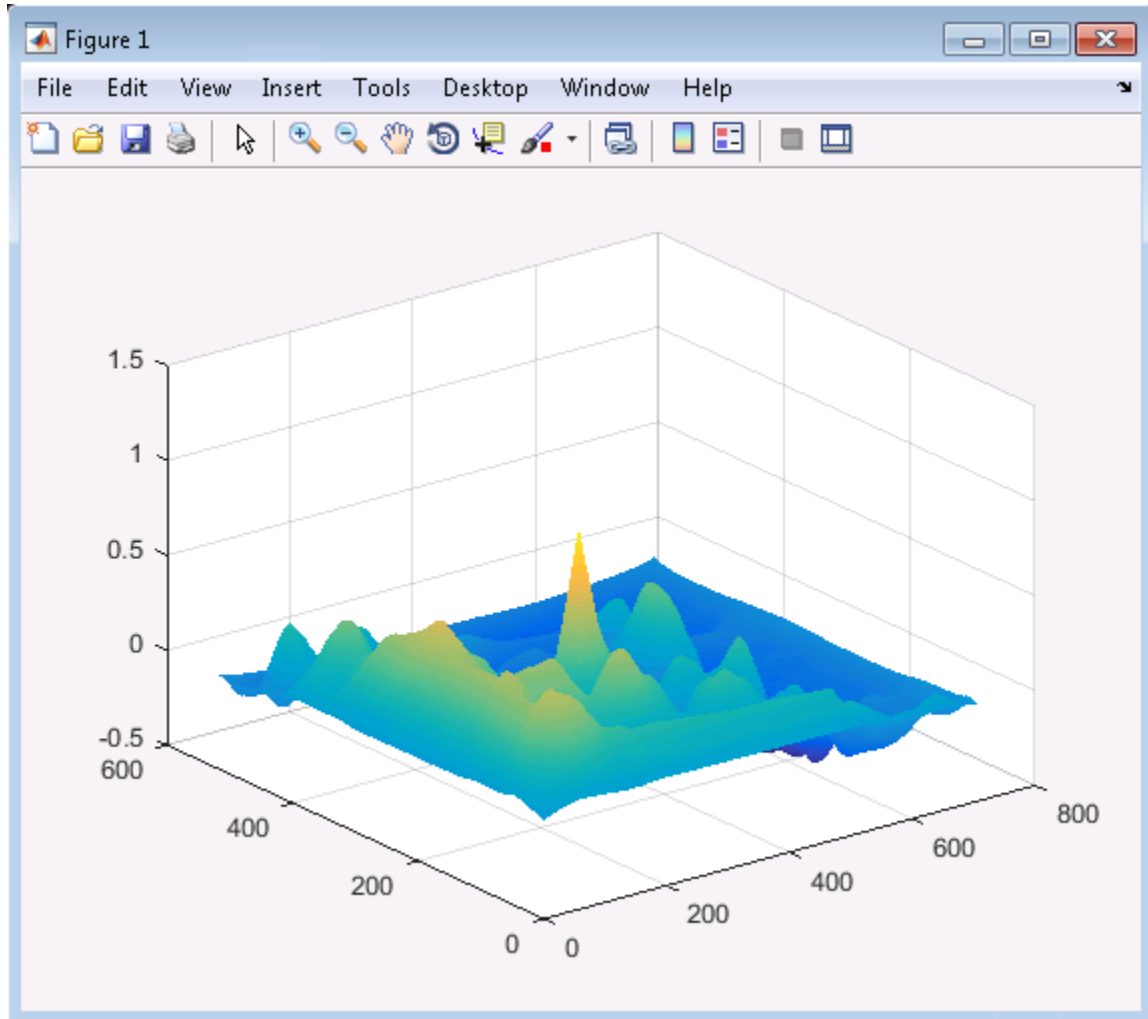
```
onion = gpuArray(imread('onion.png'));  
peppers = gpuArray(imread('peppers.png'));
```

Convert the color images to 2-D grayscale. The `rgb2gray` function accepts `gpuArrays`.

```
onion = rgb2gray(onion);  
peppers = rgb2gray(peppers);
```

Perform cross-correlation, and display the result as a surface.

```
c = normxcorr2(onion,peppers);  
figure, surf(c), shading flat
```



Find the peak in cross-correlation.

```
[ypeak, xpeak] = find(c==max(c(:)));
```

Account for the padding that normxcorr2 adds.

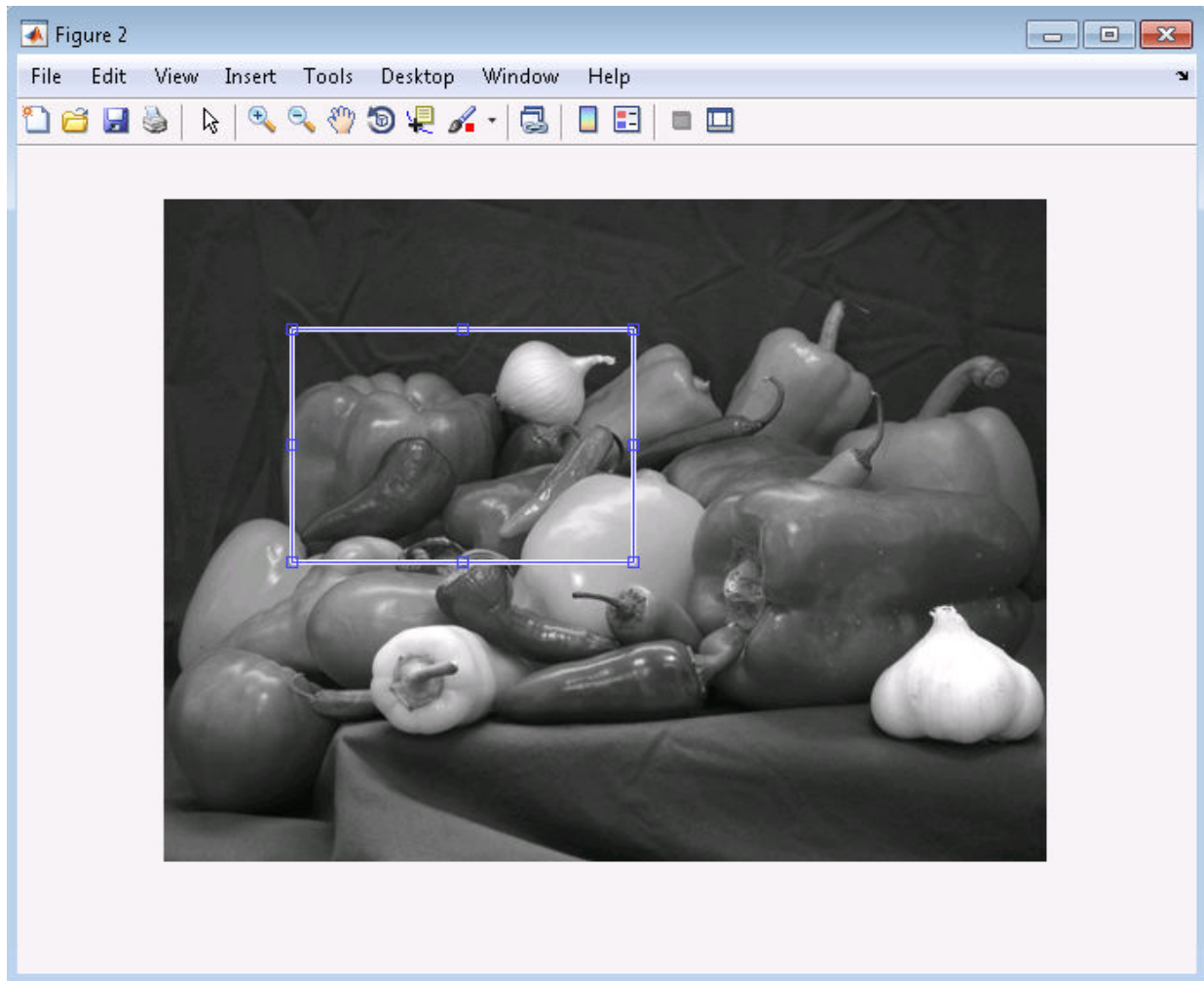
```
yoffSet = ypeak-size(onion,1);  
xoffSet = xpeak-size(onion,2);
```

Move the data back to the CPU for display.

```
yoffSet = gather(ypeak-size(onion,1));  
xoffSet = gather(xpeak-size(onion,2));
```

Display the matched area.

```
figure  
imshow(peppers);  
imrect(gca, [xoffSet+1, yoffSet+1, size(onion,2), size(onion,1)]);
```



## Input Arguments

**template** — Input template  
numeric matrix | gpuArray

Input template, specified as a numeric matrix. The values of `template` cannot all be the same.

To perform the computation using a GPU, specify `template` as a `gpuArray` that contains a numeric matrix.

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

### **A — Input image**

numeric matrix | `gpuArray`

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`.

To perform the computation using a GPU, specify `A` as a `gpuArray` that contains a numeric matrix.

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

## **Output Arguments**

### **C — Correlation coefficients**

numeric matrix | `gpuArray`

Correlation coefficients, returned as a numeric matrix with values in the range `[-1, 1]`.

If the correlation coefficients are computed using a GPU, then `C` is returned as a `gpuArray` containing a numeric matrix with values in the range `[-1, 1]`.

Data Types: `double`

## **Algorithms**

`normxcorr2` uses the following general procedure [1], [2]:

- 1 Calculate cross-correlation in the spatial or the frequency domain, depending on size of images.
- 2 Calculate local sums by precomputing running sums [1].
- 3 Use local sums to normalize the cross-correlation to get correlation coefficients.

The implementation closely follows the formula from [1]:

$$\gamma(u,v) = \frac{\sum_{x,y} [f(x,y) - \bar{f}_{u,v}] [t(x-u, y-v) - \bar{t}]}{\left\{ \sum_{x,y} [f(x,y) - \bar{f}_{u,v}]^2 \sum_{x,y} [t(x-u, y-v) - \bar{t}]^2 \right\}^{0.5}}$$

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.

# Extended Capabilities

## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see "Image Processing on a GPU".



## See Also

corr2 | 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');
```

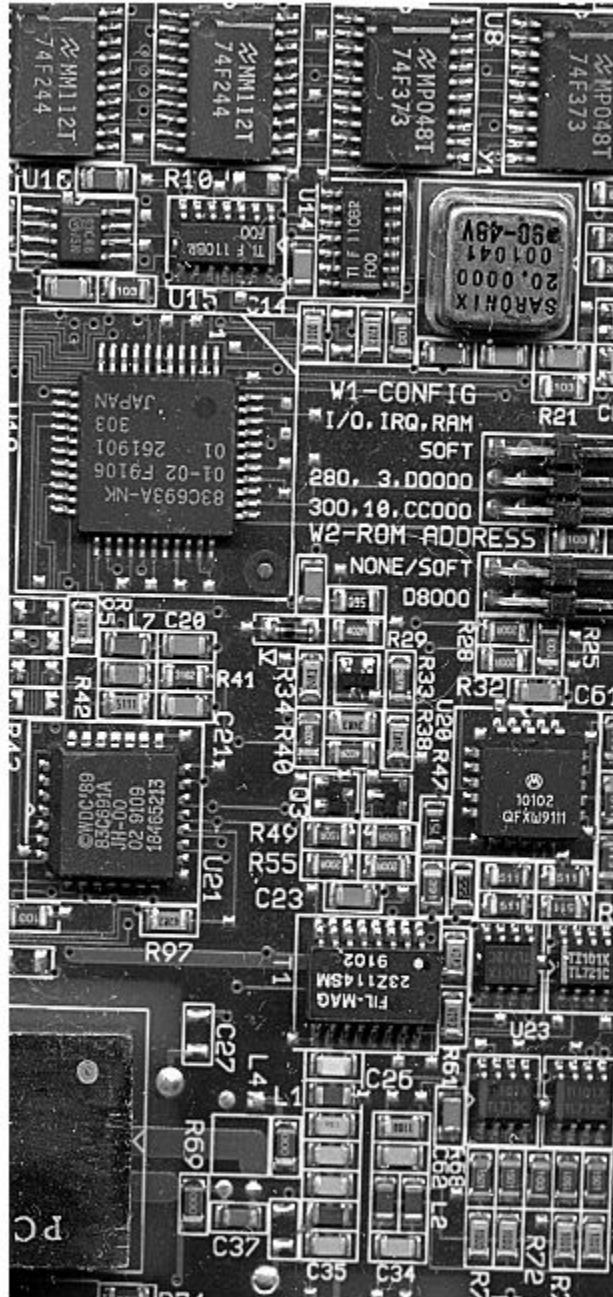
Convert the image to YIQ color space.

```
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



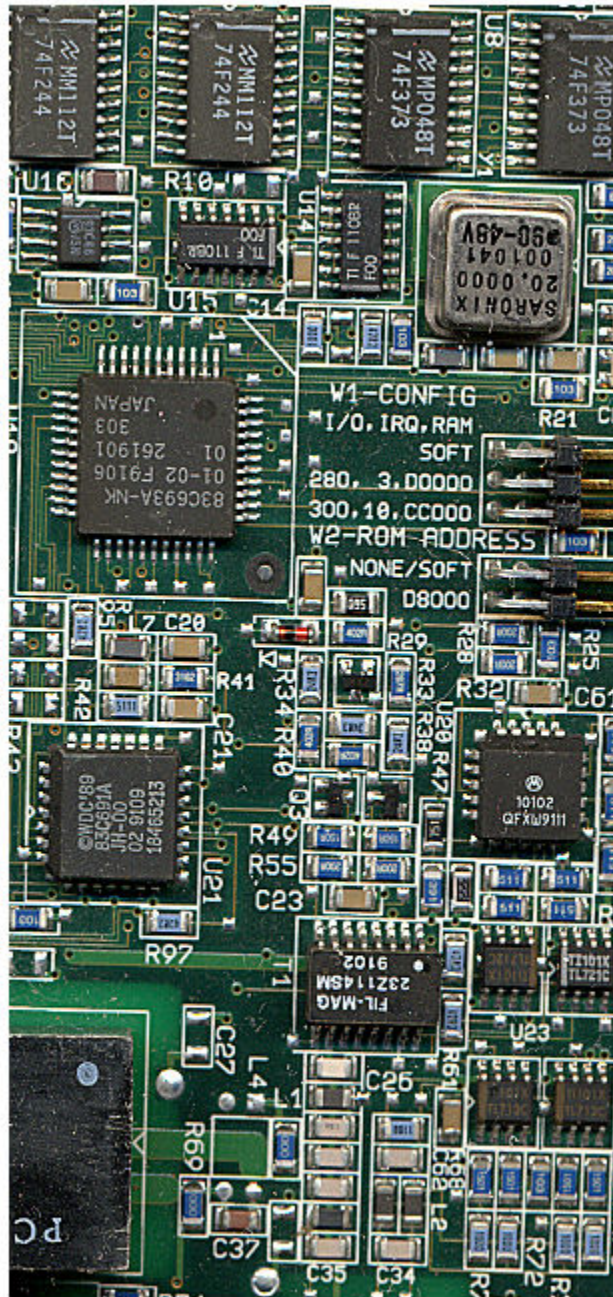
Convert the YIQ image back to RGB color space.

```
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

[ind2gray](#) | [ind2rgb](#) | [rgb2ind](#) | [rgb2ntsc](#)

**Introduced before R2006a**

## offsetstrel

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.

A nonflat structuring element is a matrix 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. The matrix is of type `double`.

To create a flat structuring element, use `strel`.

### Creation

### Syntax

```
SE = offsetstrel('ball',r,h)
SE = offsetstrel('ball',r,h,n)

SE = offsetstrel(offset)
```

### Description

`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
$n > 0$	<code>offsetstrel</code> uses a sequence of <code>n</code> nonflat, line-shaped structuring elements to approximate the shape. <code>n</code> must be an even number.

Value of $n$	Behavior
$n = 0$	<code>offsetstrel</code> 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

numeric matrix

Values to be added to each pixel location in the neighborhood when performing the morphological operation, specified as a numeric matrix. Values that are `-Inf` are not considered in the computation.

Data Types: double

## Properties

### **Offset** — Structuring element neighborhood with offsets

numeric matrix

Structuring element neighborhood with offsets, specified as a numeric matrix.

Data Types: logical

### **Dimensionality** — Dimensions of structuring element

nonnegative scalar

Dimensions of structuring element, specified as a nonnegative scalar.

Data Types: double

## Object Functions

<code>decompose</code>	Return sequence of decomposed structuring elements
<code>reflect</code>	Reflect structuring element
<code>translate</code>	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: [11x11 double]
Dimensionality: 2
```

View the structuring element.

SE.Offset

```
ans = 11x11
```

```

    -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

`strel`

## Topics

“Structuring Elements”

**Introduced before R2006a**

# OnePlusOneEvolutionary

One-plus-one evolutionary optimizer configuration

## Description

A `OnePlusOneEvolutionary` object describes a one-plus-one evolutionary optimization configuration that you pass to the function `imregister` to solve image registration problems.

## Creation

You can create a `OnePlusOneEvolutionary` object using the following methods:

- `imregconfig` — Returns a `OnePlusOneEvolutionary` object paired with an appropriate metric for registering multimodal images
- Entering

```
metric = registration.optimizer.OnePlusOneEvolutionary;
```

on the command line creates a `OnePlusOneEvolutionary` object with default settings

## 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`

## **Examples**

### **Register Images with One Plus One Evolutionary Optimizer**

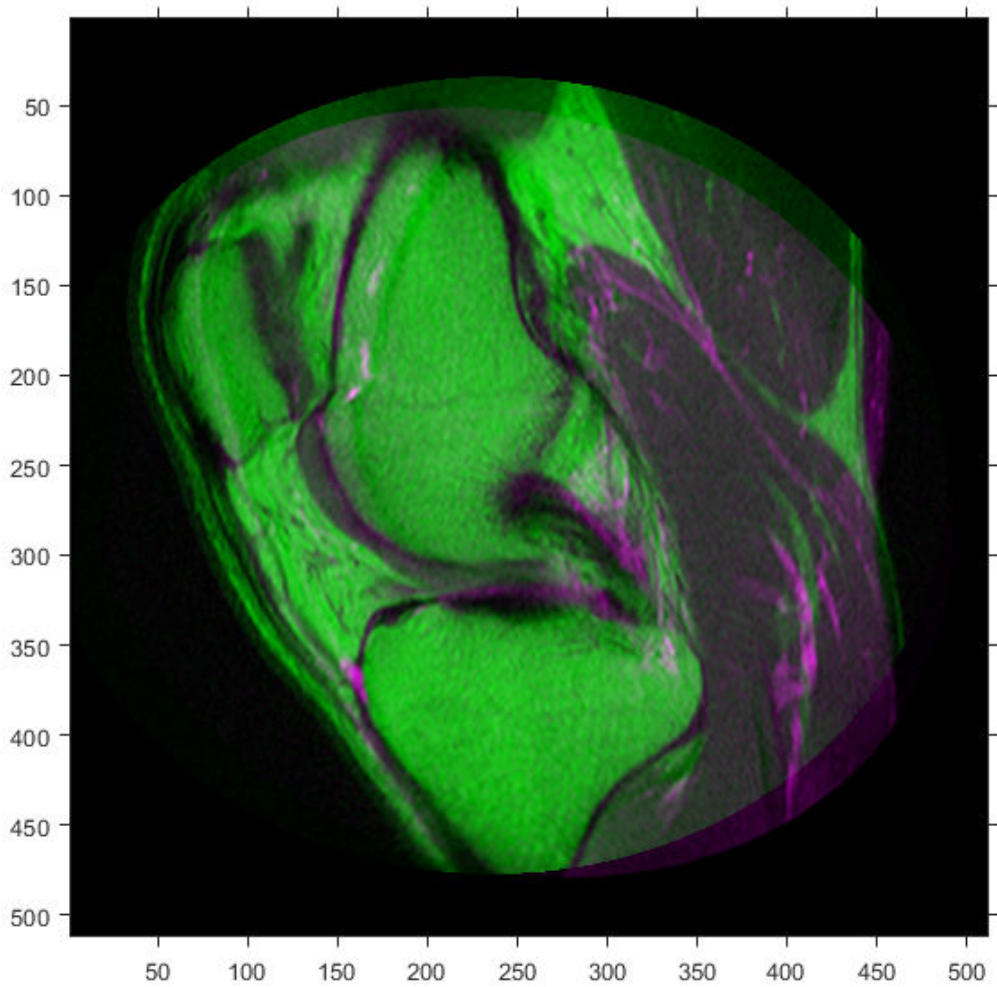
Create a `OnePlusOneEvolutionary` object and use it to register two MRI images of a knee that were obtained using different protocols.

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.

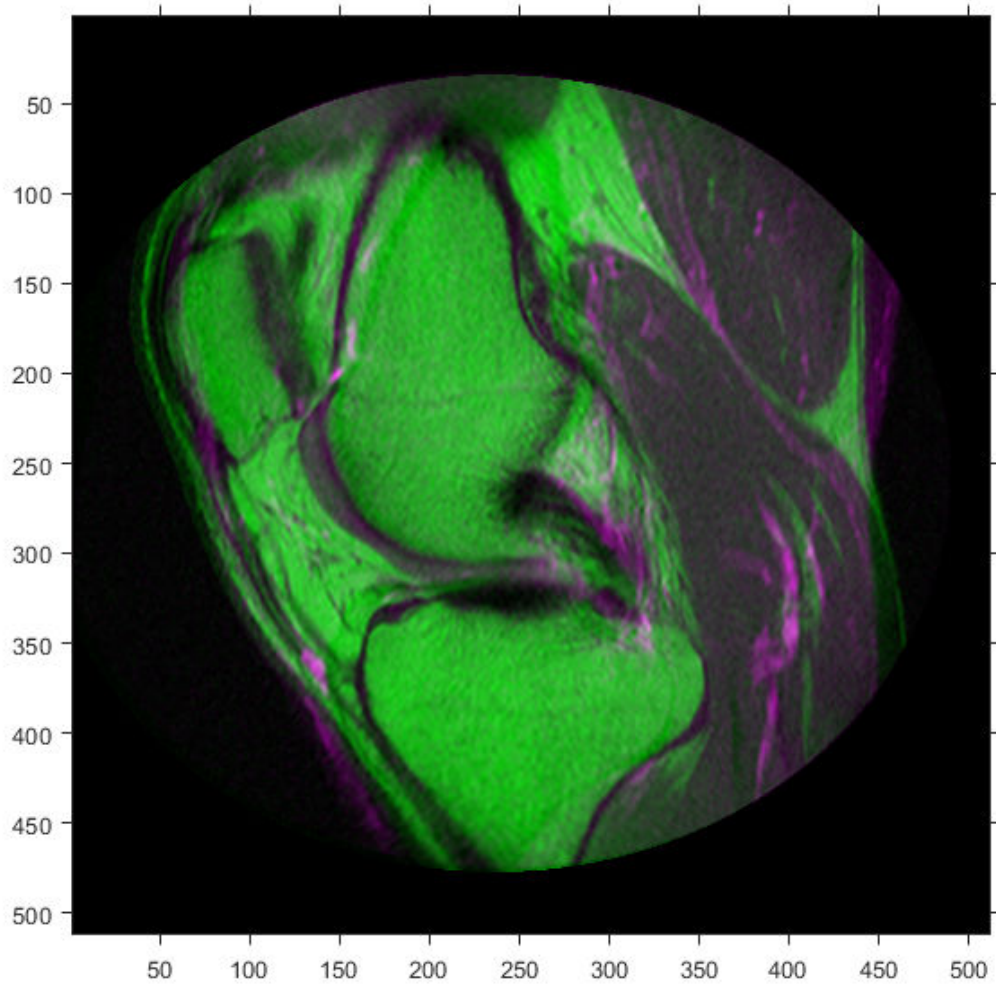
```
optimizer.InitialRadius = 0.009;
optimizer.Epsilon = 1.5e-4;
optimizer.GrowthFactor = 1.01;
optimizer.MaximumIterations = 300;
```

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

### Functions

`imregconfig` | `imregister`

### Objects

`MattesMutualInformation` | `MeanSquares` | `RegularStepGradientDescent`

## Topics

"Create an Optimizer and Metric for Intensity-Based Image Registration"

**Introduced in R2012a**

## **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 `orderth` 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.

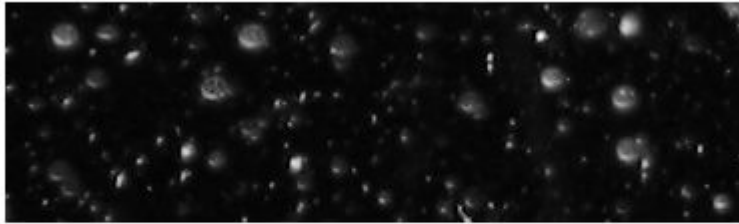
`B = ordfilt2( ____,padopt)` filters `A`, where `padopt` specifies how `ordfilt2` pads the matrix boundaries.

## 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.

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 1s and 0s. `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	Sample Image Data, Indicating Selected Element																		
Median filter	<code>B = ordfilt2(A, 5, ones(3,3))</code>	<table border="1"> <tr><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	1	1	1	1	1	1	1	1	1	<table border="1"> <tr><td>88</td><td>16</td><td>56</td></tr> <tr><td>5</td><td>3</td><td>30</td></tr> <tr><td>21</td><td>63</td><td>42</td></tr> </table>	88	16	56	5	3	30	21	63	42
1	1	1																			
1	1	1																			
1	1	1																			
88	16	56																			
5	3	30																			
21	63	42																			
Minimum filter	<code>B = ordfilt2(A, 1, ones(3,3))</code>	<table border="1"> <tr><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	1	1	1	1	1	1	1	1	1	<table border="1"> <tr><td>88</td><td>16</td><td>56</td></tr> <tr><td>5</td><td>3</td><td>30</td></tr> <tr><td>21</td><td>63</td><td>42</td></tr> </table>	88	16	56	5	3	30	21	63	42
1	1	1																			
1	1	1																			
1	1	1																			
88	16	56																			
5	3	30																			
21	63	42																			

Type of Filtering Operation	MATLAB code	Neighborhood	Sample Image Data, Indicating Selected Element																		
Maximum filter	<code>B = ordfilt2(A, 9, ones(3,3))</code>	<table border="1"> <tr><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	1	1	1	1	1	1	1	1	1	<table border="1"> <tr><td>88</td><td>16</td><td>56</td></tr> <tr><td>5</td><td>3</td><td>30</td></tr> <tr><td>21</td><td>63</td><td>42</td></tr> </table>	88	16	56	5	3	30	21	63	42
1	1	1																			
1	1	1																			
1	1	1																			
88	16	56																			
5	3	30																			
21	63	42																			
Minimum of north, east, south, and west neighbors	<code>B = ordfilt2(A, 1, [0 1 0; 1 0 1; 0 1 0])</code>	<table border="1"> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> </table>	0	1	0	1	0	1	0	1	0	<table border="1"> <tr><td>88</td><td>16</td><td>56</td></tr> <tr><td>5</td><td>3</td><td>30</td></tr> <tr><td>21</td><td>63</td><td>42</td></tr> </table>	88	16	56	5	3	30	21	63	42
0	1	0																			
1	0	1																			
0	1	0																			
88	16	56																			
5	3	30																			
21	63	42																			

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.



Option	Description
'symmetric'	Pad array with mirror reflections of itself.

Data Types: char | string

## 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 “Understand Code Generation with Image Processing Toolbox”.
- When generating code, the `padopt` argument must be a compile-time constant.

## See Also

`medfilt2`

**Introduced before R2006a**

# otf2psf

Convert optical transfer function to point-spread function

## Syntax

```
PSF = otf2psf(OTF)
PSF = otf2psf(OTF,sz)
```

## Description

`PSF = otf2psf(OTF)` computes the inverse Fast Fourier Transform of the optical transfer function (OTF) and creates a point-spread function (PSF), centered at the origin.

`PSF = otf2psf(OTF,sz)` specifies the size, `sz`, of the output point-spread function.

## 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]);
```

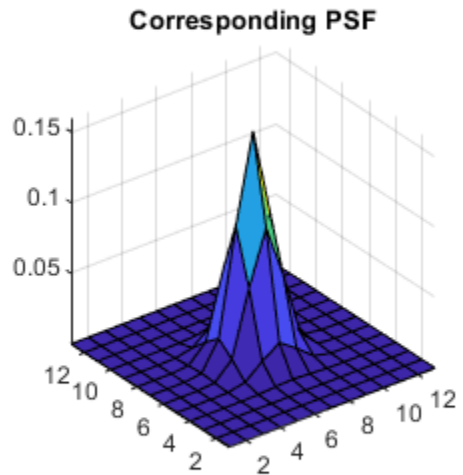
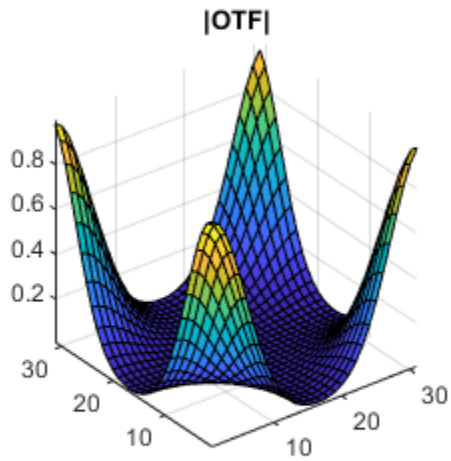
Convert the OTF back to a PSF.

```
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
```



## Input Arguments

### **OTF — Optical transfer function**

numeric array

Optical transfer function, specified as a numeric array of any dimension.

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

Complex Number Support: Yes

### **sz — Size of point-spread function**

vector of positive integers

Size of the output point-spread function PSF, specified as a vector of positive integers. The size of PSF must not exceed the size of OTF in any dimension. By default, PSF is the same size as OTF.

Data Types: `double`

## Output Arguments

### **PSF — Point-spread function**

numeric array

Point-spread function, centered at the origin, returned as a numeric array of size `sz`.

Data Types: `double`

Complex Number Support: Yes

## Tips

- 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 `sz`.
- This function is used in image convolution and deconvolution when the operations involve the FFT.

## **See Also**

`circshift` | `fft` | `ifft` | `padarray` | `psf2otf`

## **Topics**

“Create Your Own Deblurring Functions”

**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 [1]. `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

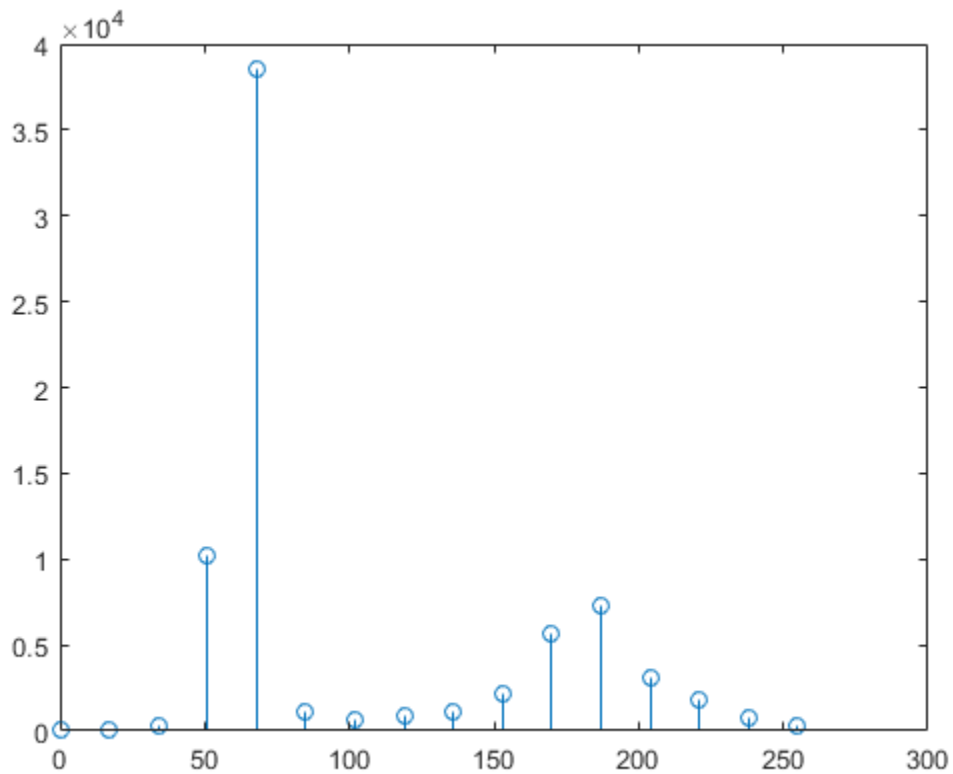
### Compute Threshold from Image Histogram and Binarize Image

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)
```



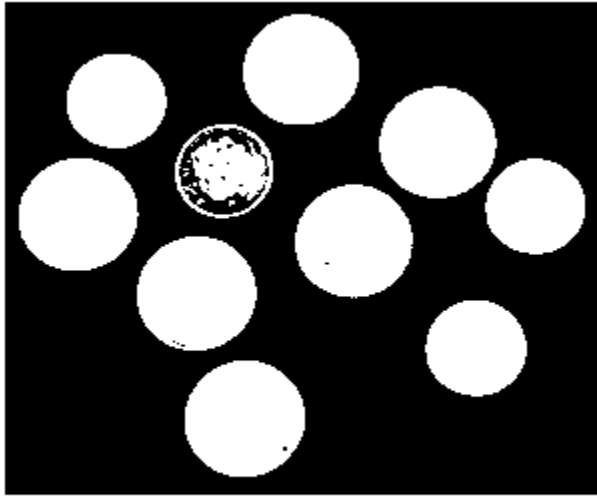
Compute a global threshold using the histogram counts.

```
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.

## **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. For more information, see "Code Generation for Image Processing".

## **See Also**

`adaptthresh` | `graythresh` | `imbinarize`

**Introduced in R2016a**

# outputLimits

## Package:

Find output spatial limits given input spatial limits

## 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 set of input spatial limits, `xLimitsIn` and `yLimitsIn`, given 2-D geometric transformation `tform`.

`[xLimitsOut,yLimitsOut,zLimitsOut] = outputLimits(tform,xLimitsIn,yLimitsIn,zLimitsIn)` estimates the output spatial limits, given 3-D geometric transformation `tform`.

## Examples

### Estimate the Output Limits for a 2-D Affine 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
```

### **Estimate the Output Limits for a 3-D Affine 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
```

## Input Arguments

### **tform — Geometric transformation**

geometric transformation object

Geometric transformation, specified as a geometric transformation object.

For 2-D geometric transformations, `tform` is an `affine2d`, `projective2d`, `LocalWeightedMeanTransformation2D`, `PiecewiseLinearTransformation2D`, or `PolynomialTransformation2D` geometric transformation object.

For 3-D geometric transformations, `tform` is an `affine3d` object.

### **xLimitsIn — Input spatial limits in the x-dimension**

1-by-2 numeric vector

Input spatial limits in the x-dimension, specified as a 1-by-2 numeric vector.

Data Types: `double`

### **yLimitsIn — Input spatial limits in the y-dimension**

1-by-2 numeric vector

Input spatial limits in the y-dimension, specified as a 1-by-2 numeric vector.

Data Types: `double`

### **zLimitsIn — Input spatial limits in the z-dimension**

1-by-2 numeric vector

Input spatial limits in the z-dimension, specified as a 1-by-2 numeric vector. Provide `zLimitsIn` only when `tform` is an `affine3d` object.

Data Types: `double`

## Output Arguments

### **xLimitsOut** — Output spatial limits in the x-dimension

1-by-2 numeric vector

Output spatial limits in the x-dimension, returned as a 1-by-2 numeric vector.

Data Types: double

### **yLimitsOut** — Output spatial limits in the y-dimension

1-by-2 numeric vector

Output spatial limits in the y-dimension, returned as a 1-by-2 numeric vector.

Data Types: double

### **zLimitsOut** — Output spatial limits in the z-dimension

1-by-2 numeric vector

Output spatial limits in the z-dimension, returned as a 1-by-2 numeric vector. `outputLimits` returns `zLimitsIn` only when `tform` is an `affine3d` object.

Data Types: double

## See Also

[LocalWeightedMeanTransformation2D](#) | [PiecewiseLinearTransformation2D](#) | [PolynomialTransformation2D](#) | [affine2d](#) | [affine3d](#) | [projective2d](#)

**Introduced in R2013a**

# padarray

Pad array

## Syntax

```
B = padarray(A,padsize)
B = padarray(A,padsize,padval)
B = padarray( ____,direction)
```

## Description

`B = padarray(A,padsize)` pads array `A` with 0s (zeros). `padsize` 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.

You optionally can perform the padding operation using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`B = padarray(A,padsize,padval)` pads array `A` where `padval` specifies a constant value to use for padded elements or a method to replicate array elements.

`B = padarray( ____,direction)` pads `A` in the direction specified by `direction`.

## Examples

### Add Padding to 2-D and 3-D Arrays

#### Pad the Beginning of a Vector

Add three elements of padding to the beginning of a vector with padding value 9.

```
A = [ 1 2 3 4 ]
```

A = 1×4

1	2	3	4
---	---	---	---

B = padarray(A,3,9, 'pre')

B = 4×4

9	9	9	9
9	9	9	9
9	9	9	9
1	2	3	4

## Pad Each Dimension of a 2-D Array

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. Use the value of the last array element on each dimension as the padding value.

A = [ 1 2; 3 4 ]

A = 2×2

1	2
3	4

B = padarray(A,[3 2], 'replicate', 'post')

B = 5×4

1	2	2	2
3	4	4	4
3	4	4	4
3	4	4	4
3	4	4	4

## Pad Each Dimension of a 3-D Array

Add three elements of padding to each dimension of a three-dimensional array. Each pad element contains the value 0.



First create the 3-D array.

```
A = [1 2; 3 4];
B = [5 6; 7 8];
C = cat(3,A,B)
```

```
C =
C(:,:,1) =
```

```
    1    2
    3    4
```

```
C(:,:,2) =
```

```
    5    6
    7    8
```

Pad the 3-D Array

```
D = padarray(C,[3 3],0,'both')
```

```
D =
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
```

```
D(:,:,2) =
```

```
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    5    6    0    0    0
    0    0    0    7    8    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
```

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)
```



## Input Arguments

**A — Array to be padded**

numeric array | logical array | gpuArray

Array to be padded, specified as a numeric or logical array of any dimension. When padding with a constant value, A must be numeric or logical. When padding using the 'circular', 'replicate', or 'symmetric' methods, A can be of any class.

To perform the padding operation using a GPU, specify A as a `gpuArray` that contains a numeric or logical array.

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

### **padsize** — Amount of padding to add to each dimension

vector of nonnegative integers

Amount of padding to add to each dimension, specified as a vector of nonnegative integers. For example, a `padsize` value of `[2 3]` means add two elements of padding along the first dimension and three 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.

Data Types: `double`

### **padval** — Value to use for the padding

0 (default) | numeric scalar | 'circular' | 'replicate' | 'symmetric'

Value to use for the padding, specified as one of the following.

Value	Meaning
numeric scalar	Pad array with elements of constant value.
'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.

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

### **direction** — Where to pad array along each dimension

'both' (default) | 'post' | 'pre'

Where to pad array along each dimension, specified as one of the following values:

<b>Value</b>	<b>Meaning</b>
'both'	Pads before the first element and after the last array element along each dimension.
'post'	Pad after the last array element along each dimension.
'pre'	Pad before the first array element along each dimension.

Data Types: `char` | `string`

## Output Arguments

### **B — Padded array**

`numeric array` | `gpuArray`

Padded array, returned as a numeric array of the same class as `A`.

If the padding operation is performed using a GPU, then `B` is returned as a `gpuArray` containing a numeric 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, `padarray` supports only up to 3-D inputs, and the input arguments, `padval` and `direction` must be compile-time constants.

### **GPU Arrays**

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

## **See Also**

`circshift` | `imfilter`

**Introduced before R2006a**

## para2fan

Convert parallel-beam projections to fan-beam

### Syntax

```
F = para2fan(P,D)
F = fan2para(P,D,Name,Value)
[F,fan_sensor_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 from the fan-beam vertex to the center of rotation.

The parallel-beam sensors are assumed to have a one-pixel spacing. The parallel-beam rotation angles are spaced equally to cover  $[0,180]$  degrees. The calculated fan-beam rotation angles have the same spacing as the parallel-beam rotation angles, and cover  $[0,360)$  degrees. The calculated fan-beam angles are equally spaced with the spacing set to the smallest angle implied by the sensor spacing.

`F = fan2para(P,D,Name,Value)` uses name-value pairs to control aspects of the data conversion. Argument names can be abbreviated, and case does not matter.

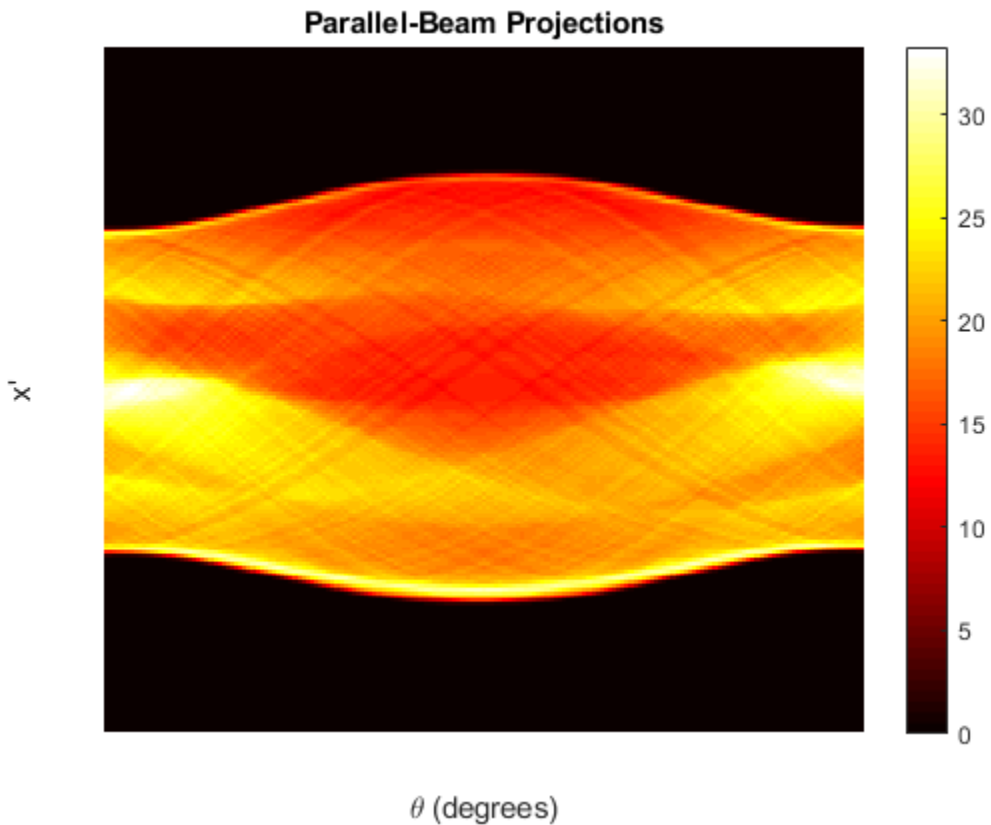
`[F,fan_sensor_positions,fan_rotation_angles] = fan2para( ___ )` returns the fan-beam sensor locations in `fan_sensor_positions` and rotation angles in `fan_rotation_angles`.

### Examples

#### Convert Parallel-beam Projections to Fan-beam Projections

Generate parallel-beam projections

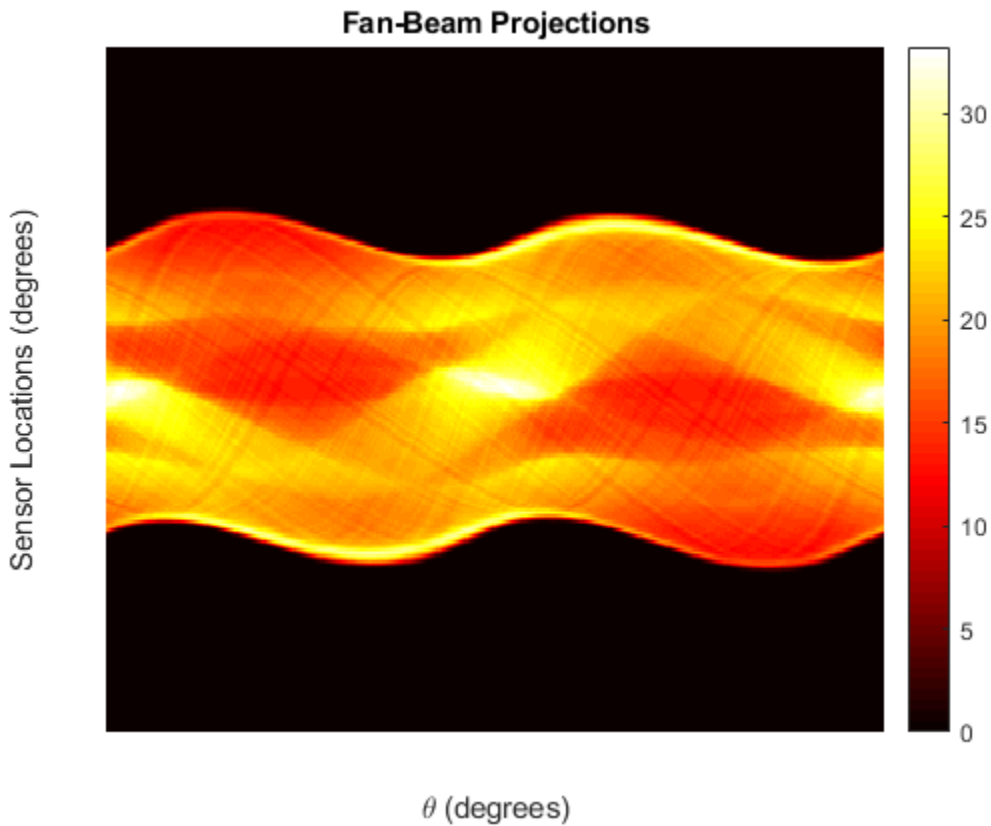
```
ph = phantom(128);  
theta = 0:180;  
[P,xp] = radon(ph,theta);  
imshow(P,[],'XData',theta,'YData',xp,'InitialMagnification','fit')  
axis normal  
title('Parallel-Beam Projections')  
xlabel('\theta (degrees)')  
ylabel('x''')  
colormap(gca,hot), colorbar
```



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
```



## Input Arguments

**P** — Parallel-beam projection data  
numeric matrix



Parallel-beam projection data, specified as a numeric matrix. Each column of  $P$  contains the parallel-beam data at one rotation angle. The number of columns indicates the number of parallel-beam rotation angles and the number of rows indicates the number of parallel-beam sensors.

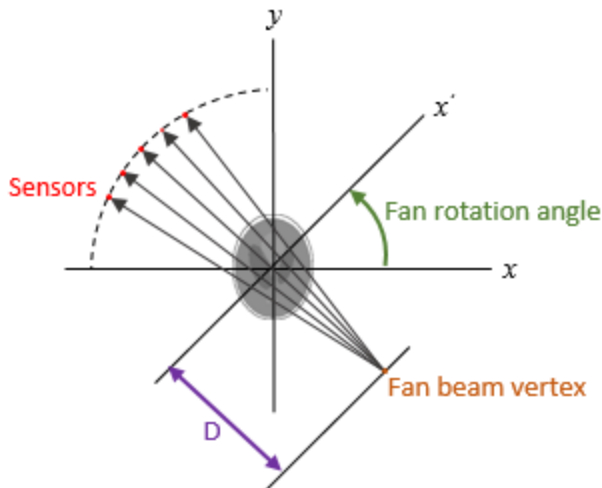
Data Types: `double` | `single`

### **D — Distance from fan beam vertex to center of rotation**

positive number

Distance in pixels from the fan beam vertex to the center of rotation, specified as a positive number. `para2fan` assumes that the center of rotation is the center point of the projections, which is defined as  $\text{ceil}(\text{size}(F,1)/2)$ . The value of  $D$  must be greater than or equal to  $\text{ParallelSensorSpacing} * (\text{SIZE}(P,1) - 1) / 2$

The figure illustrates  $D$  in relation to the fan-beam vertex for one fan-beam projection.



Data Types: `double` | `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 quotes.

You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `F = para2fan(P,D,'FanRotationIncrement',5)`

### **FanCoverage** — Range of fan-beam rotation

'cycle' (default) | 'minimal'

Range of fan-beam rotation, specified as the comma-separated pair consisting of 'FanCoverage' and one of these values.

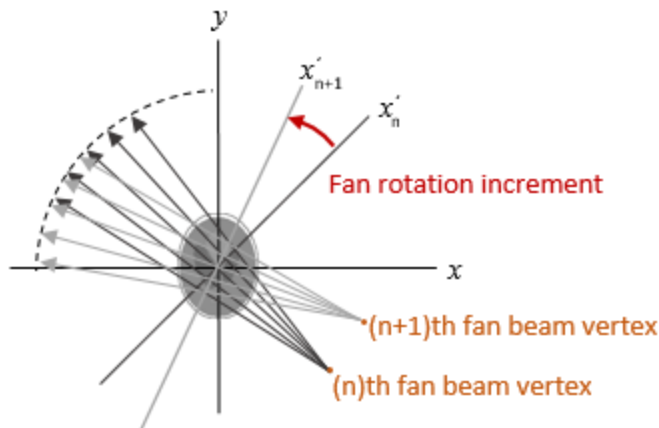
- 'cycle' — Rotate through the full range [0, 360) degrees.
- 'minimal' — Rotate through the minimum range necessary to represent the object.

### **FanRotationIncrement** — Fan-beam rotation angle increment

positive scalar

Fan-beam rotation angle increment in degrees, specified as the comma-separated pair consisting of 'FanRotationIncrement' and a positive scalar.

- If FanCoverage is 'cycle', then  $360/\text{FanRotationIncrement}$  must be an integer.
- If you do not specify FanRotationIncrement, then the default value is equal to the parallel-beam rotation angle.

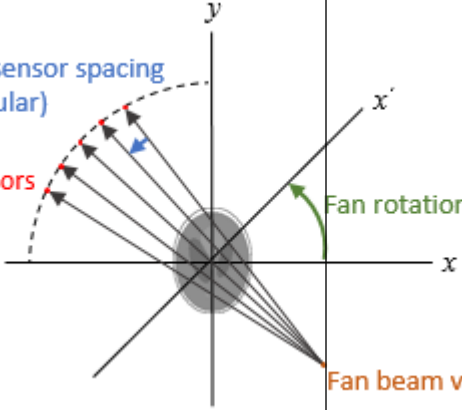


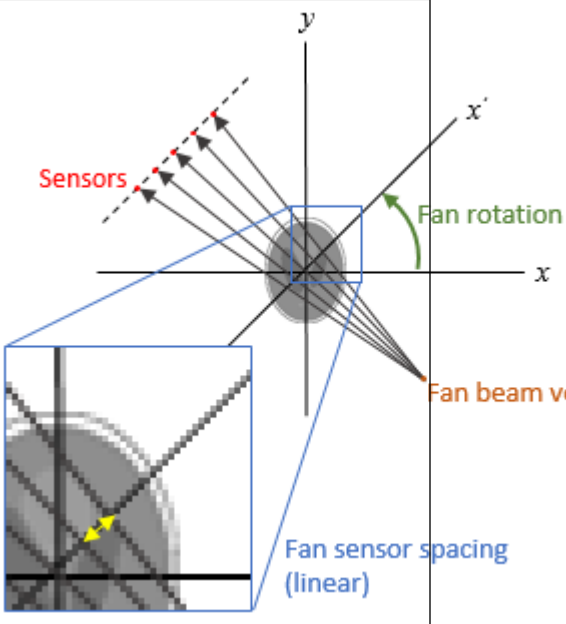
Data Types: double

## FanSensorGeometry — Fan-beam sensor positioning

'arc' (default) | 'line'

Fan-beam sensor positioning, specified as the comma-separated pair consisting of 'FanSensorGeometry' and one of the following values.

Value	Meaning	Diagram
'arc'	<p>Sensors are spaced at equal angles along a circular arc at distance D from the center of rotation.</p> <p>FanSensorSpacing defines the angular spacing in degrees.</p>	

Value	Meaning	Diagram
'line'	<p>Sensors are spaced at equal distances along a line that is parallel to the <math>x'</math> axis. The closest sensor is distance <math>D</math> from the center of rotation.</p> <p>FanSensorSpacing defines the distance between fan-beams on the <math>x'</math> axis, in pixels.</p>	

**FanSensorSpacing — Fan-beam sensor spacing**

positive scalar

Fan-beam sensor spacing, specified as the comma-separated pair consisting of 'FanSensorSpacing' and a positive scalar.

- If FanSensorGeometry is 'arc', then FanSensorSpacing defines the angular spacing in degrees.
- If FanSensorGeometry is 'line', then FanSensorSpacing defines the linear distance between fan-beams, in pixels. Linear spacing is measured on the  $x'$  axis.

If you do not specify FanSensorGeometry, then the default value of FanSensorSpacing is the smallest value implied by ParallelSensorSpacing such that:

- If FanSensorGeometry is 'arc', then FanSensorSpacing is  $180/\pi * \text{ASIN}(\text{ParallelSensorSpacing}/D)$

- If FanSensorGeometry is 'line', then FanSensorSpacing is  $D * \text{ASIN}(\text{ParallelSensorSpacing}/D)$

Data Types: double

### **Interpolation — Type of interpolation**

'Linear' (default) | 'nearest' | 'spline' | 'pchip'

Type of interpolation used between the parallel-beam and fan-beam data, specified as the comma-separated pair consisting of 'Interpolation' and one of these values.

'nearest' — Nearest-neighbor

'linear' — Linear (the default)

'spline' — Piecewise cubic spline

'pchip' — Piecewise cubic Hermite (PCHIP)

### **ParallelCoverage — Range of parallel-beam rotation**

'halfcycle' (default) | 'cycle'

Range of parallel-beam rotation, specified as the comma-separated pair consisting of 'ParallelCoverage' and one of these values.

- 'cycle' — Parallel data covers the full range of [0, 360) degrees.
- 'halfcycle' — Parallel data covers [0, 180) degrees.

### **ParallelSensorSpacing — Parallel-beam sensor spacing**

1 | positive scalar

Parallel-beam sensor spacing in pixels, specified as the comma-separated pair consisting of 'ParallelSensorSpacing' and a positive scalar.

Data Types: double

## **Output Arguments**

### **F — Fan-beam projection data**

numeric matrix

Fan-beam projection data, returned as a numeric matrix. Each column of `F` contains the fan-beam sensor samples at one rotation angle.

Parallel-beam projection data, returned as a numeric matrix. Each column of `F` contains the fan-beam data at one rotation angle. The number of columns indicates the total number of fan-beam rotation angles and is equal to the length of `fan_rotation_angles`. The number of rows indicates the total number of parallel-beam sensors and is equal to the length of `fan_sensor_positions`.

Data Types: double

### **fan\_sensor\_positions — Fan-beam sensor locations**

numeric column vector

Fan-beam sensor locations, returned as a numeric column vector.

- If 'FanSensorGeometry' is 'arc' (the default), then `fan_sensor_positions` contains the fan-beam sensor measurement angles.
- If 'FanSensorGeometry' is 'line', then `fan_sensor_positions` contains the fan-beam sensor positions along the line of sensors.

Data Types: double

### **fan\_rotation\_angles — Fan-beam rotation angles**

numeric row vector

Fan-beam rotation angles, returned as a numeric row vector.

Data Types: double

## **See Also**

`fan2para` | `fanbeam` | `ifanbeam` | `iradon` | `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 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, and `n` specifies the number of rows and columns in the phantom image.

`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.

`[P, E] = phantom( ___ )` returns the matrix `E` used to generate the phantom.

## 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)
```



## Input Arguments

**def** — Type of head phantom

'Modified Shepp-Logan' (default) | 'Shepp-Logan'

Type of head phantom to generate, specified as one of the following.

- 'Shepp-Logan' — Test image used widely by researchers in tomography
- 'Modified Shepp-Logan' — Variant of the Shepp-Logan phantom in which the contrast is improved for better visual perception

Data Types: char | string

**n** — Number of rows and columns

256 (default) | positive integer

Number of rows and columns in the phantom image, specified as a positive integer.

Data Types: double



## E — Ellipses

*e*-by-6 numeric matrix

Ellipses that define the phantom, specified as an *e*-by-6 numeric matrix defining *e* ellipses. The six columns of E are the ellipse parameters.

Column	Parameter	Meaning
Column 1	A	Additive intensity value of the ellipse
Column 2	a	Length of the horizontal semiaxis of the ellipse
Column 3	b	Length of the vertical semiaxis of the ellipse
Column 4	x0	x-coordinate of the center of the ellipse
Column 5	y0	y-coordinate of the center of the ellipse
Column 6	phi	Angle (in degrees) between the horizontal semiaxis of the ellipse and the x-axis of the image

The domains for the *x*- and *y*-axes are [-1,1]. Columns 2 through 5 must be specified in terms of this range.

Data Types: double

## Output Arguments

### P — Phantom image

*n*-by-*n* numeric matrix

Phantom image, returned as an *n*-by-*n* numeric matrix.

Data Types: double

## 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

`iradon` | `radon`

**Introduced before R2006a**

# images.geotrans.PiecewiseLinearTransformation2D

2-D piecewise linear geometric transformation

## Description

A `PiecewiseLinearTransformation2D` object encapsulates a 2-D piecewise linear geometric transformation.

## Creation

You can create a `PiecewiseLinearTransformation2D` object using the following methods:

- The `fitgeotrans` function, which estimates a geometric transformation that maps pairs of control points between two images.
- The `images.geotrans.PiecewiseLinearTransformation2D` function described here. This function creates a `PiecewiseLinearTransformation2D` object using coordinates of fixed points and moving points.

## Syntax

```
tform = images.geotrans.PiecewiseLinearTransformation2D(  
movingPoints, fixedPoints)
```

## Description

`tform = images.geotrans.PiecewiseLinearTransformation2D(movingPoints, fixedPoints)` creates a `PiecewiseLinearTransformation2D` object given control point coordinates in `movingPoints` and `fixedPoints`, which define matched control points in the moving and fixed images, 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. 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`

## Properties

### **Dimensionality** — Dimensionality of the geometric transformation

2

Dimensionality of the geometric transformation for both input and output points, specified as the value 2.

## Object Functions

`outputLimits`

Find output spatial limits given input spatial limits

`transformPointsInverse`

Apply inverse geometric transformation

## 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 the 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

### Functions

[cpselect](#) | [fitgeotrans](#) | [imwarp](#)

### Classes

[LocalWeightedMeanTransformation2D](#) | [PolynomialTransformation2D](#) | [affine2d](#) | [projective2d](#)

**Introduced in R2013b**

# piqe

Perception based Image Quality Evaluator (PIQE) no-reference image quality score

## Syntax

```
score = piqe(A)  
[score,activityMask,noticeableArtifactsMask,noiseMask] = piqe(A)
```

## Description

`score = piqe(A)` calculates the no-reference image quality score for image A using a perception based image quality evaluator.

`[score,activityMask,noticeableArtifactsMask,noiseMask] = piqe(A)` also returns the spatial quality masks computed from the input image.

## Examples

### Calculate PIQE Score for Images and Display Results

Calculate PIQE score for an image and the corresponding distorted images. Display the results with their corresponding image.

Read an image into the workspace. Generate distorted images by adding noise and blur. Use `imnoise` function to generate the noisy image and `imgaussfilt` function to generate the blurred image.

```
A = imread('lighthouse.png');  
Anoise = imnoise(A,'Gaussian',0,0.05);  
Ablur = imgaussfilt(A,2);
```

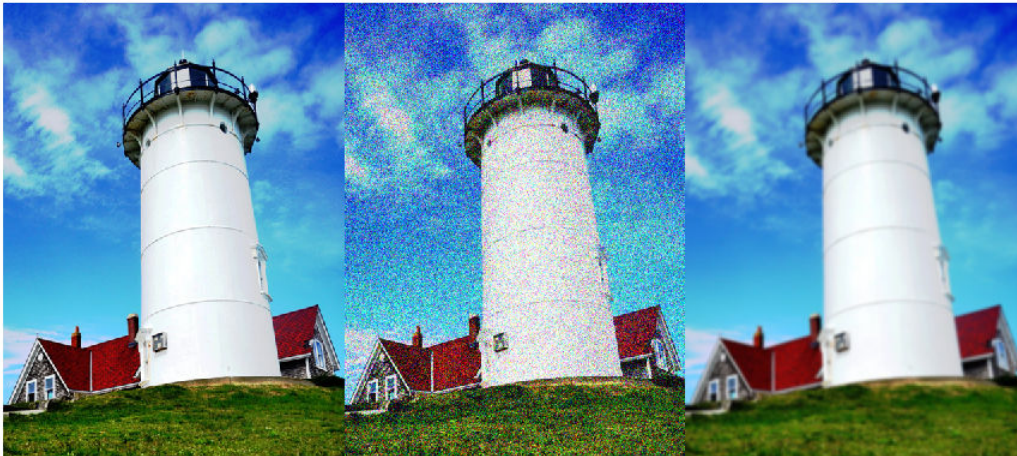
Calculate PIQE score for the original image and the distorted images.

```
score = piqe(A);  
score_noise = piqe(Anoise);  
score_blur = piqe(Ablur);
```

Display the images as a montage with their corresponding scores as a part of the figure title.

```
figure  
montage({A,Anoise,Ablur}, 'Size',[1 3])  
title({'Original Image: PIQE score = ', num2str(score), ' | Noisy Image: PIQE score = ', num2str(score_noise), ' | Blurred Image: PIQE score = ', num2str(score_blur)}, 'FontSize',12)
```

Original Image: PIQE score = 24.8481 | Noisy Image: PIQE score = 72.3375 | Blurred Image: PIQE score = 85.7362



## Calculate PIQE Score, Spatial Quality Masks for Image, and Display Results

Calculate PIQE score of an image distorted due to blocking artifacts and Gaussian noise. Generate spatial quality masks that indicate the high spatially active blocks, noticeable artifacts blocks, and noise blocks in the image. Visualize the spatial quality masks by overlaying them on the distorted image. Display the image with and without the masks and the PIQE score for the image.

Read a distorted image (distortion due to JPEG2K) into the workspace.



```
Adistorted = imread('DistortedImage.png');
```

Calculate PIQE score and the spatial quality masks.

```
[score,activityMask,noticeableArtifactsMask,noiseMask] = piqe(Adistorted);
```

Overlay the spatial quality masks on the input image.

```
mask_1 = labeloverlay(Adistorted,activityMask,'Colormap','winter','Transparency',0.25);
mask_2 = labeloverlay(Adistorted,noticeableArtifactsMask,'Colormap','autumn','Transparency',0.25);
mask_3 = labeloverlay(Adistorted,noiseMask,'Colormap','hot','Transparency',0.25);
```

Display the original distorted image and the distorted images with overlaid spatial quality masks as a montage.

```
figure
montage({Adistorted,mask_1,mask_2,mask_3},'Size',[1 4])
title('Distorted Image | Overlay activityMask | Overlay noticeableArtifactsMask | Overlay noiseMask')
```



Display PIQE score for the distorted image.

```
fprintf('PIQE score for the distorted image is %0.4f.\n',score)
```

```
PIQE score for the distorted image is 65.1855.
```

## Input Arguments

### A — Input image

2-D grayscale image | 2-D RGB image

Input image, specified as a 2-D grayscale image of size  $m$ -by- $n$  or 2-D RGB image of size  $m$ -by- $n$ -by-3.

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

## Output Arguments

### score — PIQE score

nonnegative scalar

PIQE score for the input image A, returned as a nonnegative scalar in the range [0, 100]. The PIQE score is the no-reference image quality score and it is inversely correlated to the perceptual quality of an image. A low score value indicates high perceptual quality and high score value indicates low perceptual quality.

Data Types: `double`

### activityMask — Spatial quality mask of active blocks

2-D Binary image

Spatial quality mask of active blocks, returned as a 2-D binary image of size  $m$ -by- $n$ , where  $m$  and  $n$  are the dimensions of the input image A. The `activityMask` is composed of high spatially active blocks in the input image. The high spatially active blocks in the input image are the regions with more spatial variability due to factors that include compression artifacts and noise. The high spatially active blocks are assigned a value '1' in the `activityMask`.

Data Types: `logical`

### noticeableArtifactsMask — Spatial quality mask of noticeable artifacts

2-D Binary image

Spatial quality mask of noticeable artifacts, returned as a 2-D binary image of size  $m$ -by- $n$ , where  $m$  and  $n$  are the dimensions of the input image A. The `noticeableArtifactsMask` is composed of blocks in `activityMask` that contain blocking artifacts (due to compression) or sudden distortions.

Data Types: logical

### **noiseMask — Spatial quality mask of Gaussian noise**

2-D Binary image

Spatial quality mask of Gaussian noise, returned as a 2-D binary image of size  $m$ -by- $n$ , where  $m$  and  $n$  are the dimensions of the input image **A**. The **noiseMask** is composed of blocks in **activityMask** that contain Gaussian noise.

Data Types: logical

## **Algorithms**

PIQE calculates the no-reference quality score for an image through block-wise distortion estimation, using these steps:

- 1** Compute the Mean Subtracted Contrast Normalized (MSCN) coefficient for each pixel in the image using the algorithm proposed by N. Venkatanath and others [1].
- 2** Divide the input image into nonoverlapping blocks of size 16-by-16.
- 3** Identify high spatially active blocks based on the variance of the MSCN coefficients.
- 4** Generate **activityMask** using the identified high spatially active blocks.
- 5** In each block, evaluate distortion due to blocking artifacts and noise using the MSCN coefficients.
- 6** Use threshold criteria to classify the blocks as distorted blocks with blocking artifacts, distorted blocks with Gaussian noise, and undistorted blocks.
- 7** Generate **noticeableArtifactsMask** from the distorted blocks with blocking artifacts and **noiseMask** from the distorted blocks with Gaussian noise.
- 8** Compute the PIQE score for the input image as the mean of scores in the distorted blocks.
- 9** The quality scale of the image based on its PIQE score is given in this table. The quality scale and respective score range are assigned through experimental analysis on the dataset in LIVE Image Quality Assessment Database Release 2 [2].





## References

- [1] N. Venkatanath, D. Praneeth, Bh. M. Chandrasekhar, S. S. Channappayya, and S. S. Medasani. "Blind Image Quality Evaluation Using Perception Based Features", In *Proceedings of the 21<sup>st</sup> National Conference on Communications (NCC)*. Piscataway, NJ: IEEE, 2015.
- [2] Sheikh, H. R., Z. Wang, L. Cormack and A.C. Bovik, "LIVE Image Quality Assessment Database Release 2 ", <https://live.ece.utexas.edu/research/quality>.

## See Also

### Functions

`brisque` | `immse` | `niqe` | `psnr` | `ssim`

### Topics

"Image Quality Metrics"

**Introduced in R2018b**

## plotChromaticity

Plot color reproduction on chromaticity diagram

### Syntax

```
plotChromaticity(colorTable)  
plotChromaticity(colorTable,Name,Value)
```

```
plotChromaticity  
plotChromaticity(Name,Value)
```

### Description

`plotChromaticity(colorTable)` plots on a chromaticity diagram the measured and reference colors, `colorTable`, for color patch regions of interest (ROIs) in a test chart.

`plotChromaticity(colorTable,Name,Value)` plots measured and reference colors with additional parameters to control aspects of the display.

`plotChromaticity` plots an empty chromaticity diagram.

`plotChromaticity(Name,Value)` plots an empty chromaticity diagram with additional parameter 'Parent' that specifies a handle of a parent axes of the plot object.

### Examples

#### Display Chromaticity Diagram from Color Accuracy Measurements

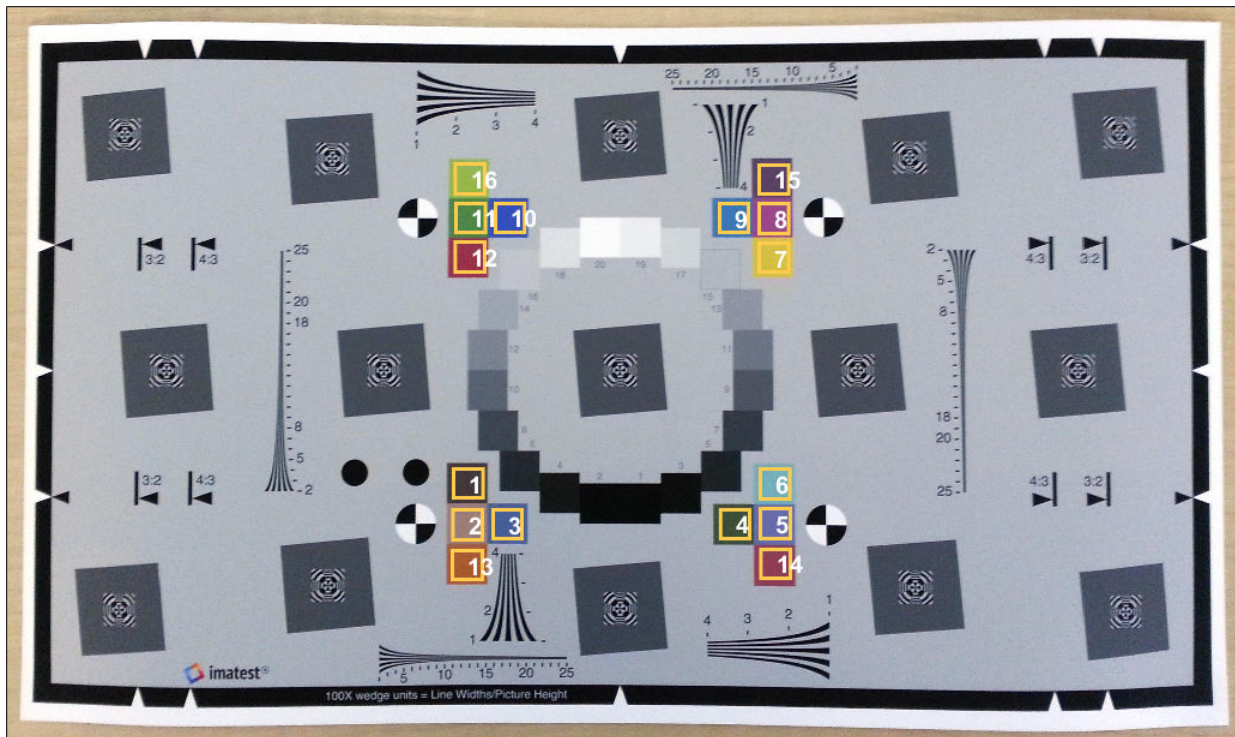
This example shows how to display the chromaticity diagram from measurements of color accuracy on an Imatest® eSFR chart.

Read an image of an eSFR chart into the workspace.

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

Create an esfrChart object. Display the chart, highlighting the 16 color patches.

```
chart = esfrChart(I);
displayChart(chart, 'displayEdgeROIs', false, ...
    'displayGrayROIs', false, 'displayRegistrationPoints', false)
```

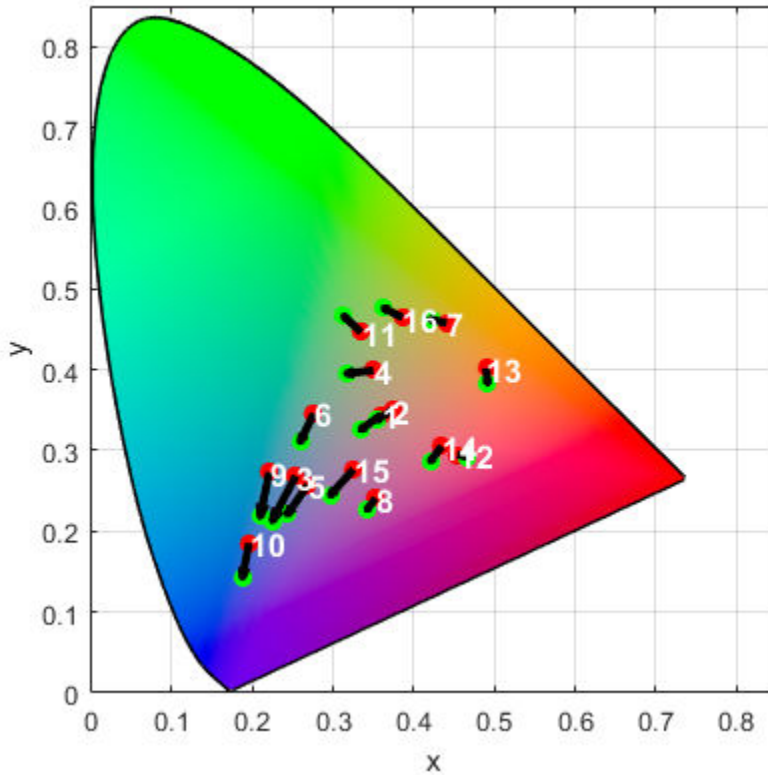


Measure the color in all color patch ROIs.

```
colorTable = measureColor(chart);
```

Plot the measured and reference colors in the CIE 1976 L\*a\*b\* color space on a chromaticity diagram. Red circles indicate the reference color and green circles indicate the measured color of each color patch. The chromaticity diagram does not portray the brightness of color.

```
figure
plotChromaticity(colorTable)
```



## Input Arguments

### **colorTable** — Color values

color table

Color values in each color patch, specified as an  $m$ -by-8 color table, where  $m$  is the number of patches. The eight columns represent these variables:



Variable	Description
ROI	Index of the sampled ROI. The value of ROI is an integer in the range [1, 16]. The indices match the ROI numbers displayed by <code>displayChart</code> .
Measured_R	Mean value of red channel pixels in an ROI. <code>Measured_R</code> is a scalar of the same data type as <code>chart.Image</code> , which can be of type <code>single</code> , <code>double</code> , <code>uint8</code> , or <code>uint16</code> .
Measured_G	Mean value of green channel pixels in an ROI. <code>Measured_G</code> is a scalar of the same data type as <code>chart.Image</code> .
Measured_B	Mean value of blue channel pixels in an ROI. <code>Measured_B</code> is a scalar of the same data type as <code>chart.Image</code> .
Reference_L	Reference L* value corresponding to the ROI. <code>Reference_L</code> is a scalar of type <code>double</code> .
Reference_a	Reference a* value corresponding to the ROI. <code>Reference_a</code> is a scalar of type <code>double</code> .
Reference_b	Reference b* value corresponding to the ROI. <code>Reference_b</code> is a scalar of type <code>double</code> .
Delta_E	Euclidean color distance between the measured and reference color values, as outlined in CIE 1976.

To obtain a color table, use the `measureColor` function.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `plotChromaticity(myColorTable, 'displayROIIndex', false)` turns off the display of the ROI indices on the chromaticity diagram.

### **displayROIIndex** — Display ROI index labels

`true` (default) | `false`

Display ROI index labels, specified as the comma-separated pair consisting of `'displayROIIndex'` and `true` or `false`. When `displayROIIndex` is `true`, then

`plotChromaticity` overlays color patch ROI index labels on the chromaticity diagram. The indices match the ROI numbers displayed by `displayChart`.

Data Types: `logical`

### **Parent — Axes handle of displayed image object**

`axes handle`

Axes handle of the displayed image object, specified as the comma-separated pair consisting of 'Parent' and an axes handle. Parent specifies the parent of the image object created by `plotChromaticity`.

## **See Also**

### **Functions**

`displayChart` | `displayColorPatch` | `measureColor`

### **Objects**

`esfrChart`

**Introduced in R2017b**

# plotSFR

Plot spatial frequency response of edge

## Syntax

```
plotSFR(sharpnessMeasurementTable)
plotSFR(sharpnessMeasurementTable, Name, Value)
```

## Description

`plotSFR(sharpnessMeasurementTable)` plots the spatial frequency response (SFR) in a sharpness measurement table or aggregate sharpness measurement table.

`plotSFR(sharpnessMeasurementTable, Name, Value)` plots the SFR, specifying additional parameters to control aspects of the display.

## Examples

### Plot Spatial Frequency Response of Specific ROIs from an eSFR Chart

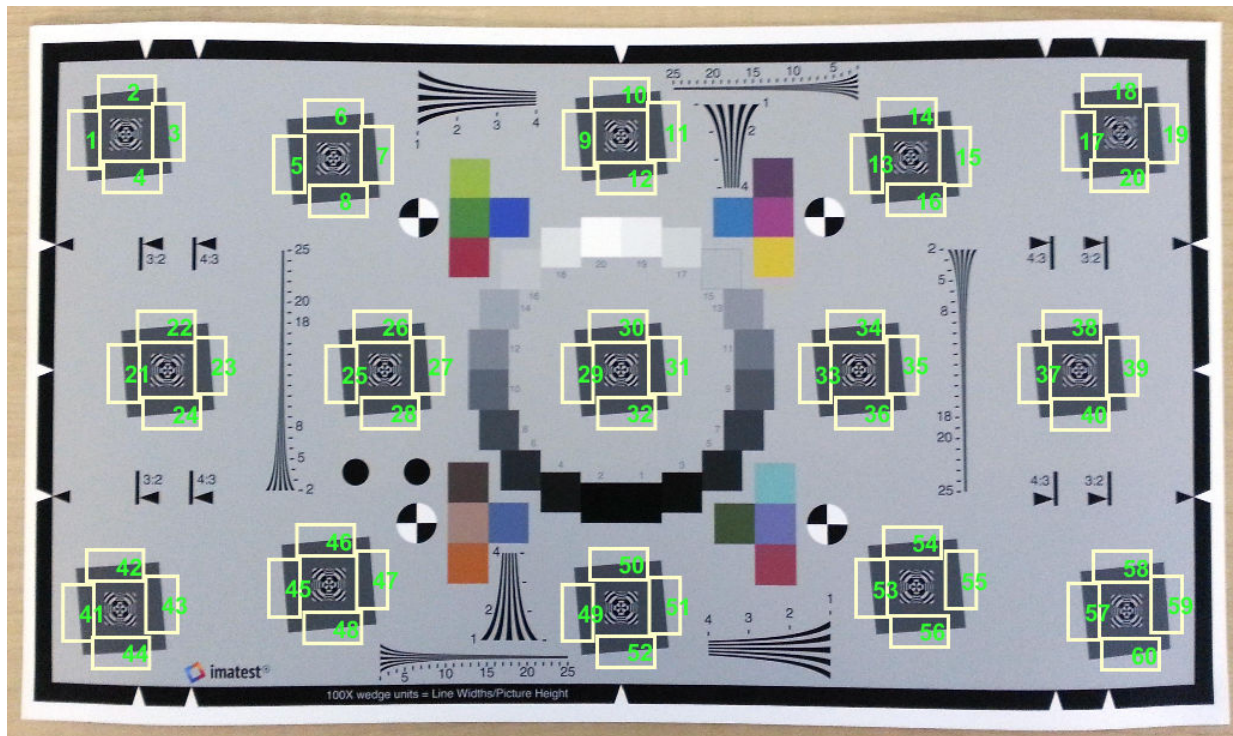
This example shows how to display the spatial frequency response (SFR) plot of a specified subset of the 60 slanted edge ROIs on an Imatest® eSFR chart.

Read an image of an eSFR chart into the workspace.

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

Create an `esfrChart` object, then display the chart with ROI annotations. The 60 slanted edge ROIs are labeled with green numbers.

```
chart = esfrChart(I);
displayChart(chart, 'displayGrayROIs', false, ...
    'displayColorROIs', false, 'displayRegistrationPoints', false)
```

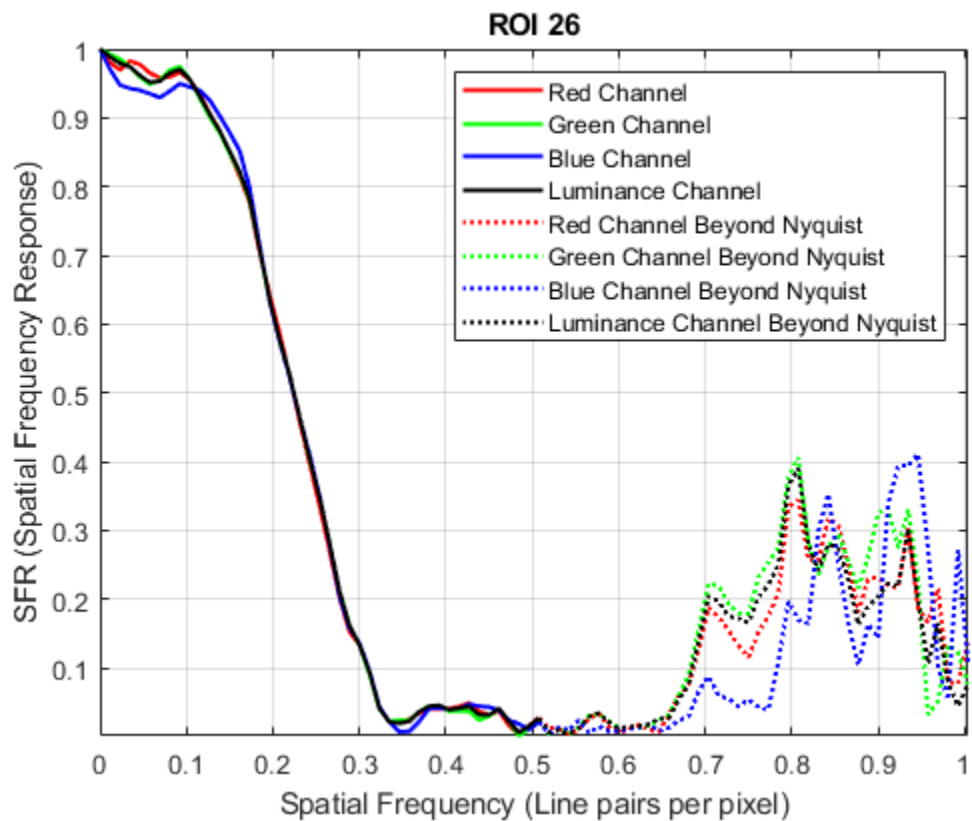


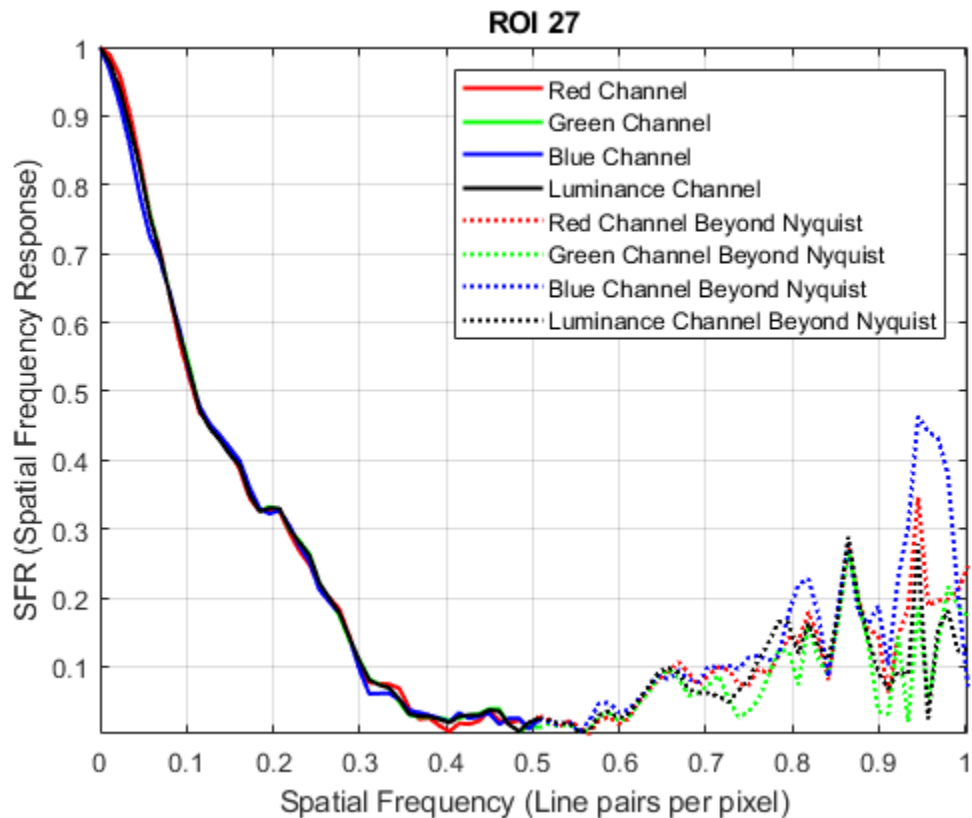
Measure the edge sharpness in all ROIs and return the measurements in sharpnessTable.

```
sharpnessTable = measureSharpness(chart);
```

Display the SFR plot of ROIs 26 and 27 only.

```
plotSFR(sharpnessTable, 'ROIIndex', [26 27]);
```





## Input Arguments

### **sharpnessMeasurementTable — SFR measurements**

sharpness table | aggregate sharpness table

SFR measurements of edges, specified as a sharpness table or aggregate sharpness table with  $m$  rows:

- When sharpnessMeasurementTable is a sharpness table,  $m$  is the number of sampled ROIs.

- When `sharpnessMeasurementTable` is an aggregate sharpness table, `m` is either 1 or 2, corresponding to the number of sampled orientations.

To obtain a sharpness table or aggregate sharpness table, use the `measureSharpness` function.

Data Types: `table`

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `plotSFR(myTable, 'ROIIndex', 2)` displays the measured sharpness only of ROI 2.

### ROIIndex — ROI indices

scalar | vector

ROI indices to display, specified as the comma-separated pair consisting of `'ROIIndex'` and a scalar or vector of integers in the range [1, 60]. The indices match the ROI numbers displayed by `displayChart`.

- When `sharpnessMeasurementTable` is a sharpness table, by default `plotSFR` creates only one figure, showing the SFR plot from the first row of the table.
- When `sharpnessMeasurementTable` is an aggregate sharpness table, `plotSFR` ignores the specified `ROIIndex`, and creates one figure for each row in the table.

Example: 29:32

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

### displayLegend — Display plot legend

`true` (default) | `false`

Display plot legend, specified as the comma-separated pair consisting of `'displayLegend'` and `true` or `false`. When `displayLegend` is `true`, the SFR plot shows a legend that identifies the different curves on the plot.

Data Types: `logical`

**displayTitle — Display plot title**

true (default) | false

Display plot title, specified as the comma-separated pair consisting of 'displayTitle' and true or false. When displayTitle is true, the SFR plot shows a title that indicates the individual ROI index or aggregate ROI orientation.

Data Types: logical

**Parent — Axes handle of displayed image object**

axes handle

Axes handle of the displayed image object, specified as the comma-separated pair consisting of 'Parent' and an axes handle. Parent specifies the parent of the image object created by plotSFR.

## See Also

**Functions**

displayChart | measureSharpness

**Objects**

esfrChart

**Introduced in R2017b**



# poly2mask

Convert region of interest (ROI) polygon to region mask

## Syntax

```
BW = poly2mask(xi,yi,m,n)
```

## Description

`BW = poly2mask(xi,yi,m,n)` computes a binary region of interest (ROI) mask, `BW`, of size `m`-by-`n`, from an ROI polygon with vertices at coordinates `xi` and `yi`. `poly2mask` closes the polygon automatically, if the polygon is not already closed.

The `poly2mask` function sets pixels that are inside the polygon to 1 and sets pixels outside the polygon to 0. For more information about classifying pixels that are partially enclosed by the ROI, see Algorithm on page 1-2423.

## 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.

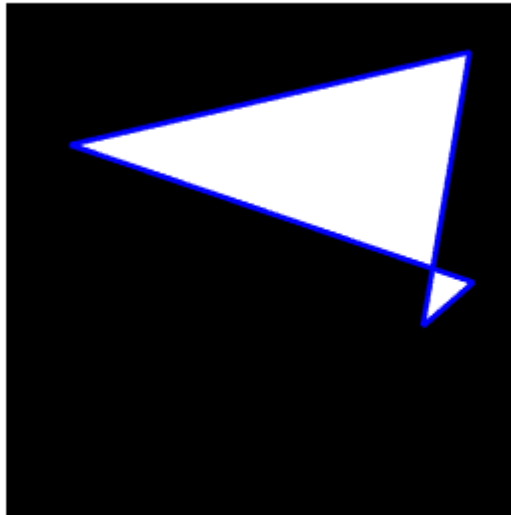
```
x = 256*rand(1,4);  
y = 256*rand(1,4);  
x(end+1) = x(1);  
y(end+1) = y(1);
```

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
```



## Input Arguments

**xi, yi** — x- or y-coordinate of polygon vertices

numeric vector

x- or y-coordinate of polygon vertices, specified as a numeric vector. The length of xi and yi must match.

Data Types: double

**m** — Number of rows in the mask

positive integer

Number of rows in the mask, specified as a positive integer.

Data Types: `double`

## **n** — Number of columns in the mask

positive integer

Number of columns in the mask, specified as a positive integer.

Data Types: `double`

## Output Arguments

### **BW** — Binary image

m-by-n logical matrix

Binary image, returned as an m-by-n logical matrix.

Data Types: `logical`

## Tips

- 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 =

```

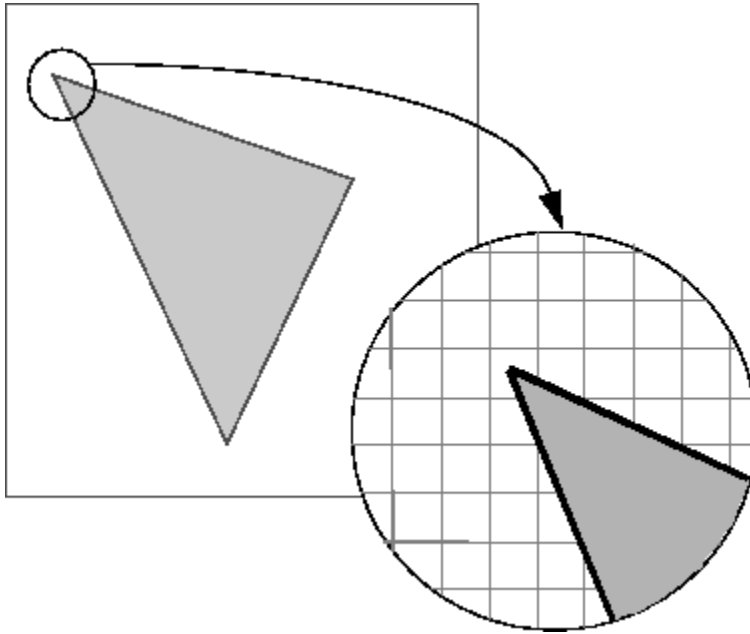
```

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

```

## 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.

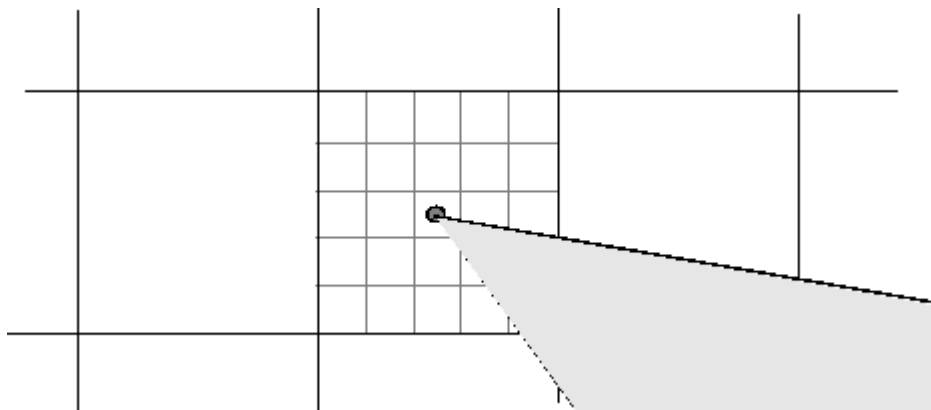


**Pixels on the Edge of an ROI Are Only Partially Covered by Border**

To determine which pixels are in the region, `poly2mask` uses the following algorithm:

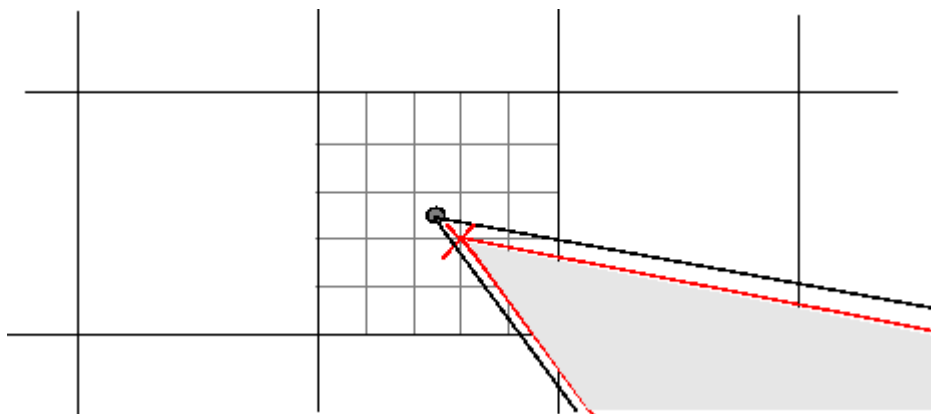
- 1** Divide each pixel into a 5-by-5 subpixel grid.

The figure shows the pixel that contains the vertex of the ROI shown previously with this 5-by-5 subpixel grid.



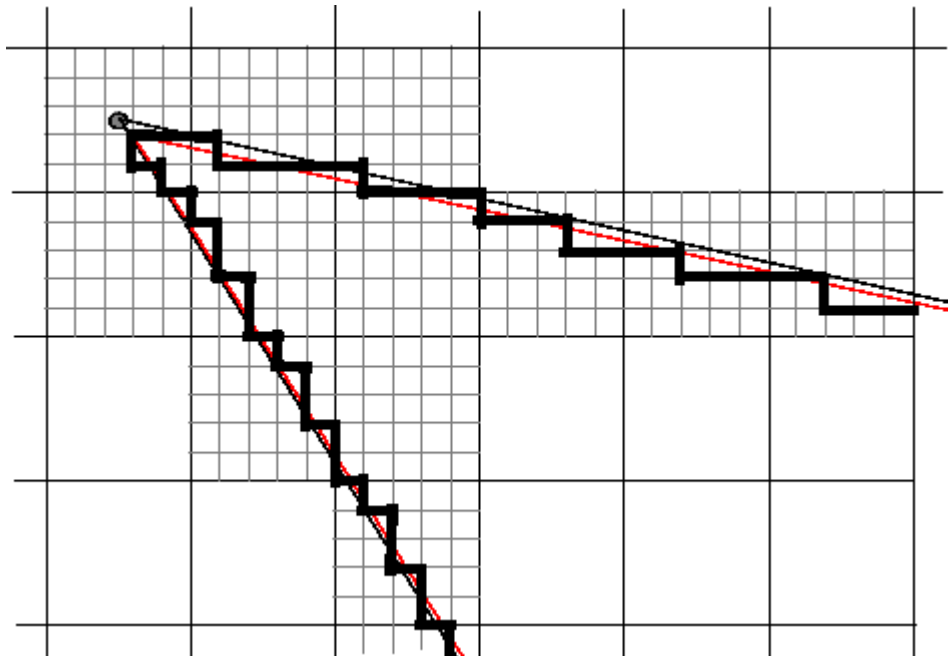
- 2** Adjust the position of the vertices.

`poly2mask` moves each vertex of the polygon to the nearest intersection of the subpixel grid. Note how `poly2mask` rounds `x` and `y` coordinates to the nearest subpixel grid corner. This creates a second, modified polygon. The figure shows the modified vertex with a red "X".



- 3** Draw a path between adjusted vertices.

`poly2mask` forms a path from each adjusted vertex to the next, following the edges of the subpixel grid. The figure shows a portion of this modified polygon by the thick dark lines.

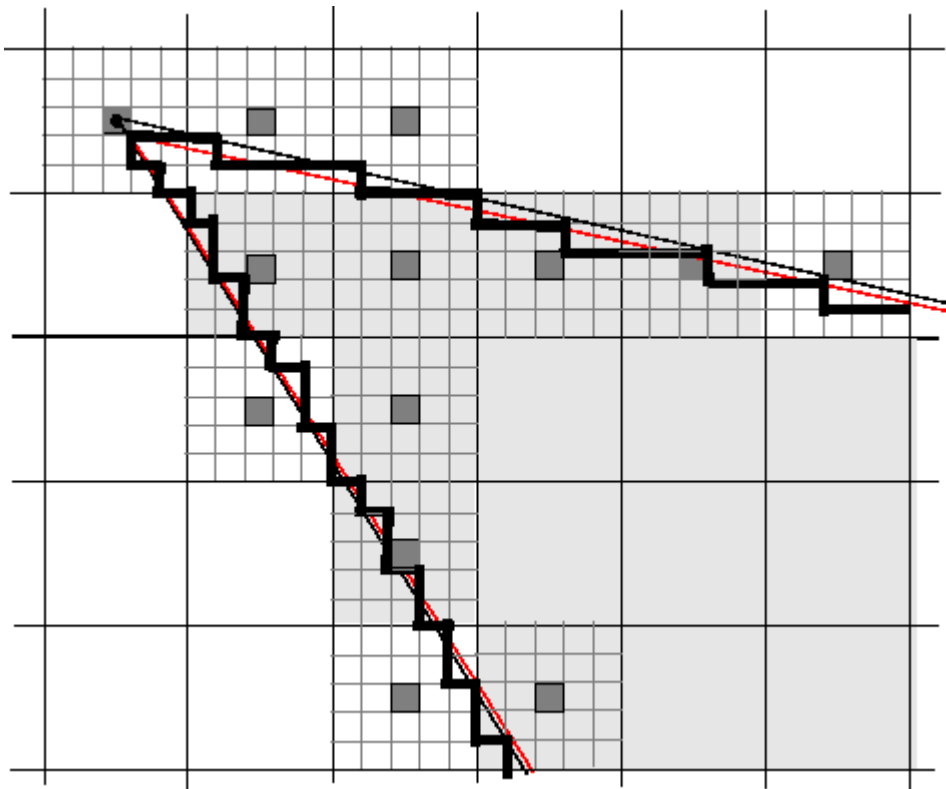


- 4 Determine which border pixels are inside the polygon.

`poly2mask` uses the following rule to determine which border pixels are inside the polygon: if the pixel's central subpixel is inside the boundaries defined by the path between adjusted vertices, then 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

`roifilt2` | `roipoly`

Introduced before R2006a

# images.geotrans.PolynomialTransformation2D

2-D polynomial geometric transformation

## Description

A `PolynomialTransformation2D` object encapsulates a 2-D polynomial geometric transformation.

## Creation

You can create a `PolynomialTransformation2D` object using the following methods:

- The `fitgeotrans` function, which estimates a geometric transformation that maps pairs of control points between two images.
- The `images.geotrans.PolynomialTransformation2D` function described here. This function creates a `PolynomialTransformation2D` object using coordinates of fixed points and moving points, or the known polynomial coefficients for the forward and inverse transformation.

## Syntax

```
tform = images.geotrans.PolynomialTransformation2D(movingPoints,  
fixedPoints,degree)  
tform = images.geotrans.PolynomialTransformation2D(a,b)
```

## Description

`tform = images.geotrans.PolynomialTransformation2D(movingPoints, fixedPoints, degree)` creates a `PolynomialTransformation2D` object and sets the `Degree` property. The function estimates the polynomial coefficients A and B from matrices `movingPoints` and `fixedPoints` that define matched control points in the moving and fixed images, respectively.

`tform = images.geotrans.PolynomialTransformation2D(a,b)` creates a `PolynomialTransformation2D` object and sets the `A` and `B` properties.

## 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

**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 — Dimensionality of the geometric transformation

2

Dimensionality of the geometric transformation for both input and output points, specified as the value 2.

## Object Functions

outputLimits Find output spatial limits given input spatial limits  
transformPointsInverse Apply inverse geometric transformation

## 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 the `PolynomialTransformation2D` transformation at the control points.

```
movingPointsEstimated = transformPointsInverse(tformPolynomial, fixedPoints);  
errorInFit = hypot(movingPointsEstimated(:,1)-movingPoints(:,1), ...  
                  movingPointsEstimated(:,2)-movingPoints(:,2))
```

## Definitions

### ***U* and *V***

*U* and *V* are the *x*- and *y*-coordinates of control points in the original coordinate system. This is the same coordinate system as obtained by performing a forward transformation followed by its inverse transformation.

### ***X* and *Y***

*X* and *Y* are the *x*- and *y*-coordinates of control points in the forward transformed coordinate system.

## See Also

### **Functions**

`cpselect` | `fitgeotrans` | `imwarp`

### **Objects**

`LocalWeightedMeanTransformation2D` | `PiecewiseLinearTransformation2D` | `affine2d` | `projective2d`

**Introduced in R2013b**

## projective2d

2-D projective geometric transformation

### Description

A `projective2d` object encapsulates a 2-D projective geometric transformation.

### Creation

A `projective2d` object encapsulates a 2-D projective geometric transformation.

You can create a `projective2d` object using the following methods:

- `fitgeotrans` — Estimates a geometric transformation that maps pairs of control points between two images
- The `projective2d` function described here

### Syntax

```
tform = projective2d  
tform = projective2d(A)
```

### Description

`tform = projective2d` creates an `affine2d` object with default property settings that correspond to the identity transformation.

`tform = projective2d(A)` sets the property `T` with a valid projective transformation defined by nonsingular matrix `A`.

## Properties

### T — Forward 2-D projective transformation

nonsingular 3-by-3 numeric matrix

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

The matrix T uses the convention:

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

where T has the form:

$$\begin{bmatrix} a & b & c; \\ d & e & f; \\ g & h & i; \end{bmatrix}$$

The default of T is the identity transformation.

Data Types: double | single

### Dimensionality — Dimensionality of the geometric transformation

2

Dimensionality of the geometric transformation for both input and output points, specified as the value 2.

## Object Functions

invert	Invert geometric transformation
outputLimits	Find output spatial limits given input spatial limits
transformPointsForward	Apply forward geometric transformation
transformPointsInverse	Apply inverse geometric transformation

## 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.

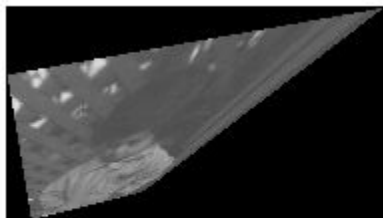
```
I = imread('pout.tif');
```

Create a geometric transformation object. This example combines rotation and tilt into a transformation matrix, `tm`. Use this transformation matrix to construct a `projective2d` 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);
```



## 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

### Functions

`fitgeotrans` | `imwarp`

### Objects

`LocalWeightedMeanTransformation2D` | `PiecewiseLinearTransformation2D` | `PolynomialTransformation2D` | `affine2d` | `geometricTransform2d`

### Topics

“Register an Aerial Photograph to a Digital Orthophoto”

“2-D and 3-D Geometric Transformation Process Overview”

“Matrix Representation of Geometric Transformations”

**Introduced in R2013a**

## psf2otf

Convert point-spread function to optical transfer function

### Syntax

```
OTF = psf2otf(PSF)
OTF = psf2otf(PSF,sz)
```

### 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.

`OTF = psf2otf(PSF,sz)` specifies the size, `sz`, of the optical transfer function.

### Examples

#### Convert PSF to OTF

Create a point-spread function (PSF).

```
PSF = fspecial('gaussian',13,1);
```

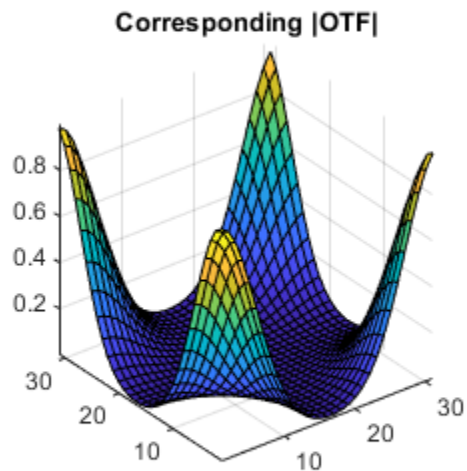
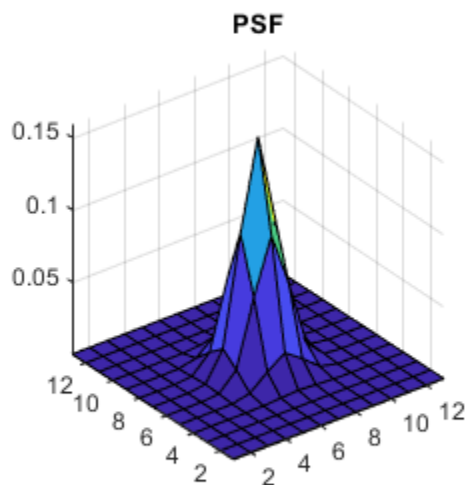
Convert the PSF to an Optical Transfer Function (OTF).

```
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
```



## Input Arguments

### PSF — Point-spread function

numeric array

Point-spread function, specified as a numeric array of any dimension.

Example: `PSF = fspecial('gaussian',13,1);`

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

Complex Number Support: Yes

## **sz — Size of optical transfer function**

vector of positive integers

Size of the output optical transfer function OTF, specified as a vector of positive integers. The size of OTF must not exceed the size of PSF in any dimension. By default, OTF is the same size as PSF.

Data Types: `double`

## **Output Arguments**

### **OTF — Optical transfer function**

numeric array

Optical transfer function, returned as a numeric array of size `sz`.

Data Types: `double`

Complex Number Support: Yes

## **Tips**

- To ensure that OTF is not altered because of PSF off-centering, `psf2otf` postpads PSF (down or to the right) with 0s to match dimensions specified in `sz`. Then `psf2otf` circularly shifts the values of PSF up (or to the left) until the central pixel reaches (1,1) position.
- This function is used in image convolution and deconvolution when the operations involve the FFT.

## **See Also**

`circshift` | `fft` | `ifft` | `otf2psf` | `padarray`

## **Topics**

“Create Your Own Deblurring Functions”

**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`

## **Algorithms**

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**

`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

`S = qtdecomp(I)` performs a quadtree decomposition on the grayscale image `I` and returns the quadtree structure in the sparse matrix `S`. 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`.

`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.

`S = qtdecomp(I,fun)` uses the function `fun` to determine whether to split a block.

### 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.

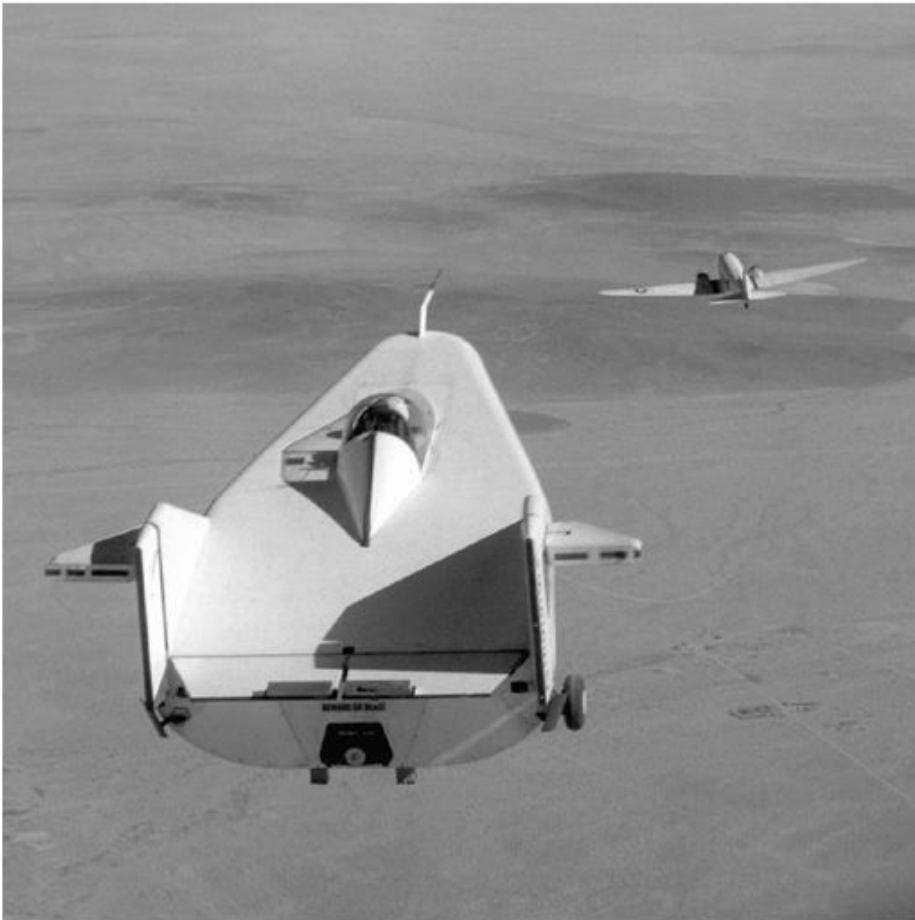
```
I = imread('liftingbody.png');
```

Perform the quadtree decomposition and display the block representation in a figure.

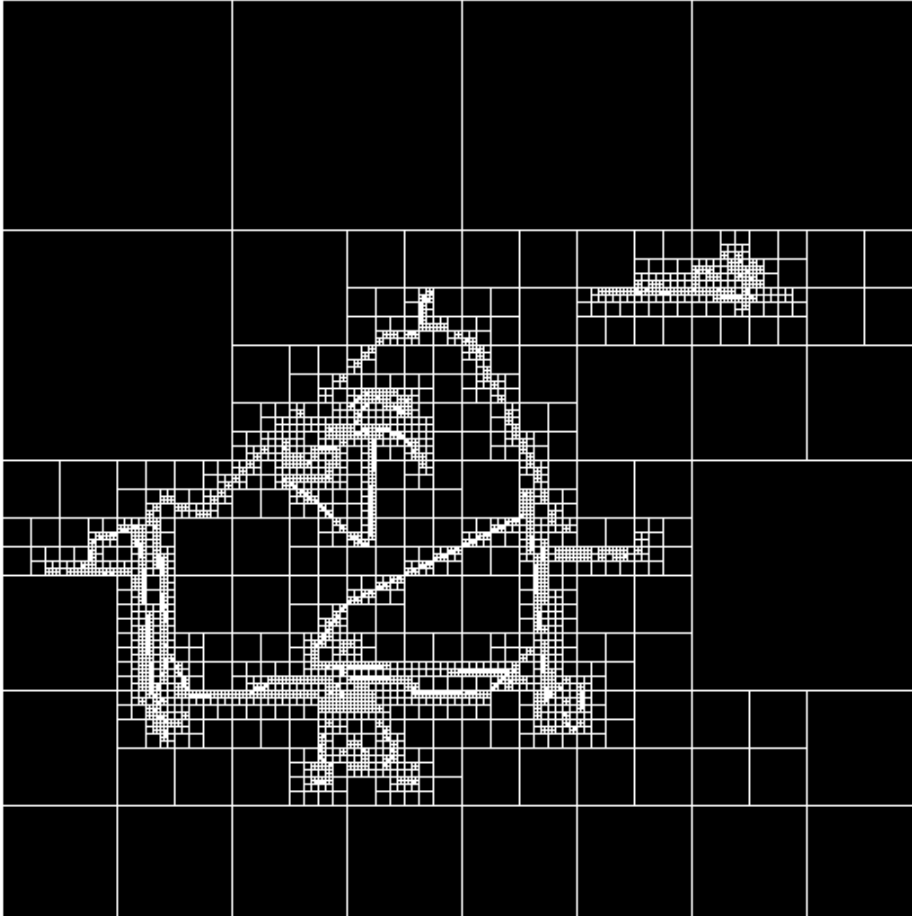
```
S = qtdecomp(I, .27);
blocks = repmat(uint8(0), size(S));

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, [])
```



## Input Arguments

### **I — Grayscale image**

*m*-by-*n* numeric matrix

Grayscale image, specified as an *m*-by-*n* numeric matrix. If the syntax includes a function handle, `fun`, then the image can be of any class supported by the function.

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

### **threshold — Threshold of block homogeneity**

scalar in the range [0, 1]

Threshold of block homogeneity, specified as a scalar in the range [0, 1].

- If `I` is of class `uint8`, then `qtdecomp` multiplies the value of `threshold` by 255 to determine the actual threshold to use.
- If `I` is of class `uint8`, then `qtdecomp` multiplies the value of `threshold` by 65535 to determine the actual threshold to use.

### **mindim — Minimum block dimension**

positive integer

Minimum block size, specified as a positive integer. `mindim` must be a factor of the image size.

### **maxdim — Maximum block dimension**

positive integer

Maximum block size, specified as a positive integer. `maxdim`/`mindim` must be a power of 2.

### **fun — Function handle**

handle

Function handle, specified as a handle. The function must accept as input all *m*-by-*m* blocks stacked into an *m*-by-*m*-by-*k* array, where *k* is the number of blocks. The function must return 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, then the third *m*-by-*m* block should not be split.

For more information about function handles, see “Create Function Handle” (MATLAB).

## Output Arguments

### S — Quadtree structure

sparse matrix

Quadtree structure, returned as a sparse matrix. 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  $S(k, m)$ .

Data Types: double

## 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, `fun`, the function must return 0 at the point when the block cannot be divided further.

## Algorithms

The `qtdecomp` function 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.

## See Also

`qtgetblk` | `qtsetblk`

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

### Description

`R = radon(I)` returns the Radon transform `R` of 2-D grayscale image `I` for angles in the range `[0, 179]` degrees. The Radon transform is the projection of the image intensity along a radial line oriented at a specific angle.

You optionally can compute the Radon transform using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`R = radon(I, theta)` returns the Radon transform for the angles specified by `theta`.

`[R, xp] = radon( ___ )` returns a vector `xp` containing the radial coordinates corresponding to each row of the image.

### Examples

#### Calculate Radon Transform and Display Plot

Make the axes scale visible for this image.

```
iptsetpref('ImshowAxesVisible','on')
```

Create a sample image.

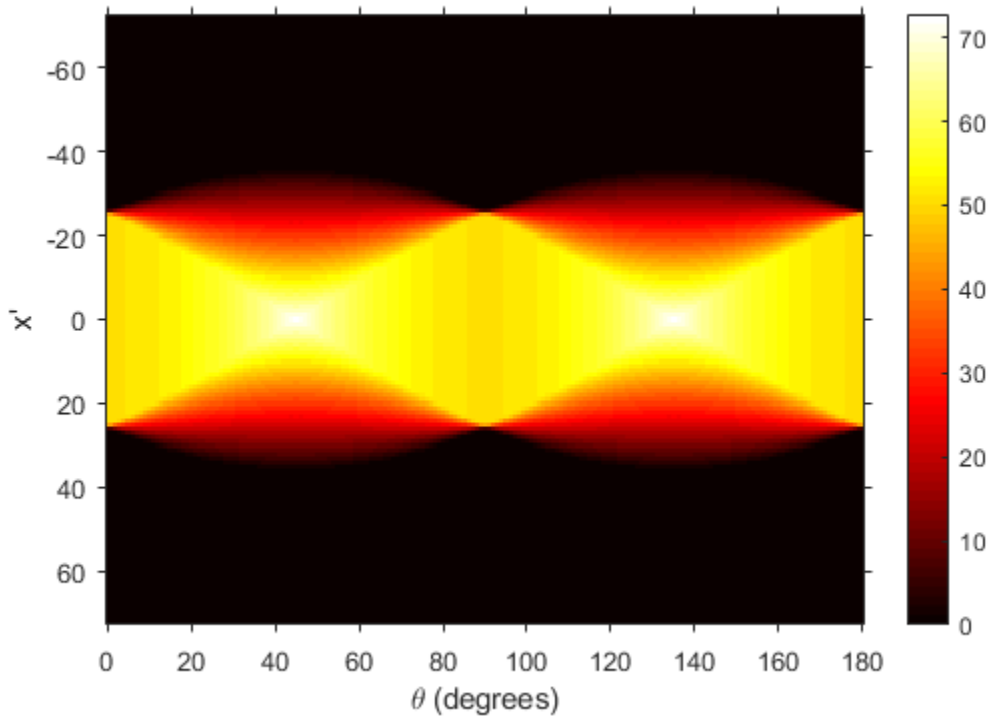
```
I = zeros(100,100);
I(25:75, 25:75) = 1;
```

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')
```

## Input Arguments

### **I — Image**

2-D grayscale image

Image, specified as a 2-D grayscale image.

To calculate the Radon transform using a GPU, specify **I** as a `gpuArray` that contains a 2-D grayscale image of data type `single`, `double`, `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`.

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

### **theta — Projection angles**

0:179 (default) | numeric scalar | numeric vector | `gpuArray`

Projection angles (in degrees), specified as a numeric scalar or numeric vector.

If you calculate the Radon transform using a GPU, then **theta** can optionally be a `gpuArray` containing a numeric scalar or vector.

Data Types: `double`

## Output Arguments

### **R — Radon transform**

numeric column vector | numeric matrix | `gpuArray`

Radon transform of image **I**, returned as one of the following.

- If **theta** is a scalar, then **R** is a numeric column vector containing the Radon transform for **theta** degrees.
- If **theta** is a vector, then **R** is a matrix in which each column is the Radon transform for one of the angles in **theta**.

If the Radon transform is computed using a GPU, then **R** is returned as a `gpuArray` containing a numeric column vector or numeric matrix.

**xp — Radial coordinates**

numeric vector | gpuArray

Radial coordinates corresponding to each row of `R`, returned as a numeric vector. The radial coordinates 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)
```

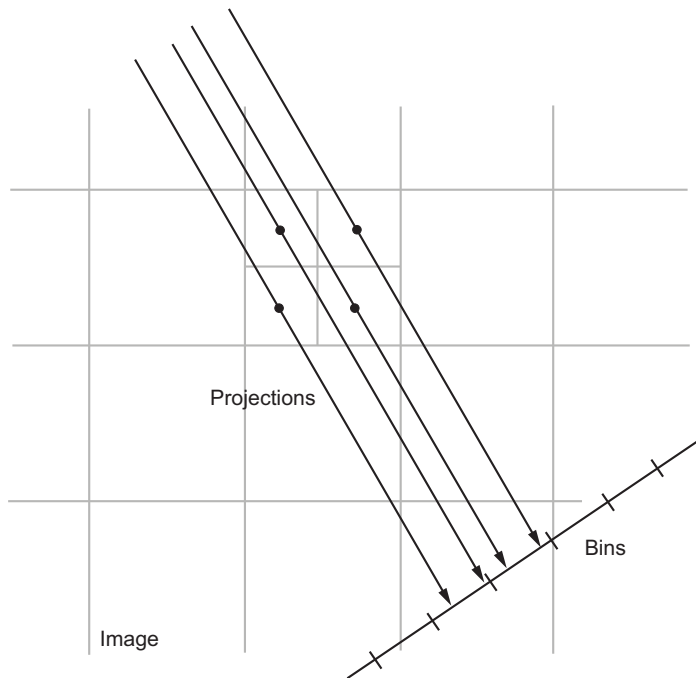
For example, in a 20-by-30 image, the center pixel is (10,15).

If the Radon transform is computed using a GPU, then `xp` is returned as a `gpuArray` containing a numeric vector.

## 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

- [1] Bracewell, Ronald N., *Two-Dimensional Imaging*, Englewood Cliffs, NJ, Prentice Hall, 1995, pp. 505-537.
- [2] Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, pp. 42-45.



## Extended Capabilities

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

### See Also

fan2para | fanbeam | ifanbeam | iradon | para2fan | phantom

**Introduced before R2006a**

## randomPatchExtractionDatastore

Datastore for extracting random patches from images or pixel label images

### Description

A `randomPatchExtractionDatastore` extracts corresponding randomly-positioned patches from two datastores. The input datastores can be two image datastores that contain the network inputs and desired network responses for training deep neural networks, or ground truth images and pixel label data for training semantic segmentation networks.

This object requires that you have Deep Learning Toolbox.

---

**Note** When you use a `randomPatchExtractionDatastore` as a source of training data, the datastore extracts multiple random patches from each image for each epoch, so that each epoch uses a slightly different data set. The actual number of training patches at each epoch is the number of training images multiplied by `PatchesPerImage`. The image patches are not stored in memory.

---

### Creation

### Syntax

```
patchds = randomPatchExtractionDatastore(imds1,imds2,patchSize)
patchds = randomPatchExtractionDatastore(imds1,pxds,patchSize)
patchds = randomPatchExtractionDatastore( ___,Name,Value)
```

### Description

```
patchds = randomPatchExtractionDatastore(imds1,imds2,patchSize)
```

creates a datastore that extracts randomly-positioned patches of size `patchSize` from images in image datastore `imds1` and corresponding patches from images in image datastore `imds2`.

`patchds = randomPatchExtractionDatastore(imds1,pxds,patchSize)` creates a datastore that extracts randomly-positioned patches of size `patchSize` from ground truth images in image datastore `imds1` and corresponding patches from pixel label images in pixel label datastore `pxds`.

This syntax requires Computer Vision System Toolbox.

`patchds = randomPatchExtractionDatastore( ___,Name,Value)` uses name-value pairs to set the `PatchesPerImage`, `DataAugmentation`, and `DispatchInBackground` properties. You can specify multiple name-value pairs. Enclose each argument or property name in quotes.

For example,  
`randomPatchExtractionDatastore(imds1,imds2,50,'PatchesPerImage',40)` creates a datastore that randomly generates 40 patches of size 50-by-50 pixels from each image in image datastores `imds1` and `imds2`.

## Input Arguments

### **imds1** — Input image data

ImageDatastore object

Input image data containing training input to the network, specified as an ImageDatastore object.

ImageDatastore allows batch-reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, then prefetching does not happen.

### **imds2** — Response image data

ImageDatastore object

Response image data representing the desired network responses, specified as an ImageDatastore object.

### **pxds** — Response pixel label data

pixelLabelDatastore object

Response pixel label data representing the desired network responses, specified as a pixelLabelDatastore object.

### **patchSize** — Patch size

positive integer | 2-element vector of positive integers

Patch size, specified as one of the following.

- A positive integer that specifies the number of rows and columns of square patches.
- A 2-element vector of positive integers, of the form  $[r\ c]$ . The first element specifies the number of rows in the patch, and the second element specifies the number of columns.

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

## Properties

### **PatchesPerImage — Number of random patches per image**

128 (default) | positive integer

Number of random patches per image, specified as a positive integer.

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

### **DataAugmentation — Preprocessing applied to input images**

'none' (default) | `imageDataAugmenter` object

Preprocessing applied to input images, specified as an `imageDataAugmenter` object or 'none'. When `DataAugmentation` is 'none', no preprocessing is applied to input images.

Augment data with random transformations, such as resizing, rotation, and reflection, to help prevent the network from overfitting and memorizing the exact details of the training data. The `randomPatchExtractionDatastore` applies the same random transformation to both patches in each pair.

### **DispatchInBackground — Dispatch observations in background**

false (default) | true

Dispatch observations in the background during training, prediction, or classification, specified as `false` or `true`. To use background dispatching, you must have Parallel Computing Toolbox.

### **MiniBatchSize — Number of observations in each batch**

positive integer

This property is read-only.

Number of observations that are returned in each batch. For training, prediction, and classification, the `MiniBatchSize` property is set to the mini-batch size defined in `trainingOptions`.

### **NumObservations** — Total number of observations in the datastore

positive integer

This property is read-only.

Total number of observations in the `randomPatchExtractionDatastore`. The number of observations is the length of one training epoch.

## **Object Functions**

<code>hasdata</code>	Determine if data is available to read
<code>partitionByIndex</code>	Partition <code>randomPatchExtractionDatastore</code> according to indices
<code>preview</code>	Subset of data in datastore
<code>read</code>	Read data from <code>randomPatchExtractionDatastore</code>
<code>readall</code>	Read all data in datastore
<code>readByIndex</code>	Read data specified by index from <code>randomPatchExtractionDatastore</code>
<code>reset</code>	Reset datastore to initial state
<code>shuffle</code>	Shuffle data in <code>randomPatchExtractionDatastore</code>

## **Examples**

### **Create Random Patch Extraction Datastore**

Create an image datastore containing training images. The datastore in this example contains JPEG color images.

```
imageDir = fullfile(toolboxdir('images'),'imdata');  
imds1 = imageDatastore(imageDir,'FileExtensions','.jpg');
```

Create a second image datastore containing desired network responses. This image datastore includes a custom read function called `myreadfcn` (defined at the end of the example) that smooths images using a Gaussian blur.

```
imds2 = imageDatastore(imageDir,'FileExtensions','.jpg','ReadFcn',@myreadfcn);
```

Create an `imageDataAugmenter` that rotates images by random angles in the range [0, 90] degrees and randomly reflects image data horizontally.

```
augmenter = imageDataAugmenter('RandRotation',[0 90],'RandXReflection',true)
```

```
augmenter =  
  imageDataAugmenter with properties:
```

```
    FillValue: 0  
    RandXReflection: 1  
    RandYReflection: 0  
    RandRotation: [0 90]  
        RandScale: [1 1]  
        RandXScale: [1 1]  
        RandYScale: [1 1]  
        RandXShear: [0 0]  
        RandYShear: [0 0]  
    RandXTranslation: [0 0]  
    RandYTranslation: [0 0]
```

Create a `randomPatchExtractionDatastore` object that extracts random patches of size [100 100] from the unprocessed training images and corresponding smoothed response images. Specify the augmentation options by setting the `DataAugmentation` property.

```
patchds = randomPatchExtractionDatastore(imds1,imds2,[100 100], ...  
    'DataAugmentation',augmenter)
```

```
patchds =  
  randomPatchExtractionDatastore with properties:
```

```
    PatchesPerImage: 128  
        PatchSize: [100 100]  
    DataAugmentation: [1x1 imageDataAugmenter]  
    MiniBatchSize: 128  
    NumObservations: 4736  
    DispatchInBackground: 0
```

Preview a set of augmented image patches and the corresponding smoothed image patches.

```
minibatch = preview(patchds);  
inputs = minibatch.InputImage;
```

```
responses = minibatch.ResponseImage;  
test = cat(2,inputs,responses);  
montage(test','Size',[8 2])  
title('Inputs (Left) and Respones (Right)')
```

**Inputs (Left) and Responses (Right)**





This example defines the `myreadfcn` function that reads a file from disk then smooths the image by applying a Gaussian blur. The function returns the smoothed image.

```
function J = myreadfcn(filename)
    I = imread(filename);
    J = imgaussfilt(I,2);
end
```

## Train Semantic Segmentation Network Using Random Patch Extraction Datastore

Create an image datastore containing training images.

```
dataDir = fullfile(toolboxdir('vision'),'visiondata','triangleImages');
imageDir = fullfile(dataDir,'trainingImages');
imds = imageDatastore(imageDir);
```

Define class names and their associated label IDs. Then, create a pixel label datastore containing the ground truth pixel labels for the training images.

```
classNames = ["triangle","background"];
labelIDs = [255 0];
labelDir = fullfile(dataDir,'trainingLabels');
pxds = pixelLabelDatastore(labelDir,classNames,labelIDs);
```

Create a random patch extraction datastore to extract random patches of size 32-by-32 pixels from the images and corresponding pixel labels. Set the optional `PatchesPerImage` property to extract 512 random patches from each image and pixel label pair.

```
patchds = randomPatchExtractionDatastore(imds,pxds,32, ...
    'PatchesPerImage',512);
```

Create a network for semantic segmentation.

```
layers = [
    imageInputLayer([32 32 1])
    convolution2dLayer(3,64,'Padding',1)
    reluLayer()
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,64,'Padding',1)
    reluLayer()
    transposedConv2dLayer(4,64,'Stride',2,'Cropping',1)
```

```
convolution2dLayer(1,2)
softmaxLayer()
pixelClassificationLayer()
]
```

```
layers =
```

```
10x1 Layer array with layers:
```

1	''	Image Input	32x32x1 images with 'zerocenter' normalization
2	''	Convolution	64 3x3 convolutions with stride [1 1] and padding
3	''	ReLU	ReLU
4	''	Max Pooling	2x2 max pooling with stride [2 2] and padding
5	''	Convolution	64 3x3 convolutions with stride [1 1] and padding
6	''	ReLU	ReLU
7	''	Transposed Convolution	64 4x4 transposed convolutions with stride [1 1] and padding
8	''	Convolution	2 1x1 convolutions with stride [1 1] and padding
9	''	Softmax	softmax
10	''	Pixel Classification Layer	Cross-entropy loss

Set up training options. To reduce training time, set `MaxEpochs` to 5.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',1e-3, ...
    'MaxEpochs',5, ...
    'Verbose',false);
```

Train the network.

```
net = trainNetwork(patchds, layers, options);
```

## Tips

- The `randomPatchExtractionDatastore` expects that the output from the `read` operation on the input datastores return arrays of the same size.
- The `randomPatchExtractionDatastore` always reads one image at a time from each input datastore. It accomplishes this by setting the `ReadSize` property of the input datastores to 1.
- If the input datastore is an `ImageDatastore`, then the values in its `Labels` property are ignored by the `randomPatchExtractionDatastore`.
- To visualize the data in a `randomPatchExtractionDatastore`, you can use the `preview` function, which returns a subset of data in a table. Visualize all of the

patches in the same figure by using the `montage` function. For example, this code displays a preview of image patches from a `randomPatchExtractionDatastore` called `patchds`.

```
minibatch = preview(patchds);  
montage(minibatch.InputImage)
```

## See Also

[augmentedImageDatastore](#) | [imageDataAugmenter](#) | [imageDatastore](#) | [pixelLabelDatastore](#) | [pixelLabelImageDatastore](#) | [trainNetwork](#)

## Topics

[“Single Image Super-Resolution Using Deep Learning”](#)

[“Image Processing Operator Approximation Using Deep Learning”](#)

[“Deep Learning in MATLAB”](#) (Deep Learning Toolbox)

[“Preprocess Images for Deep Learning”](#) (Deep Learning Toolbox)

## Introduced in R2018b

## partitionByIndex

Partition `randomPatchExtractionDatastore` according to indices

### Syntax

```
patchds2 = partitionByIndex(patchds, ind)
```

### Description

`patchds2 = partitionByIndex(patchds, ind)` partitions a subset of observations in a random patch extraction datastore, `patchds`, into a new datastore, `patchds2`. The desired observations are specified by indices, `ind`.

### Input Arguments

**patchds** — Random patch extraction datastore  
`randomPatchExtractionDatastore`

Random patch extraction datastore, specified as a `randomPatchExtractionDatastore` object.

**ind** — Indices  
vector of positive integers

Indices of observations, specified as a vector of positive integers.

### Output Arguments

**patchds2** — Output datastore  
`randomPatchExtractionDatastore` object

Output datastore, returned as a `randomPatchExtractionDatastore` object containing a subset of files from `patchds`.

## See Also

randomPatchExtractionDatastore | read | readByIndex | readall

**Introduced in R2018b**

## read

Read data from `randomPatchExtractionDatastore`

## Syntax

```
data = read(patchds)
[data,info] = read(patchds)
```

## Description

`data = read(patchds)` returns a batch of data from a random patch extraction datastore, `patchds`. Subsequent calls to the `read` function continue reading from the endpoint of the previous call.

`[data,info] = read(patchds)` also returns information about the extracted data, including metadata, in `info`.

## Input Arguments

### **patchds** — Random patch extraction datastore

`randomPatchExtractionDatastore`

Random patch extraction datastore, specified as a `randomPatchExtractionDatastore` object. The datastore specifies a `MiniBatchSize` number of observations in each batch, and a `numObservations` total number of observations.

## Output Arguments

### **data** — Output data

table

Output data, returned as a table with `MiniBatchSize` number of rows. The first variable is `InputImage`, which contains input image patches. If the network responses are images

in an image datastore, then the second variable is `ResponseImage`. If the network responses are pixel label images in a pixel label datastore, then the second variable is `ResponsePixelLabelImage`.

For the last batch of data in the datastore `patchds`, if `numObservations` is not cleanly divisible by `MiniBatchSize`, then `read` returns a partial batch containing all the remaining observations in the datastore.

### info — Information about read data

structure array

Information about read data, returned as a structure array. The structure array can contain the following fields.

Field Name	Description
<code>RandomPatchRectangles</code>	<code>MiniBatchSize</code> -by-4 numeric matrix. Each row specifies the size and position of the patch in the format <code>[xywidthheight]</code> . The elements define the x- and y-coordinate of the top left corner, and the width and height of the patch.
<code>ImageIndices</code>	<code>MiniBatchSize</code> -by-1 numeric vector that specifies the indices of the read images in the input datastores.
<code>InputImageFilename</code>	<code>MiniBatchSize</code> -by-1 cell array that specifies the fully resolved path containing the path string, name of the file, and file extension of each input image.
<code>ResponseImageFilename</code>	<code>MiniBatchSize</code> -by-1 cell array that specifies the fully resolved path containing the path string, name of the file, and file extension of each response image or pixel label image.

## See Also

`matlab.io.datastore.read` | `randomPatchExtractionDatastore` | `readByIndex` | `readall`

**Introduced in R2018b**

## readByIndex

Read data specified by index from `randomPatchExtractionDatastore`

### Syntax

```
data = readByIndex(patchds,ind)
[data,info] = readByIndex(patchds,ind)
```

### Description

`data = readByIndex(patchds,ind)` returns a subset of observations from a random patch extraction datastore, `patchds`. The desired observations are specified by indices, `ind`.

`[data,info] = readByIndex(patchds,ind)` also returns information about the observations, including metadata, in `info`.

### Input Arguments

**patchds** — Random patch extraction datastore  
`randomPatchExtractionDatastore`

Random patch extraction datastore, specified as a `randomPatchExtractionDatastore` object.

**ind** — Indices  
vector of positive integers

Indices of observations, specified as a vector of positive integers.

### Output Arguments

**data** — Observations from datastore  
table



Observations from the datastore, returned as a table with `length(ind)` number of rows.

### info — Information about read data

structure array

Information about read data, returned as a structure array. The structure array can contain the following fields.

Field Name	Description
RandomPatchRectangles	<code>length(ind)</code> -by-4 numeric matrix. Each row specifies the size and position of the patch in the format <code>[xywidthheight]</code> . The elements define the x- and y-coordinate of the top left corner, and the width and height of the patch.
ImageIndices	<code>length(ind)</code> -by-1 numeric vector that specifies the indices of the read images in the input datastores.
InputImageFilename	<code>length(ind)</code> -by-1 cell array that specifies the fully resolved path containing the path string, name of the file, and file extension of each input image.
ResponseImageFilename	<code>length(ind)</code> -by-1 cell array that specifies the fully resolved path containing the path string, name of the file, and file extension of each response image or pixel label image.

## See Also

`partitionByIndex` | `randomPatchExtractionDatastore` | `read` | `readall`

**Introduced in R2018b**

## shuffle

Shuffle data in `randomPatchExtractionDatastore`

### Syntax

```
patchds2 = shuffle(patchds)
```

### Description

`patchds2 = shuffle(patchds)` returns a `randomPatchExtractionDatastore` object containing a random ordering of the data from random patch extraction datastore `patchds`.

### Input Arguments

**patchds** — Random patch extraction datastore

`randomPatchExtractionDatastore`

Random patch extraction datastore, specified as a `randomPatchExtractionDatastore` object.

### Output Arguments

**patchds2** — Output datastore

`randomPatchExtractionDatastore` object

Output datastore, returned as a `randomPatchExtractionDatastore` object containing randomly ordered files from `patchds`.

### See Also

`randomPatchExtractionDatastore` | `read` | `readByIndex` | `readall`

**Introduced in R2018b**

## 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`.

`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.

### Examples

#### Identify Objects in 2-D Image

Read an image into the workspace.

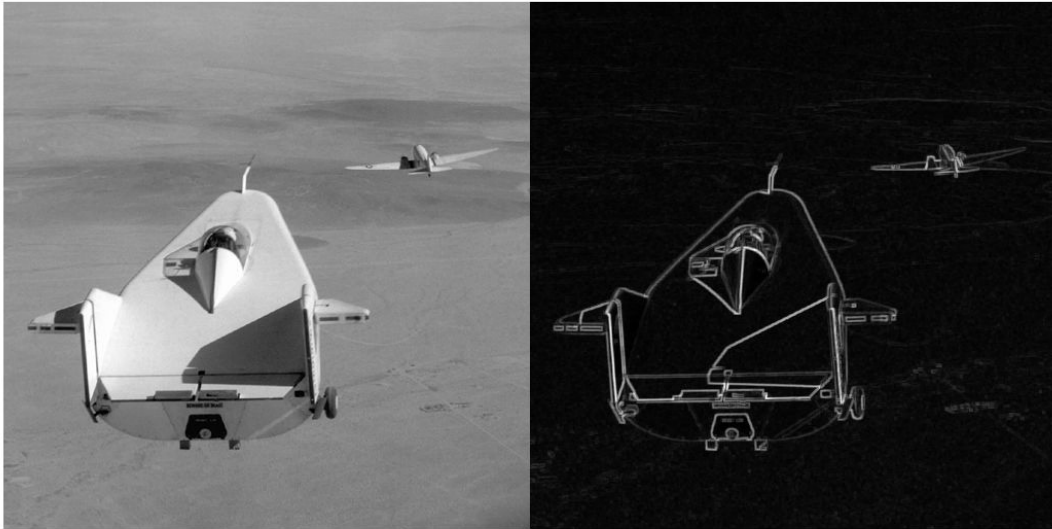
```
I = imread('liftingbody.png');
```

Filter the image. The `rangefilt` function returns an array 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.

```
J = rangefilt(I);
```

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

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



### Quantify Land Cover Changes in RGB Image

Read image into the workspace.

```
RGB = imread('autumn.tif');
```

Convert the RGB image into a L\*a\*b\* image.

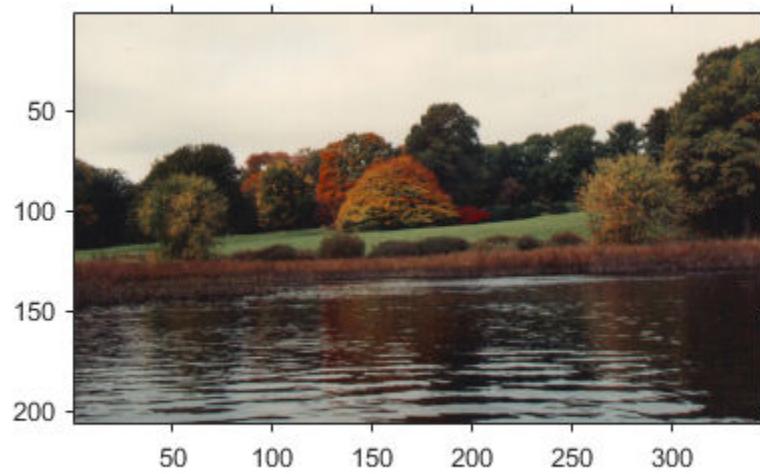
```
LAB = rgb2lab(RGB);
```

Perform the range filtering on the LAB image.

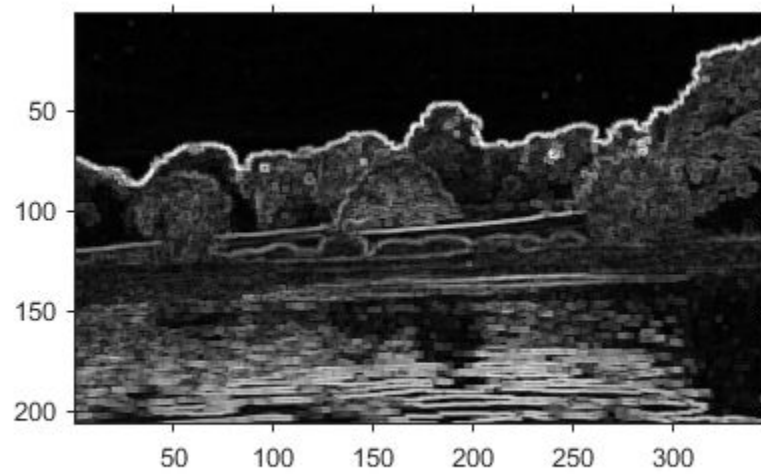
```
rLAB = rangefilt(LAB);
```

Display the images.

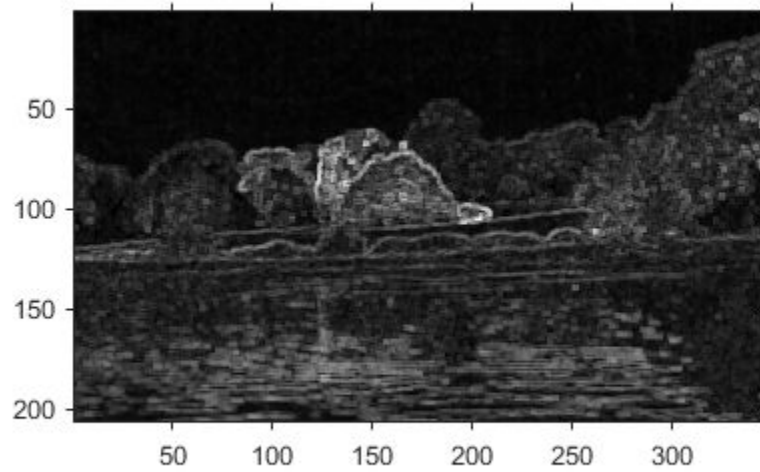
```
imshow(RGB);
```



```
figure, imshow(rLAB(:,:,1),[]);
```

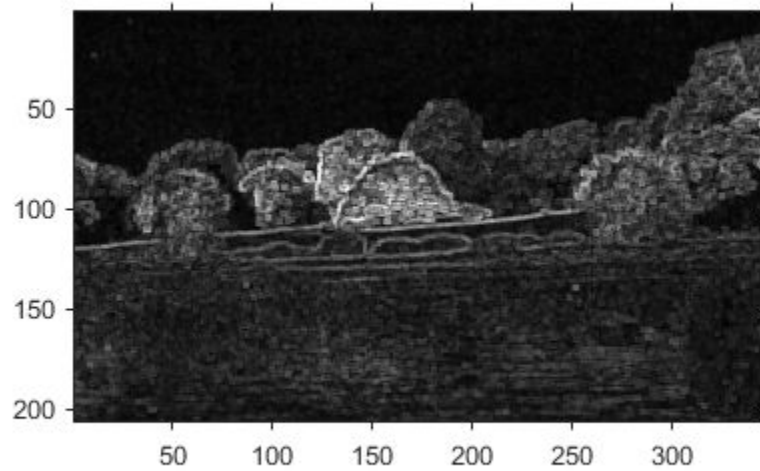


```
figure, imshow(rLAB(:,:,2),[]);
```



```
figure, imshow(rLAB(:,:,3),[]);
```

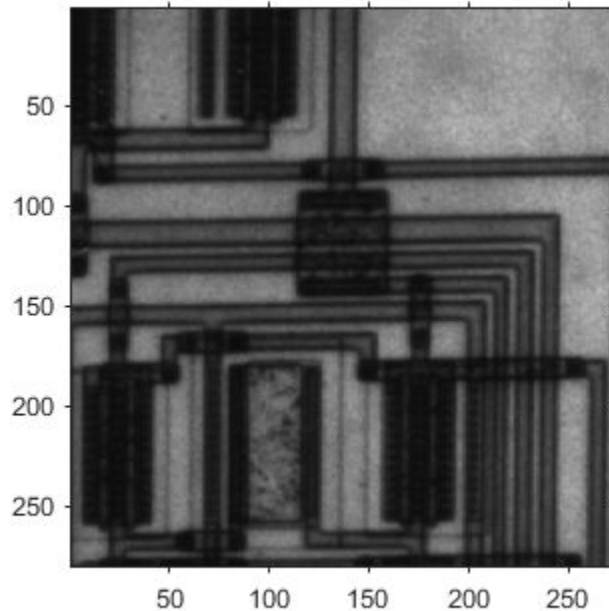




### Identify Vertical Edges Using Range Filtering

Read an image into the workspace, and display it.

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

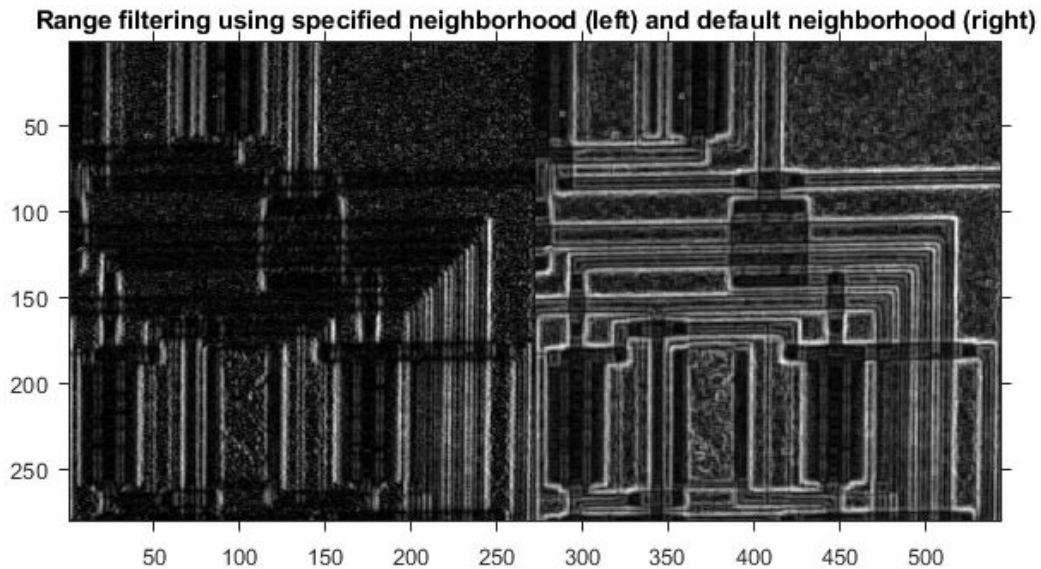


Define a neighborhood. In this example, the neighborhood returns a large value when there is a large difference between pixel values to the left and right of an input pixel. The filtering does not consider pixels above and below the input pixel. Thus, this neighborhood emphasizes vertical edges.

```
nhood = [1 1 1];
```

Perform the range filtering operation using this neighborhood. For comparison, also perform range filtering using the default 3-by-3 neighborhood. Compare the results.

```
J = rangefilt(I, nhood);  
K = rangefilt(I);  
figure  
imshowpair(J, K, 'montage');  
title('Range filtering using specified neighborhood (left) and default neighborhood (right)');
```



The result using the specified neighborhood emphasizes vertical edges, as expected. In comparison, the default filter is not sensitive to edge directionality.

## Input Arguments

### **I** — Image to be filtered

real, nonsparse, numeric array

Image to be filtered, specified as a real, nonsparse, numeric array of any dimension.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `logical`

### **nhood** — Neighborhood

`true(3)` (default) | multidimensional, logical or numeric array containing zeros and ones

Neighborhood, specified as a multidimensional, logical or numeric array containing zeros and ones. `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)`.

To specify neighborhoods of other shapes, such as a disk, use the `strel` function to create a structuring element object of the desired shape. Then, extract the neighborhood from the structuring element object's `neighborhood` property.

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

## Output Arguments

### J — Filtered image

numeric array

Filtered image, returned as a numeric array, the same size and class as the input image `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`.

## Algorithms

`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.

## See Also

### Functions

`entropyfilt` | `getnhood` | `imdilate` | `imerode` | `stdfilt`

### Objects

`offsetstrel` | `strel`

## Topics

“What Is Image Filtering in the Spatial Domain?”

**Introduced before R2006a**

## reflect

Reflect 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).

## 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: [3x3 logical]  
    Dimensionality: 2
```

Look at the neighborhood.

```
se.Neighborhood
```

```
ans = 3x3 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: [3x3 logical]
    Dimensionality: 2

```

Look at the reflected neighborhood.

```
se2.Neighborhood
```

```
ans = 3x3 logical array
```

```

0  0  0
0  0  0
1  0  0

```

## 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: [11x11 double]
    Dimensionality: 2

```

Reflect the structuring element.

```
se2 = se.reflect()
```

`se2 =`  
`offsetstrel` is a ball shaped offset structuring element with properties:

Offset: [11x11 double]  
Dimensionality: 2

## Input Arguments

### SE — Structuring elements

`strel` or `offsetstrel` object or array of objects

Structuring element, specified as a `strel` or `offsetstrel` object or array of objects. If SE is an array of structuring element objects, then `reflect` reflects each element of SE.

## Output Arguments

### SE2 — Reflected structuring elements

`strel` or `offsetstrel` object or array of objects

Reflected structuring elements, returned as a `strel` or `offsetstrel` object or array of objects. SE2 has the same size as SE.

## See Also

`translate`

**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')
```



## 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.

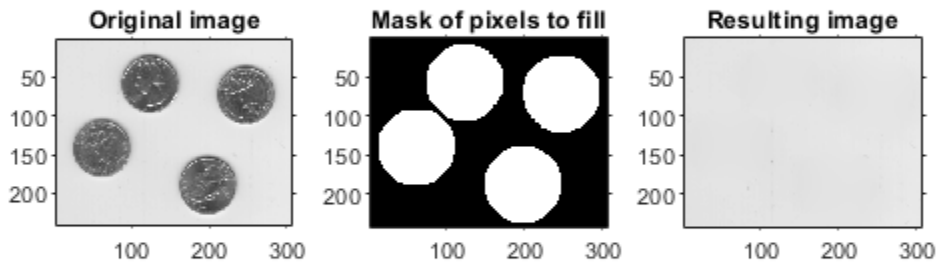
```
mask = imerode(mask,strel('disk',10));  
mask = imdilate(mask,strel('disk',20));
```

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

**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: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

## 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

`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, ____)
```

## 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 1-2516).

---

**Note** To return measurements of a 3-D volumetric image, consider using `regionprops3`. While `regionprops` can accept 3-D images, `regionprops3` calculates more statistics for 3-D images than `regionprops`.

---

For all syntaxes, if you do not specify the `properties` argument, then `regionprops` returns the 'Area', 'Centroid', and 'BoundingBox' measurements.

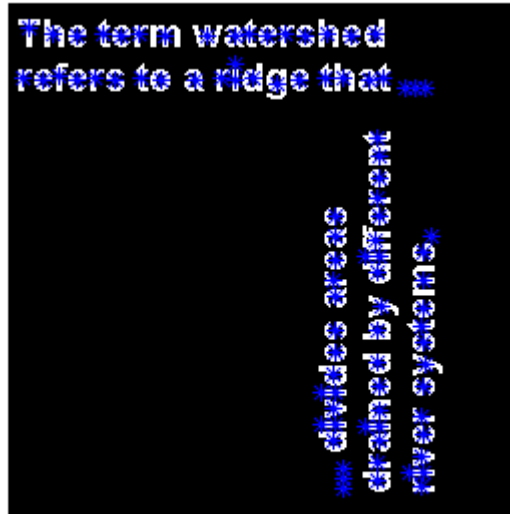
You optionally can measure properties of image regions using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”. The 'ConvexArea', 'ConvexHull', 'ConvexImage', 'EulerNumber', 'FilledArea', 'FilledImage', and 'Solidity' properties are not supported on a GPU.

`stats = regionprops(CC,properties)` measures a set of properties for each connected component (object) in `CC`, which is a structure returned by `bwconncomp`.

This syntax is not supported on a GPU.







### Calculate Centroids and Superimpose Locations on Image on 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.

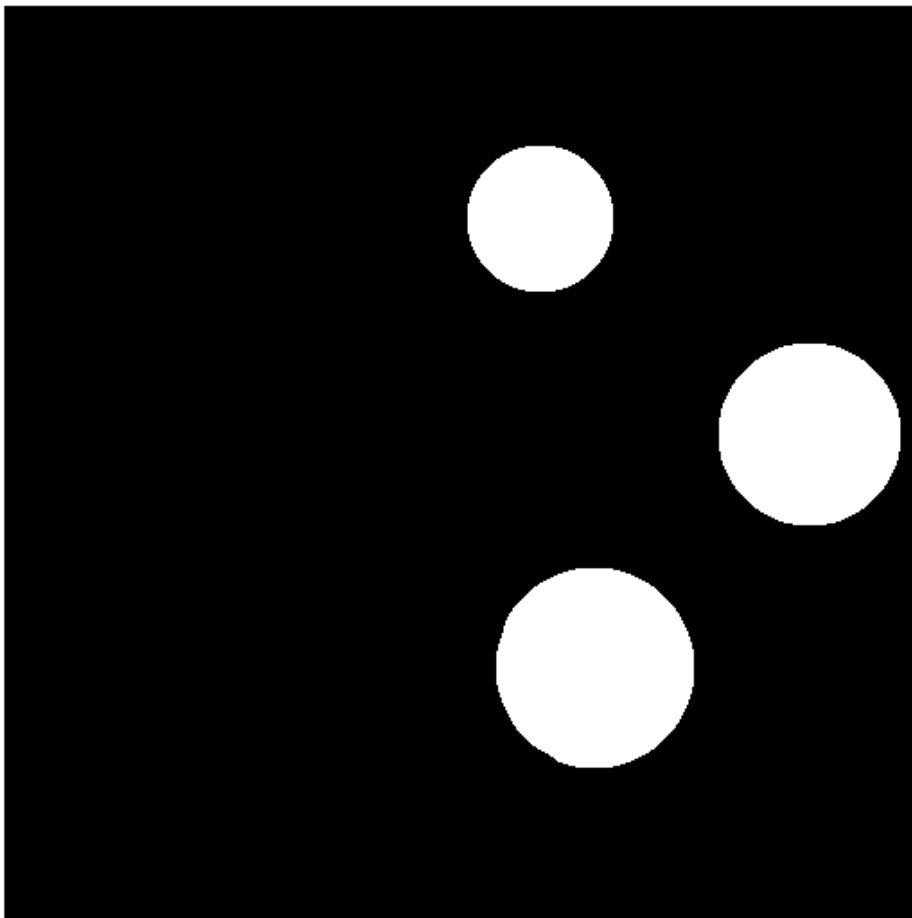
Read an image into workspace.

```
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=4x3 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
```

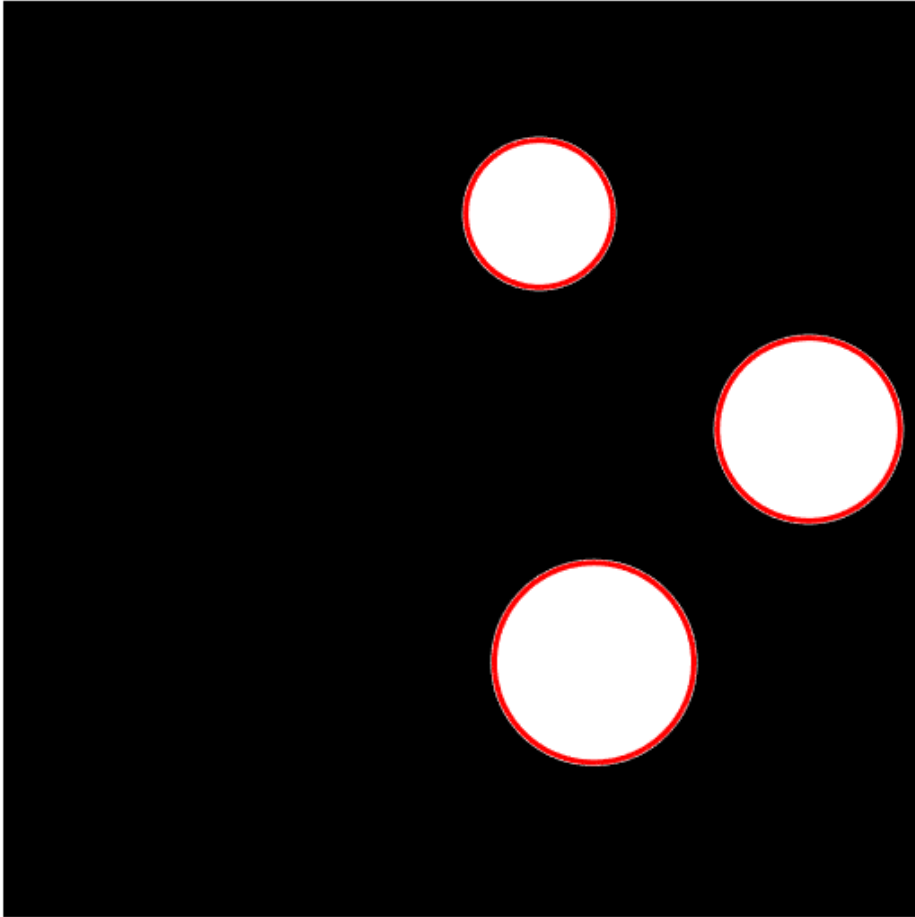
Get centers and radii of the circles.

```
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 — Binary image**

logical array of any dimension | gpuArray

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

To measure properties of image regions using a GPU, specify **BW** as a `gpuArray` that contains a 2-D logical matrix.

Data Types: `logical`

### **CC — Connected components**

structure

Connected components, specified as a structure returned by `bwconncomp`.

Data Types: `struct`

### **L — Label matrix**

numeric array | gpuArray

Label matrix, specified as a numeric array of 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.

To measure properties of image regions using a GPU, specify **L** as a `gpuArray` that contains a 2-D label matrix.

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

### **properties — Type of measurement**

comma-separated list of string scalars or character vectors | cell array of string scalars or character vectors | `'all'` | `'basic'`

Type of measurement, specified as a comma-separated list of string scalars or character vectors, a cell array of string scalars or character vectors, or as `'all'` or `'basic'`. Property names are case-insensitive and can be abbreviated. When used with code generation, `regionprops` does not support cell arrays of string scalars or character vectors.

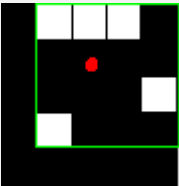
- If you specify 'all', `regionprops` computes all the shape measurements and, for grayscale images, the pixel value measurements as well.
- If you specify 'basic', or do not specify the `properties` argument, `regionprops` computes only the 'Area', 'Centroid', and 'BoundingBox' measurements.

The following tables list all the properties that provide shape measurements. The properties listed in the Pixel Value Measurements table are valid only when you specify a grayscale image.


### Shape Measurements

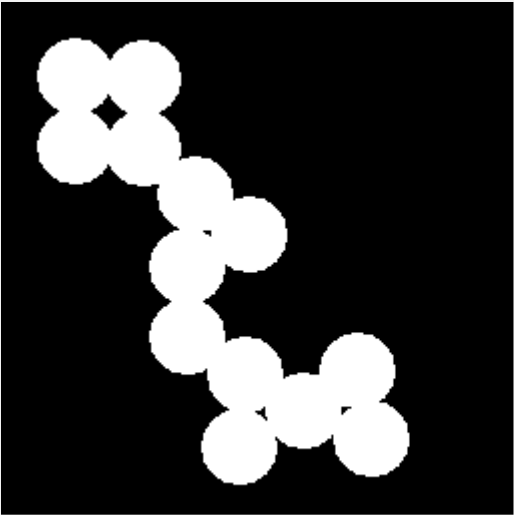
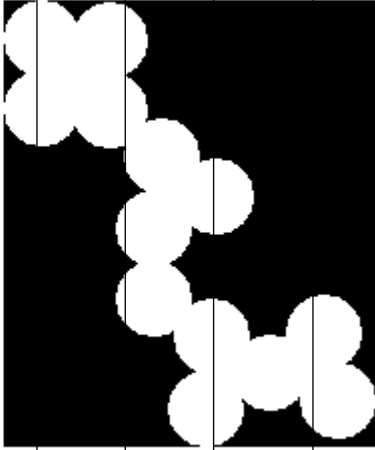
Property Name	Description	N-D Support	GPU Support	Code Generation
'Area'	<p>Actual number of pixels in the region, returned as a scalar. (This value might differ slightly from the value returned by <code>bwarea</code>, which weights different patterns of pixels differently.)</p> <p>To find the equivalent to the area of a 3-D volume, use the 'Volume' property of <code>regionprops3</code>.</p>	Yes	Yes	Yes
'BoundingBox'	<p>Smallest rectangle containing the region, returned as a 1-by-Q*2 vector, where Q is the number of image dimensions. For example, in the vector <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> specifies the width of the bounding box along each dimension in the form <code>[x_width y_width ...]</code>. <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>.</p>	Yes	Yes	Yes




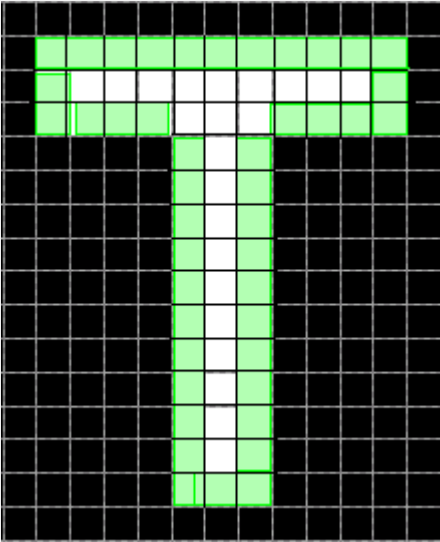
Property Name	Description	N-D Support	GPU Support	Code Generation
'Centroid'	<p>Center of mass of the region, returned as a 1-by-<math>Q</math> vector. The first element of <code>Centroid</code> is the horizontal coordinate (or <math>x</math>-coordinate) of the center of mass. The second element is the vertical coordinate (or <math>y</math>-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.</p> 	Yes	Yes	Yes
'ConvexArea'	Number of pixels in 'ConvexImage', returned as a scalar.	2-D only	No	No
'ConvexHull'	Smallest convex polygon that can contain the region, returned as a $p$ -by-2 matrix. Each row of the matrix contains the $x$ - and $y$ -coordinates of one vertex of the polygon.	2-D only	No	No
'ConvexImage'	Image that specifies the convex hull, with all pixels within the hull filled in (set to on), returned as a binary image (logical). 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

<b>Property Name</b>	<b>Description</b>	<b>N-D Support</b>	<b>GPU Support</b>	<b>Code Generation</b>
'Eccentricity'	Eccentricity of the ellipse that has the same second-moments as the region, returned as a scalar. 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
'EquivDiameter'	Diameter of a circle with the same area as the region, returned as a scalar. Computed as $\sqrt{4 * \text{Area} / \pi}$ .	2-D only	Yes	Yes
'EulerNumber'	Number of objects in the region minus the number of holes in those objects, returned as a scalar. 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".	2-D only	No	Yes
'Extent'	Ratio of pixels in the region to pixels in the total bounding box, returned as a scalar. Computed as the Area divided by the area of the bounding box.	2-D only	Yes	Yes

Property Name	Description	N-D Support	GPU Support	Code Generation
'Extrema'	<p>Extrema points in the region, returned as an 8-by-2 matrix. Each row of the matrix contains the x- and y-coordinates of one of the points. The format of the vector is [top-left top-right right-top right-bottom bottom-right bottom-left left-bottom left-top]. This 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.</p> 	2-D only	Yes	Yes
'FilledArea'	Number of on pixels in FilledImage, returned as a scalar.	Yes	No	Yes

Property Name	Description	N-D Support	GPU Support	Code Generation
'FilledImage'	Image the same size as the bounding box of the region, returned as a binary (logical) array. The on pixels correspond to the region, with all holes filled in, as shown in this figure.  <div style="display: flex; justify-content: space-around; align-items: center;">   </div>	Yes	No	Yes
	<b>Original Image, Containing a Single Region</b>		<b>Image Returned</b>	
'Image'	Image the same size as the bounding box of the region, returned as a binary (logical) array. The on pixels correspond to the region, and all other pixels are off.	Yes	Yes	Yes
'MajorAxisLength'	Length (in pixels) of the major axis of the ellipse that has the same normalized second central moments as the region, returned as a scalar.	2-D only	Yes	Yes
'MinorAxisLength'	Length (in pixels) of the minor axis of the ellipse that has the same normalized second central moments as the region, returned as a scalar.	2-D only	Yes	Yes

Property Name	Description	N-D Support	GPU Support	Code Generation
'Orientation'	<p>Angle between the x-axis and the major axis of the ellipse that has the same second-moments as the region, returned as a scalar. The value is in degrees, ranging from -90 degrees 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. The orientation is the angle between the horizontal dotted line and the major axis.</p> 	2-D only	Yes	Yes

Property Name	Description	N-D Support	GPU Support	Code Generation
'Perimeter'	<p>Distance around the boundary of the region. returned as a scalar. <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.</p> 	2-D only	Yes	Yes
'PixelId xList'	Linear indices of the pixels in the region, returned as a $p$ -element vector.	Yes	Yes	Yes
'PixelList'	Locations of pixels in the region, returned as a $p$ -by- $Q$ matrix. 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'	Proportion of the pixels in the convex hull that are also in the region, returned as a scalar. Computed as $\text{Area}/\text{ConvexArea}$ .	2-D only	No	No

Property Name	Description	N-D Support	GPU Support	Code Generation
'SubarrayIdx'	Elements of <code>L</code> inside the object bounding box, returned as a cell array that contains indices such that <code>L(idx{:})</code> extracts the elements.	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
'MaxIntensity'	Value of the pixel with the greatest intensity in the region, returned as a scalar.	Yes	Yes	Yes
'MeanIntensity'	Mean of all the intensity values in the region, returned as a scalar.	Yes	Yes	Yes
'MinIntensity'	Value of the pixel with the lowest intensity in the region, returned as a scalar.	Yes	Yes	Yes
'PixelValues'	Number of pixels in the region, returned as 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.	Yes	Yes	Yes
'WeightedCentroid'	Center of the region based on location and intensity value, returned as a $p$ -by- $Q$ vector of coordinates. 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.	Yes	Yes	Yes

Data Types: `char` | `string` | `cell`

### I — Image to be measured

grayscale image

Image to be measured, specified as a grayscale image. The size of the image must match the size of the binary image `BW`, connected component structure `CC`, or label matrix `L`.

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
<code>'struct'</code>	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> .
<code>'table'</code>	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` | `string`

## 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 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 useful when you work with the output of `regionprops`. 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 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. To specify nondefault connectivity, call `bwconncomp` and pass the result to `regionprops`.

```
CC = bwconncomp(BW, CONN);  
S = regionprops(CC);
```

- Most of the measurements take little time to compute. However, the following measurements can take 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. `regionprops` takes advantage of intermediate computations useful to each computation. 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 discontinuous 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 “Understand 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'`.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `gpuArray` input must be a 2-D logical matrix or a 2-D label matrix.

- The connected component structure (CC) input is not supported.
- The following properties are not supported: 'ConvexArea', 'ConvexHull', 'ConvexImage', 'EulerNumber', 'FilledArea', 'FilledImage', and 'Solidity'.
- 'struct' is the only return type supported.

For more information, see “Image Processing on a GPU”.

## See Also

`bwconncomp` | `bwlabel` | `bwlabeln` | `bwpropfilt` | `ismember` | `labelmatrix` | `regionprops3` | `watershed`

**Introduced before R2006a**

# regionprops3

Measure properties of 3-D volumetric image regions

## Syntax

```
stats = regionprops3(BW,properties)
stats = regionprops3(CC,properties)
stats = regionprops3(L,properties)
stats = regionprops3( ____,V,properties)
```

## Description

`stats = regionprops3(BW,properties)` measures a set of properties for each connected component (object) in the 3-D volumetric binary image `BW`. The output `stats` is a table with height (number of rows) equal to the number of objects in `BW`. The variables (columns) of the table denote different properties for each region, as specified by `properties`.

For all syntaxes, if you do not specify the `properties` argument, `regionprops3` returns the 'Volume', 'Centroid', and 'BoundingBox' measurements.

`stats = regionprops3(CC,properties)` measures a set of properties for each connected component (object) in `CC`, which is a structure returned by `bwconncomp`.

`stats = regionprops3(L,properties)` measures a set of properties for each labeled region in the 3-D label matrix `L`.

`stats = regionprops3( ____,V,properties)` measures a set of properties for each labeled region in the 3-D volumetric grayscale image `V`. The first input (`BW`, `CC`, or `L`) identifies the regions in `V`.

## Examples

### Estimate Centers and Radii of Objects in 3-D Volumetric Image

Create a binary image with two spheres.

```
[x,y,z] = meshgrid(1:50,1:50,1:50);  
bw1 = sqrt((x-10).^2 + (y-15).^2 + (z-35).^2) < 5;  
bw2 = sqrt((x-20).^2 + (y-30).^2 + (z-15).^2) < 10;  
bw = bw1 | bw2;
```

Get the centers and radii of the two spheres.

```
s = regionprops3(bw, "Centroid", "PrincipalAxisLength");  
centers = s.Centroid
```

```
centers = 2×3
```

```
    20    30    15  
    10    15    35
```

```
diameters = mean(s.PrincipalAxisLength,2)
```

```
diameters = 2×1
```

```
    17.8564  
     8.7869
```

```
radii = diameters/2
```

```
radii = 2×1
```

```
     8.9282  
     4.3935
```

### Get All Statistics for Cube Within a Cube

Make a 9-by-9 cube of 0s that contains a 3-by-3 cube of 1s at its center.

```
innercube = ones(3,3,3);  
cube_in_cube = padarray(innercube,[3 3],0,'both');
```

Get all statistics on the cube within the cube.

```
stats = regionprops3(cube_in_cube, 'all')
```

*stats=1x18 table*

Volume	Centroid	BoundingBox	SubarrayIdx
27	5 5 2	[1x6 double]	[1x3 double] [1x3 double] [1x3 double]

## Input Arguments

### **BW** — Volumetric binary image

3-D logical array

Volumetric binary image, specified as a 3-D logical array.

Data Types: `logical`

### **CC** — Connected components

structure

Connected components, specified as a structure returned by `bwconncomp`. The `CC` structure must represent a 3-D image, that is, `CC.ImageSize` must be a 1-by-3 vector. The `CC` structure must also have been created using a 3-D connectivity value, such as 6, 18, or 26.

Data Types: `struct`

### **L** — Label matrix

3-D numeric array

Label matrix, specified as a 3-D numeric array. `regionprops3` 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`

## **properties — Type of measurement**

'basic' (default) | comma-separated list of strings or character vectors | cell array of strings or character vectors | 'all'

Type of measurement, specified as a comma-separated list of strings or character vectors, a cell array of strings or character vectors, 'all' or 'basic'. Property names are case-insensitive and can be abbreviated.

- If you specify 'all', `regionprops3` computes all the shape measurements and, if you specified a grayscale image, all the pixel value measurements.
- If you specify 'basic' or do not specify the `properties` argument, then `regionprops3` computes only the 'Volume', 'Centroid', and 'BoundingBox' measurements.

The following table lists all the properties that provide shape measurements. The Voxel Value Measurements table lists additional properties that are valid only when you specify a grayscale image.



## Shape Measurements

Property Name	Description
'BoundingBox'	Smallest cuboid containing the region, returned as a 1-by-6 vector of the form [ulf_x ulf_y ulf_z width_x width_y width_z]. ulf_x, ulf_y, and ulf_z specify the upper-left front corner of the cuboid. width_x, width_y, and width_z specify the width of the cuboid along each dimension.
'Centroid'	Center of mass of the region, returned as a 1-by-3 vector of the form [centroid_x centroid_y and centroid_z]. The first element, centroid_x, is the horizontal coordinate (or x-coordinate) of the center of mass. The second element, centroid_y, is the vertical coordinate (or y-coordinate). The third element, centroid_z, is the planar coordinate (or z-coordinate).
'ConvexHull'	Smallest convex polygon that can contain the region, returned as a <i>p</i> -by-3 matrix. Each row of the matrix contains the x-, y-, and z-coordinates of one vertex of the polygon.
'ConvexImage'	Image of the convex hull, returned as a volumetric binary image (logical) with all voxels within the hull filled in (set to on). The image is the size of the bounding box of the region.
'ConvexVolume'	Number of voxels in 'ConvexImage', returned as a scalar.
'EigenValues'	Eigenvalues of the voxels representing a region, returned as a 3-by-1 vector. regionprops3 uses the eigenvalues to calculate the principal axes lengths.
'EigenVectors'	Eigenvectors of the voxels representing a region, returned as a 3-by-3 vector. regionprops3 uses the eigenvectors to calculate the orientation of the ellipsoid that has the same normalized second central moments as the region.
'EquivalentDiameter'	Diameter of a sphere with the same volume as the region, returned as a scalar. Computed as $(6 * \text{Volume} / \pi)^{(1/3)}$ .
'Extent'	Ratio of voxels in the region to voxels in the total bounding box, returned as a scalar. Computed as the value of Volume divided by the volume of the bounding box. [Volume / (bounding box width * bounding box height * bounding box depth)]

Property Name	Description
'Image'	Bounding box of the region, returned as a volumetric binary image (logical) that is the same size as the bounding box of the region. The on voxels correspond to the region, and all other voxels are off.
'Orientation'	Euler angles, returned as a 1-by-3 vector. The angles are based on the right-hand rule. <code>regionprops3</code> interprets the angles by looking at the origin along the <i>x</i> -, <i>y</i> -, and <i>z</i> -axis representing roll, pitch, and yaw respectively. A positive angle represents a rotation in the counterclockwise direction. Rotation operations are not commutative so they must be applied in the correct order to have the intended effect. For more information, see “References” on page 1-2526.
'PrincipalAxisLength'	Length (in voxels) of the major axes of the ellipsoid that have the same normalized second central moments as the region, returned as 1-by-3 vector. <code>regionprops3</code> sorts the values from highest to lowest.
'Solidity'	Proportion of the voxels in the convex hull that are also in the region, returned as a scalar. Computed as <code>Volume/ConvexVolume</code> .
'SubarrayIdx'	Indices used to extract elements inside the object bounding box, returned as a cell array such that <code>L(idx{:})</code> extracts the elements of <code>L</code> inside the object bounding box.
'SurfaceArea'	Distance around the boundary of the region, returned as a scalar. For more information, see “References” on page 1-2526.
'Volume'	Count of the actual number of 'on' voxels in the region, returned as a scalar. Volume represents the metric or measure of the number of voxels in the regions within the volumetric binary image, <code>BW</code> .
'VoxelIdxList'	Linear indices of the voxels in the region, returned as a <i>p</i> -element vector.
'VoxelList'	Locations of voxels in the region, returned as a <i>p</i> -by-3 matrix. Each row of the matrix has the form [ <i>x y z</i> ] and specifies the coordinates of one voxel in the region.

The voxel value measurement properties in the following table are valid only when you specify a grayscale volumetric image, `V`.

## Voxel Value Measurements

Property Name	Description
'MaxIntensity'	Value of the voxel with the greatest intensity in the region, returned as a scalar.
'MeanIntensity'	Mean of all the intensity values in the region, returned as a scalar.
'MinIntensity'	Value of the voxel with the lowest intensity in the region, returned as a scalar.
'VoxelValues'	Value of the voxels in the region, returned as a $p$ -by-1 vector, where $p$ is the number of voxels in the region. Each element in the vector contains the value of a voxel in the region.
'WeightedCentroid'	Center of the region based on location and intensity value, returned as a $p$ -by-3 vector of coordinates. 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). The third element is the planar coordinate (or z-coordinate).

Data Types: char | string | cell

### V — Volumetric grayscale image

3-D numeric array

Volumetric grayscale image, specified as a 3-D numeric array. The size of the image must match the size of the binary image `BW`, connected component structure `CC`, or label matrix `L`.

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

## Output Arguments

### stats — Measurement values

table

Measurement values, returned as a table. The number of rows in the table corresponds to the number of objects in `BW`, `CC.NumObjects`, or `max(L(:))`. The variables (columns) in

each table row denote the properties calculated for each region, as specified by properties.

## References

- [1] Lehmann, Gaetan and David Legland, *Efficient N-Dimensional surface estimation using Crofton formula and run-length encoding*, <https://hdl.handle.net/10380/3342>
- [2] Shoemake, Ken, *Graphics Gems IV* Edited by Paul S. Heckbert, Morgan Kaufmann, 1994, Pg 222-229.

## See Also

`bwconncomp` | `bwlabeln` | `ismember` | `regionprops`

**Introduced in R2017b**

# RegularStepGradientDescent

Regular step gradient descent optimizer configuration

## Description

A `RegularStepGradientDescent` object describes a regular step gradient descent optimization configuration that you pass to the function `imregister` to solve image registration problems.

## Creation

You can create a `RegularStepGradientDescent` object using the following methods:

- `imregconfig` — Returns a `RegularStepGradientDescent` object paired with an appropriate metric for registering monomodal images
- Entering

```
metric = registration.optimizer.RegularStepGradientDescent;
```

on the command line creates a `RegularStepGradientDescent` object with default settings

## 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

## 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

### Functions

`imregconfig` | `imregister`

### Objects

`MattesMutualInformation` | `MeanSquares` | `OnePlusOneEvolutionary`

## Topics

“Create an Optimizer and Metric for Intensity-Based Image Registration”

**Introduced in R2012a**

# 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 = 1×3
```

```
    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 = 1×3  
    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 = 1×3  
    31.3294    -4.0732   -18.1750
```

## Convert RGB Image to L\*a\*b\* and Display L\* Component

Read RGB image into the workspace.

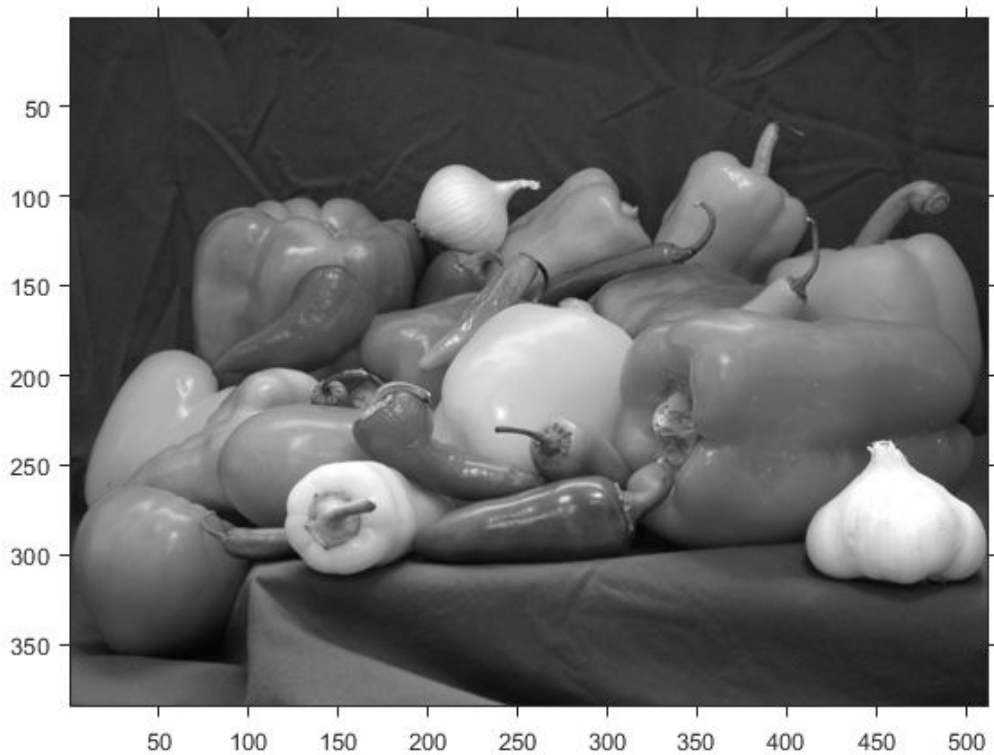
```
rgb = imread('peppers.png');
```

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

*c*-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 *c*-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 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, [1.000, 1.000, 1.000]. Useful as a theoretical reference.
'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 horizon light.
'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.

Value	White Point
'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 [0.9642, 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

`lab2rgb` | `rgb2xyz` | `xyz2lab`

**Introduced in R2014b**



# rgb2lin

Linearize gamma-corrected RGB values

## Syntax

```
B = rgb2lin(A)
B = rgb2lin(A,Name,Value)
```

## Description

`B = rgb2lin(A)` undoes the gamma correction of the sRGB values in image `A` so that `B` contains linear RGB values.

`B = rgb2lin(A,Name,Value)` undoes gamma correction using name-value pairs to control additional options.

## Examples

### Linearize an sRGB Image

Open an image. The JPEG file format saves images in the gamma-corrected sRGB color space.

```
A = imread('foosball.jpg');
```

Display the image. To shrink the image so that it appears fully on the screen, set the optional initial magnification to a value less than 100.

```
figure
imshow(A,'InitialMagnification',25)
title('Scene With sRGB Gamma Correction')
```

Scene With sRGB Gamma Correction



To undo the gamma correction and linearize the image, use the `rgb2lin` function. Optionally, specify the data type of the linearized values.

```
B = rgb2lin(A, 'OutputType', 'double');
```

Display the linearized image, setting the optional magnification.

```
figure  
imshow(B, 'InitialMagnification', 25)  
title('Scene Without sRGB Gamma Correction')
```

Scene Without sRGB Gamma Correction



Shadows in the linearized image are darker than in the original image, as expected.

## Input Arguments

### **A** — Gamma-corrected RGB image

real, nonsparse,  $m$ -by- $n$ -by-3 array

Gamma-corrected RGB image, specified as a real, nonsparse,  $m$ -by- $n$ -by-3 array.

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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `B = lin2rgb(I, 'ColorSpace', 'adobe-rgb-1998')` linearizes the gamma-corrected image, `I`, according to the Adobe RGB (1998) standard.

### **ColorSpace** — Color space of the input image

`'srgb'` (default) | `'adobe-rgb-1998'`

Color space of the input image, specified as the comma-separated pair consisting of `'ColorSpace'` and `'srgb'` or `'adobe-rgb-1998'`.

Data Types: `char` | `string`

### **OutputType** — Data type of output RGB values

`'double'` | `'single'` | `'uint8'` | `'uint16'`

Data type of the output RGB values, specified as the comma-separated pair consisting of `'OutputType'` and `'double'`, `'single'`, `'uint8'`, or `'uint16'`. By default, the output data type is the same as the data type of `A`.

Data Types: `char` | `string`

## Output Arguments

### **B** — Linearized RGB image

real, nonsparse, *m*-by-*n*-by-3 array

Linearized RGB image, returned as a real, nonsparse *m*-by-*n*-by-3 array.

## Algorithms

### Linearization Using the sRGB Standard

sRGB tristimulus values are linearized using the following parametric curve:

$$f(u) = -f(-u), \quad u < 0$$

$$f(u) = c \cdot u, \quad 0 \leq u < d$$

$$f(u) = (a \cdot u + b)^\gamma, \quad u \geq d,$$

where  $u$  represents a color value with these parameters:

$$a = 1/1.055$$

$$b = 0.055/1.055$$

$$c = 1/12.92$$

$$d = 0.04045$$

$$\gamma = 2.4$$

## Linearization Using the Adobe RGB (1998) Standard

Adobe RGB (1998) tristimulus values are linearized using a simple power function:

$$v = u^\gamma,$$

with

$$\gamma = 2.19921875$$

## References

- [1] Ebner, Marc. "Gamma Correction." *Color Constancy*. Chichester, West Sussex: John Wiley & Sons, 2007.
- [2] Adobe Systems Incorporated. "Inverting the color component transfer function." *Adobe RGB (1998) Color Image Encoding*. Section 4.3.5.2, May 2005, p.12.

## See Also

lin2rgb

**Introduced in R2017b**

## rgb2ntsc

Convert RGB color values to NTSC color space

### Syntax

```
yiqmap = rgb2ntsc(rgbmap)  
YIQ = rgb2ntsc(RGB)
```

### Description

`yiqmap = rgb2ntsc(rgbmap)` converts the RGB values in `rgbmap` to NTSC color space. `rgb2ntsc` returns in `yiqmap` the NTSC luminance (*Y*) and chrominance (*I* and *Q*) color components that are equivalent to the colors in RGB color space.

`YIQ = rgb2ntsc(RGB)` converts the truecolor image RGB to the equivalent NTSC image YIQ.

### 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.

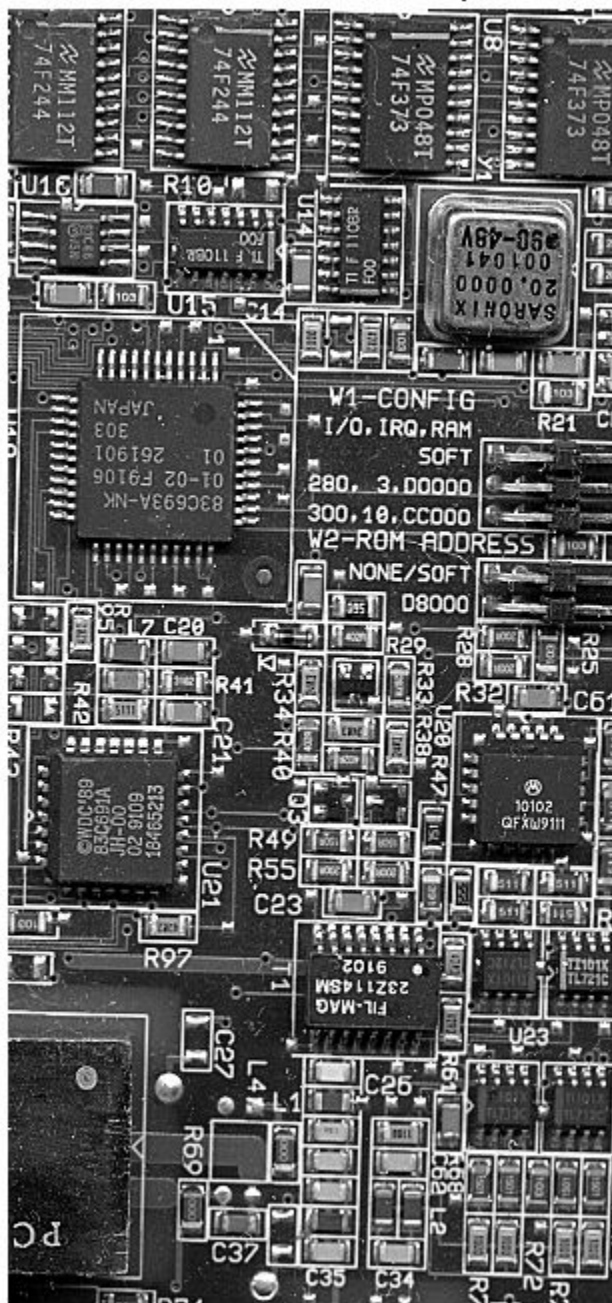
```
YIQ = rgb2ntsc(RGB);
```

Display the NTSC luminance value, represented by the first color channel in the YIQ image.

```
imshow(YIQ(:,:,1));  
title('Luminance in YIQ Color Space');
```



### Luminance in YIQ Color Space



## Input Arguments

### **rgbmap — RGB color space values**

*c*-by-3 array

RGB color space values, specified as an *c*-by-3 array.

Data Types: `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` | `int16`

## Output Arguments

### **yiqmap — NTSC color space values**

*c*-by-3 array

NTSC color space values corresponding to the *c* colors in `rgbmap`, returned as an *c*-by-3 array. The three columns represent the luminance (*Y*) and chrominance (*I* and *Q*) components of each color.

Data Types: `double`

### **YIQ — Image in NTSC color space**

*m*-by-*n*-by-3 array

Image in NTSC color space, returned as an *m*-by-*n*-by-3 array.

Data Types: `double`

## Algorithms

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](#) | [rgb2lab](#) | [rgb2xyz](#) | [rgb2ycbcr](#)

**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 (2° observer).

`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 = 1×3
```

```
    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

*c*-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 *c*-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 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 the comma-separated pair consisting of '`ColorSpace`' and one of '`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 the comma-separated pair consisting of 'WhitePoint' and a 1-by-3 vector or one of the CIE standard illuminants, listed in the 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, [1.000, 1.000, 1.000]. Useful as a theoretical reference.
'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 horizon light.
'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 [0.9642, 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

numeric array



Converted color values, returned as a numeric array of 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

`lab2xyz` | `rgb2lab` | `xyz2rgb`

**Introduced in R2014b**

## rgb2ycbcr

Convert RGB color values to YCbCr color space

### Syntax

```
ycbcrmap = rgb2ycbcr(rgbmap)  
YCBCR = rgb2ycbcr(RGB)
```

### 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`.

You optionally can perform the conversion using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`YCBCR = rgb2ycbcr(RGB)` converts the truecolor image `RGB` to the equivalent image in YCbCr color space.

### Examples

#### Convert RGB to YCbCr

#### Convert Image from RGB to YCbCr

Read an RGB image into the workspace.

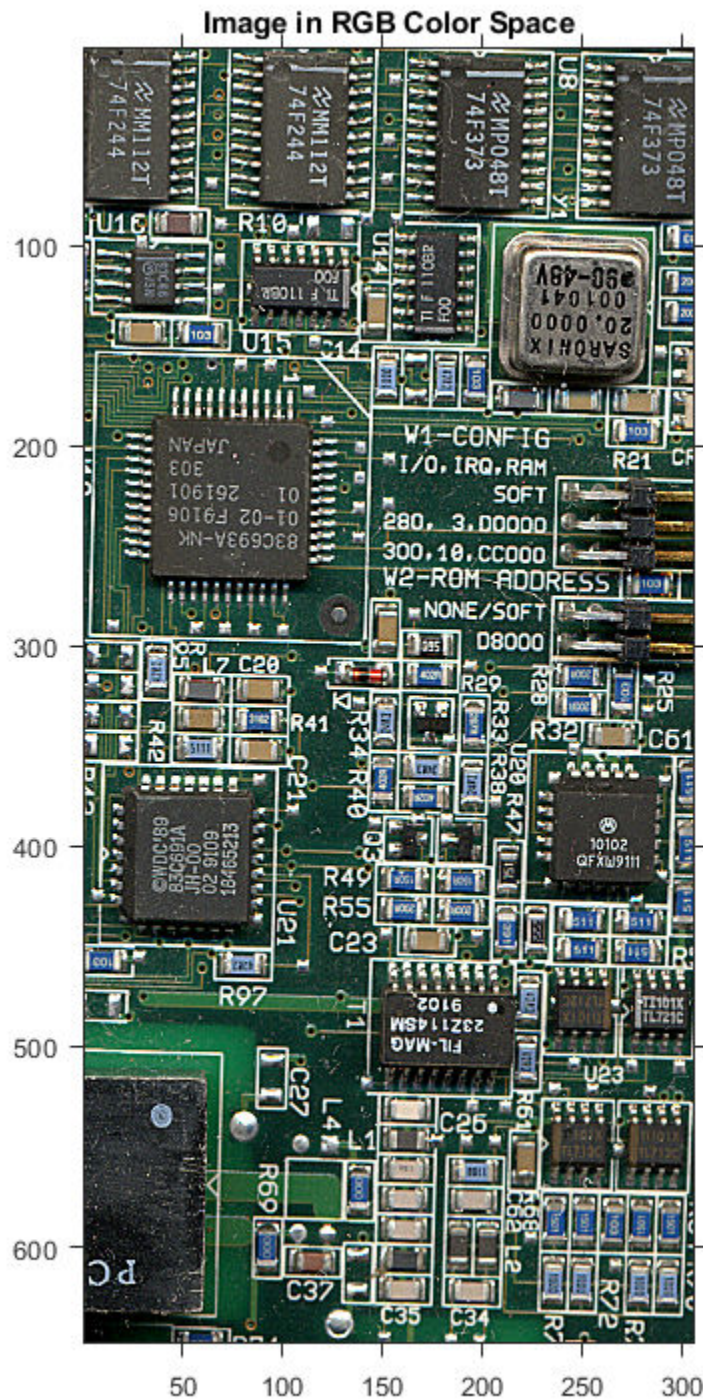
```
RGB = imread('board.tif');
```

Convert the image to YCbCr.

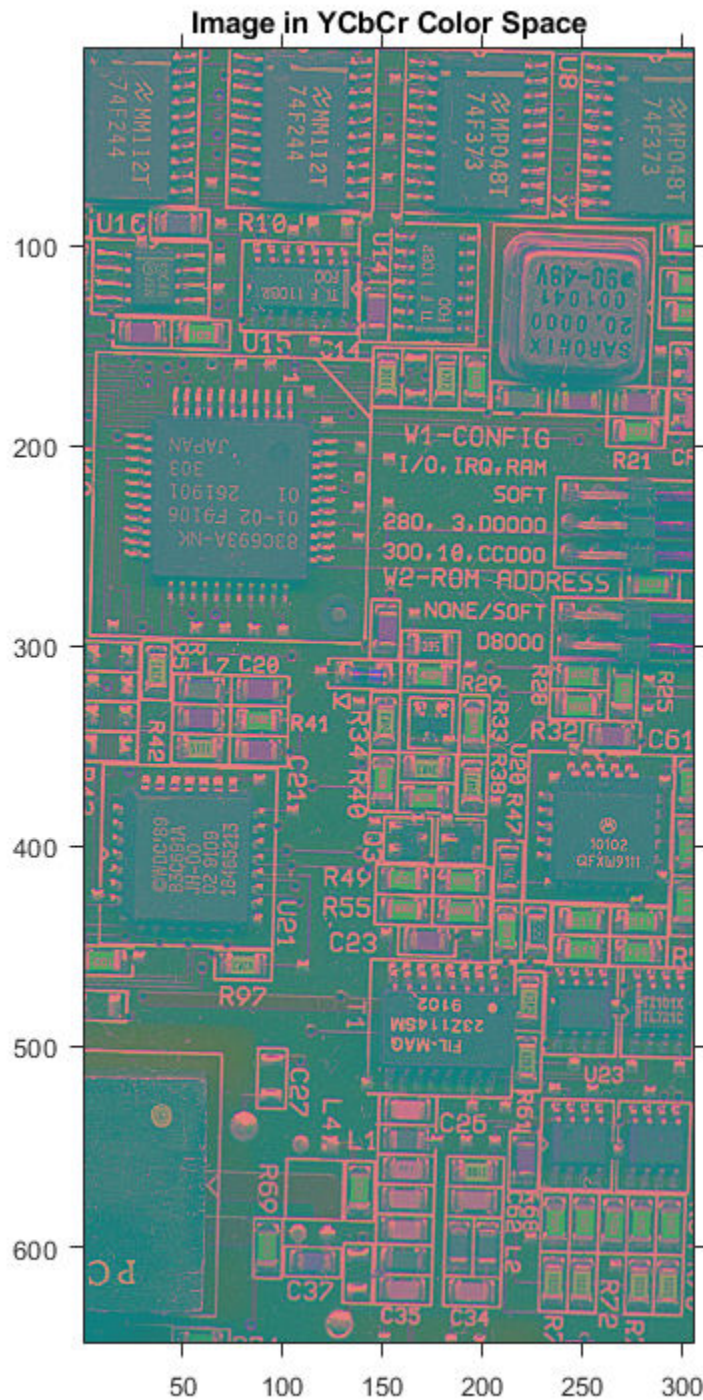
```
YCBCR = rgb2ycbcr(RGB);
```

Display the original image and the new image

```
figure
imshow(IMG);
title('Image in RGB Color Space');
```



```
figure
imshow(YCBCR);
title('Image in YCbCr Color Space');
```



### Convert Colormap from RGB to YCbCr.

Load an indexed image into the workspace. The colormap is in RGB colorspace.

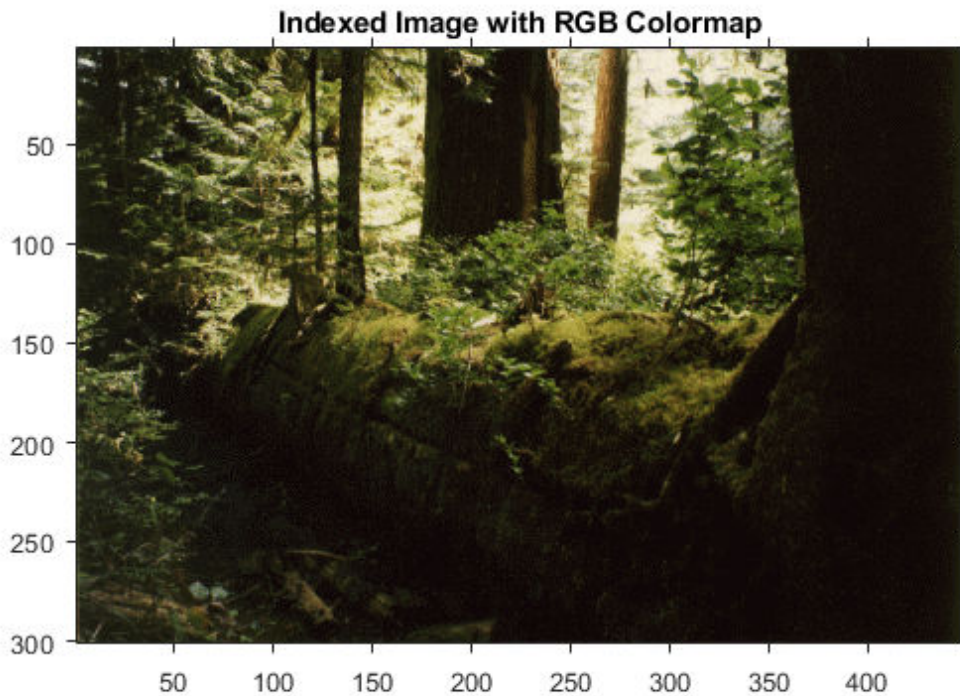
```
[I,map] = imread('forest.tif');
```

Convert the colormap to YCbCr.

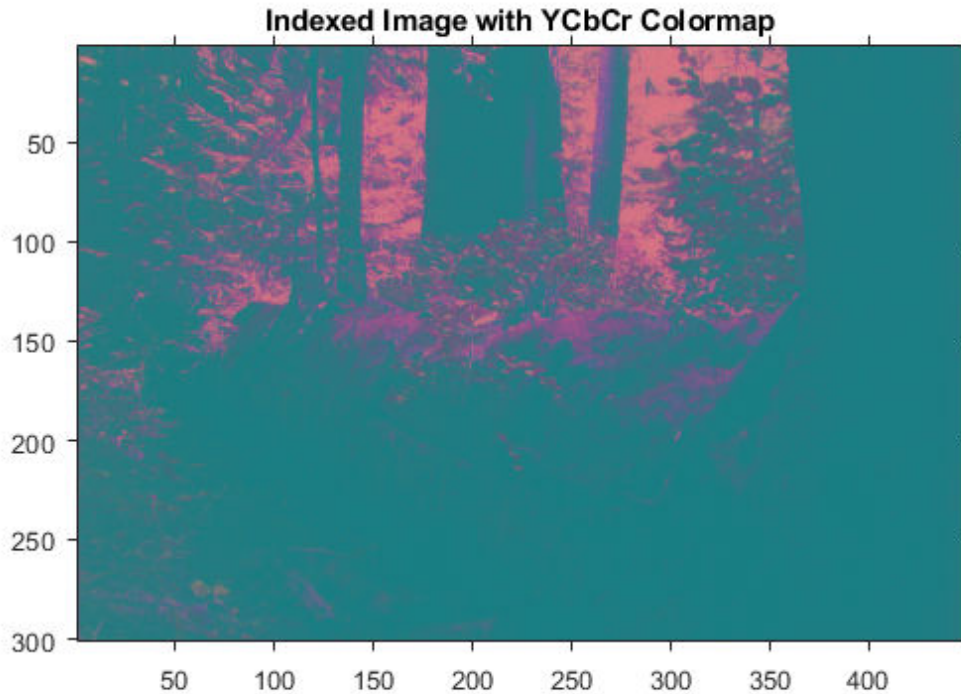
```
newmap = rgb2ycbcr(map);
```

Display the grayscale image with the original map and with the new map.

```
figure  
imshow(I,map)  
title('Indexed Image with RGB Colormap');
```



```
figure
imshow(I,newmap)
title('Indexed Image with YCbCr Colormap');
```



## Input Arguments

### **rgbmap — RGB color space values**

c-by-3 matrix | gpuArray

RGB color space values, specified as an c-by-3 matrix representing *c* colors.

To perform the conversion using a GPU, specify `rgbmap` as a `gpuArray` that contains a c-by-3 matrix.

Data Types: `single` | `double`



**RGB — RGB image**

m-by-n-by-3 array | gpuArray

RGB image, specified as an m-by-n-by-3 array.

To perform the conversion using a GPU, specify RGB as a gpuArray that contains an m-by-n-by-3 array.

Data Types: single | double | uint8 | uint16

## Output Arguments

**ycbcrmap — YCbCr color space values**

c-by-3 matrix | gpuArray

YCbCr color space values, returned as an c-by-3 matrix. 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].

If the conversion is performed using a GPU, then ycbcrmap is returned as a gpuArray containing a c-by-3 matrix.

**YCBCR — Image in YCbCr color space**

m-by-n-by-3 array | gpuArray

Image in YCbCr color space, returned as an m-by-n-by-3 array.

- If the input is double or single, then 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, then Y is in the range [16, 235] and Cb and Cr are in the range [16, 240].
- If the input is uint16, then Y is in the range [4112, 60395] and Cb and Cr are in the range [4112, 61680].

If the conversion is performed using a GPU, then YCBCR is returned as a gpuArray containing an m-by-n-by-3 array.

## 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 “Understand Code Generation with Image Processing Toolbox”.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

### See Also

rgb2lab | rgb2ntsc | rgb2xyz | ycbcr2rgb

**Introduced before R2006a**

## roicolor

Select region of interest (ROI) based on color

### Syntax

```
BW = roicolor(I,low,high)
BW = roicolor(I,v)
```

### Description

`BW = roicolor(I,low,high)` returns an ROI selected as those pixels in image `I` that lie within the range `[low high]`. The returned value, `BW`, is a binary image with `0`s outside the region of interest and `1`s inside.

`BW = roicolor(I,v)` returns an ROI selected as those pixels in image `I` that match values in vector `v`.

### 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)
```



## Input Arguments

**I — Indexed or grayscale image**

*m*-by-*n* numeric matrix

Indexed or grayscale image, specified as an *m*-by-*n* numeric matrix.

**low — Minimum value**

numeric scalar

Minimum value to include in the ROI, specified as a numeric scalar.

**high — Maximum value**

numeric scalar

Minimum value to include in the ROI, specified as a numeric scalar.

**v — Set of values**

numeric vector

Set of values to include in the ROI, specified as a numeric vector.

## Output Arguments

### **BW** — Binary image

*m*-by-*n* logical matrix

Binary image, returned as an *m*-by-*n* logical matrix.

Data Types: `logical`

## Tips

- You can use the returned image as a mask for masked filtering using `roifilt2`.
- If you specify a colormap range, `[low high]`, then

```
BW = (I >= low) & (I <= high)
```

- If you specify a set of colormap values, `v`, then the mask generated by `roicolor` is equivalent to:

```
BW = ismember(I,v)
```

## See Also

`ismember` | `roifilt2` | `roipoly`

**Introduced before R2006a**

## roifill

Fill in specified region of interest (ROI) polygon in grayscale image

---

**Note** `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 1-2571. 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 1-2573).

---

**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.

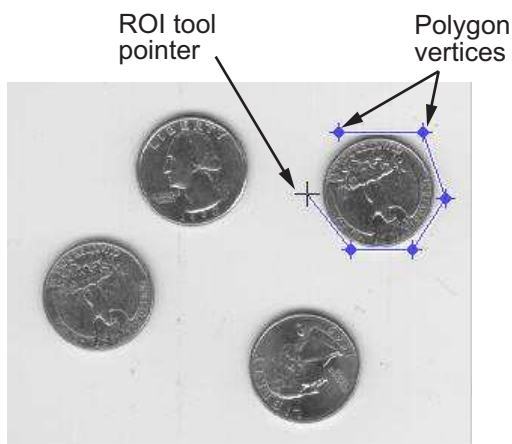
`J = roifill(x, y, I, xi, yi)` uses the vectors `x` and `y` to establish a nondefault spatial coordinate system. `xi` and `yi` are equal-length vectors that specify polygon vertices as locations in this coordinate system.

`[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
Closing the polygon. (Completing the region-of-interest.)	Use any of the following mechanisms: <ul style="list-style-type: none"> <li>• Move the pointer over the initial vertex of the polygon that you selected. The shape changes to a circle ○. Click either mouse button.</li> <li>• 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.</li> <li>• 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.</li> </ul>
Deleting the polygon	Press <b>Backspace</b> , <b>Escape</b> or <b>Delete</b> , or right-click inside the region and select <b>Cancel</b> from the context menu.  Note: If you delete the ROI, the function returns empty values.
Moving the polygon	Move the pointer inside the region. The pointer changes to a fleur ✚. Click and drag the mouse to move the polygon.

Interactive Behavior	Description
Changing the color of the polygon	Move the pointer inside the region. Right-click and select <b>Set color</b> from the context menu.
Adding a new vertex.	Move the pointer over an edge of the polygon and press the <b>A</b> 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 <b>Delete Vertex</b> 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 <b>Copy position</b> 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 regions of interest (ROIs) in the 2-D image `I` using the 2-D linear filter `h`. `BW` is a binary mask, the same size as `I`, that defines the ROIs in `I`. `roifilt2` returns an image that consists of filtered values for pixels in locations where `BW` contains 1s, and unfiltered values for pixels in locations where `BW` contains 0s.

`J = roifilt2(I,BW,fun)` processes the data in ROIs of `I` using the function `fun`. The value `fun` must be a function handle.

### Examples

#### 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)
```



## Input Arguments

### **h** — Linear filter

2-D numeric matrix

Linear filter, specified as a 2-D numeric matrix.

Data Types: double

### **I** — Image

2-D numeric matrix

Image, specified as a 2-D numeric matrix.

- If you specify a filter, **h**, then **I** can be any of the listed data types.
- If you specify a function handle, **fun**, then **I** can be any class supported by **fun**.

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

## **BW — Mask**

2-D logical matrix | 2-D numeric matrix

Mask, specified as a 2-D logical matrix or a 2-D numeric matrix, the same size as `I`, containing 0s and 1s.

Data Types: `logical`

## **fun — Function handle**

handle

Function handle, specified as a handle. For more information about function handles, see “Create Function Handle” (MATLAB).

## **Output Arguments**

### **J — Filtered image**

2-D matrix

Filtered image, returned as a 2-D matrix.

- If you specify a filter, `h`, then `J` has the same class as the input image, `I`.
- If you specify a function handle, `fun`, then the class of `J` is determined by `fun`.

## **Algorithms**

If you specify a filter, `h`, then `roifilt2` calls `filter2` to implement the filter.

## **See Also**

`filter2` | `imfilter` | `roipoly`

**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,xi2,yi2] = roipoly(____)  
[x2,y2,BW,xi2,yi2] = roipoly(____)  
roipoly(____)
```

## Description

`BW = roipoly` creates an interactive polygon tool associated with the image displayed in the current figure. With this syntax and the other interactive syntaxes, the polygon selection tool blocks the MATLAB command line until you complete the operation. `roipoly` returns the mask as a binary image, setting pixels inside the ROI to 1 and pixels outside the ROI to 0.

For more information about using the polygon selection tool, see “Interactive Behavior” on page 1-2584.

`BW = roipoly(I)` displays the grayscale or RGB image `I` in a figure window and creates an interactive polygon selection tool associated with the image.

`BW = roipoly(I,c,r)` returns a polygonal ROI, `BW` with vertices defined by pixel column and row indices, `c` and `r`.

`BW = roipoly(x,y,I,xi,yi)` returns a polygonal ROI with vertices defined in a nondefault spatial coordinate system. `x` and `y` specify the image limits in the world coordinate system. `xi` and `yi` specify coordinates of polygon vertices as locations in this coordinate system.

`[BW,xi2,yi2] = roipoly( ___ )` also returns the coordinates of the vertices of the closed polygon, `xi2` and `yi2`.

`[x2,y2,BW,xi2,yi2] = roipoly( ___ )` also returns the image limits in `x2` and `y2`.

`roipoly( ___ )` without an output argument displays the resulting mask image in a new figure window.

## Examples

### 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)



## Input Arguments

**I — Grayscale or RGB image**

*m-by-n* numeric matrix | *m-by-n-by-3* numeric array

Grayscale or RGB image, specified as an *m-by-n* numeric matrix for a grayscale image, or an *m-by-n-by-3* numeric array for an RGB image.

**c, r — Column or row index of vertices**

vector of positive integers

Column or row index of vertices, specified as a vector of positive integers. The length of *c* and *r* must match.

**x, y — Image limits in world coordinates**

2-element numeric vector

Image limits in world coordinates along the  $x$ - or  $y$ -dimension, specified as a 2-element numeric vector of the form `[min max]`. The values of  $x$  and  $y$  set the image `XData` and `YData`.

**$xi$ ,  $yi$  —  $x$ - or  $y$ -coordinate of vertices**

numeric vector

$x$ - or  $y$ -coordinate of vertices in the world coordinate system defined by  $x$  and  $y$ , specified as a numeric vector. The length of  $xi$  and  $yi$  must match.

## Output Arguments

**BW — Binary image**

$m$ -by- $n$  logical matrix

Binary image, returned as an  $m$ -by- $n$  logical matrix.

Data Types: `logical`

**$xi2$ ,  $yi2$  — Position of vertices**

numeric vector

Position of vertices of the closed polygon along the  $x$ - or  $y$ -dimension, returned as a numeric vector. The first and last element in the vector are identical, so that the polygon is closed.

- If you select vertices interactively by using the polygon selection tool, or if you specify the position of vertices using row and column indices  $c$  and  $r$ , then  $xi2$  and  $yi2$  are interpreted as pixel indices.
- If you specify the image limits,  $x$  and  $y$ , in a nondefault spatial coordinate system, then  $xi2$  and  $yi2$  are interpreted as  $x$ - and  $y$ -coordinates in that coordinate system.

---

**Note** `roipoly` always produces a closed polygon. If you specify input vertex positions of a closed polygon (such that the last pair of coordinates is identical to the first pair), then the length of the output coordinate vectors is equal to the number of points specified. If the points specified do not describe a closed polygon, then `roipoly` adds a final point having the same coordinates as the first point. In this case the length of the output coordinate vectors is one greater than the number of points specified.

---

## x2, y2 — Image limits


2-element numeric vector

Image limits in the x or y dimension, returned as a 2-element numeric vector of the form [min max]. If you specify image limits x and y, then x2 and y2 are equal to these values. Otherwise, x2 and y2 are equal to the original image XData and YData.

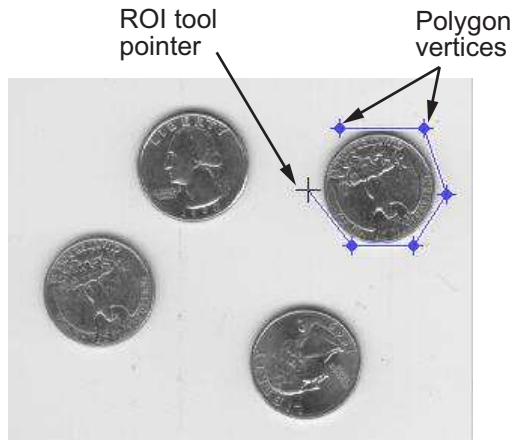
## Definitions






### Interactive Behavior

The polygon selection tool enables you to select and adjust polygon vertices interactively using the mouse.

When the polygon tool is 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. 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.

The figure illustrates a polygon defined by multiple vertices. The following table describes all the interactive behavior of the polygon tool.



Interactive Behavior	Description
Closing the polygon. (Completing the region-of-interest.)	Use any of the following mechanisms: <ul style="list-style-type: none"> <li>• Move the pointer over the initial vertex of the polygon that you selected. The pointer changes to a circle . Click either mouse button.</li> <li>• 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.</li> <li>• 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.</li> </ul>
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.
Deleting the polygon	Press <b>Backspace</b> , <b>Escape</b> or <b>Delete</b> , or right-click inside the region and select <b>Cancel</b> 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 <b>A</b> 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 <b>Delete vertex</b> 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 <b>Set color</b> from the context menu.

<b>Interactive Behavior</b>	<b>Description</b>
Retrieving the coordinates of the vertices	Move the pointer inside the region. Right-click and select <b>Copy position</b> 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.

## 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. Use the mouse to specify a polygon within this image.

## See Also

`drawpolygon` | `poly2mask` | `regionfill` | `roicolor` | `roifilt2`

**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)` creates a reduced resolution data set (R-Set) from the specified file, where `File_Name` is a TIFF or NITF image 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 1-1000 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**

## sizesMatch

Determine if object and image are size-compatible

### Syntax

```
TF = sizesMatch(R,A)
```

### Description

`TF = sizesMatch(R,A)` returns `True` if the size of image `A` is consistent with the `ImageSize` property of spatial referencing object `R`.

### Examples

#### Check If 2-D Grayscale Image and 2-D Spatial Referencing Object Are Size-Compatible

Read a 2-D grayscale image into the workspace. View the size of the image.

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

```
ans = 1×2
```

```
    256    256
```

Create an `imref2d` spatial referencing object with the same dimensions as the image.

```
R = imref2d(size(I))
```

```
R =  
imref2d with 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]

```

Confirm that the size of the image matches the `ImageSize` property of the object.

```

res = sizesMatch(R,I)

res = logical
     1

```

Read another 2-D grayscale image that has a different size. View the size of this image.

```

I2 = imread('coins.png');
size(I2)

ans = 1×2

     246     300

```

Check if the size of this image matches the size of the original spatial referencing object.

```

res2 = sizesMatch(R,I2)

res2 = logical
      0

```

The result is false, as expected.

### Check If 2-D RGB Image and 2-D Spatial Referencing Object Are Size-Compatible

Read an RGB image into the workspace. View the size of the image.

```

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

```

```
ans = 1×3  
    384    512     3
```

Create an `imref2d` spatial referencing object with the same dimensions as the image. The object does not retain information about the third dimension of the image array.

```
R = imref2d(size(I))
```

```
R =  
  imref2d with properties:  
    XWorldLimits: [0.5000 512.5000]  
    YWorldLimits: [0.5000 384.5000]  
    ImageSize: [384 512]  
  PixelExtentInWorldX: 1  
  PixelExtentInWorldY: 1  
  ImageExtentInWorldX: 512  
  ImageExtentInWorldY: 384  
  XIntrinsicLimits: [0.5000 512.5000]  
  YIntrinsicLimits: [0.5000 384.5000]
```

Check if the size of the image is compatible with the `ImageSize` property of the object.

```
res = sizesMatch(R,I)  
  
res = logical  
     1
```

## Check If 3-D Image Array and 3-D Spatial Referencing Object Are Size-Compatible

Read a 3-D volume into the workspace. This image consists of 27 frames of 128-by-128 pixel grayscale images.

```
load mri;  
D = squeeze(D);  
D = ind2gray(D,map);  
size(D)  
  
ans = 1×3
```

```
128 128 27
```

Create an `imref3d` spatial referencing object associated with the volume.

```
R = imref3d(size(D))
```

```
R =  
  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]
```

Confirm that the size of the volume matches the `ImageSize` property of the object.

```
res = sizesMatch(R,D)
```

```
res = logical  
     1
```

The sizes match, as expected.

Read another image that has a different size. This image a 3-D array representing an RGB image.

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

```
ans = 1×3
```

```
384 512 3
```

Check if the size of this image matches the size of the original spatial referencing object.

```
res2 = sizesMatch(R,I)
```

```
res2 = logical  
      0
```

The result is false, as expected.

## Input Arguments

### R — Spatial referencing object

imref2d or imref3d object

Spatial referencing object, specified as an imref2d or imref3d object.

### A — Input image

numeric *m*-by-*n* or *m*-by-*n*-by-*p* array

Input image, specified as a numeric *m*-by-*n* or *m*-by-*n*-by-*p* array.

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

## Output Arguments

### TF — Flag indicating size compatibility

logical scalar

Flag indicating size compatibility, returned as a logical scalar. TF is True if the size of the image A is consistent with the referencing object R. When R is:

- An imref2d spatial referencing object, TF returns true when `R.ImageSize == [size(A,1) size(A,2)]`.

---

**Note** The dimensionality of A does not need to match the dimensionality of an imref2d spatial referencing object. For example, an RGB image can be consistent with an imref2d object. In this case, sizesMatch ignores the third image dimension.

---



- An `imref3d` spatial referencing object, `TF` returns true when `R.ImageSize == size(A)`. `A` must be a 3-D array.

Data Types: `logical`

## See Also

**Introduced in R2013a**

## 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');

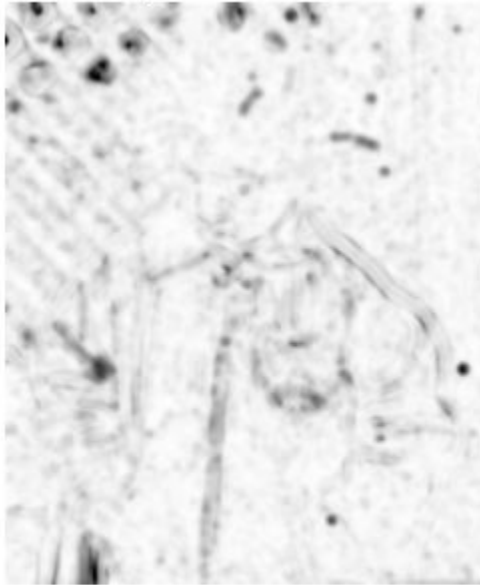
subplot(1,2,1); imshow(ref); title('Reference Image');
subplot(1,2,2); imshow(A); title('Blurred Image');
```



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**



## 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 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.

### Algorithms

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  $\mu_x$ ,  $\mu_y$ ,  $\sigma_x$ ,  $\sigma_y$ , and  $\sigma_{xy}$  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] Zhou, W., A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. "Image Quality Assessment: From Error Visibility to Structural Similarity." *IEEE Transactions on Image Processing*. Vol. 13, Issue 4, April 2004, pp. 600-612.

## 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)
```

### Description

`B = std2(A)` computes the standard deviation of all values in array `A`.

You optionally can compute the standard deviation using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

### 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 array | logical array | gpuArray

Input data, specified as a numeric or logical array.

To perform the computation using a GPU, specify A as a `gpuArray` that contains a numeric or logical array.

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

## Output Arguments

### B — Standard deviation

numeric scalar | gpuArray

Standard deviation of input data, returned as a numeric scalar. If the data type of A is `single`, then the data type of B is also `single`. Otherwise, the data type of B is `double`.

If the standard deviation is computed using a GPU, then B is returned as a `gpuArray` containing a numeric scalar.

Data Types: `single` | `double`

## Extended Capabilities

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

## **See Also**

corr2 | mean | mean2 | std

**Introduced before R2006a**

## stdfilt

Local standard deviation of image

### Syntax

```
J = stdfilt(I)
J = stdfilt(I,nhood)
```

### Description

`J = stdfilt(I)` performs standard deviation filtering of image `I` and returns the filtered image `J`. The value of each output pixel is the standard deviation of the 3-by-3 neighborhood around the corresponding input pixel. 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`.

You optionally can perform the operation using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`J = stdfilt(I,nhood)` specifies the neighborhood, `nhood`, used to compute the standard deviation.

### 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.

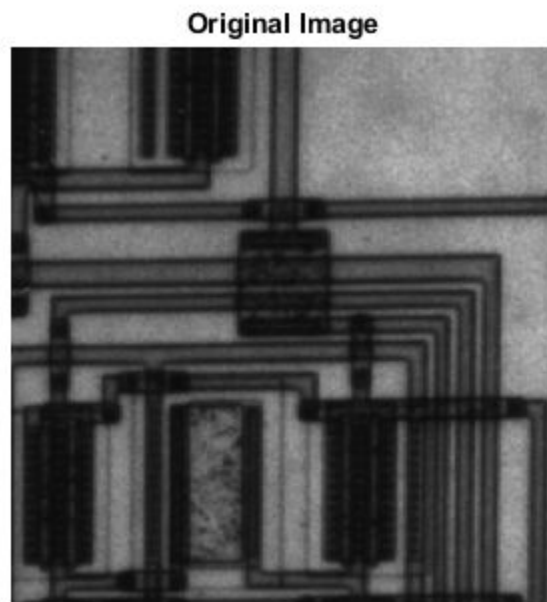
```
I = imread('circuit.tif');
```

Perform standard deviation filtering using `stdfilt`.

```
J = stdfilt(I);
```

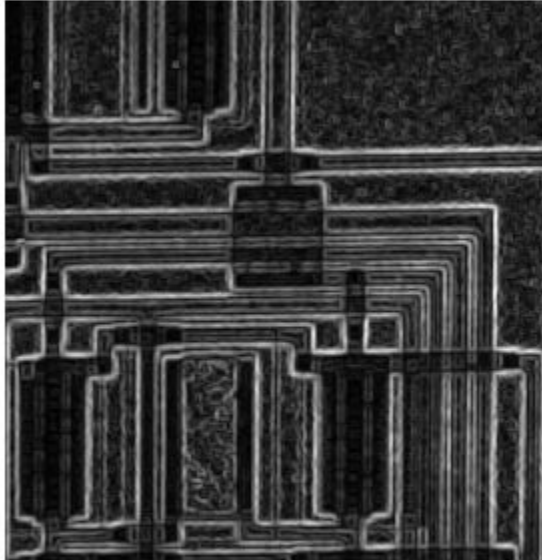
Show the original image and the processed image.

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



```
figure  
imshow(J,[])  
title('Result of Standard Deviation Filtering')
```

**Result of Standard Deviation Filtering**



### **Perform Standard Deviation Filtering on a GPU**

Read an images into a gpuArray.

```
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')
```

## Input Arguments

### **I** — Image to be filtered

numeric array | logical array | gpuArray

Image to be filtered, specified as a numeric or logical array of any dimension.

To perform the operation using a GPU, specify **I** as a **gpuArray** that contains a numeric or logical array.

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

### **nhood** — Neighborhood

`true(3)` (default) | multidimensional, logical, or numeric array containing zeros and ones

Neighborhood, specified as a multidimensional, logical, or numeric array containing zeros and ones. The size of **nhood** must be odd in each dimension.

By default, `stdfilt` uses the neighborhood `true(3)`. `stdfilt` 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 of the desired shape. Then extract the neighborhood from the `neighborhood` property of the structuring element.

---

**Note** When running this function on a GPU, the neighborhood must be 2-D.

---

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

## Output Arguments

### **J** — Filtered image

numeric array | gpuArray

Filtered image, returned as a numeric array of the same size as the input image `I`. The class of `J` is `double`.

If the filtered image is computed using a GPU, then `J` is returned as a `gpuArray` containing a numeric array of the same class as in `I`.

## Algorithms

If the image contains `Infs` or `NaNs`, then the behavior of `stdfilt` is undefined. Propagation of `Infs` or `NaNs` might not be localized to the neighborhood around the `Inf` or `NaN` pixel.

## Extended Capabilities

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The filtering neighborhood must be two-dimensional.

For more information, see “Image Processing on a GPU”.

## See Also

### Functions

`entropyfilt` | `getnhood` | `rangefilt` | `std2`

### Objects

`offsetstrel` | `strel`

### Topics

“What Is Image Filtering in the Spatial Domain?”



**Introduced before R2006a**

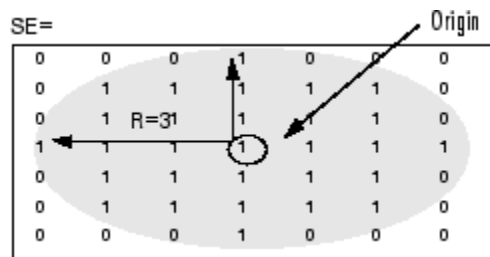
## strel

Morphological structuring element

### Description

A `strel` object represents a flat morphological structuring element, which is an essential part of morphological dilation and erosion operations.

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 (described below) 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.



To create a nonflat structuring element, use `offsetstrel`.

### Creation

### Syntax

```
SE = strel('diamond',r)
SE = strel('disk',r,n)
SE = strel('line',len,deg)
```

```
SE = strel('octagon',r)
SE = strel('rectangle',mn)
SE = strel('square',w)

SE = strel('cube',w)
SE = strel('cuboid',xyz)
SE = strel('sphere',r)

SE = strel('arbitrary',nhood)
```

## Description

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)`.

The following syntaxes still work, but `offsetstrel` is the preferred way to create these nonflat structuring element shapes:

- `SE = strel('arbitrary', nhood, h)`
- `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)`

## 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 1s and 0s. The location of the 1s 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 logical matrix.

Data Types: logical

### Dimensionality — Dimensions of structuring element

nonnegative scalar

Dimensions of structuring element, specified as a nonnegative scalar.

Data Types: double

## Object Functions

`decompose` Return sequence of decomposed structuring elements

`reflect` Reflect structuring element

`translate` Translate structuring element

## 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: [11x11 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: [7x7 logical]
Dimensionality: 2

```

View the structuring element.

```
SE.Neighborhood
```

```
ans = 7x7 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: [29x29 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)
```

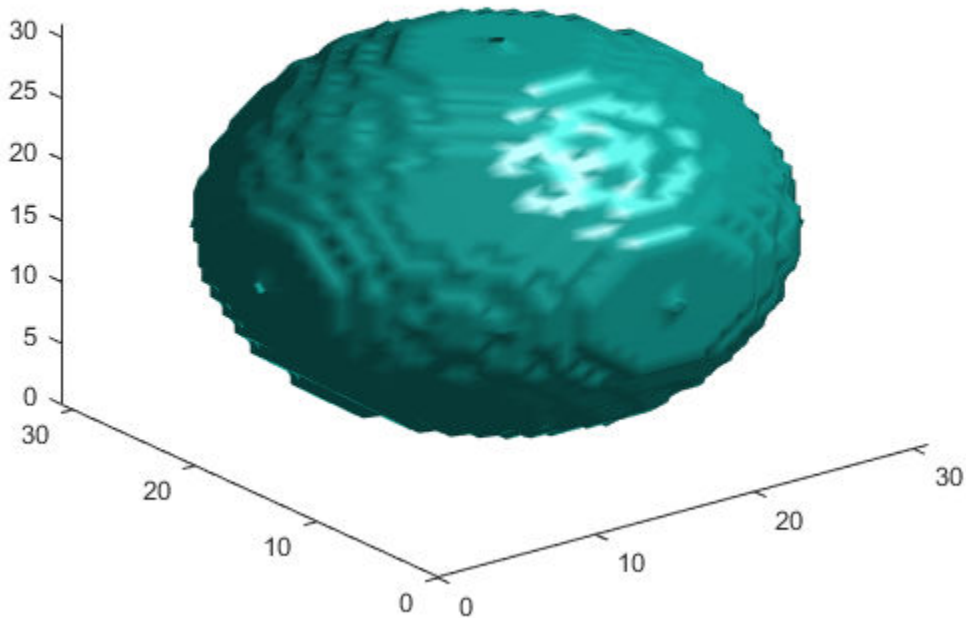
```
SE =
strel is a sphere shaped structuring element with properties:
```

```
Neighborhood: [31x31x31 logical]
Dimensionality: 3
```

Display the structuring element.

```
figure
isosurface(SE.Neighborhood)
```





## 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 approximation—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**

offsetstrel

## **Topics**

“Structuring Elements”

**Introduced before R2006a**

## stretchlim

Find limits to contrast stretch image

### Syntax

```
lowhigh = stretchlim(I)  
lowhigh = stretchlim(I,Tol)
```

### Description

`lowhigh = stretchlim(I)` computes the lower and upper limits that can be used for contrast stretching grayscale or RGB image `I`. The limits are returned in `lowhigh`. By default, the limits specify the bottom 1% and the top 1% of all pixel values.

You optionally can calculate the limits using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`lowhigh = stretchlim(I,Tol)` specifies the fraction, `Tol`, of the image to saturate at low and high pixel values.

### 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** — Image to be contrast stretched

2-D grayscale image | 2-D RGB image | gpuArray

Image to be contrast stretched, specified as a 2-D grayscale or RGB image.

To compute the limits using a GPU, specify **I** as a **gpuArray** that contains a 2-D grayscale or RGB image.

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

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

Example: `I = gpuArray(imread('pout.tif'));`

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

### **Tol** — Fraction of image to saturate

[0.01 0.99] (default) | numeric scalar | 2-element numeric vector

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>Tol</code> is a scalar, then <code>Low_Fract = Tol</code> , and <code>High_Fract = 1 - Low_Fract</code> , which saturates equal fractions at low and high pixel values.
0	If <code>Tol = 0</code> , then <code>lowhigh = [min(I(:)); max(I(:))]</code> .
Default	If you omit the <code>Tol</code> argument, then [ <code>Low_Fract High_Fract</code> ] defaults to [0.01 0.99], saturating 2%.
Too big	If <code>Tol</code> is too big, such that no pixels would be left after saturating low and high pixel values, then <code>stretchlim</code> returns [0 1].

Example: `lohi = stretchlim(I,[.02 .80]);`

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

## Output Arguments

### **lowhigh** — Lower and upper limits for contrast stretching

2-element numeric vector | 2-by-3 numeric matrix | gpuArray

Lower and upper limits for contrast stretching, returned as one of the following.

- A 2-element numeric vector when **I** is a grayscale image.
- A 2-by-3 numeric matrix when **I** is an RGB image. The columns indicate the lower and upper limit for each of the three color channels.

If the limits are computed using a GPU, then **lowhigh** is returned as a `gpuArray` containing a 2-element numeric vector or a 2-by-3 numeric array.

Data Types: `double`

## Tips

- Use the `imadjust` function to adjust the contrast of image **I** using the limits, **lowhigh**.

## 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 “Understand Code Generation with Image Processing Toolbox”.



## GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

## See Also

brighten | decorrstretch | histeq | imadjust

**Introduced before R2006a**

## subimage

Display multiple images in single figure

---

**Note** `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.

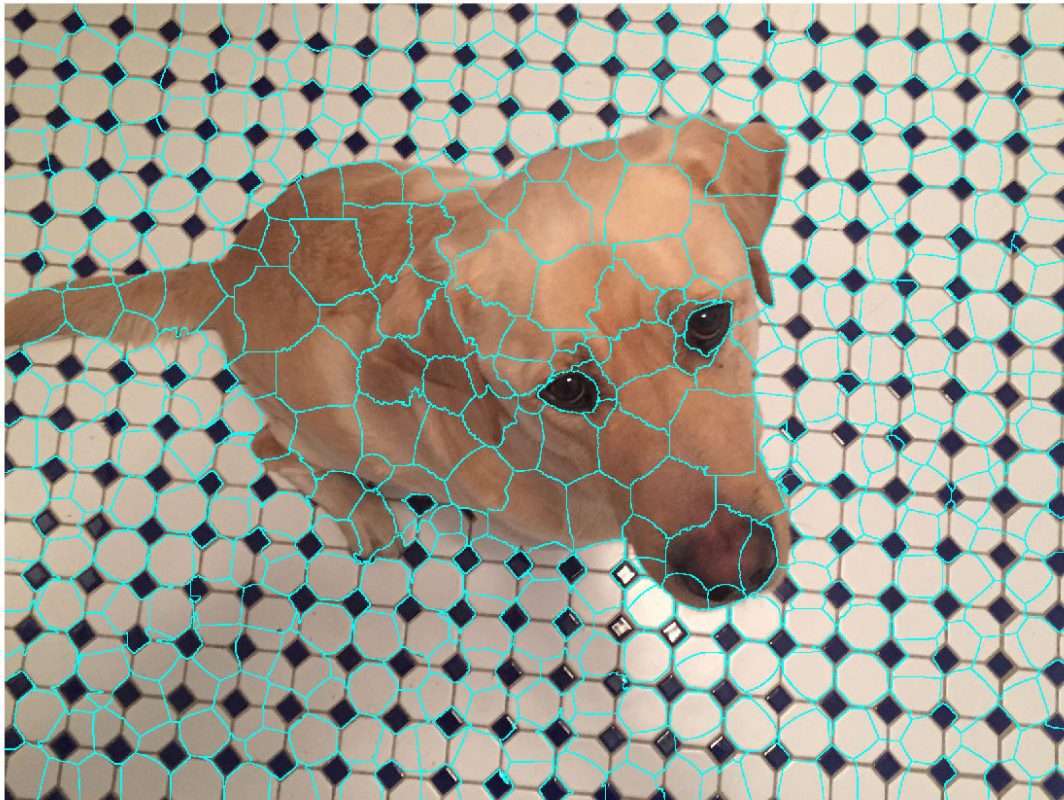
```
A = imread('kobi.png');
```

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)
```



## 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 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  $(0 \text{ Inf})$ . 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 'Compactness' adaptively after the first iteration. This is the default.
<code>'slic'</code>	'Compactness' is constant during clustering.

Data Types: `char` | `string`

### **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

[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.

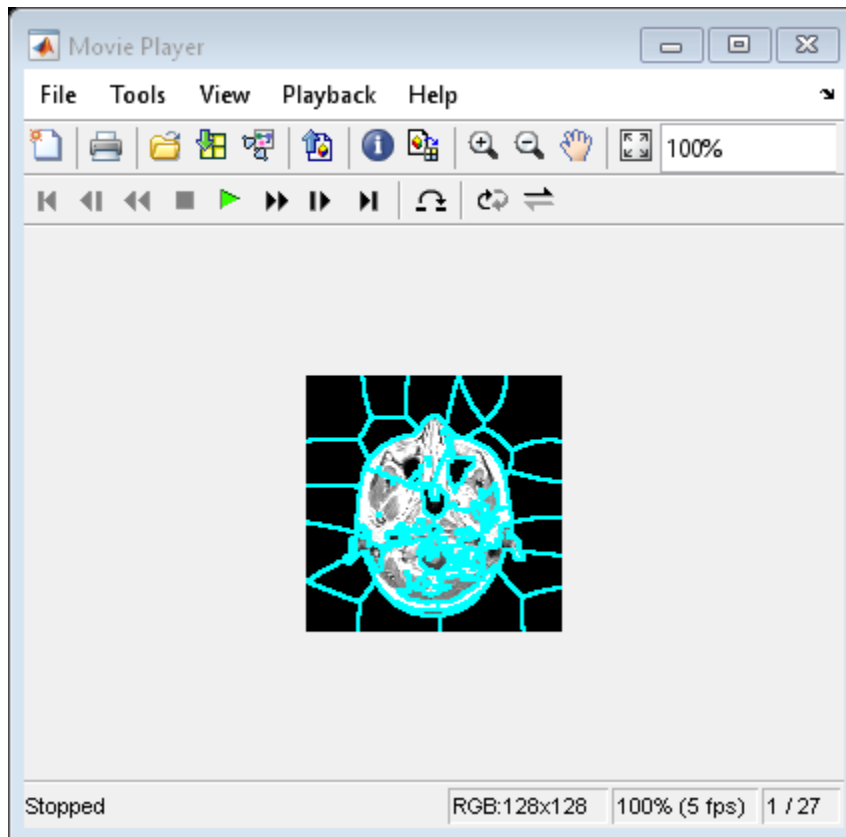
```
[L,N] = superpixels3(A,34);
```

Show all xy-planes progressively with superpixel boundaries.

```
imSize = size(A);
```

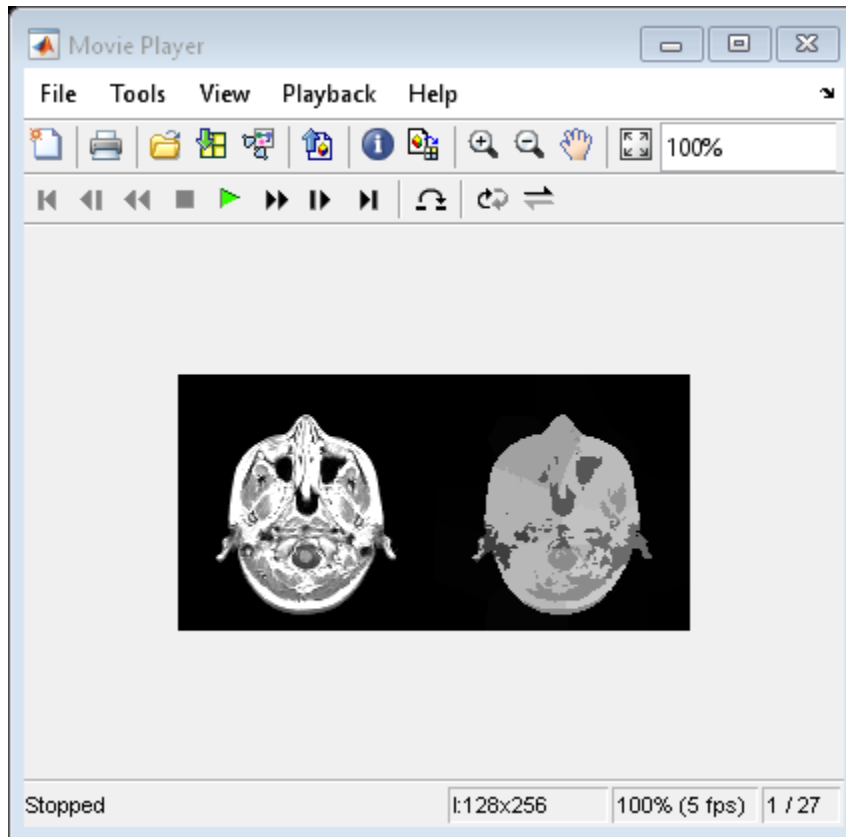
Create a stack of RGB images to display the boundaries in color.

```
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 `imshow` 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
imshow([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 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 1-2641.

Value	Meaning
<code>'slic0'</code>	<code>superpixels3</code> uses the <code>SLIC0</code> algorithm to refine <code>'Compactness'</code> adaptively after the first iteration. This is the default.

Value	Meaning
'slic'	'Compactness' is constant during clustering.

Data Types: char | string

### **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

`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

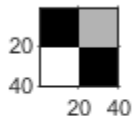
`B = tformarray(A,T,R,tdims_A,tdims_B,tsize_B,tmap_B,F)` applies a spatial transformation to array `A` to produce array `B`.

## Examples

### Transform Checkerboard Image

Create a 2-by-2 square checkerboard image where each square is 20 pixels wide. Display the image.

```
I = checkerboard(20,1,1);  
figure  
imshow(I)
```



Transform the checkerboard with a projective transformation. First create a spatial transformation structure.

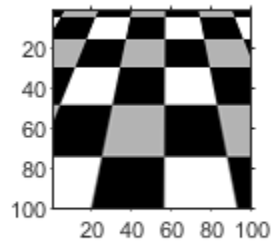
```
T = maketform('projective',[1 1; 41 1; 41 41; 1 41],...  
              [5 5; 40 5; 35 30; -10 30]);
```

Create a resampler. Use the pad method 'circular' when creating the resampler, so that the output appears to be a perspective view of an infinite checkerboard.

```
R = makesampler('cubic','circular');
```

Perform the transformation, specifying the transformation structure and the resampler. For this example, swap the output dimensions, and specify a 100-by-100 output image. Leave argument `tmap_B` empty since you specify argument `tsize_B`. Leave argument `F` empty since the fill value is not needed.

```
J = tformarray(I,T,R,[1 2],[2 1],[100 100],[],[]);  
figure  
imshow(J)
```



### **Transform Checkerboard Image, with Nonuniform Mapping from Input to Output Space**

Create a 2-by-2 square checkerboard image where each square is 20 pixels wide. Display the image.

```
I = checkerboard(20,1,1);  
figure  
imshow(I)
```



Transform the checkerboard with a projective transformation. First create a spatial transformation structure.

```
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 the resampler, so that the output appears to be a perspective view of an infinite checkerboard.

```
R = makesampler('cubic','circular');
```

Create arrays that specify the mapping of points from input space to output space. This example uses anisotropic sampling, where the distance between samples is larger in one direction than the other.

```
samp_x = 1:1.5:150;
samp_y = 1:100;
[x,y] = meshgrid(samp_x,samp_y);
tmap = cat(3,x,y);
size(tmap)
```

```
ans = 1x3
```

```
100 100 2
```

Note the size of `tmap`. The output image will have dimensions 100-by-100.

Perform the transformation, specifying the transformation structure and the resampler. Specify the output map as `tmap`. Leave argument `tsize_B` empty, since you specify argument `tmap_B`. The fill value does not matter since the resampler is circular.

```
J = tformarray(I,T,R,[1 2],[1 2],[],tmap,[]);
figure
imshow(J)
```



The length of checkerboard squares is larger in the  $y$ -direction than in the  $x$ -direction, which agrees with the larger sampling distance between points in the vector `samp_x`. Compared to the result using isotropic point mapping (see example “Transform Checkerboard Image” on page 1-2643), three additional columns of the checkerboard appear at the right of the transformed image, and no new rows are added to the transformed image.

## Input Arguments

### **A** — Input image

nonsparse numeric array

Input image, specified as a nonsparse numeric array. `A` can be real or complex.

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

### **T** — Spatial transformation

TFORM spatial transformation structure

Spatial transformation, specified as a TFORM spatial transformation structure. You typically use the `maketform` 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, then `tformarray` operates as a direct resampling function. Further, if `tmap_B` is:

- Not empty, then `tformarray` applies the resampler defined in `R` to compute values at each transform space location defined in `tmap_B`
- Empty, then `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. The entries need not be listed in increasing order, but the order matters. The order specifies the precise correspondence between dimensions of arrays `A` and `B` and the input and output spaces of the transformation `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 row and column dimensions of `A` correspond to the second and first 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 — Output transform dimensions**

row vector of finite, positive integers

Output 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 10]`.

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` is

$$[D_1 \ D_2 \ D_3 \ \dots \ D_N \ L]$$

where N equals `length(tdims_B)`. `tsize_B` should be `[]`.

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
- 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

numeric array or scalar

Fill values, specified as a numeric array or scalar. 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'`.

When `F` is:

- A scalar (including NaN), its value is replicated across all the nontransform dimensions.
- Nonscalar, its 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, `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**

numeric array

Transformed image, returned as a numeric array.

## 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,...,X_ndims_out] = tformfwd(T,U)
X = tformfwd(T,U1,U2,...,U_ndims_in)
```

## Description

`[X,Y] = tformfwd(T,U,V)` applies the 2D-to-2D forward spatial transformation defined in `T` to coordinate arrays `U` and `V`, mapping the point  $[U(k) \ V(k)]$  to the point  $[X(k) \ Y(k)]$ .

Both `T.ndims_in` and `T.ndims_out` must equal 2. `U` and `V` are typically column vectors, but they can have any dimensionality. `X` and `Y` are the same size as `U` and `V`.

`[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`. The transformation maps the point  $[U1(k) \ U2(k) \ \dots \ U\_ndims\_in(k)]$  to the point  $[X1(k) \ X2(k) \ \dots \ X\_ndims\_out(k)]$ .

The number of input coordinate arrays, `ndims_in`, must equal `T.ndims_in`. The number of output coordinate arrays, `ndims_out`, must equal `T.ndims_out`. The arrays `U1,U2,...,U_ndims_in` can have any dimensionality, but must be the same size. The output arrays `X1,X2,...,X_ndims_out` must be this size also.

`X = tformfwd(T,U)` applies the spatial transformation defined in `T` to coordinate array `U`.

- When `U` is a 2-D matrix with dimensions `m-by-ndims_in`, `X` is a 2-D matrix with dimensions `m-by-ndims_out`. `tformfwd` applies the `ndims_in`-to-`ndims_out` transformation to each row of `U`. `tformfwd` maps the point  $U(k, :)$  to the point  $X(k, :)$ .

- When  $U$  is an  $(N+1)$ -dimensional array, `tformfwd` maps the point  $U(k_1, k_2, \dots, k_N, :)$  to the point  $X(k_1, k_2, \dots, k_N, :)$ .

`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  $I = 1, \dots, N$ , 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,...,X_ndims_out] = tformfwd(T,U)` maps one  $(N+1)$ -dimensional array to `ndims_out` equally sized  $N$ -dimensional arrays.

`X = tformfwd(T,U1,U2,...,U_ndims_in)` maps `ndims_in`  $N$ -dimensional arrays to one  $(N+1)$ -dimensional array.

## Examples

### Create an Affine Transformation and Validate It with Forward 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 `tformfwd`. The results should equal  $x$  and  $y$ .

```
[xm, ym] = tformfwd(tform, u, v)
```

```
xm = 3×1
```

```
    -1  
     0  
     4
```

```
ym = 3×1
```

-1  
-10  
4

## Input Arguments

### **T** — Spatial transformation

TFORM spatial transformation structure

Spatial transformation, specified as a TFORM spatial transformation structure. Create T using `maketform`, `fliptform`, or `cp2tform`.

Data Types: `struct`

### **U** — Input coordinate points

numeric array

Input coordinate points, specified as a numeric array. The size and dimensionality of U can have additional limitations depending on the syntax used.

Data Types: `double`

### **V** — Input coordinate points

numeric array

Input coordinate points, specified as a numeric array. V must be the same size as U.

Data Types: `double`

### **U1,U2,...,U\_ndims\_in** — Input coordinate points

multiple numeric arrays

Input coordinate points, specified as multiple numeric arrays. The size and dimensionality of U1,U2,...,U\_ndims\_in can have additional limitations depending on the syntax used.

Data Types: `double`

## Output Arguments

### **X — Coordinate array of output points**

numeric array

Coordinate array of output points, returned as a numeric array. The size and dimensionality of X can have additional limitations depending on the syntax used.

### **Y — Coordinate array of output points**

numeric array

Coordinate array of output points, returned as a numeric array. Y is the same size as V.

### **X1,X2,...,X\_ndims\_out — Coordinates of output points**

multiple numeric arrays

Coordinates of output points, returned as multiple numeric arrays. The size and dimensionality of X1,X2,...,X\_ndims\_out can have additional limitations depending on the syntax used.

## See Also

`cp2tform` | `fliptform` | `maketform` | `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_out)
U = tforminv(T,X)
[U1,U2,...,U_ndims_in] = tforminv(T,X)
U = tforminv(T,X1,X2,...,X_ndims_in)
```

### Description

`[U,V] = tforminv(T,X,Y)` applies the 2D-to-2D inverse spatial transformation defined in `T` to coordinate arrays `X` and `Y`, mapping the point  $[X(k) \ Y(k)]$  to the point  $[U(k) \ V(k)]$ .

Both `T.ndims_in` and `T.ndims_out` must equal 2. `X` and `Y` are typically column vectors, but they can have any dimensionality. `U` and `V` are the same size as `X` and `Y`.

`[U1,U2,...,U_ndims_in] = tforminv(T,X1,X2,...,X_ndims_out)` applies the `ndims_out`-to-`ndims_in` inverse transformation defined in `T` to the coordinate arrays `X1,X2,...,X_ndims_out`. The transformation maps the point  $[X1(k) \ X2(k) \ \dots \ X\_ndims\_out(k)]$  to the point  $[U1(k) \ U2(k) \ \dots \ U\_ndims\_in(k)]$ .

The number of input coordinate arrays, `ndims_out`, must equal `T.ndims_out`. The number of output coordinate arrays, `ndims_in`, must equal `T.ndims_in`. The arrays `X1,X2,...,X_ndims_out` can have any dimensionality, but must be the same size. The output arrays `U1,U2,...,U_ndims_in` must be this size also.

`U = tforminv(T,X)` applies the `ndims_out`-to-`ndims_in` inverse transformation defined in `T` to array `X`.

- When `X` is a 2-D matrix with dimensions `m`-by-`ndims_out` matrix, `U` is a 2-D matrix with dimensions `m`-by-`ndims_in`. `tforminv` applies the transformation to to each row of `X`. `tforminv` maps the point  $X(k, :)$  to the point  $U(k, :)$ .

- When  $X$  is an  $(N+1)$ -dimensional array, `tforminv` maps the point  $X(k_1, k_2, \dots, k_N, :)$  to the point  $U(k_1, k_2, \dots, k_N, :)$ .

`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  $I = 1, \dots, N$ , and `size(U,N+1)` equal to `ndims_in`.

The syntax `U = tforminv(X,T)` is an older form of this syntax that remains supported for backward compatibility.

`[U1,U2,...,U_ndims_in] = tforminv(T,X)` maps one  $(N+1)$ -dimensional array to `ndims_in` equally sized  $N$ -dimensional arrays.

`U = tforminv(T,X1,X2,...,X_ndims_in)` maps `ndims_out`  $N$ -dimensional arrays to one  $(N+1)$ -dimensional array.

## Examples

### Create an Affine Transformation and Validate It with 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`. The results should equal `u` and `v`.

```
[um, vm] = tforminv(tform, x, y)
```

```
um = 3×1
```

```
    0  
 6.0000  
-2.0000
```

```
vm = 3×1
```

```

    0
    3.0000
    5.0000

```

## Input Arguments

### **T** — Spatial transformation

TFORM spatial transformation structure

Spatial transformation, specified as a TFORM spatial transformation structure. Create T using `maketform`, `fliptform`, or `cp2tform`.

Data Types: `struct`

### **X** — Input coordinate points

numeric array

Input coordinate points, specified as a numeric array. The size and dimensionality of X can have additional limitations depending on the syntax used.

Data Types: `double`

### **Y** — Input coordinate points

numeric array

Input coordinate points, specified as a numeric array. Y must be the same size as X.

Data Types: `double`

### **X1,X2,...,X\_ndims\_out** — Input coordinate points

multiple numeric arrays

Input coordinate points, specified as multiple numeric arrays. The size and dimensionality of X1,X2,...,X\_ndims\_out can have additional limitations depending on the syntax used.

Data Types: `double`

## Output Arguments

### **U — Coordinate array of output points**

numeric array

Coordinate array of output points, returned as a numeric array. The size and dimensionality of U can have additional limitations depending on the syntax used.

### **V — Coordinate array of output points**

numeric array

Coordinate array of output points, returned as a numeric array. V is the same size as Y.

### **U1,U2,...,U\_ndims\_in — Coordinates of output points**

multiple numeric arrays

Coordinates of output points, returned as multiple arrays. The size and dimensionality of U1,U2,...,U\_ndims\_in can have additional limitations depending on the syntax used.

## See Also

`cp2tform` | `fliptform` | `maketform` | `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)` uses name-value pairs 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 as 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 quotes.

You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `RGB = tonemap(HDR, 'AdjustLightness', [0.05 0.95]);`

## **AdjustLightness — Overall lightness of the rendered image**

2-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) | 2-element vector of positive integers

Number of tiles used during the adaptive histogram equalization part of the tone mapping operation, specified as a 2-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**

`adapthisteq` | `hdrread` | `stretchlim` | `tonemapfarbman`

**Introduced in R2007b**

## tonemapfarbman

Convert HDR image to LDR using edge-preserving multiscale decompositions

### Syntax

```
LDR = tonemapfarbman(HDR)
LDR = tonemapfarbman(HDR, Name, Value)
```

### Description

`LDR = tonemapfarbman(HDR)` converts the high dynamic range (HDR) image to a low dynamic range (LDR) image, suitable for display, using a process called edge-preserving decompositions for multiscale tone and detail manipulation.

`LDR = tonemapfarbman(HDR, Name, Value)` uses one or more name-value pairs to control various aspects of the tone mapping.

### Examples

#### Compress Dynamic Range of HDR Image Using Edge-Preserving Multiscale Decompositions

Load a high dynamic range (HDR) image into the workspace. Convert the HDR image to a low dynamic range (LDR) image using the basic tone mapping function `tonemap`. Display the result.

```
HDR = hdrread('office.hdr');
LDR = tonemap(HDR);
imshow(LDR)
```



The LDR image is displayed with an acceptable dynamic range, but colors are muted.

Convert the HDR image to an LDR image using edge-preserving multiscale decompositions. Display the result.

```
RGB = tonemapfarbman(HDR);  
imshow(RGB)
```



Colors appear more saturated, but the image is bright and has poor contrast. This brightness and contrast indicate that the default value of 'Exposure' is too large. The poor contrast is especially evident in the shadow of the tree.

Repeat the conversion with a lower value of 'Exposure' to darken the image. Display the result.

```
RGB2 = tonemapfarbman(HDR, 'Exposure', 1.5);  
imshow(RGB2)
```





The image contrast is improved. Also, the image shows a decrease in the clipping of pixel values in bright regions, such as the sky, road, and monitor.

## Input Arguments

### **HDR — High dynamic range image**

*m*-by-*n* matrix | *m*-by-*n*-by-3 array

High dynamic range image, specified as an *m*-by-*n* matrix or 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `LDR = tonemapFarbman(HDR, 'Saturation', 2.1);`

### **RangeCompression — Range compression**

0.3 (default) | number in the range [0, 1]

Range compression, specified as the comma-separated pair consisting of `'RangeCompression'` and a number in the range [0, 1]. A value of 1 represents maximum compression and a value of 0 represents minimum compression.

### **Saturation — Saturation**

1.6 (default) | nonnegative number

Saturation, specified as the comma-separated pair consisting of `'Saturation'` and a nonnegative number. The recommended range for `'Saturation'` is [0, 5]. As the saturation value increases, colors become more rich and intense. As the saturation value decreases, colors fade away to gray. The `'Saturation'` argument does not affect grayscale HDR images.

### **Exposure — Exposure**

3 (default) | positive number

Exposure, specified as the comma-separated pair consisting of `'Exposure'` and a positive number. The recommended range for `'Exposure'` is (0, 5]. As this value decreases, the exposure length decreases, so the image darkens. As this value increases, the exposure length increases, so the image brightens.

### **NumberOfScales — Number of scales**

3 (default) | positive integer

Number of scales, specified as the comma-separated pair consisting of `'NumberOfScales'` and a positive integer. The recommended range for `'NumberOfScales'` is [1, 5]. The default number of scales is `length(Weights)` when you specify `'Weights'`. Otherwise, the default number of scales is 3.

### **Weights — Weights of detail layers**

[1.5 1.5 1.5] (default) | *n*-element vector of positive numbers

Weights of detail layers, specified as the comma-separated pair consisting of 'Weights' and an  $n$ -element vector of positive numbers, where  $n$  is the number of scales specified by 'NumberOfScales'. The recommended range of each element in `Weights` is (0, 3]. The default value of 'Weights' is an  $n$ -element numeric vector with all elements set to 1.5. For `Weights` <1, the amount of detail in the output image decreases and `Weights` >1, the amount of detail in the output image increases.

## Output Arguments

### LDR — Low dynamic range image

numeric array

Low dynamic range image, specified as a numeric array of the same size as HDR.

Data Types: `uint8`

## Tips

- This function uses an anisotropic diffusion filter, `imdiffusefilt`, for the approximation of the weighted least squares filter, as proposed by Farbman et al. [1]

## References

[1] Farbman, Z., R. Fattal, D. Lischinski, and R. Szeliski. "Edge-Preserving Decompositions for Multi-Scale Tone and Detail Manipulation." *ACM Transactions on Graphics*. Vol. 27, Number 3, August 2008, pp. 1-10.

## See Also

`hdrread` | `imdiffusefilt` | `locallapfilt` | `localtonemap` | `makehdr` | `tonemap`

**Introduced in R2018b**

## transformPointsForward

Apply forward geometric transformation

### Syntax

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

### Description

`[x,y] = transformPointsForward(tform,u,v)` applies the forward transformation of 2-D geometric transformation `tform` to the points specified by coordinates `u` and `v`.

`[x,y,z] = transformPointsForward(tform,u,v,w)` applies the forward transformation of 3-D geometric transformation `tform` to the points specified by coordinates `u`, `v`, and `w`.

`X = transformPointsForward(tform,U)` applies the forward transformation of `tform` to the input coordinate matrix `U` and returns the coordinate matrix `X`. `transformPointsForward` maps the  $k$ th point `U(k,:)` to the point `X(k,:)`.

### Examples

#### Apply Forward Transformation of 2-D 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
```

### Transform Coordinate Arrays Using Custom 2-D Transformation

Specify the x- and y-coordinates vectors of five points to transform.

```
x = [10 11 15 2 2];
y = [15 32 34 7 10];
```

Define the inverse and forward mapping functions. Both functions accept and return points in packed (x,y) format.

```
inversefn = @(c) [c(:,1).^2,sqrt(c(:,2))];
forwardfn = @(c) [sqrt(c(:,1)),c(:,2).^2];
```

Create a 2-D geometric transform object, `tform`, that stores the inverse mapping function and the optional forward mapping function.

```
tform = geometricTransform2d(inversefn,forwardfn)
```

tform =

```
geometricTransform2d with properties:
```

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

Apply the inverse geometric transform to the input points.

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

```
u = 1×5
```

```
    100    121    225     4     4
```

```
v = 1×5
```

```
    3.8730    5.6569    5.8310    2.6458    3.1623
```

Apply the forward geometric transform to the transformed points u and v.

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

```
x = 1×5
```

```
    10    11    15     2     2
```

```
y = 1×5
```

```
    15.0000    32.0000    34.0000     7.0000    10.0000
```

### **Apply Forward Transformation of 3-D Geometric Transformation**

Create an `affine3d` object that defines the transformation.

```
tform = affine3d([3 1 2 0;4 5 8 0;6 2 1 0;0 0 0 1])
```

```
tform =
```

```
    affine3d with properties:
```

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

Apply forward transformation of 3-D geometric transformation to an input (u,v,w) point.

```
[X,Y,Z] = transformPointsForward(tform,2,3,5)
```

```

X =
    48

Y =
    27

Z =
    33

```

### Transform Coordinate Arrays Using Custom 3-D Transformation

Specify the x-, y- and the z-coordinate vectors of five points to transform.

```

x = [3 5 7 9 11];
y = [2 4 6 8 10];
z = [5 9 13 17 21];

```

Define the inverse and forward mapping functions that accept and return points in packed (x,y,z) format.

```

inverseFcn = @(c)[c(:,1).^2,c(:,2).^2,c(:,3).^2];
forwardFcn = @(c)[sqrt(c(:,1)),sqrt(c(:,2)),sqrt(c(:,3))];

```

Create a 3-D geometric transformation object, `tform`, that stores these inverse and forward mapping functions.

```

tform = geometricTransform3d(inverseFcn,forwardFcn)

tform =
    geometricTransform3d with properties:

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

```

Apply the inverse transformation of this 3-D geometric transformation to the input points.

```

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

```

$u = 1 \times 5$

9    25    49    81    121

$v = 1 \times 5$

4    16    36    64    100

$w = 1 \times 5$

25    81    169    289    441

Apply the forward geometric transform to the transformed points  $u$ ,  $v$ , and  $w$ .

$[x, y, z] = \text{transformPointsForward}(tform, u, v, w)$

$x = 1 \times 5$

3    5    7    9    11

$y = 1 \times 5$

2    4    6    8    10

$z = 1 \times 5$

5    9    13    17    21

## Input Arguments

**tform** — Geometric transformation

geometric transformation object

Geometric transformation, specified as a geometric transformation object.

For 2-D geometric transformations, **tform** is an `affine2d`, `projective2d`, or `geometricTransform2d` geometric transformation object.



For 3-D geometric transformations, `tform` is an `affine3d` object or `geometricTransform3d` geometric transformation object.

**u — x-coordinates of points to be transformed**

*m-by-n* or *m-by-n-by-p* numeric array

x-coordinates of points to be transformed, specified as an *m-by-n* or *m-by-n-by-p* numeric array. The number of dimensions of `u` matches the dimensionality of `tform`.

Data Types: `single` | `double`

**v — y-coordinates of points to be transformed**

*m-by-n* or *m-by-n-by-p* numeric array

y-coordinates of points to be transformed, specified as an *m-by-n* or *m-by-n-by-p* numeric array. The size of `v` must match the size of `u`.

Data Types: `single` | `double`

**w — z-coordinates of points to be transformed**

*m-by-n-by-p* numeric array

z-coordinates of points to be transformed, specified as an *m-by-n-by-p* numeric array. `w` is used only when `tform` is a 3-D geometric transformation. The size of `w` must match the size of `u`.

Data Types: `single` | `double`

**U — Coordinates of points to be transformed**

*l-by-2* or *l-by-3* numeric array

Coordinates of points to be transformed, specified as an *l-by-2* or *l-by-3* numeric array. The number of columns of `U` matches the dimensionality of `tform`.

The first column lists the x-coordinate of each point to transform, and the second column lists the y-coordinate. If `tform` represents a 3-D geometric transformation, `U` has size *l-by-3* and the third column lists the z-coordinate of the points to transform.

Data Types: `single` | `double`

## Output Arguments

### **x** — x-coordinates of points after transformation

*m*-by-*n* or *m*-by-*n*-by-*p* numeric array

x-coordinates of points after transformation, returned as an *m*-by-*n* or *m*-by-*n*-by-*p* numeric array. The number of dimensions of **x** matches the dimensionality of `transform`.

Data Types: `single` | `double`

### **y** — y-coordinates of points after transformation

*m*-by-*n* or *m*-by-*n*-by-*p* numeric array

y-coordinates of points after transformation, returned as an *m*-by-*n* or *m*-by-*n*-by-*p* numeric array. The size of **y** matches the size of **x**.

Data Types: `single` | `double`

### **z** — z-coordinates of points after transformation

*m*-by-*n*-by-*p* numeric array

z-coordinates of points after transformation, returned as an *m*-by-*n*-by-*p* numeric array. The size of **z** matches the size of **x**.

Data Types: `single` | `double`

### **X** — Coordinates of points after transformation

numeric array

Coordinates of points after transformation, returned as a numeric array. The size of **X** matches the size of **U**.

The first column lists the x-coordinate of each point after transformation, and the second column lists the y-coordinate. If `transform` represents a 3-D geometric transformation, the third column lists the z-coordinate of the points after transformation.

Data Types: `single` | `double`

## See Also

`imwarp` | `transformPointsInverse`

**Introduced in R2013a**

## transformPointsInverse

### Package:

Apply inverse geometric transformation

### Syntax

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

### Description

`[u,v] = transformPointsInverse(tform,x,y)` applies the inverse transformation of 2-D geometric transformation `tform` to the points specified by coordinates `x` and `y`.

`[u,v,w] = transformPointsInverse(tform,x,y,z)` applies the inverse transformation of 3-D geometric transformation `tform` to the points specified by coordinates `x`, `y`, and `z`.

`U = transformPointsInverse(tform,X)` applies the inverse transformation of `tform` to the input coordinate matrix `X` and returns the coordinate matrix `U`. `transformPointsInverse` maps the  $k$ th point  $X(k,:)$  to the point  $U(k,:)$ .

### Examples

#### Apply Inverse Transformation of 2-D 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 transformation of 2-D geometric transformation to an input point.

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

```
X =  
    6.6605
```

```
Y =  
    8.9798
```

Apply inverse transformation of 2-D 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
```

### Transform Packed Coordinates Using Custom 2-D Transformation

Specify the packed (x,y) coordinates of five input points. The packed coordinates are stored in a 5-by-2 matrix, where the x-coordinate of each point is in the first column, and the y-coordinate of each point is in the second column.

```
XY = [10 15;11 32;15 34;2 7;2 10];
```

Define the inverse mapping function. The function accepts and returns points in packed (x,y) format.

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

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

Create a 2-D geometric transform object, `tform`, that stores the inverse mapping function.

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

```
tform =  
    geometricTransform2d with properties:  
        InverseFcn: @(c)[c(:,1)+c(:,2),c(:,1)-c(:,2)]  
        ForwardFcn: []  
        Dimensionality: 2
```

Apply the inverse geometric transform to the input points.

```
UV = transformPointsInverse(tform,XY)
```

```
UV = 5×2
```

```
    25    -5  
    43   -21  
    49   -19  
     9    -5  
    12    -8
```

### **Apply Inverse Transformation of 3-D Geometric Transformation**

Create an `affine3d` object that defines the transformation.

```
tform = affine3d([3 1 2 0;4 5 8 0;6 2 1 0;0 0 0 1])
```

```
tform =  
    affine3d with properties:
```

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

Apply forward transformation of 3-D geometric transformation to an input point.

```
[X,Y,Z] = transformPointsForward(tform,2,3,5)
```

```
X =
```

```
48
```

```
Y =
```

```
27
```

```
Z =
```

```
33
```

Apply inverse transformation of 3-D geometric transformation to output point from the previous step to recover the original coordinates.

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

```
U =
```

```
2.0000
```

```
V =
```

```
3
```

```
W =
```

```
5.0000
```

### Transform Packed Coordinates Using Custom 3-D Transformation

Specify the packed  $(x,y,z)$  coordinates of five input points. The packed coordinates are stored as a 5-by-3 matrix, where the first, second, and third columns contain the  $x$ -,  $y$ -, and  $z$ - coordinates, respectively.

```
XYZ = [5 25 20;10 5 25;15 10 5;20 15 10;25 20 15];
```

Define an inverse mapping function that accepts and returns points in packed  $(x,y,z)$  format.

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

Create a 3-D geometric transformation object, `tform`, that stores this inverse mapping function.

```
tform = geometricTransform3d(inverseFcn)
```

```
tform =  
    geometricTransform3d with properties:  
        InverseFcn: @(c)[c(:,1)+c(:,2),c(:,1)-c(:,2),c(:,3).^2]  
        ForwardFcn: []  
        Dimensionality: 3
```

Apply the inverse transformation of this 3-D geometric transformation to the input points.

```
UVW = transformPointsInverse(tform,XYZ)
```

```
UVW = 5×3
```

```
    30    -20    400  
    15     5    625  
    25     5     25  
    35     5    100  
    45     5    225
```

## Input Arguments

**tform** — Geometric transformation

geometric transformation object



Geometric transformation, specified as a geometric transformation object.

For 2-D geometric transformations, `tform` is an `affine2d`, `projective2d`, `geometricTransform2d`, `LocalWeightedMeanTransformation2D`, `PiecewiseLinearTransformation2D`, or `PolynomialTransformation2D` geometric transformation object.

For 3-D geometric transformations, `tform` is an `affine3d` object or `geometricTransform3d` geometric transformation object.

### **x — x-coordinates of points to be transformed**

*m-by-n* or *m-by-n-by-p* numeric array

x-coordinates of points to be transformed, specified as an *m-by-n* or *m-by-n-by-p* numeric array. The number of dimensions of `x` matches the dimensionality of `tform`.

Data Types: `single` | `double`

### **y — y-coordinates of points to be transformed**

*m-by-n* or *m-by-n-by-p* numeric array

y-coordinates of points to be transformed, specified as an *m-by-n* or *m-by-n-by-p* numeric array. The size of `y` must match the size of `x`.

Data Types: `single` | `double`

### **z — z-coordinates of points to be transformed**

*m-by-n-by-p* numeric array

z-coordinates of points to be transformed, specified as an *m-by-n-by-p* numeric array. `z` is used only when `tform` is a 3-D geometric transformation. The size of `z` must match the size of `x`.

Data Types: `single` | `double`

### **X — Coordinates of points to be transformed**

*l-by-2* or *l-by-3* numeric array

Coordinates of points to be transformed, specified as an *l-by-2* or *l-by-3* numeric array. The number of columns of `X` matches the dimensionality of `tform`.

The first column lists the x-coordinate of each point to transform, and the second column lists the y-coordinate. If `tform` represents a 3-D geometric transformation, `X` has size *l-by-3* and the third column lists the z-coordinate of the points to transform.

Data Types: `single` | `double`

## Output Arguments

### **u** — x-coordinates of points after transformation

*m*-by-*n* or *m*-by-*n*-by-*p* numeric array

x-coordinates of points after transformation, returned as an *m*-by-*n* or *m*-by-*n*-by-*p* numeric array. The number of dimensions of **u** matches the dimensionality of `tform`.

Data Types: `single` | `double`

### **v** — y-coordinates of points after transformation

*m*-by-*n* or *m*-by-*n*-by-*p* numeric array

y-coordinates of points after transformation, returned as an *m*-by-*n* or *m*-by-*n*-by-*p* numeric array. The size of **v** matches the size of **u**.

Data Types: `single` | `double`

### **w** — z-coordinates of points after transformation

*m*-by-*n*-by-*p* numeric array

z-coordinates of points after transformation, returned as an *m*-by-*n*-by-*p* numeric array. The size of **w** matches the size of **u**.

Data Types: `single` | `double`

### **U** — Coordinates of points after transformation

numeric array

Coordinates of points after transformation, returned as a numeric array. The size of **U** matches the size of **X**.

The first column lists the x-coordinate of each point after transformation, and the second column lists the y-coordinate. If `tform` represents a 3-D geometric transformation, the third column lists the z-coordinate of the points after transformation.

Data Types: `single` | `double`

## See Also

`imwarp` | `transformPointsForward`

**Introduced in R2013a**

## translate

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.

### 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**



### **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: [11x11 double]  
 Dimensionality: 2
```

```
SE.Offset
```

```
ans = 11x11
```

```

-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    1.0
0      1.6246    2.4369    3.2492    4.0615    4.0615    4.0615    3.2492    2.4
0.8123    2.4369    3.2492    4.0615    4.8738    4.8738    4.8738    4.0615    3.2
1.6246    3.2492    4.0615    4.8738    5.6861    5.6861    5.6861    4.8738    4.0
2.4369    3.2492    4.0615    4.8738    5.6861    6.4984    5.6861    4.8738    4.0
1.6246    3.2492    4.0615    4.8738    5.6861    5.6861    5.6861    4.8738    4.0
0.8123    2.4369    3.2492    4.0615    4.8738    4.8738    4.8738    4.0615    3.2
0      1.6246    2.4369    3.2492    4.0615    4.0615    4.0615    3.2492    2.4
-Inf      0.8123    1.6246    2.4369    3.2492    3.2492    3.2492    2.4369    1.0
:

```

Translate the structuring element.

```

V = [2 2];
SE2 = translate(SE,V)

```

```

SE2 =
offsetstrel is a ball shaped offset structuring element with properties:

```

```

        Offset: [15x15 double]
Dimensionality: 2

```

SE2.Offset

```

ans = 15x15

```

```

-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    1.0
-Inf      -Inf      -Inf      -Inf      -Inf      0.8123    1.6246    2.4369    3.2
-Inf      -Inf      -Inf      -Inf      0      1.6246    2.4369    3.2492    4.0
-Inf      -Inf      -Inf      -Inf      0.8123    2.4369    3.2492    4.0615    4.8
-Inf      -Inf      -Inf      -Inf      1.6246    3.2492    4.0615    4.8738    5.6
-Inf      -Inf      -Inf      -Inf      2.4369    3.2492    4.0615    4.8738    5.6
:

```

## Input Arguments

### **SE — Structuring element**

`strel` or `offsetstrel` object

Structuring element, specified as a `strel` or `offsetstrel` object.

### **v — Translation offsets**

numeric vector

Translation offsets, specified as a numeric vector. Each element specifies the amount of desired translation in the corresponding dimension.

## Output Arguments

### **SE2 — Translated structuring element**

`strel` or `offsetstrel` object

Translated structuring elements, returned as a `strel` or `offsetstrel` object.

## See Also

`reflect`

**Introduced before R2006a**



# truesize

Adjust display size of image

## Syntax

```
truesize(fig,[mrows ncols])  
truesize(fig)
```

## Description

`truesize(fig,[mrows ncols])` adjusts the display size of an image in a figure, `fig`, to the dimensions `[mrows ncols]`, in pixels.

`truesize(fig)` adjusts the display size such that each image pixel covers one screen pixel.

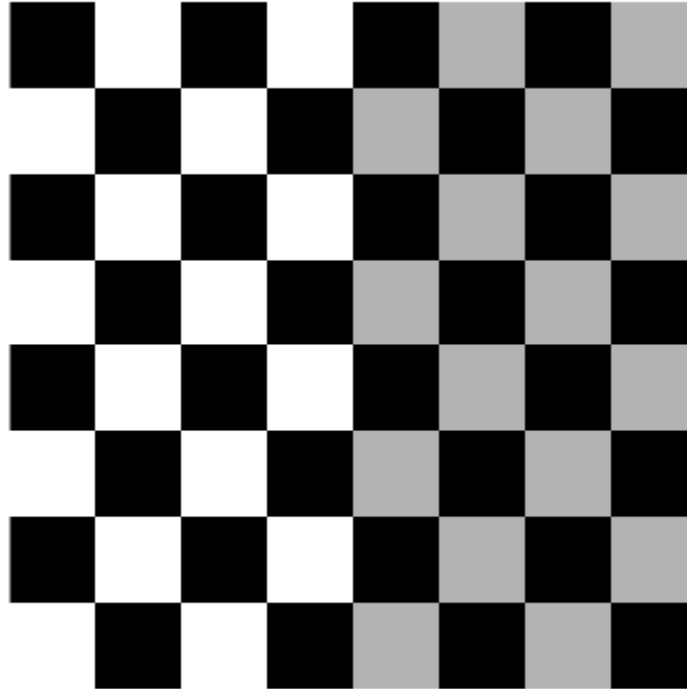
If you do not specify a figure, `truesize` adjusts the display size of the current figure.

## Examples

### Adjust Display Size of Image

Create a default checkerboard image, which has size 80-by-80 pixels. Display the checkerboard image to fill the full size of the figure window. The image is magnified to fill the window.

```
c = checkerboard;  
imshow(c,'InitialMagnification','fit')
```



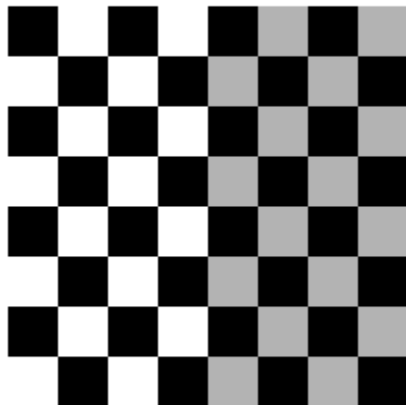
Display the checkerboard image so that each image pixel covers one screen pixel.

`trueSize`



You can adjust the size of the figure window to arbitrary dimension. The image scales to fit within the figure window.

```
truesize([300 200]);
```



## Input Arguments

### **fig** — Figure

gcf (default) | figure handle

Figure containing a single image or a single image with a colorbar, specified as a figure handle. By default, `truesize` uses the current figure, with handle `gcf`.

### **[mrows ncols]** — Screen dimensions

2-element numeric row vector

Screen dimensions (in pixels) that the image should occupy, specified as a 2-element numeric row vector. By default, `[mrows ncols]` is equal to the image size, so each image pixel covers one screen pixel.

## See Also

`imshow` | `iptgetpref` | `iptsetpref`

**Introduced before R2006a**

# visboundaries

Plot region boundaries

## Syntax

```
visboundaries(BW)  
visboundaries(B)  
visboundaries(ax, ___)  
visboundaries(___,Name,Value)  
h = visboundaries(___)
```

## 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`.

`visboundaries(___,Name,Value)` passes the name-value pair arguments to specify additional properties of the boundaries. Parameter names can be abbreviated.

`h = visboundaries(___)` returns a handle `h`, for the boundaries.

## Examples

### Compute Boundaries and Plot on Image

Read image.

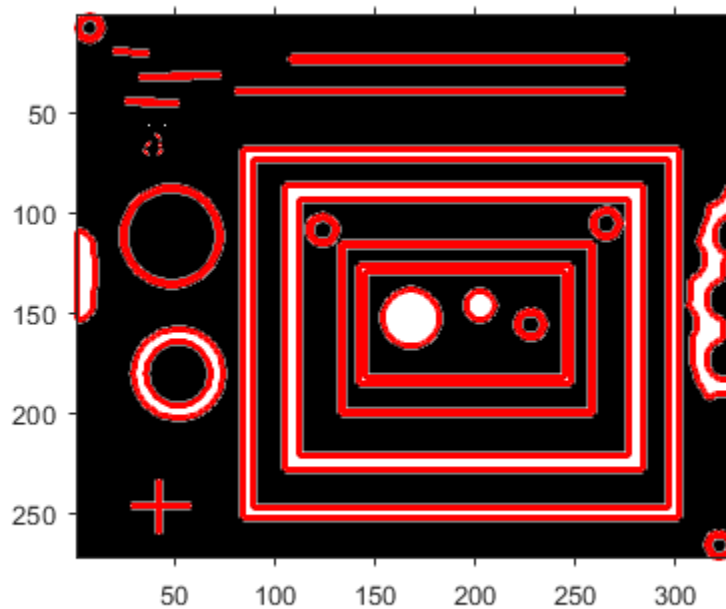
```
BW = imread('blobs.png');
```

Compute boundaries.

```
B = bwboundaries(BW);
```

Display image and plot boundaries on image.

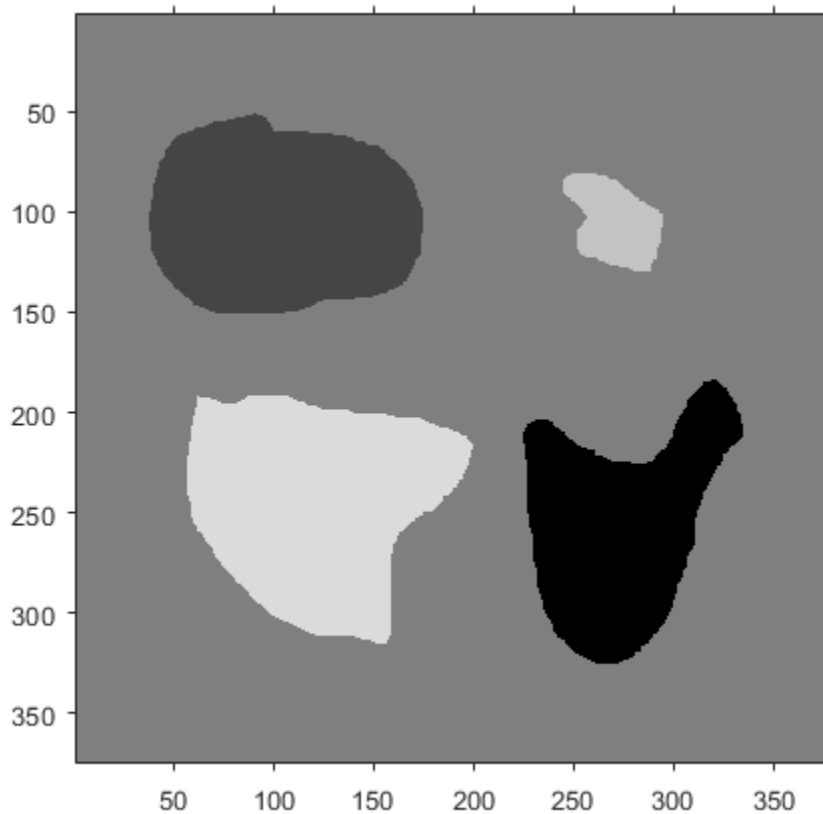
```
imshow(BW)  
hold on  
visboundaries(B)
```



## Visualize Segmentation Result

Read image and display it.

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

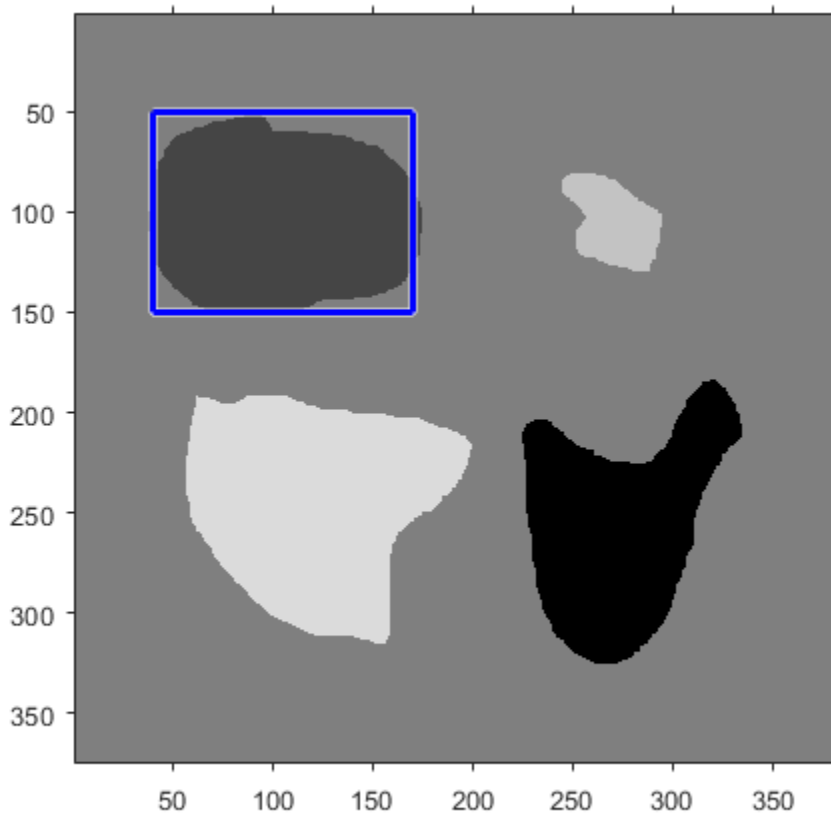


Segment the image using the active contours (snakes) algorithm. First, specify the initial contour location close to the object that is to be segmented.

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

Display the initial contour on the original image in blue.

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



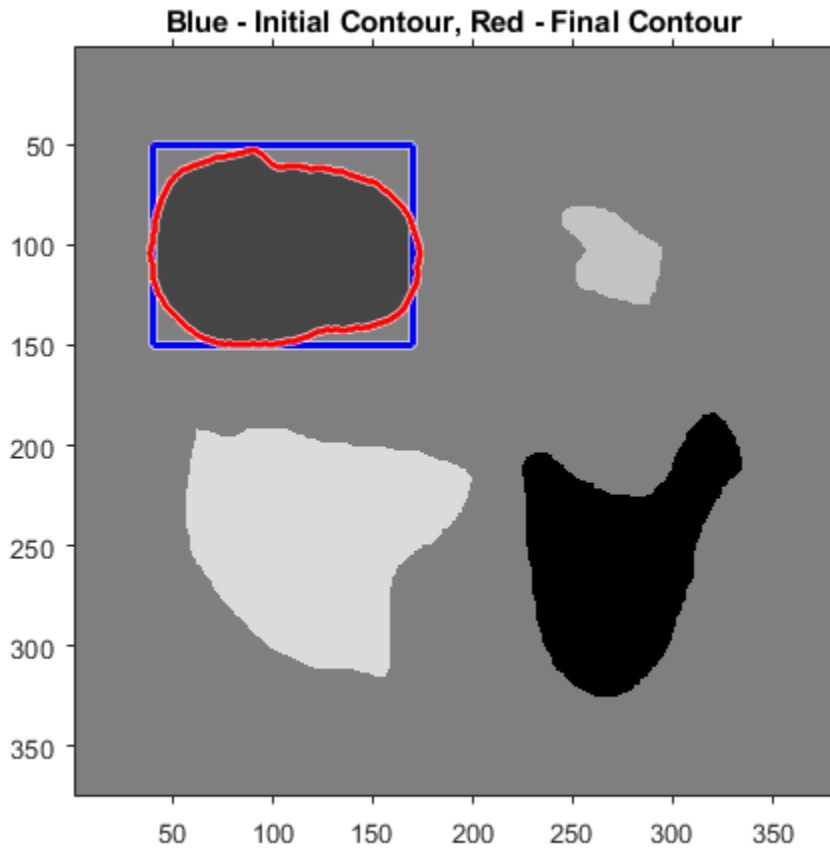
Segment the image using the 'edge' method using 200 iterations.

```
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');
```



## 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 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**

'red' (default) | [R G B] | short name | long name

Color of the boundary, specified as a MATLAB `ColorSpec`.

Example: `visboundaries(bw, 'Color', 'r');`





Example: `visboundaries(bw, 'Color', 'green');`

Example: `visboundaries(bw, 'Color', [0 0 1]);`

## **LineStyle — Style of boundary line**

'-' (default) | '--' | ':' | '-.' | 'none'

Line style of boundary 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

Example: `visboundaries(bw, 'LineStyle', '-.');`

### **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**

### **h** — Boundary lines

hggroup object

Boundary line, returned as an hggroup object. `h` is the child of the axes `ax` if specified, otherwise `h` is the child of the current axes.

## **See Also**

`bwboundaries` | `bwperim` | `bwtraceboundary` | `viscircles`

**Introduced in R2015a**

# viscircles

Create circle

## Syntax

```
viscircles(centers, radii)  
viscircles(ax, centers, radii)  
viscircles( ____, Name, Value)  
h = viscircles( ____ )
```

## 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`.

`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.

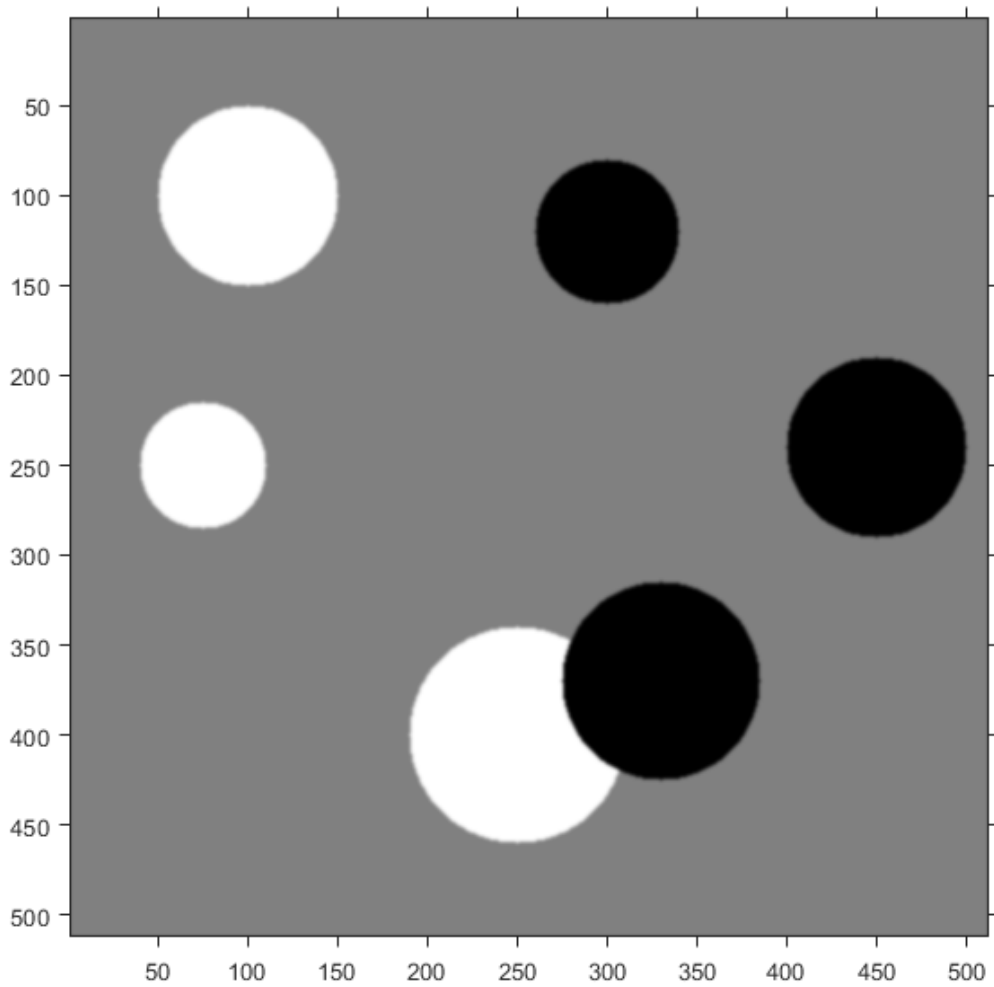
`h = viscircles( ____ )` returns a handle, `h`, to the drawn circles.

## Examples

### Draw Lines Around Bright and Dark Circles in 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.

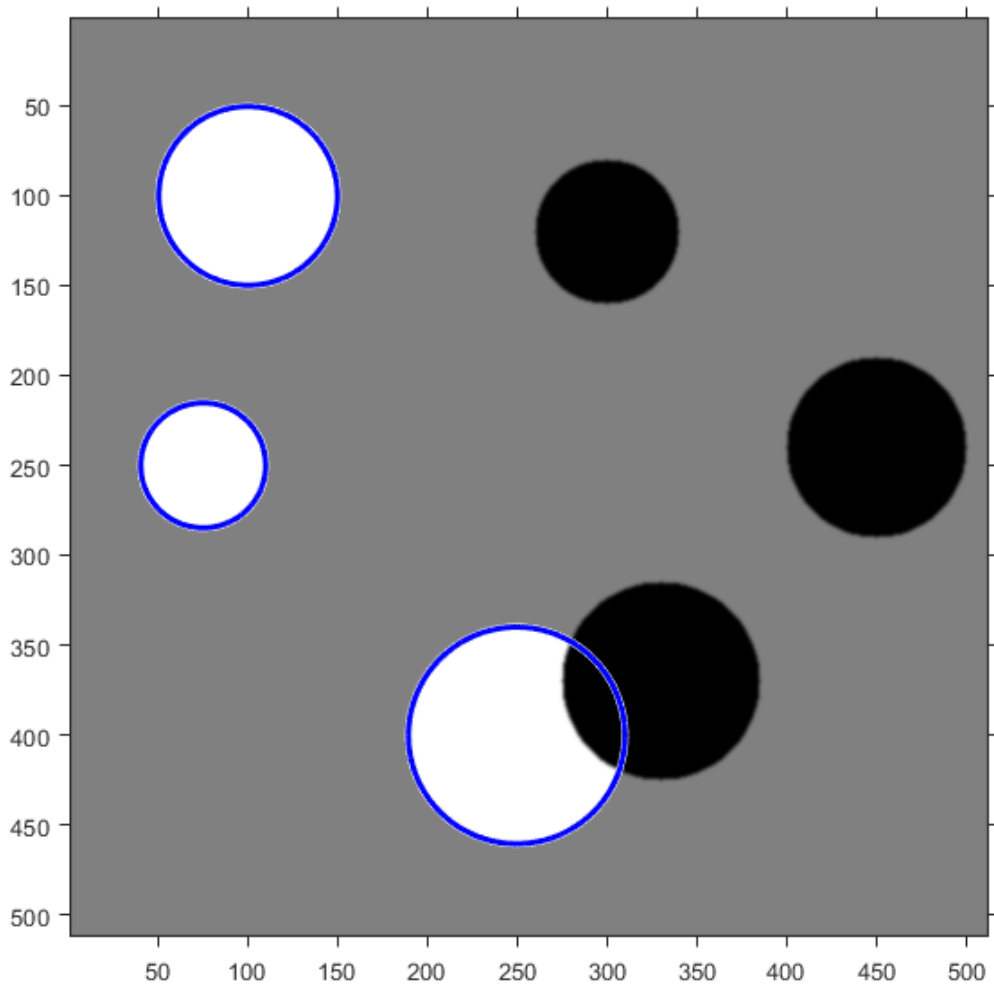
```
[centersBright, radiiBright] = imfindcircles(A,[Rmin Rmax], 'ObjectPolarity', 'bright');
```

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.

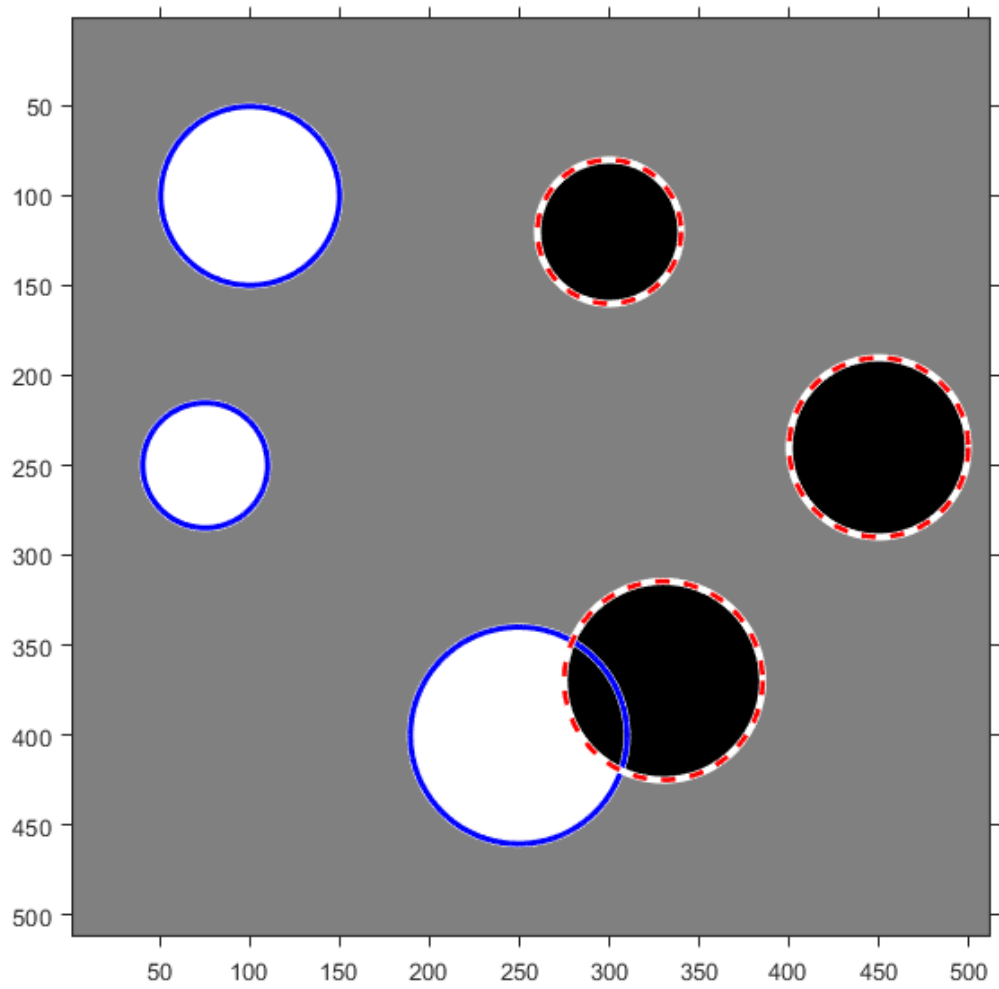
```
viscircles(centersBright, radiiBright, 'Color', 'b');
```



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

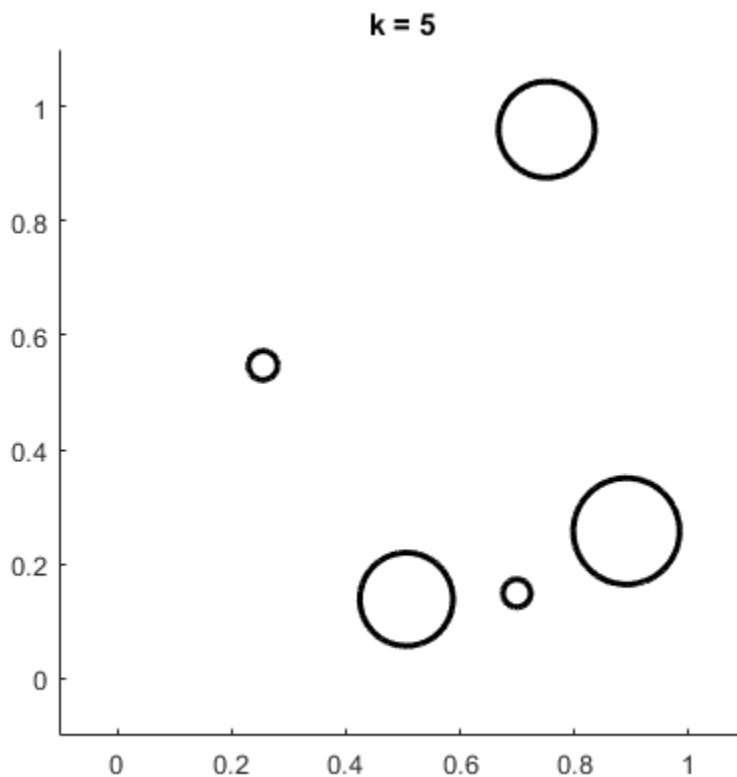
    % Fix the axis limits.
    xlim([-0.1 1.1])
    ylim([-0.1 1.1])

    % 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 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`.

Example: `viscircles(centers, radii, 'Color', 'r');`

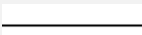
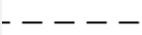
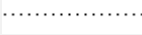

Example: `viscircles(centers,radii,'Color','green');`

Example: `viscircles(centers,radii,'Color',[0 0 1]);`

### LineStyle — Line style of circle edge

'-' (default) | '--' | ':' | '-.' | 'none'

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

hggroup object

Circles drawn, returned as an hggroup object. h is the child of the axes ax if specified, otherwise h is the child of the current axes.

## See Also

`imdistline` | `imfindcircles` | `imtool` | `visboundaries`

**Introduced in R2012a**

# volshow

Display volume

## Syntax

```
vol = volshow(V)  
vol = volshow(V,Name,Value)
```

## Description

`vol = volshow(V)` displays the 3-D grayscale volume *V* and returns a `volshow` object, `vol`. Use `vol` to modify the display settings after the volume is displayed. You can also rotate and zoom the display interactively by using the mouse.

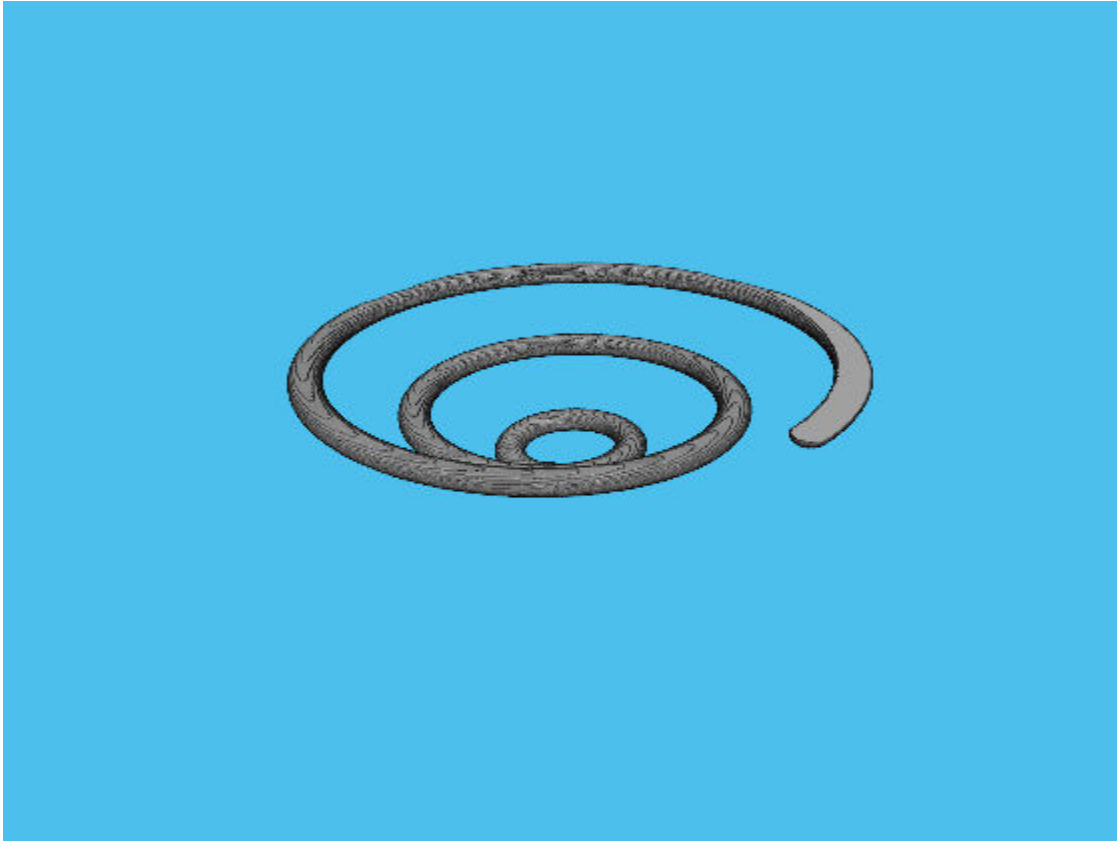
`vol = volshow(V,Name,Value)` adjusts the initial display settings using name-value pairs.

## Examples

### Display Animation of Rotating Volume

Load a volume into the workspace and view it using `volshow`.

```
load('spiralVol.mat');  
h = volshow(spiralVol);
```



Create an array of camera positions around the unit circle.

```
vec = linspace(0,2*pi(),120)';  
myPosition = [cos(vec) sin(vec) ones(size(vec))];
```

Update the display at each camera position.

```
framerate = 24;  
for idx = 1:120  
    h.CameraPosition = myPosition(idx,:);  
    pause(1/framerate)  
end
```



## Visualize Volume of MRI Data

Load MRI data and remove the singleton dimension.

```
load mri
V = squeeze(D);
```

Generate a color map and transparency (alpha) map suited for MRI images.

```
intensity = [0 20 40 120 220 1024];
alpha = [0 0 0.15 0.3 0.38 0.5];
color = ([0 0 0; 43 0 0; 103 37 20; 199 155 97; 216 213 201; 255 255 255]) ./ 255;
queryPoints = linspace(min(intensity),max(intensity),256);
alphamap = interp1(intensity,alpha,queryPoints)';
colormap = interp1(intensity,color,queryPoints);
```

View the volume with the custom color map and transparency map. Click and drag the mouse to rotate the volume. Use the scroll wheel to zoom in and out of the volume.

```
vol = volshow(V, 'Colormap', colormap, 'Alphamap', alphamap);
```



## Visualize Volume of CT Data

This example uses 3-D volumetric human chest CT scan data. To run this example, you must download the sample data using the Add-Ons Explorer. Select **Get Add-ons** from the **Add-ons** drop-down menu from the MATLAB® desktop. The support file is in the **MathWorks Features** section. You must have write privileges for the installation folder.

Load the data into the MATLAB workspace.

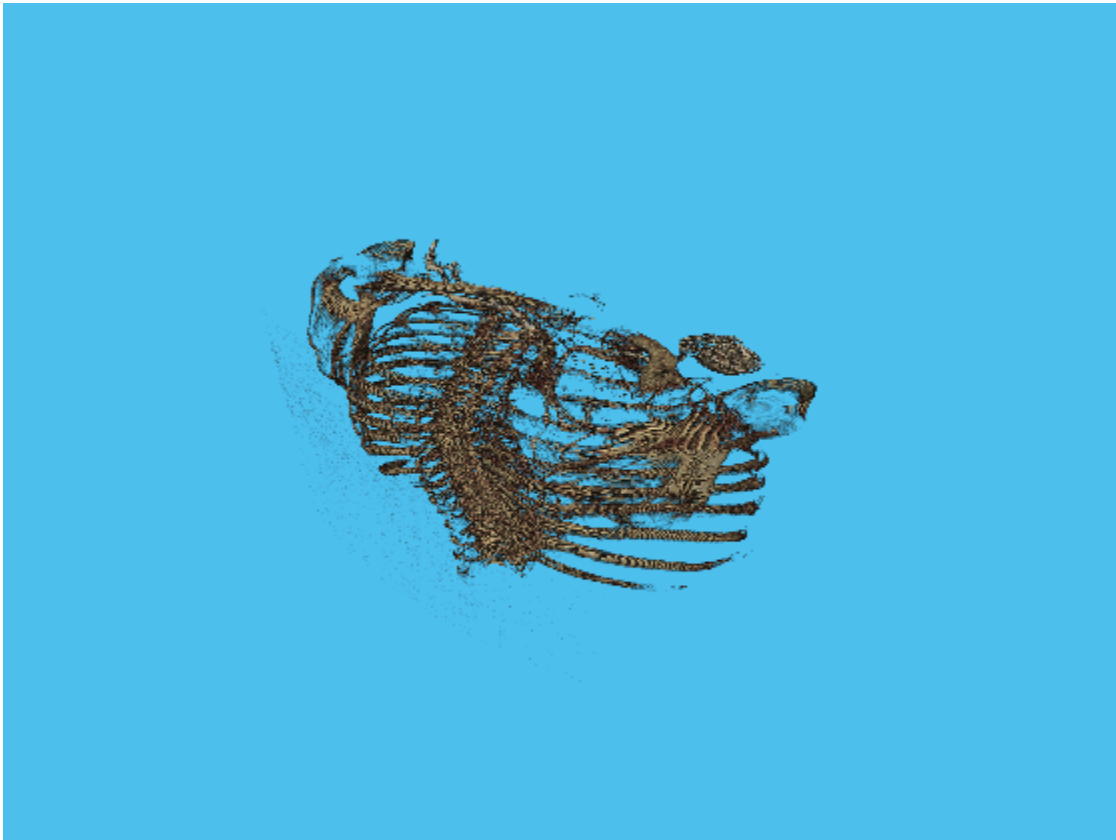
```
load chestVolume
```

Generate a color map and transparency (alpha) map suited for CT images.

```
intensity = [-3024, -16.45, 641.38, 3071];  
alpha = [0, 0, 0.72, 0.72];  
color = ([0 0 0; 186 65 77; 231 208 141; 255 255 255]) ./ 255;  
queryPoints = linspace(min(intensity),max(intensity),256);  
alphamap = interp1(intensity,alpha,queryPoints)';  
colormap = interp1(intensity,color,queryPoints);
```

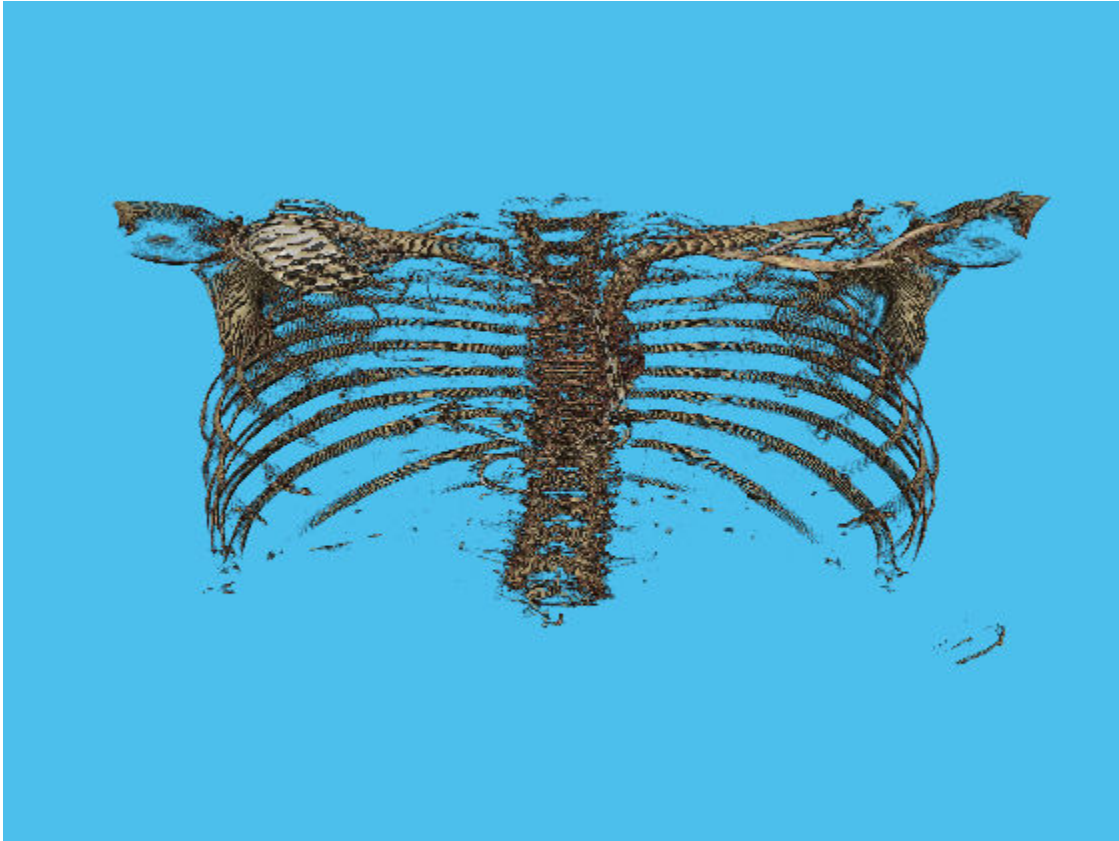
View the volume with the custom color map and transparency map. Click and drag the mouse to rotate the volume. Use the scroll wheel to zoom in and out of the volume.

```
vol = volshow(V1, 'Colormap', colormap, 'Alphamap', alphamap);
```



Programmatically change the camera position to see a different view of the volume.

```
vol.CameraPosition = [-1.0533 -0.0093, 0.1593];
```



## Input Arguments

### **V** — 3-D grayscale image

*m*-by-*n*-by-*p* numeric array

3-D grayscale image, specified as an *m*-by-*n*-by-*p* numeric array.

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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `vol = volshow(V, 'Colormap', cmap, 'Alphamap', amap);`

### Parent — Parent of the volshow object

`gcf` (default) | panel object | figure

Parent container in which to plot, specified as a handle to a figure or panel object. If you do not specify a parent, then `volshow` uses the handle to the current figure, `gcf`. If a figure does not exist, then `volshow` creates a new figure.

### Colormap — Colormap of the volume content

`gray(256)` (default) | 256-by-3 numeric array

Colormap of the volume content, specified as a 256-by-3 numeric array, with values in the range `[0 1]`. The default colormap is `gray(256)`.

Data Types: double

### Alphamap — Transparency map for the volume content

`linspace(0,1,256)'` | 256-by-1 numeric vector

Transparency map for the volume content, specified as a 256-by-1 numeric vector with values in the range `[0 1]`. The default value of `Alphamap` is `linspace(0,1,256)'`.

Data Types: double

### BackgroundColor — Background color

`[0.3 0.75 0.93]` (default) | `ColorSpec`

Background color, specified as a MATLAB `ColorSpec`.

Data Types: single | double

### CameraPosition — Camera viewpoint

`[1 1 1]` (default) | 3-element numeric vector

Camera viewpoint, specified as a 3-element numeric vector of the form `[x y z]`. Changing the `CameraPosition` property changes the point from which you view the

volume. The camera is oriented along the view axis, which is a straight line that connects the camera position and the camera target. This property updates when you interactively rotate the volume.

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

### **CameraUpVector — Vector defining upwards direction**

`[0 0 1]` (default) | 3-element numeric vector

Vector defining upwards direction, specified as a 3-element numeric vector of the form `[x y z]`. This property updates when you interactively rotate the volume.

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

### **CameraTarget — Point used as camera target**

`[0 0 0]` (default) | 3-element numeric vector

Point used as camera target, specified as a 3-element numeric vector of the form `[x y z]`. The camera is oriented along the view axis, which is a straight line that connects the camera position and the camera target.

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

### **Renderer — Rendering style**

`'Isosurface'` | `'VolumeRendering'` | `'MaximumIntensityProjection'`

Rendering style, specified as one of the following values.

Value	Description
<code>'Isosurface'</code>	View an isosurface of the volume specified by the value in <code>Isovalue</code> .
<code>'MaximumIntensityProjection'</code>	View the voxel with the highest intensity value for each ray projected through the data.
<code>'VolumeRendering'</code>	View the volume based on the specified color and transparency for each voxel.

If volume `V` is a numeric data type, then the default rendering style is `'VolumeRendering'`. If `V` is logical, then the default rendering style is `'Isosurface'`.

**IsoSurfaceColor — IsoSurface color**`[1 0 0]` (default) | `ColorSpec`

IsoSurface color, specified as a MATLAB `ColorSpec`. This property specifies the volume color when `Renderer` is set to `'Isosurface'`.

**IsoValue — Isovalue**`0.49` (default) | numeric scalar in the range `[0 1]`

Isovalue, specified as a numeric scalar in the range `[0 1]`. The isovalue specifies which surface of the volume to draw when `Renderer` is set to `'Isosurface'`.

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

**InteractionsEnabled — Display is interactive**`true` (default) | `false`

Display is interactive, specified as `true` or `false`. When `true` (default), you can zoom using the mouse scroll wheel, and rotate by clicking and dragging on the volume. Rotation and zoom is performed about the value specified by `CameraTarget`. When `false`, you cannot interact with the volume.

**ScaleFactors — Scale factors**`[1 1 1]` (default) | 1-by-3 vector of positive numbers

Scale factors used to rescale the volume in the  $x$ ,  $y$ , and  $z$  directions, specified as a 1-by-3 vector of positive numbers.

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

## Tips

- The `volshow` function creates a panel object in the specified parent figure. Panels are containers that group UI components together. The volumetric data is displayed in the panel.

## See Also

**Volume Viewer**

**Introduced in R2018b**



## 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.

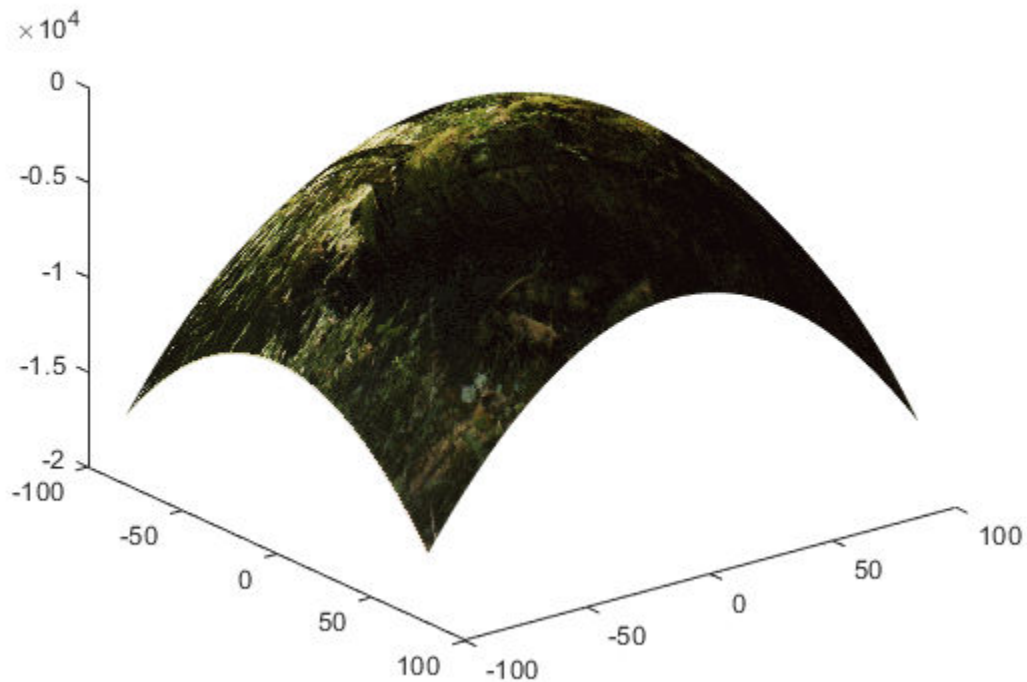
```
[X,Y] = meshgrid(-100:100,-80:80);
```

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.

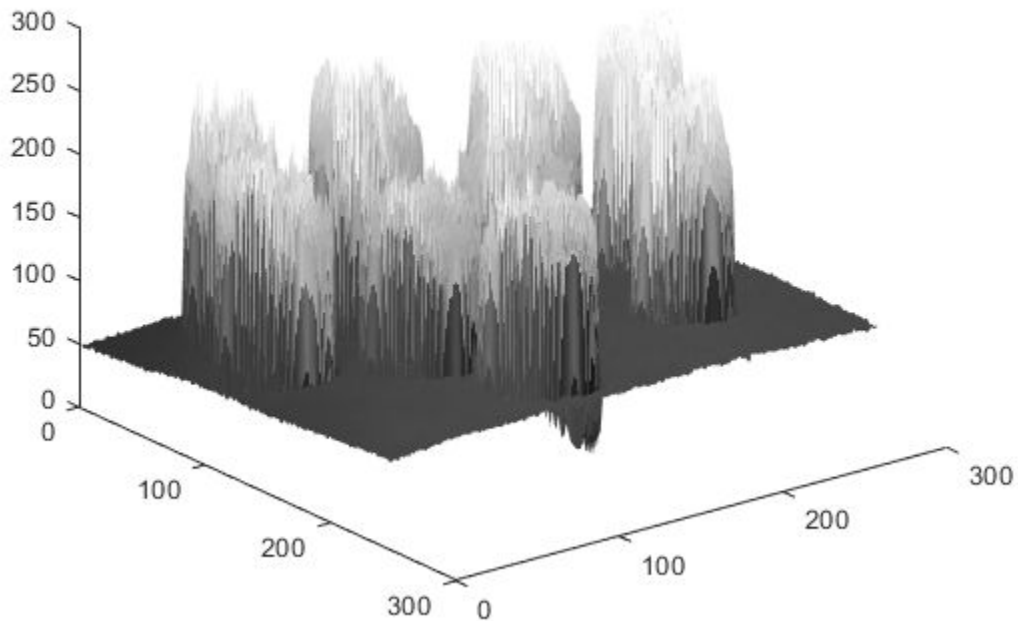
### **Warp Grayscale Image Based on Intensity**

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**

`image` | `imagesc` | `imshow` | `surf`

**Introduced before R2006a**

# images.geotrans.Warper

Apply same geometric transformation to many images efficiently

## Description

A Warper object applies an `affine2d` or `projective2d` geometric transformation to images with a specific size.

## Creation

## Syntax

```
w = images.geotrans.Warper(tform, inputSize)
w = images.geotrans.Warper(tform, inputRef)
w = images.geotrans.Warper(tform, inputRef, outputRef)
w = images.geotrans.Warper(sourceX, sourceY)
w = images.geotrans.Warper( ____, Name, Value)
```

## Description

`w = images.geotrans.Warper(tform, inputSize)` creates an image warper from the geometric transformation object `tform` and sets the `ImageSize` property.

`w = images.geotrans.Warper(tform, inputRef)` specifies the coordinate system of the input images, `inputRef`.

`w = images.geotrans.Warper(tform, inputRef, outputRef)` specifies the coordinate system of the output image, `outputRef`. This syntax can be used to improve performance by limiting the application of the geometric transformation to a specific output region of interest.

`w = images.geotrans.Warper(sourceX, sourceY)` specifies the input image coordinates, `sourceX` and `sourceY`, required to perform the geometric transformation.

`w = images.geotrans.Warper( ____, Name, Value)` sets the Interpolation and FillValue properties using one or more name-value pair arguments. Enclose each property name in single quotes.

For example, `warper = images.geotrans.Warper(tform,size(im),'FillValue',1)` specifies a fill value of 1 for pixels outside the original image.

## Input Arguments

### **tform — Geometric transformation**

affine2d object | projective2d object

Geometric transformation, specified as an affine2d or projective2d geometric transformation object.

### **inputRef — Referencing object associated with input image**

imref2d object

Referencing object associated with the input image, specified as an imref2d spatial referencing object.

### **outputRef — Referencing object associated with output image**

imref2d object

Referencing object associated with the output image, specified as an imref2d spatial referencing object.

### **sourceX, sourceY — Input image coordinates**

2-D matrix

Input image coordinates, specified as a 2-D matrix the same size as the required output image. Each (x, y) index in sourceX and sourceY specifies the location in the input image for the corresponding output pixel.

Data Types: single

## Properties

### **InputSize — Size of the input images**

2-element vector of positive integers | 3-element vector of positive integers



Size of the input images, specified as a 2- or 3-element vector of positive integers.

### **OutputSize — Size of the first two dimensions of the output image**

2-element vector of positive integers

Size of the first two dimensions of the output image, specified as a 2-element vector of positive integers.

### **Interpolation — Interpolation method**

'linear' (default) | 'nearest' | 'cubic'

Interpolation method, specified as 'linear', 'nearest', or 'cubic'.

Data Types: char | string

### **FillValue — Value used for output pixels outside the input image boundaries**

0 (default) | numeric scalar

Value used for output pixels outside the input image boundaries, specified as a numeric scalar. Warper casts the fill value to the data type of the input image.

## **Object Functions**

warp Apply geometric transformation

## **Examples**

### **Apply Shear to Multiple Images**

Pick a set of images of the same size. The example uses a set of images that show cells.

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

Create a geometric transform to rotate each image by 45 degrees and to shrink each image.

```
tform = affine2d([ 0.5*cos(pi/4) sin(pi/4) 0;
                  -sin(pi/4) 0.5*cos(pi/4) 0;
                  0 0 1]);
```

Create a Warper object, specifying the geometric transformation object, `tform`, and the size of the input images.

```
im = readimage(imds,1);  
warper = images.geotrans.Warper(tform,size(im));
```

Determine the number of images to be processed and preallocate the output array.

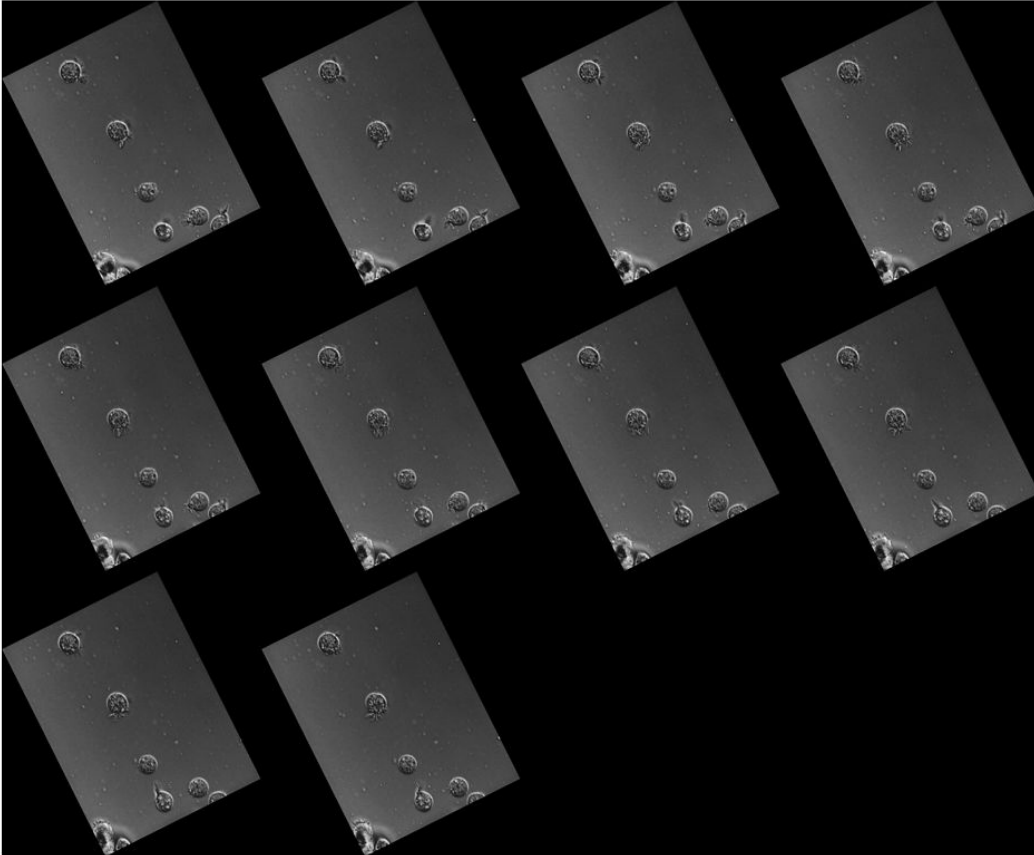
```
numFiles = numel(imds.Files);  
imr = zeros([warper.OutputSize 1 numFiles], 'like', im);
```

Apply the geometric transformation to each of the input images by calling the `warp` function of the Warper object.

```
for ind = 1:numFiles  
    im = read(imds);  
    imr(:,:,1,ind) = warp(warper,im);  
end
```

Visualize the output images. (Turn off the warning message about the images being scaled for display.)

```
warning('off', 'images:initSize:adjustingMag')  
montage(imr);
```



## Tips

- If the input images are RGB images or 3-D grayscale images of size  $m$ -by- $n$ -by- $p$ , then `warp` applies the transformation to each color channel or plane  $p$  independently.

## Algorithms

Warper is optimized to apply the same geometric transformation across a batch of same size images. Warper achieves this optimization by splitting the warping process into two steps: computation of the transformed coordinates (done once) and interpolation on the image (done for each image). Compared to `imwarp`, this approach speeds up the whole process significantly for small to medium-sized images, with diminishing returns for larger images.

## See Also

### Functions

`imrotate` | `imtranslate` | `imwarp` | `warp`

### Objects

`affine2d` | `imref2d` | `projective2d`

### Introduced in R2017b

# warp

Apply geometric transformation

## Syntax

```
B = warp(w,A)
```

## Description

`B = warp(w,A)` performs the geometrical transformation defined in `w` on input image `A` and returns the warped image in `B`.

## Input Arguments

### **w** — Image warper

Warper object

Image warper, specified as a Warper object.

### **A** — Input image

numeric matrix

Input image, specified as a numeric matrix, with size *m*-by-*n* or *m*-by-*n*-by-*p*. The size of `A` must match `w.InputSize`.

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

## Output Arguments

### **B** — Transformed image

numeric matrix

Transformed image, returned as a numeric matrix. `B` has the same type as `A` and its first two dimensions are `w.OutputSize`. If `A` has *p* planes, `B` will also have *p* planes.

## **See Also**

**Introduced in R2017b**

# 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`. 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.

`L = watershed(A,conn)` specifies the connectivity to be used in the watershed computation.

## Examples

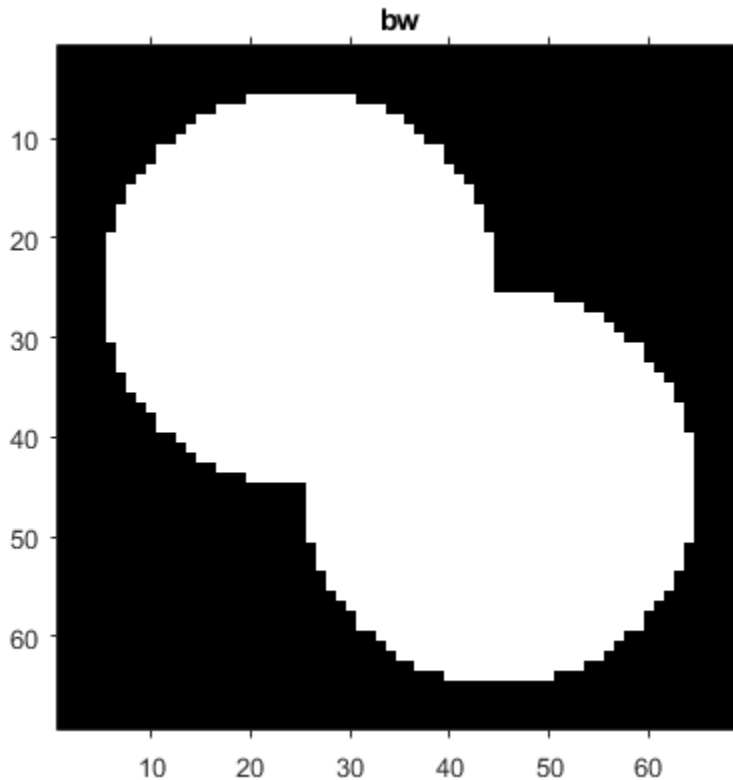
### Compute Watershed Transform and Display Resulting Label Matrix

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;
```

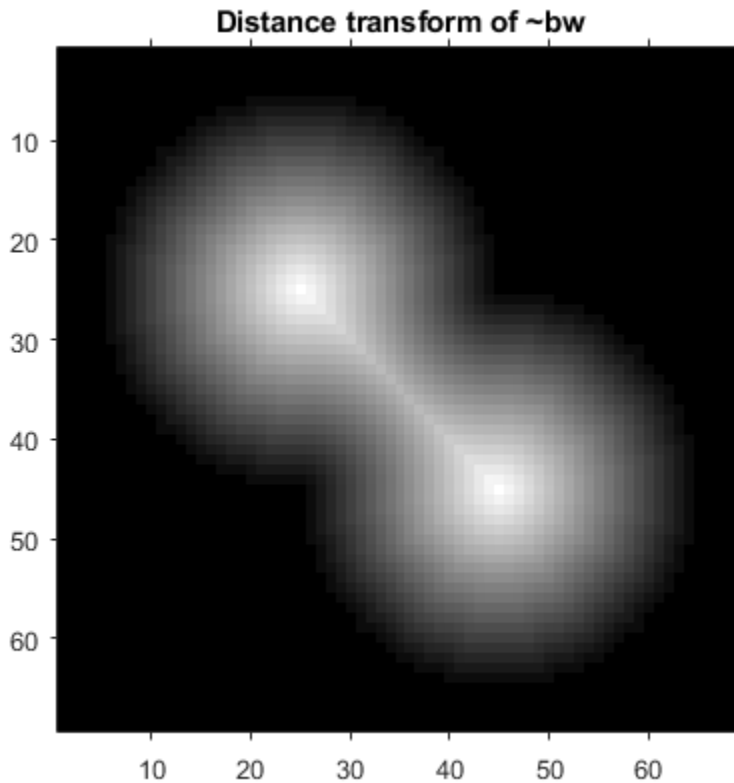
```
bw2 = sqrt((x-center2).^2 + (y-center2).^2) <= radius;  
bw = bw1 | bw2;  
figure  
imshow(bw, 'InitialMagnification', 'fit'), title('bw')
```



Compute the distance transform of the complement of the binary image.

```
D = bwdist(~bw);  
figure  
imshow(D, [], 'InitialMagnification', 'fit')  
title('Distance transform of ~bw')
```





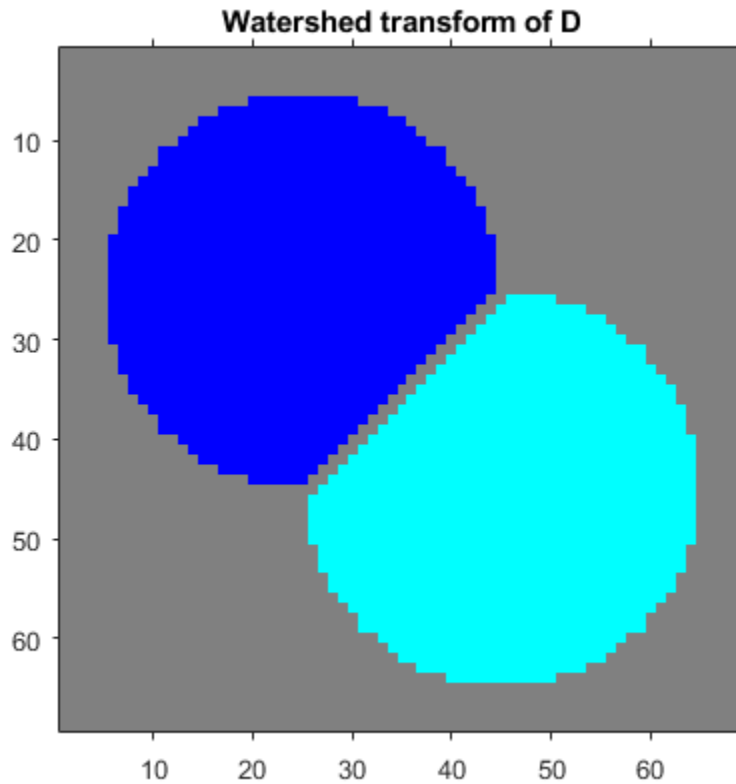
Complement the distance transform, and force pixels that don't belong to the objects to be at Inf .

```
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')
```

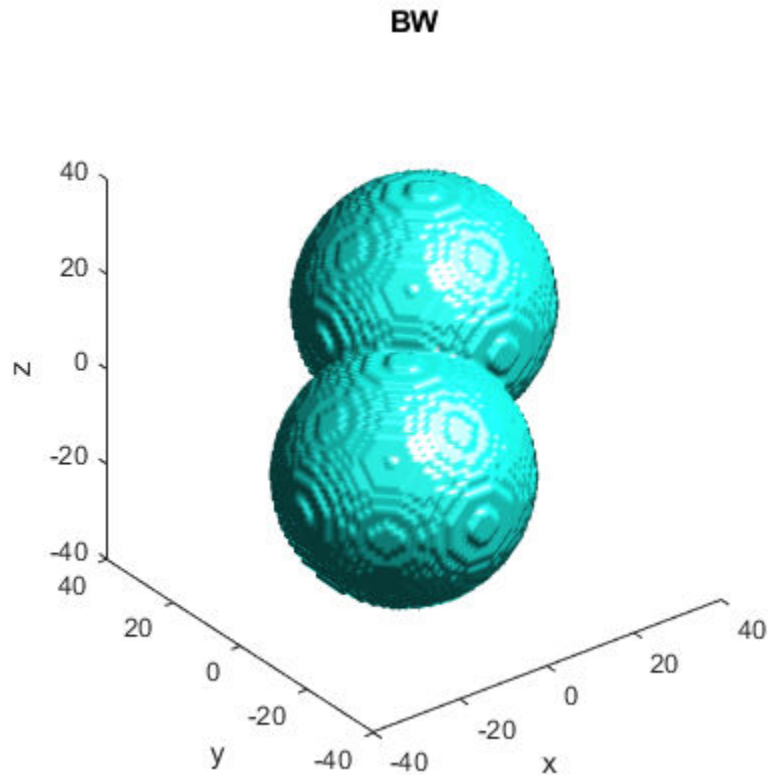


### Compute Watershed Transform of 3-D Binary Image

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)];
```

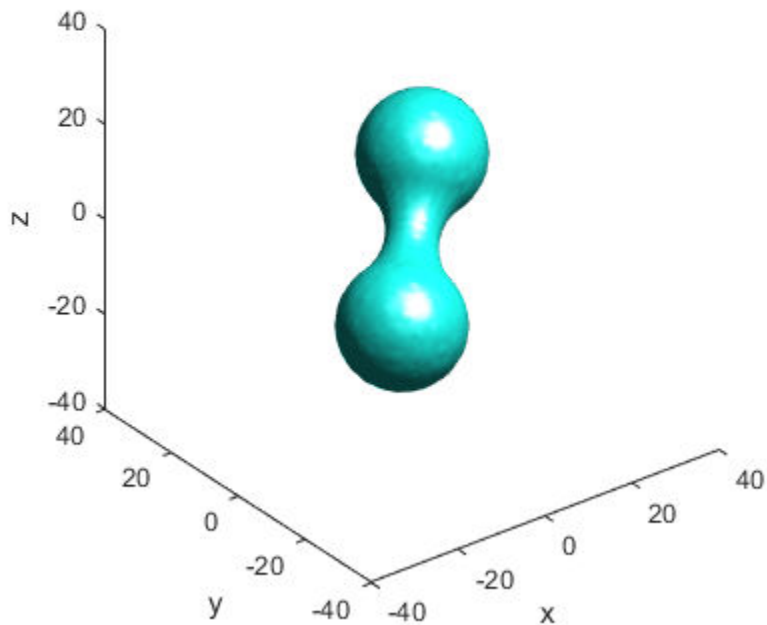
```
[x,y,z] = meshgrid(lims(1):lims(2));  
bw1 = sqrt((x-center1).^2 + (y-center1).^2 + ...  
           (z-center1).^2) <= radius;  
bw2 = sqrt((x-center2).^2 + (y-center2).^2 + ...  
           (z-center2).^2) <= radius;  
bw = bw1 | bw2;  
figure, isosurface(x,y,z,bw,0.5), axis equal, title('BW')  
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
```

**Isosurface of distance transform**

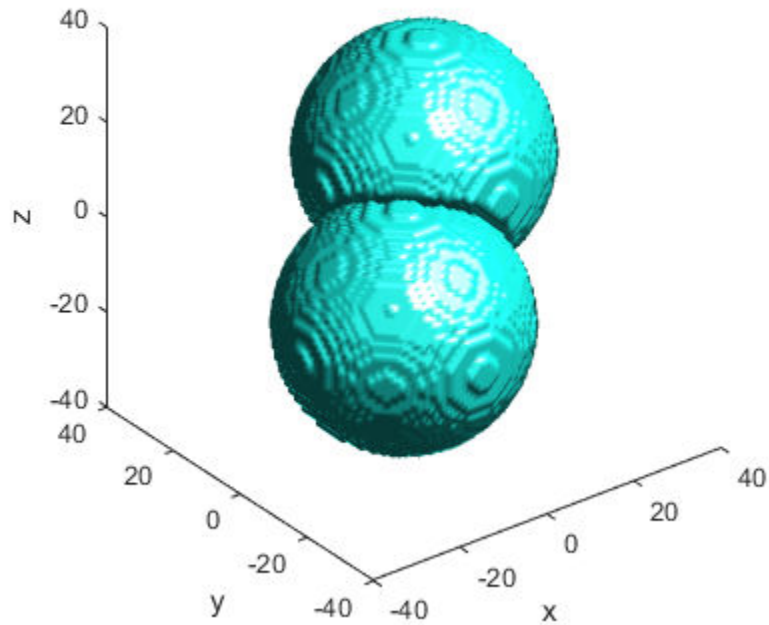


Complement the distance transform, force nonobject pixels to be `Inf`, and then compute the watershed transform.

```
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
```

**Segmented objects**



## Input Arguments

### A — Input image

numeric array | logical array

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

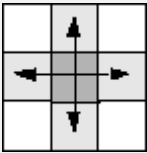
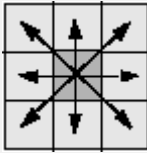
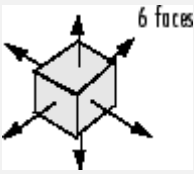
Example: `RGB = imread('pears.png'); I = rgb2gray(RGB);`

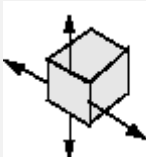
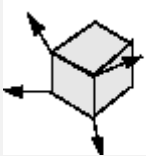
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### conn — Pixel connectivity

4 | 8 | 6 | 18 | 26 | 3-by-3-by- ... -by-3 matrix of 0s and 1s

Pixel connectivity, specified as one of the values in this table. The default connectivity is 8 for 2-D images, and 26 for 3-D images.

Value	Meaning	
<b>Two-Dimensional Connectivities</b>		
4-connected	Pixels are connected if their edges touch. The neighborhood of a pixel are the adjacent pixels in the horizontal or vertical direction.	
8-connected	Pixels are connected if their edges or corners touch. The neighborhood of a pixel are the adjacent pixels in the horizontal, vertical, or diagonal direction.	
<b>Three-Dimensional Connectivities</b>		
6-connected	Pixels are connected if their faces touch. The neighborhood of a pixel are the adjacent pixels in: <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> </ul>	

Value	Meaning	
18-connected	<p>Pixels are connected if their faces or edges touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> </ul>	 <p>6 faces + 12 edges</p>
26-connected	<p>Pixels are connected if their faces, edges, or corners touch. The neighborhood of a pixel are the adjacent pixels in:</p> <ul style="list-style-type: none"> <li>• One of these directions: in, out, left, right, up, and down</li> <li>• A combination of two directions, such as right-down or in-up</li> <li>• A combination of three directions, such as in-right-up or in-left-down</li> </ul>	 <p>6 faces + 12 edges + 8 corners</p>

For higher dimensions, `watershed` uses the default value `conndef(ndims(A), 'maximal')`.

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 center element of `conn`. Note that `conn` must be symmetric about its center element. See “Specifying Custom Connectivities” for more information.

---

**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.

---

Data Types: `double` | `logical`

## Output Arguments

### **L** — Label matrix

numeric array of nonnegative integers

Label matrix, specified as a numeric array of nonnegative integers. 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.

## 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 “Understand 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

`bwdist` | `bwlabel` | `bwlabeln` | `regionprops`

**Introduced before R2006a**

## whitepoint

XYZ color values of standard illuminants

### Syntax

```
xyz = whitepoint  
xyz = whitepoint(illuminant)
```

### Description

`xyz = whitepoint` returns the XYZ value corresponding to the default ICC white reference illuminant, scaled so that  $Y = 1$ .

`xyz = whitepoint(illuminant)` returns the XYZ value corresponding to the white reference illuminant, `illuminant`, scaled so that  $Y = 1$ .

### Examples

#### Get XYZ Value of ICC Illuminant

Return the XYZ color space representation of the default white reference illuminant, 'icc'.

```
wp_icc = whitepoint  
wp_icc =  
    0.9642    1.0000    0.8249
```

Note that the second element, corresponding to the Y value, is 1.

#### Get XYZ Value of d65 Illuminant

Return the XYZ color space representation of the 'd65' white reference illuminant.

```
wp_d65 = whitepoint('d65')
wp_d65 =
    0.9504    1.0000    1.0888
```

## Input Arguments

### illuminant — White reference illuminant

'icc' (default) | 'a' | 'c' | 'e' | 'd50' | 'd55' | 'd65'

White reference illuminant, specified as one of these 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, [1.000, 1.000, 1.000]. Useful as a theoretical reference.
'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 horizon light.
'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 [0.9642, 1.000, 0.8249] using fixed-point, signed, 32-bit numbers with 16 fractional bits. Actual value: [31595,32768, 27030]/32768.

Data Types: char | string

## Output Arguments

### **xyz** — XYZ values

3-element numeric row vector

XYZ values corresponding to the illuminant, returned as a 3-element numeric row vector. The values are scaled so that  $Y = 1$ .

Data Types: double

## See Also

[applycform](#) | [makecform](#) | [xyz2double](#) | [xyz2lab](#) | [xyz2rgb](#) | [xyz2uint16](#)

**Introduced before R2006a**

# wiener2

2-D adaptive noise-removal filtering

---

**Note** 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_out] = wiener2(I,[m n])
```

## Description

`J = wiener2(I,[m n],noise)` filters the grayscale image `I` using a pixel-wise adaptive low-pass Wiener filter. `[m n]` specifies the size (m-by-n) of the neighborhood used to estimate the local image mean and standard deviation. The additive noise (Gaussian white noise) power is assumed to be `noise`.

The input image 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,noise_out] = wiener2(I,[m n])` returns the estimates of the additive noise power `wiener2` calculates before doing the filtering.

## 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);
```

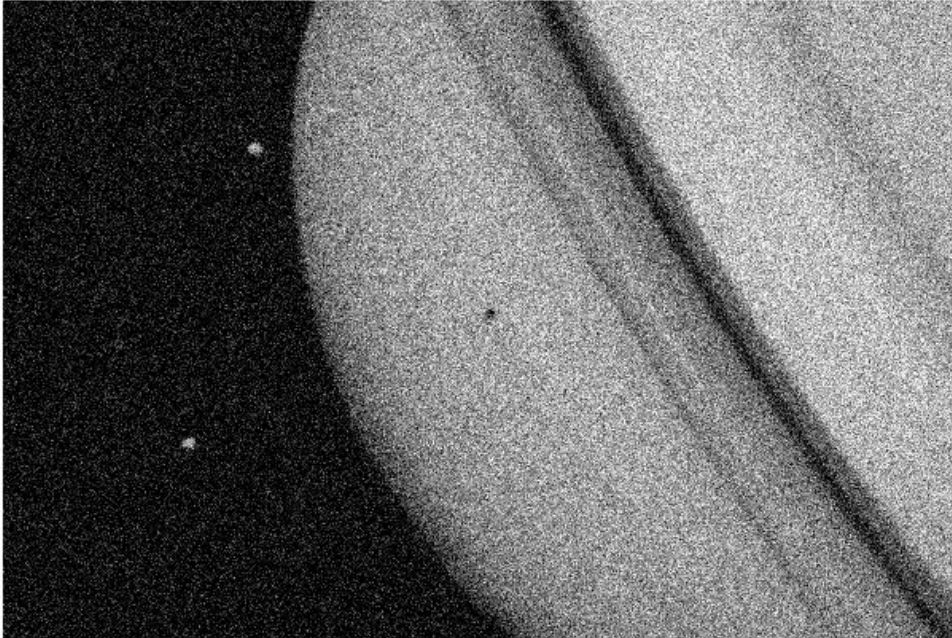
Add Gaussian noise to the image

```
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



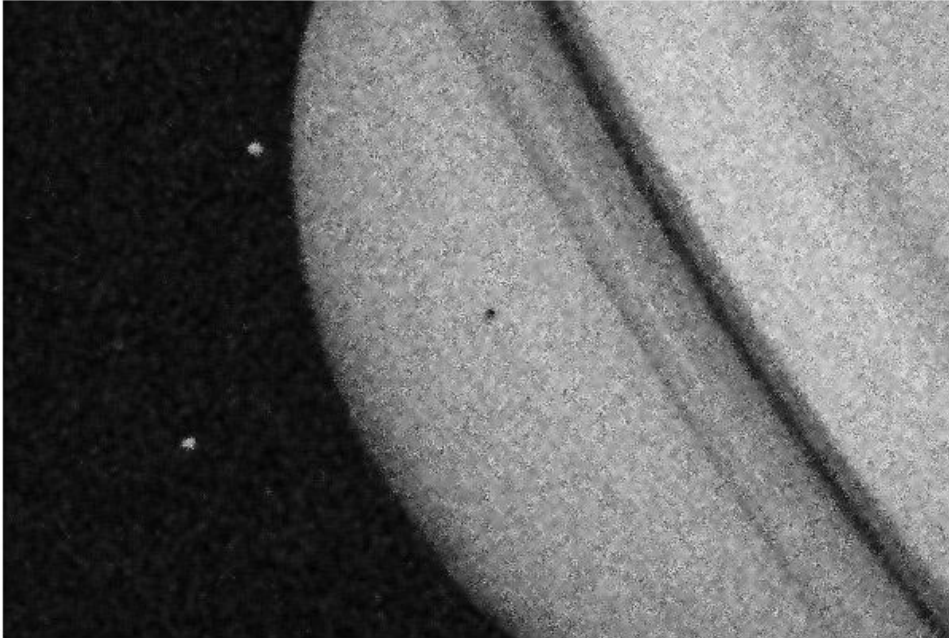
Remove the noise using the `wiener2` function.

```
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



## Input Arguments

### **I** — Input image

2-D numeric array

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

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

### **[m n]** — Neighborhood size

`[3 3]` (default) | 2-element numeric vector of the form `[m n]`

Neighborhood size, specified as a 2-element vector `[m n]` where `m` is the number of rows and `n` is the number of columns. If you omit the `[m n]` argument, `m` and `n` default to 3.



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

### **noise** — Additive noise

`mean2(localVar)` (default) | numeric array

Additive noise, specified as a numeric array. If you do not specify noise, `wiener2` calculates the mean of the local variance, `mean2(localVar)`.

Data Types: `single` | `double`

## Output Arguments

### **J** — Filtered image

numeric array

Filtered image, returned as a numeric array the same size and class as the input image `I`.

### **noise\_out** — Estimate of additive noise power

numeric array

Estimate of additive noise power, returned as a numeric array.

## 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  $\eta$  is the  $N$ -by- $M$  local neighborhood of each pixel in the image `A`. `wiener2` then creates a pixelwise Wiener filter using these estimates,

$$b(n_1, n_2) = \mu + \frac{\sigma^2 - v^2}{\sigma^2} (a(n_1, n_2) - \mu),$$

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`

## Topics

“Remove Noise By Adaptive Filtering”

**Introduced before R2006a**

# worldToIntrinsic

Convert from world to intrinsic coordinates

## Syntax

```
[xIntrinsic, yIntrinsic] = worldToIntrinsic(R,xWorld,yWorld)  
[xIntrinsic,yIntrinsic,zIntrinsic] = worldToIntrinsic(R,xWorld,  
yWorld,zWorld)
```

## Description

`[xIntrinsic, yIntrinsic] = worldToIntrinsic(R,xWorld,yWorld)` maps points from the 2-D world system (`xWorld,yWorld`) to the 2-D intrinsic system (`xIntrinsic,yIntrinsic`) based on the relationship defined by 2-D spatial referencing object `R`.

If the  $k$ th input coordinates (`xWorld(k),yWorld(k)`) fall outside the image bounds in the world coordinate system, `worldToIntrinsic` extrapolates `xIntrinsic(k)` and `yIntrinsic(k)` outside the image bounds in the intrinsic coordinate system.

`[xIntrinsic,yIntrinsic,zIntrinsic] = worldToIntrinsic(R,xWorld,yWorld,zWorld)` maps points from the world coordinate system to the intrinsic coordinate system using 3-D spatial referencing object `R`.

## Examples

### Convert 2-D World Coordinates to Intrinsic Coordinates

Read a 2-D grayscale image of a knee into the workspace.

```
m = dicominfo('knee1.dcm');  
A = dicomread(m);
```

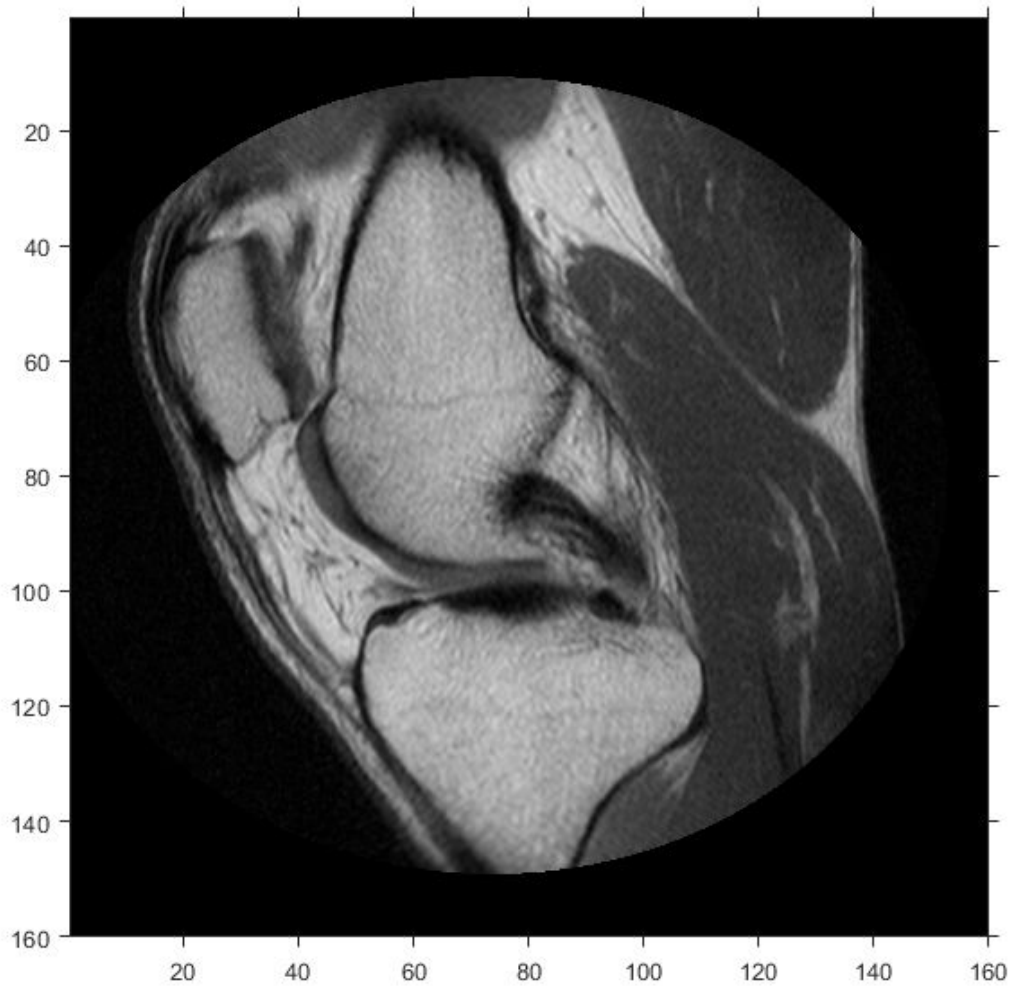
Create an `imref2d` object, specifying the size and the resolution of the pixels. The DICOM file contains a metadata field `PixelSpacing` that specifies the image resolution in each dimension in millimeters per pixel.

```
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]
```

Display the image, including the spatial referencing object. The axes coordinates reflect the world coordinates. Notice that the coordinate (0,0) is in the upper left corner.

```
figure  
imshow(A,RA,'DisplayRange',[0 512])
```



Select sample points, and store their world  $x$ - and  $y$ - coordinates in vectors. For example, the first point has world coordinates  $(38.44, 68.75)$ , the second point is 1 mm to the right of it, and the third point is 7 mm below it. The last point is outside the image boundary.

```
xW = [38.44 39.44 38.44 -0.2];  
yW = [68.75 68.75 75.75 -1];
```

Convert the world coordinates to intrinsic coordinates using `worldToIntrinsic`.

```
[xI, yI] = worldToIntrinsic(RA,xW,yW)
```

```
xI = 1×4
```

```
123.0080 126.2080 123.0080 -0.6400
```

```
yI = 1×4
```

```
220.0000 220.0000 242.4000 -3.2000
```

The resulting vectors are the intrinsic  $x$ - and  $y$ - coordinates in units of pixels. Note that the intrinsic coordinate system is continuous, and some returned intrinsic coordinates have noninteger values. Also, `worldToIntrinsic` extrapolates the intrinsic coordinates of the point outside the image boundary.

## Convert 3-D World Coordinates to Intrinsic Coordinates

Read a 3-D volume into the workspace. This image consists of 27 frames of 128-by-128 pixel images.

```
load mri;  
D = squeeze(D);  
D = ind2gray(D,map);
```

Create an `imref3d` spatial referencing object associated with the volume. For illustrative purposes, provide a pixel resolution in each dimension. The resolution is in millimeters per pixel.

```
R = imref3d(size(D),2,2,4)
```

```
R =  
imref3d with properties:
```

```
    XWorldLimits: [1 257]  
    YWorldLimits: [1 257]
```

```

        ZWorldLimits: [2 110]
        ImageSize: [128 128 27]
PixelExtentInWorldX: 2
PixelExtentInWorldY: 2
PixelExtentInWorldZ: 4
ImageExtentInWorldX: 256
ImageExtentInWorldY: 256
ImageExtentInWorldZ: 108
    XIntrinsicLimits: [0.5000 128.5000]
    YIntrinsicLimits: [0.5000 128.5000]
    ZIntrinsicLimits: [0.5000 27.5000]

```

Select sample points, and store their world  $x$ -,  $y$ -, and  $z$ -coordinates in vectors. For example, the first point has world coordinates (108,92,52), the second point is 3 mm above it in the  $+z$ -direction, and the third point is 0.2 mm to the right of it in the  $+x$ -direction. The last point is outside the image boundary.

```

xW = [108 108 108.2 2];
yW = [92 92 92 -1];
zW = [52 55 52 0.33];

```

Convert the world coordinates to intrinsic coordinates using `worldToIntrinsic`.

```
[xI, yI, zI] = worldToIntrinsic(R,xW,yW,zW)
```

```
xI = 1×4
```

```
54.0000    54.0000    54.1000    1.0000
```

```
yI = 1×4
```

```
46.0000    46.0000    46.0000   -0.5000
```

```
zI = 1×4
```

```
13.0000    13.7500    13.0000    0.0825
```

The resulting vectors are the intrinsic  $x$ -,  $y$ -, and  $z$ -coordinates in units of pixels. Note that the intrinsic coordinate system is continuous, and some returned intrinsic coordinates have noninteger values. Also, `worldToIntrinsic` extrapolates the intrinsic coordinates of the point outside the image boundary.

## Input Arguments

### **R — Spatial referencing object**

imref2d or imref3d object

Spatial referencing object, specified as an imref2d or imref3d object.

### **xWorld — Coordinates along the x-dimension in the world coordinate system**

numeric scalar or vector

Coordinates along the x-dimension in the world coordinate system, returned as a numeric scalar or vector.

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

### **yWorld — Coordinates along the y-dimension in the world coordinate system**

numeric scalar or vector

Coordinates along the y-dimension in the world coordinate system, returned as a numeric scalar or vector. yWorld is the same length as xWorld.

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

### **zWorld — Coordinates along the z-dimension in the world coordinate system**

numeric scalar or vector

Coordinates along the z-dimension in the world coordinate system, returned as a numeric scalar or vector. zWorld is the same length as xWorld.

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

## Output Arguments

### **xIntrinsic — Coordinates along the x-dimension in the intrinsic coordinate system**

numeric scalar or vector



Coordinates along the x-dimension in the intrinsic coordinate system, specified as a numeric scalar or vector. `xIntrinsic` is the same length as `xWorld`.

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

### **yIntrinsic — Coordinates along the y-dimension in the intrinsic coordinate system**

numeric scalar or vector

Coordinates along the y-dimension in the intrinsic coordinate system, specified as a numeric scalar or vector. `yIntrinsic` is the same length as `xWorld`.

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

### **zIntrinsic — Coordinates along the z-dimension in the intrinsic coordinate system**

numeric scalar or vector

Coordinates along the z-dimension in the intrinsic coordinate system, specified as a numeric scalar or vector. `zIntrinsic` is the same length as `xWorld` and `yWorld`.

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

## **See Also**

`imref2d` | `imref3d` | `intrinsicToWorld` | `worldToSubscript`

**Introduced in R2013a**

## worldToSubscript

Convert world coordinates to row and column subscripts

### Syntax

```
[I, J] = worldToSubscript(R,xWorld,yWorld)
[I, J, K] = worldToSubscript(R,xWorld,yWorld,zWorld)
```

### Description

[I, J] = worldToSubscript(R,xWorld,yWorld) maps points from the 2-D world system (xWorld,yWorld) to subscript arrays I and J based on the relationship defined by 2-D spatial referencing object R.

If the *k*th input coordinates (xWorld(*k*),yWorld(*k*)) fall outside the image bounds in the world coordinate system, worldToSubscript sets the corresponding subscripts I(*k*) and J(*k*) to NaN.

[I, J, K] = worldToSubscript(R,xWorld,yWorld,zWorld) maps points from the 3-D world system to subscript arrays I, J, and K, using 3-D spatial referencing object R.

### Examples

#### Convert 2-D World Coordinates to Row and Column Subscripts

Read a 2-D grayscale image of a knee into the workspace.

```
m = dicominfo('knee1.dcm');
A = dicomread(m);
```

Create an imref2d object, specifying the size and the resolution of the pixels. The DICOM file contains a metadata field PixelSpacing that specifies the image resolution in each dimension in millimeters per pixel.

```
RA = imref2d(size(A),m.PixelSpacing(2),m.PixelSpacing(1))
```

```
RA =
```

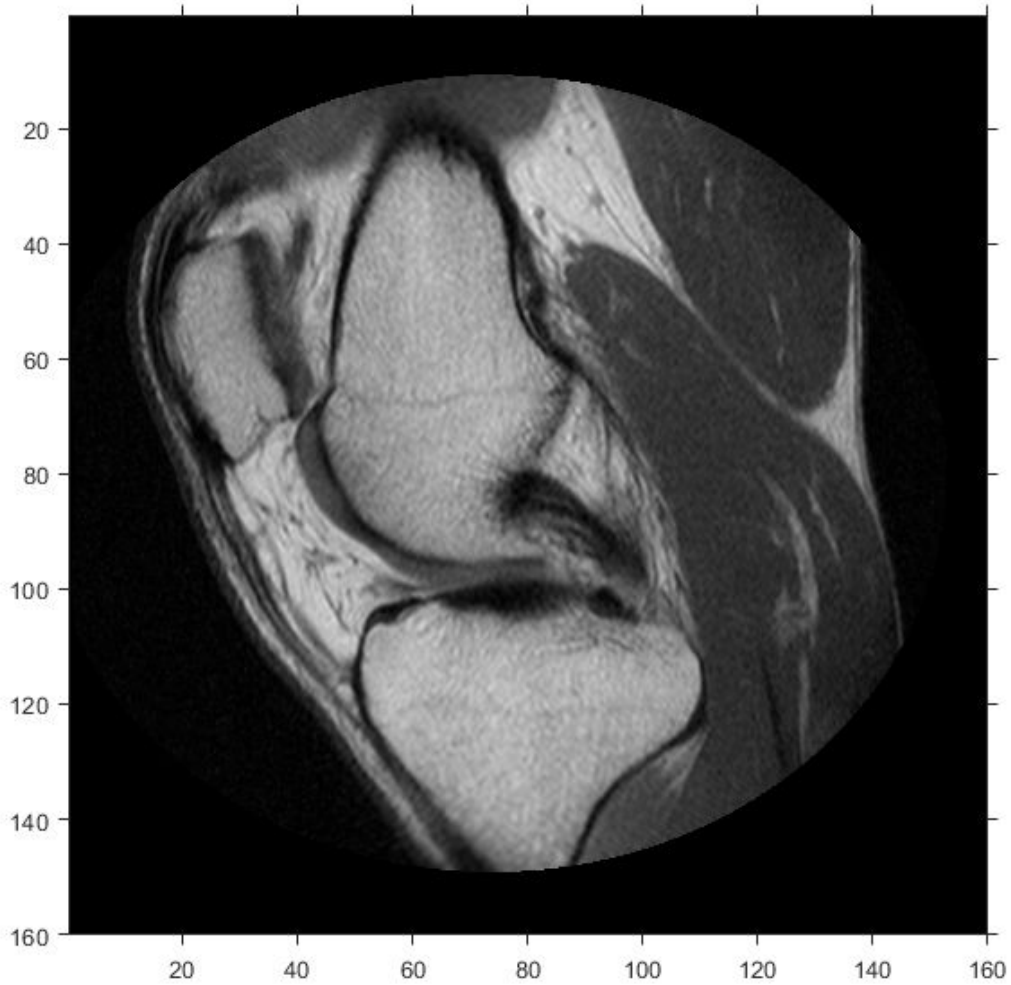
```
imref2d with properties:
```

```
    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]
```

Display the image, including the spatial referencing object. The axes coordinates reflect the world coordinates. Notice that the coordinate (0,0) is in the upper left corner.

```
figure
```

```
imshow(A,RA, 'DisplayRange', [0 512])
```



Select sample points, and store their world  $x$ - and  $y$ - coordinates in vectors. For example, the first point has world coordinates  $(38.44, 68.75)$ , the second point is 1 mm to the right of it, and the third point is 7 mm below it. The last point is outside the image boundary.

```
xW = [38.44 39.44 38.44 -0.2];
yW = [68.75 68.75 75.75 1];
```

Convert the world coordinates to row and column subscripts using `worldToSubscript`.

```
[rS, cS] = worldToSubscript(RA,xW,yW)
```

```
rS = 1×4
```

```
    220    220    242    NaN
```

```
cS = 1×4
```

```
    123    126    123    NaN
```

The resulting vectors contain the row and column indices that are closest to the point. Note that the indices are discrete, and that points outside the image boundary have `NaN` for both row and column indices.

Also, the order of the input and output coordinates is reversed. The world  $x$ -coordinate vector, `xW`, corresponds to the second output vector, `cS`. The world  $y$ -coordinate vector, `yW`, corresponds to the first output vector, `rS`.

## Convert 3-D World Coordinates to Row, Column, and Plane Subscripts

Read a 3-D volume into the workspace. This image consists of 27 frames of 128-by-128 pixel images.

```
load mri;
D = squeeze(D);
D = ind2gray(D,map);
```

Create an `imref3d` spatial referencing object associated with the volume. For illustrative purposes, provide a pixel resolution in each dimension. The resolution is in millimeters per pixel.

```
R = imref3d(size(D),2,2,4)
```

```
R =
    imref3d with properties:
```

```
XWorldLimits: [1 257]
YWorldLimits: [1 257]
ZWorldLimits: [2 110]
  ImageSize: [128 128 27]
PixelExtentInWorldX: 2
PixelExtentInWorldY: 2
PixelExtentInWorldZ: 4
ImageExtentInWorldX: 256
ImageExtentInWorldY: 256
ImageExtentInWorldZ: 108
  XIntrinsicLimits: [0.5000 128.5000]
  YIntrinsicLimits: [0.5000 128.5000]
  ZIntrinsicLimits: [0.5000 27.5000]
```

Select sample points, and store their world  $x$ -,  $y$ -, and  $z$ -coordinates in vectors. For example, the first point has world coordinates (108,92,52), the second point is 3 mm above it in the  $+z$ -direction, and the third point is 5.2 mm to the right of it in the  $+x$ -direction. The last point is outside the image boundary.

```
xW = [108 108 113.2 2];
yW = [92 92 92 -1];
zW = [52 55 52 0.33];
```

Convert the world coordinates to row, column, and plane subscripts using `worldToSubscript`.

```
[rS, cS, pS] = worldToSubscript(R,xW,yW,zW)
```

```
rS = 1×4
```

```
    46    46    46   NaN
```

```
cS = 1×4
```

```
    54    54    57   NaN
```

```
pS = 1×4
```

```
    13    14    13   NaN
```

The resulting vectors contain the column, row, and plane indices that are closest to the point. Note that the indices are discrete, and that points outside the image boundary have index values of NaN.

Also, the order of the input and output coordinates is reversed. The world x-coordinate vector, `xW`, corresponds to the second output vector, `cS`. The world y-coordinate vector, `yW`, corresponds to the first output vector, `rS`.

## Input Arguments

### **R** — Spatial referencing object

`imref2d` or `imref3d` object

Spatial referencing object, specified as an `imref2d` or `imref3d` object.

### **xWorld** — Coordinates along the x-dimension in the world coordinate system

numeric scalar or vector

Coordinates along the x-dimension in the world coordinate system, specified as a numeric scalar or vector.

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

### **yWorld** — Coordinates along the y-dimension in the world coordinate system

numeric scalar or vector

Coordinates along the y-dimension in the world coordinate system, specified as a numeric scalar or vector. `yWorld` is the same length as `xWorld`.

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

### **zWorld** — Coordinates along the z-dimension in the world coordinate system

numeric scalar or vector

Coordinates along the z-dimension in the world coordinate system, specified as a numeric scalar or vector. `zWorld` is the same length as `xWorld`.

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

## Output Arguments

### **I — Row indices**

positive integer scalar or vector

Row indices, returned as a positive integer scalar or vector. I is the same length as `yWorld`. For an  $m$ -by- $n$  or  $m$ -by- $n$ -by- $p$  image,  $1 \leq I \leq m$ .

Data Types: double

### **J — Column indices**

positive integer scalar or vector

Column indices, returned as a positive integer scalar or vector. J is the same length as `xWorld`. For an  $m$ -by- $n$  or  $m$ -by- $n$ -by- $p$  image,  $1 \leq J \leq n$ .

Data Types: double

### **K — Plane indices**

positive integer scalar or vector

Plane indices, returned as a positive integer scalar or vector. K is the same length as `zWorld`. For an  $m$ -by- $n$ -by- $p$  image,  $1 \leq K \leq p$ .

Data Types: double

## See Also

`imref2d` | `imref3d` | `worldToIntrinsic`

**Introduced in R2013a**



# 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. `xyzd` has the same size as `XYZ`.

The Image Processing Toolbox software follows the convention that double-precision XYZ arrays contain 1931 CIE XYZ values (2° observer). XYZ arrays that are `uint16` follow the convention in the ICC profile specification (ICC.1:2001-4, [www.color.org](http://www.color.org)) for representing XYZ values as unsigned 16-bit integers. There is no standard representation of XYZ values as unsigned 8-bit integers. The ICC encoding convention is illustrated by this table.

Value (X, Y, or Z)	uint16 Value
0.0	0
1.0	32768
1.0 + (32767/32768)	65535

## Class Support

`xyz` is a `uint16` or `double` array that must be real and nonsparse. `xyzd` is of class `double`.

## Examples

**Convert 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 = 1×3
```

```
    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 (2° observer) 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 = 1×3
```

```
    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 = 1×3  
    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 = 1×3  
    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

*c*-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 *c*-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 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, [1.000, 1.000, 1.000]. Useful as a theoretical reference.
'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 horizon light.
'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.

Value	White Point
'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 [0.9642, 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**

numeric array

Converted color values, returned as a numeric array of the same shape as the input. The output type is the same as the input type unless you specify the `OutputType` parameter.

## **See Also**

`lab2rgb` | `rgb2xyz` | `xyz2lab`

**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 (2° observer) 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 = 1×3
```

```
    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 = 1×3  
    69.4695   -49.5717    48.3864
```

## Input Arguments

### xyz — Color values to convert

*c*-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 *c*-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 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.



Value	White Point
'e'	Equal-energy radiator, [1.000, 1.000, 1.000]. Useful as a theoretical reference.
'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 horizon light.
'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 [0.9642, 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 a numeric array of the same shape and type as the input.

## See Also

lab2xyz | rgb2lab | 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 (2° observer). XYZ arrays that are uint16 follow the convention in the ICC profile specification (ICC.1:2001-4, [www.color.org](http://www.color.org)) for representing XYZ values as unsigned 16-bit integers. There is no standard representation of XYZ values as unsigned 8-bit integers. The ICC encoding convention is illustrated by this table.

Value (X, Y, or Z)	uint16 Value
0.0	0
1.0	32768
1.0 + (32767/32768)	65535

### Class Support

`xyz` is a uint16 or double array that must be real and nonsparse. `xyz16` is uint8.

### Examples

### Convert XYZ 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 = 1x3 uint16 row vector
```

```
    3277    16384    32768
```

### See Also

[applycform](#) | [lab2double](#) | [lab2uint16](#) | [lab2uint8](#) | [makecform](#) | [whitepoint](#) | [xyz2double](#)

**Introduced before R2006a**

## ycbcr2rgb

Convert YCbCr color values to RGB color space

### Syntax

```
rgbmap = ycbcr2rgb(ycbcrmap)  
RGB = ycbcr2rgb(YCBCR)
```

### Description

`rgbmap = ycbcr2rgb(ycbcrmap)` converts the YCbCr color space values in `ycbcrmap` to the RGB color space. `ycbcrmap` is an  $m$ -by-3 matrix that contains the YCbCr luminance ( $Y$ ) and chrominance ( $C_b$  and  $C_r$ ) color values as columns. Each row in `rgbmap` represents the equivalent color to the corresponding row in `ycbcrmap`.

You optionally can perform the conversion using a GPU (requires Parallel Computing Toolbox). For more information, see “Image Processing on a GPU”.

`RGB = ycbcr2rgb(YCBCR)` converts the YCbCr image `YCBCR` to the equivalent truecolor image `RGB`.

### 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.

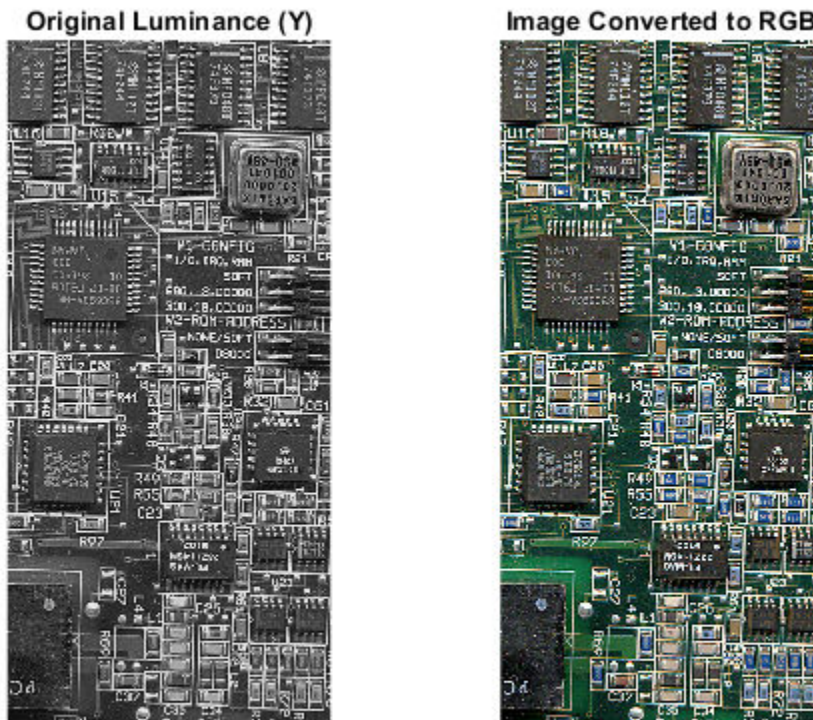
```
YCBCR = rgb2ycbcr(RGB);
```

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');
```



## Input Arguments

### **ycbcrmap — YCbCr color space values**

`c-by-3 matrix` | `gpuArray`

YCbCr color space values, specified as a `c-by-3` matrix representing `c` colors. The first column corresponds to luminance `Y`. The second and third columns correspond to chrominance `Cb` and `Cr`.

To perform the conversion using a GPU, specify `ycbcrmap` as a `gpuArray` that contains a `c-by-3` matrix.

Data Types: `single` | `double`

### **YCBCR — YCbCr image**

`m-by-n-by-3 array` | `gpuArray`

YCbCr image, specified as an `m-by-n-by-3` array.

To perform the conversion using a GPU, specify `YCBCR` as a `gpuArray` that contains an `m-by-n-by-3` array.

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

## Output Arguments

### **rgbmap — RGB color space values**

`c-by-3 matrix` | `gpuArray`

RGB color space values, returned as a `c-by-3` matrix. The three columns represent the red, green, and blue channels.

If the conversion is performed using a GPU, then `rgbmap` is returned as a `gpuArray` containing a `c-by-3` matrix.

### **RGB — Image in RGB color space**

`m-by-n-by-3 array` | `gpuArray`

Image in RGB color space, returned as an `m-by-n-by-3` array.

If the conversion is performed using a GPU, then RGB is returned as a `gpuArray` containing an `m-by-n-by-3` array.

## References

- [1] Poynton, C. A. *Technical Introduction to Digital Video*, John Wiley & Sons, Inc., 1996, p. 175.
- [2] Rec. ITU-R BT.601-5, *Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-screen 16:9 Aspect Ratios*, (1982-1986-1990-1992-1994-1995), Section 3.5.

## 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 “Understand Code Generation with Image Processing Toolbox”.

### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Image Processing on a GPU”.

## **See Also**

ntsc2rgb | rgb2ntsc | rgb2ycbcr

**Introduced before R2006a**