# Image Processing with MATLAB

## 1. Course Overview

## 2. Working with Image Data

### Summary: Working with Image Data

#### Supported File Types

MATLAB supports many file formats that you load with `imread`. Some file types, like DICOM files used in medical image equipment, have their own read function and set of processing functions.

For a complete list of file formats accepted by `imread`, see the documentation page for Supported File Formats.

#### Extracting Metadata from Image Files

You can extract metadata from image files using by using the `imfinfo` function.

```
>> info = imfinfo(filename)
```

| Outputs | | Inputs | |
|---|---|---|---|
| `info` | A structure containing the metadata of an image file. | `filename` | The filename as a string or character vector |

The metadata depends on the image type, but may include file size, color type, file format, comments, creation time, and GPS coordinates. You can extract information from the output structure with dot notation.

```
info.FieldName
```

#### Loading, Displaying, and Saving Different Image Types

There are several kinds of image types in MATLAB.

Grayscale Truecolor Indexed

| | Grayscale Images | RGB Images | Indexed Images |
|---|---|---|---|
| **Color information** | Intensities from black to white | The amount of red, green, and blue in each pixel in the image | The image represented by $n$ colors |
| **Size** | 2-dimensional array | 3-dimensional array | 2-dimensional array of indices from 0 to $n-1$ along with a colormap of size $n \times 3$ where $n$ is the number of colors |

| | Grayscale Images | RGB Images | Indexed Images |
|---|---|---|---|
| **Reading, displaying, and saving the image** | `I = imread("imFile.tif");`<br><br>`imshow(I)`<br><br>`% use full intensity range`<br>`imshow(I,[])`<br><br>`imwrite(I,"newName.jpg")` | `I = imread("imFile.tif");`<br><br>`imshow(I)`<br><br>`imwrite(I,"newName.jpg")` | `[I,map] = imread("imFile.tif");`<br><br>`imshow(I,map)`<br><br>`imwrite(I,map,"newName.jpg")` |

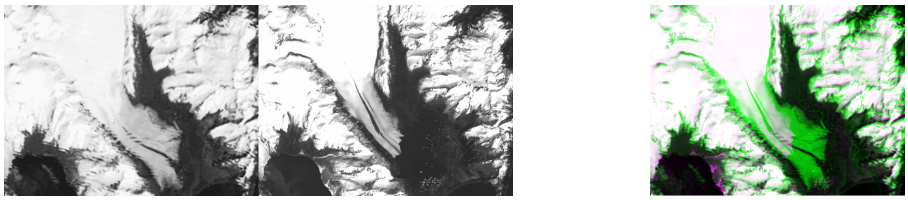
You visually compare two images using the `imshowpair` function.

```
>> imshowpair(I,method)
```

**Inputs**

| | |
|---|---|
| `I` | A grayscale, RGB, or binary image. |
| `method` | An optional string that indicates the display method. |

Using the `"montage"` method displays the images side by side. If no method is specified, the images are superimposed and highlighted green where the first image is brighter and magenta where the second image is brighter.



*The `"montage"` method is displayed left. The default is on the right.*
*Landsat satellite images of Bear Glacier courtesy of the U.S. Geological Survey.*

See the documentation for a list of functions to convert between grayscale, RGB, and indexed images. You can use the imapprox function to reduce the number of colors in an indexed image.

### Image Data Types

By default, images are imported into MATLAB as **`uint8`**, an integer data type with values from 0 to 255. To avoid rounding and saturation in computation, you may need to convert your image to type **`double`** using the `im2double` function.

Images of type **`double`** have values in the range from 0 to 1. However, your computations are not required to stay within that range. After completing your computations, you can rescale your image back to the range 0 to 1.

| | |
|---|---|
| `rescale` | *Scale all pixels to between 0 and 1.* |

| | |
|---|---|
| `imadjust` | *Scale all pixels to between 0 and 1 while saturating the bottom and top 1%.* |

See the documentation for a complete list of image data types.

### Binary Images

Binary images are 2-dimensional logical arrays. They can be created by thresholding with relational operators (`<`, `>`, `<=<` `code>`, `>=`) and manipulated with **`logical`** operators like `~`.

You can burn a binary mask onto an image using `imoverlay`. Once you have created the burned

image, you can use imshow to display it.

```
>> B = imoverlay(A,mask,color)
```

| Outputs | |
|---|---|
| B | An RGB image. |

| Inputs | |
|---|---|
| A | A grayscale or RGB image. |
| mask | The binary image you want to burn onto A. |
| color | Optional. A MATLAB color specification or RGB triplet with intensities between 0 and 1. |

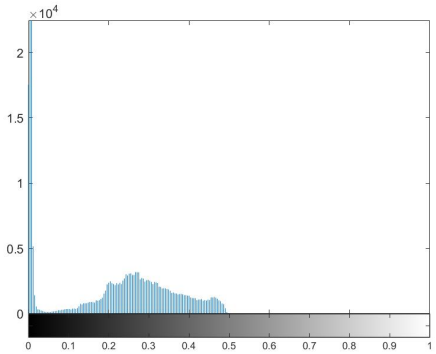
---

# 3. Preprocessing

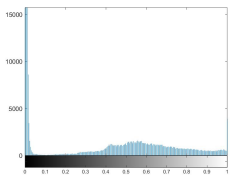
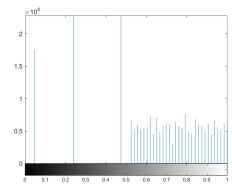
## Summary: Preprocessing

### Adjusting Contrast

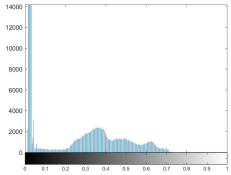
You may need to adjust contrast to improve appearance of an image or the performance of an image processing algorithm. Below are some techniques and their effects on this dimly illuminated medical slide and its intensity histogram.




Medical slide images from the "C.elegans infection live/dead image set Version 1" provided by Fred Ausubel and available from the Broad Bioimage Benchmark Collection.

| Method | Command | Effect on an Image | Intensity Histogram |
|---|---|---|---|
| **Histogram Stretching**<br>Stretch the intensity histogram to span the entire range. The bottom and top 1% of the intensities are saturated to emphasize the difference between very bright and very dark regions. | `I2 = imadjust(I);`<br><br>See the documentation for finer control of input and output intensity limits. |  |  |
| **Histogram Equalization**<br>Stretch the intensity histogram to approximate a uniform distribution. | `I2 = histeq(I);`<br><br>See the documentation to fit the intensity histogram to other distributions. |  |  |

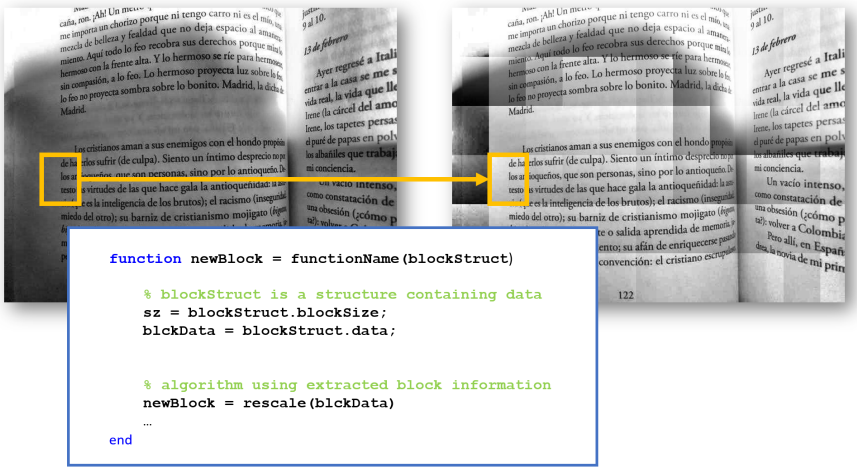
---

| Method | Command | Effect on an Image | Intensity Histogram |
|---|---|---|---|
| **Adaptive Histogram Equalization** Enhance contrast locally. The transformed histogram does not necessarily take up the entire intensity range. | `I2 = adapthisteq(I);` See the documentation to fit the intensity histogram to other distributions. |  |  |

## Block Processing

You can process your image by blocks by wrapping your algorithm in a block processing function and passing it to `blockproc`. This is helpful to

- make your own algorithm adaptive,
- parallelize your algorithm, and
- work with large data files out of memory by reading, processing, and writing output one block at a time.




```
imBP = blockproc(im,blockSize,@functionName);
```

The block processing function can be an anonymous or local function within the file or be contained in a separate file on the path. The input is a *block struct* with fields containing information about the block being processed.

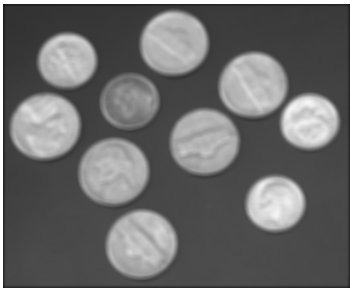
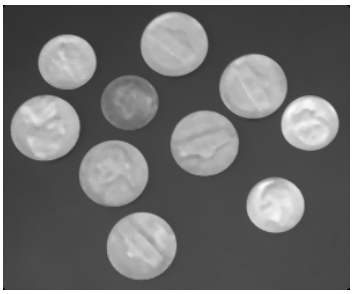
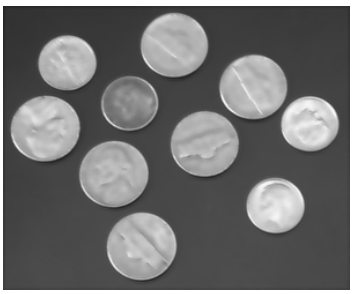
```
blockIm = blockStruct.data       % image data for the current block
blockSz = blockStruct.blockSize  % size of the block of data
imSz = blockStruct.imageSize     % size of the original image
```

See the documentation for `blockproc` for other parameters, fields of the block struct, and instructions for processing images out of memory.

## Spatial Filtering

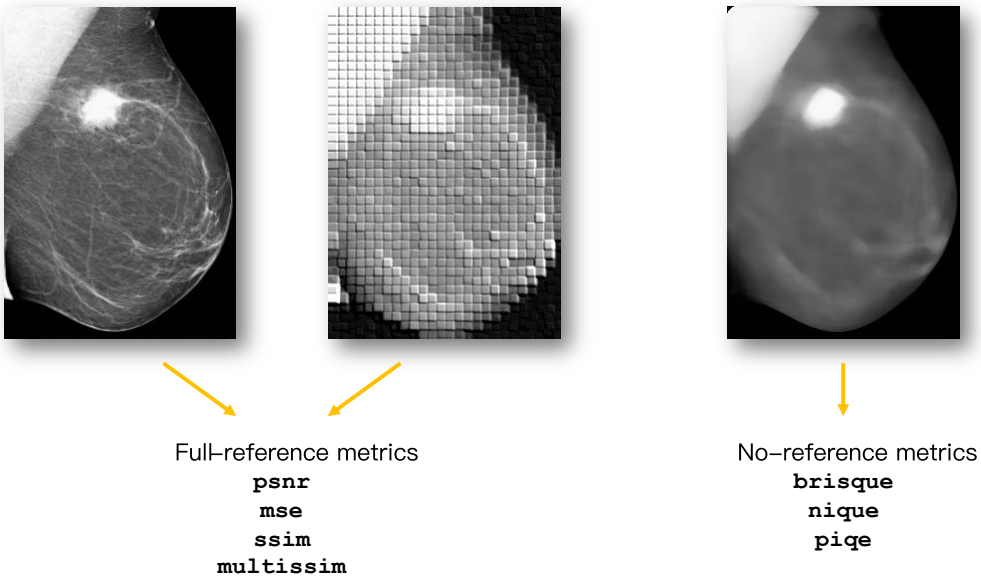
Spatial filtering can be used to smooth images and reduce noise. This can improve the appearance of an image or the output of an image processing algorithm. When filtering an image, you often balance removing noise with blurring detail.

| Method | Syntax | Effect on Image of 5-by-5 Filter |
|---|---|---|

| Method | Syntax | Effect on Image of 5-by-5 Filter |
|---|---|---|
| **Linear Filter**<br>You can create your own linear filter. The code for an averaging filter with symmetric padding is here.<br><br>See the documentation for other types of filters and border padding. | `avg = fspecial("average",[m n]);`<br>`imAvg = imfilter(I,avg,"symmetric");` |  |
| **Median Filter**<br>Take the median value within a neighborhood. This is good at removing salt and pepper noise while preserving edges.<br><br>See the documentation for other types of border padding. | `imMed = medfilt2(I,[m n],"symmetric");` |  |
| **Wiener Filter**<br>The Wiener filter is an adaptive filter that smooths less in areas of high variation (therefore preserving edges) and more in areas of low variation, like the background. | `imMed = wiener2(I,[m n]);` |  |

## Image Quality Metrics

There are two types of quality metrics depending on the type of image and your application—full-reference and no-reference metrics. The type of image quality metric you use depends on whether or not you have a reference image with no distortion.



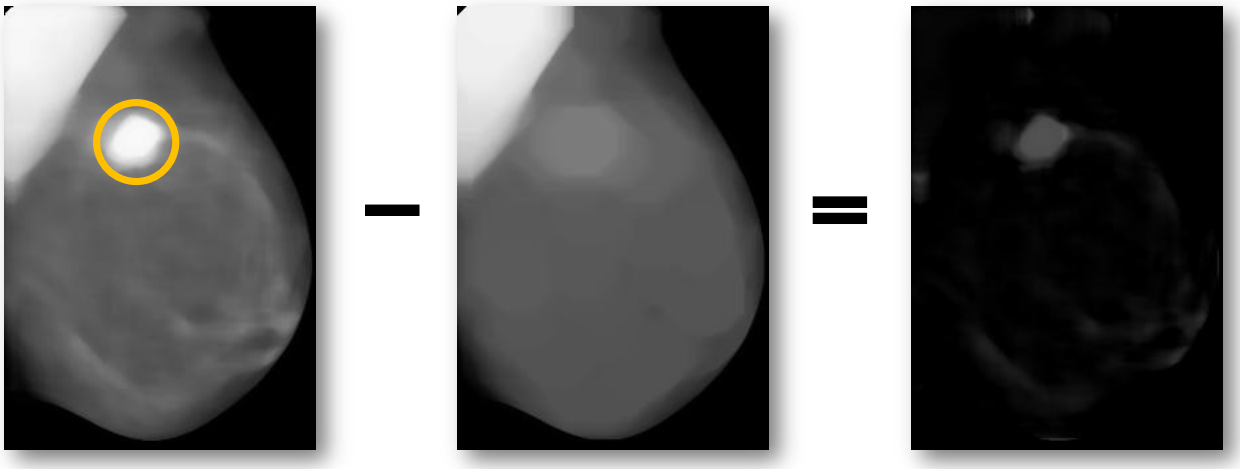

For example:

```
scoreFullRef = psnr(I,ref)
scoreNoRef = niqe(I)
```

## Background Subtraction

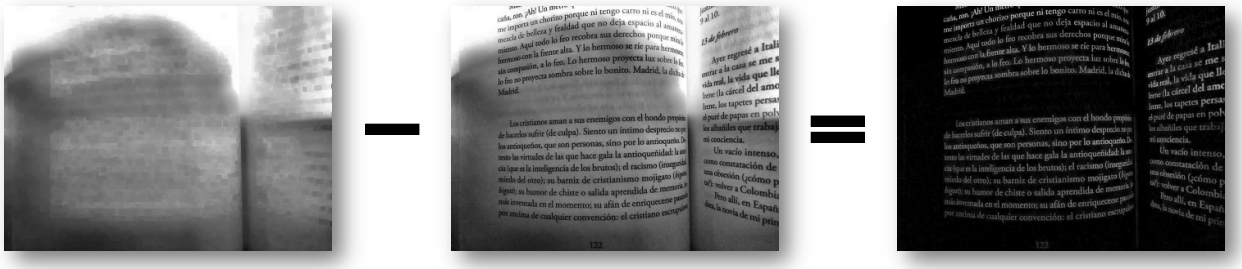
When the background is dark, you can use `imtophat` to perform background subtraction. The neighborhood needs to be large enough to include a bit of background when covering a foreground object.

```
I2 = imtophat(I,nhood);
```



Use `imbothat` when the background is bright. This inverts the intensities. You can use `imcomplement` to make the foreground (the text below) dark again.

```
I2 = imbothat(I,nhood);
```



---

# 4. Segmenting Based on Color

## Summary: Segmenting Based on Color

### Thresholding by Color

You can threshold one or more color planes to identify pixels with desired color profiles. You can threshold programmatically (as shown below), or by using the Color Thresholder app.

Import and display the image.

```
imgRGB = imread("strawberryPlant.jpg");
imshow(imgRGB)
```

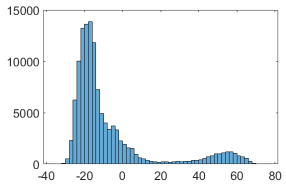
Convert the image to the desired color space and extract the color planes.

The a* plane contains values outside the range 0 to 1. Use `imshow` with the second input `[]` to display the scaled image.

```
imgLab = rgb2lab(imgRGB);
[L,a,b] = imsplit(imgLab);
imshow(a,[])
```

Examine a histogram of the color plane's values.

```
histogram(a)
```

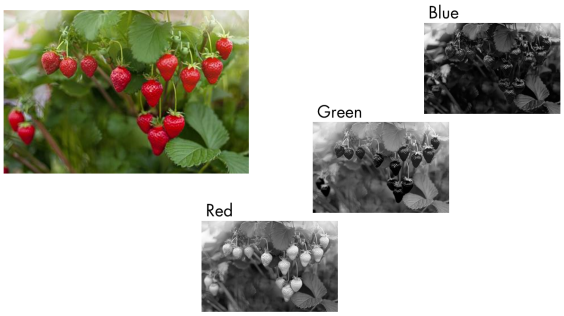


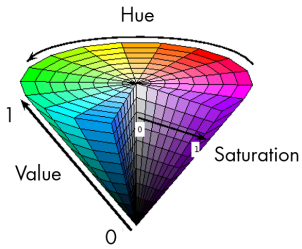
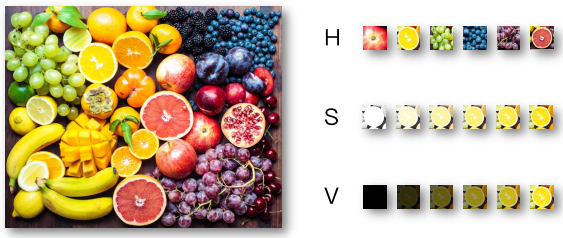
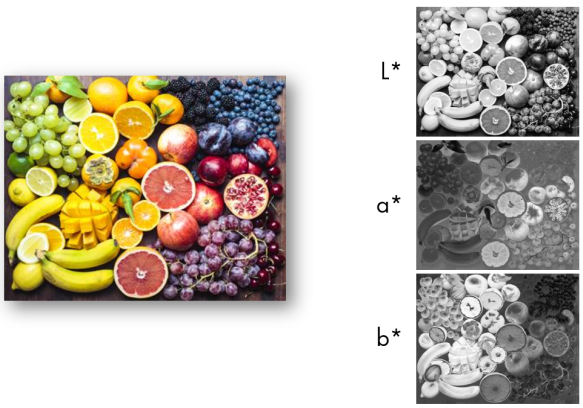

Segment the image by thresholding the color plane.

```
redMask = a > 21;
imshow(redMask)
```

## Color Spaces

Color spaces represent color in an image in different ways.

| Color Space | Channels | Description |
|---|---|---|
| RGB | **R** - red intensities<br>**G** - green intensities<br>**B** - blue intensities | Elements represent the intensities of the red, green, and blue color channels. The range depends on the data type: for example, 0 to 255 for `uint8` arrays, and 0 to 1 for `double` arrays.<br>  |

| Color Space | Channels | Description |
|---|---|---|
| HSV | **H** - hue<br>**S** - saturation<br>**V** - value | Hue corresponds to the color's position on a color wheel. Saturation is the amount of hue, or departure from neutral (shades of gray). Value quantifies brightness. Hue, saturation, and value all have the range 0 to 1.<br>  <br>  |
| L*a*b* | **L*** - luminance<br>**a*** - hue from green to red<br>**b*** - hue from blue to yellow | Luminance represents brightness from black (0) to white (100). The a* channel and b* channel represent the balance between two hues, and although there is no single range for a* and b*, values commonly fall in the range [-100, 100] or [-128, 127).<br>  |

### Segmenting Based on a Region's Average Color

Identify the ROI by displaying the image then calling `drawpolygon`.

Here, the blue polygon indicates the region of interest.

Create a mask to isolate the pixels in the selected region.

```
imshow(img)
roi = drawpolygon;
```

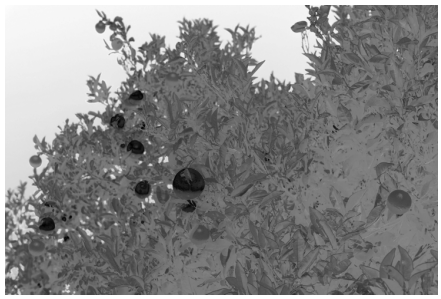
```
BW = createMask(roi);
```

Convert to the Lab color space, and calculate the average pixel color in the region.

```
imgLab = rgb2lab(img);
[L,a,b] = imsplit(imgLab);
aROI = a(BW);
bROI = b(BW);

meanROI = [mean(aROI) mean(bROI)];


aMean = meanROI(1);
bMean = meanROI(2);
distLab = sqrt((a - aMean).^2 + (b - bMean).^2);
imshow(distLab,[])
```



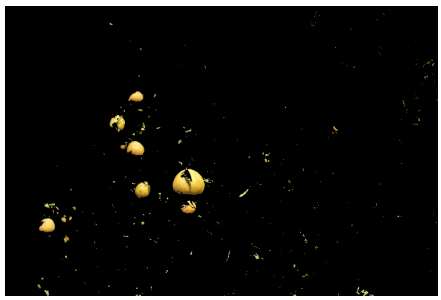
Create a distance matrix by calculating the distance between the average color and every pixel in the image.

$$\sqrt{(a^* - a_{mean})^2 + (b^* - b_{mean})^2}$$

Colors similar to the sample color appear dark, and dissimilar colors appear bright.

```
mask = distLab < 25;

nearColor  = imoverlay(img,~mask,"k");
imshow(nearColor)
```



Threshold the distance matrix to identify pixels similar to the sample color.

---

# 5. Segmenting Based on Texture

## Summary: Segmenting Based on Texture

**Texture Filters**
Texture quantifies the variation of intensity values in an image. Segmentation by texture uses spatial filtering to identify regions with high or low variation.

Import an image and convert it to grayscale.

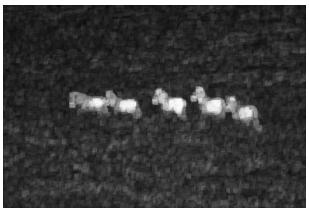
```
zebras = imread("zebras.jpg");
zebras = im2gray(zebras);
imshow(zebras)
```

Apply a texture filter to the image. By default, the filter uses a 3-by-3 neighborhood; defining `nHood` is optional.

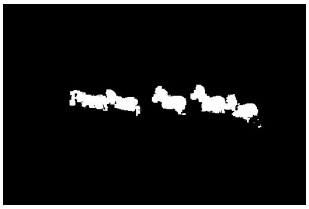
Use `rescale` to rescale the image when using a standard deviation or entropy filter.

Binarize the filtered image.

```
nHood = true(7);
zebraRNG = rangefilt(zebras,nHood);
imshow(zebraRNG)
```

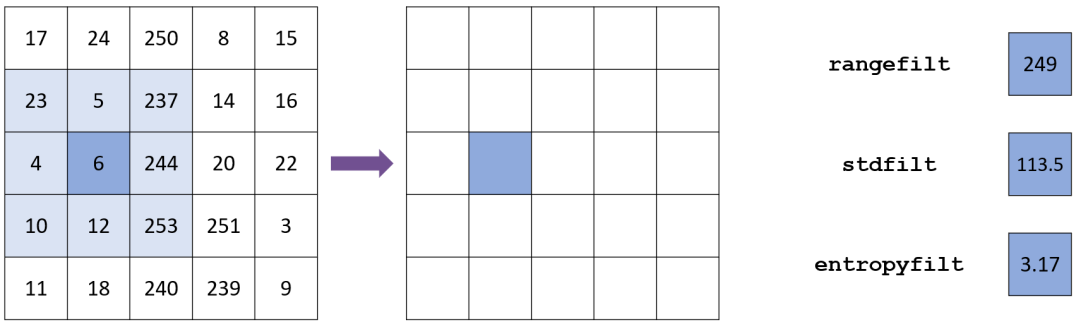


```
zebraBW = imbinarize(zebrasRNG);
imshow(zebraBW)
```



**Texture Filters**

There are three types of filters: range filter, standard deviation filter, and entropy filter. Each filter looks at a neighborhood around each pixel and replaces that pixel with a measure of the texture around it.




`rangefilt` *Perform range filtering.*

`stdfilt` *Perform standard deviation filtering.*

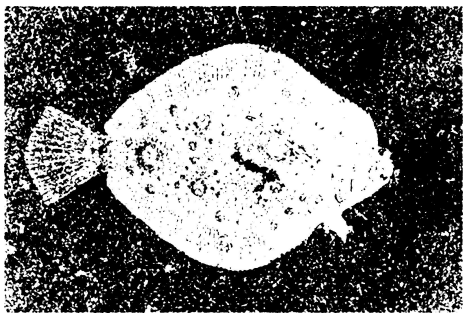
`entropyfilt` *Perform entropy filtering.*

---

# 6. Improving Segmentations

## Summary: Improving Segmentations

### Cleaning Binary Masks



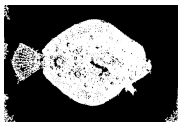
*A flatfish camouflaged in the sand and an initial segmentation based on texture.*

**Cleaning Functions**

There are several functions to remove stray noise from a mask.

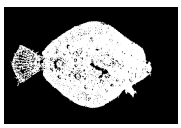
Remove pieces of foreground that are smaller than `n` pixels.

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



Remove foreground along the boundary.

```
BW2 = imclearborder(BW);
```

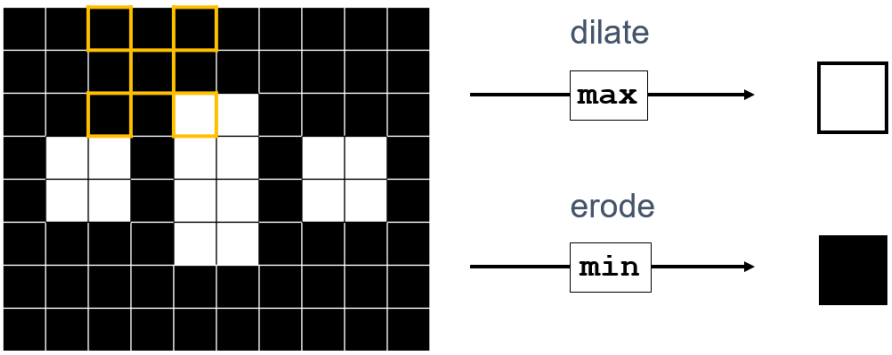


Fill holes.

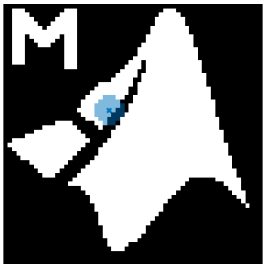
```
BW2 = imfill(BW,"holes");
```

**Morphological Operations**

Morphological operations are shape-based filtering operations that replace pixels with the minimum or maximum of a neighborhood or some combination of these.




The four main operations are erosion, dilation, opening, and closing. Let's see the effect on this mask.



*A binary image with a disk-shaped structuring element marked in blue.*

Create a disk shaped structuring element with radius `r`.

```
SE = strel("disk",n);
```

Erode a binary mask.

```
BW2 = imerode(BW,SE);
```

Dilate a binary mask.

```
BW2 = imdilate(BW,SE);
```

An opening operation first erodes then dilates an image. This has the effect of removing small areas of foreground surrounded by background while keeping the remaining objects approximately the same size.

```
BW2 = imopen(BW,SE);
```

A closing operation first dilates then erodes an image. This has the effect of closing small areas of background that protrude into foreground while keeping the objects approximately the same size.

```
BW2 = imclose(BW,SE);
```

## Growing a Mask from a Seed

Active contours and fast marching method (FMM) are two methods you can use to grow a mask from a seed.

### Active Contours

Active contours expands or shrinks edges of a seed to grow to the boundary of a region. This often works well when you have a decent initial guess at the shape.

*An initial segmentation of a river (left) expanded to identify the flood water with active contours (right).*
*Flood waters in Montezuma Wash courtesy of the National Park Service.*

You can use the `activecontour` function to grow a seed mask to fill a region based on the original image.

```
>> BW = activecontour(I,mask,n,"method")
```

| Outputs | |
|---|---|
| `BW` | The output mask |

| Inputs | |
|---|---|
| `I` | The original image |
| `mask` | The seed mask |
| `n` | The number of iterations |
| `"method"` | The method to use (`"Chan-Vese"` or `"edge"`) |

Use the `"ContractionBias"` property to control whether the segmentation tends to contract (positive values) or grow (negative values). The values are between -1 and 1, and the defaults are 0 and 0.3 for the Chan-Vese and edge methods respectively.

Use the `"SmoothFactor"` property to control the smoothness or regularity of the edges. Higher values produce smoother region boundaries but can also smooth out details. Lower values allow finer details to be captured. The default smoothness value is 0 for the Chan-Vese method and 1 for the edge method. Typical values for this parameter are between -1 and 1. The values are between -1 and 1, and the defaults are 0 and 0.3 for the Chan-Vese and edge methods respectively.

```
BW = activecontour(I,mask,n,"Chan-Vese",...
                           "ContractionBias",-0.4,...
                           "SmoothFactor",0.3);
```

**Fast Marching Method (FMM)**
When you have an initial mask that marks the approximate locations of areas of foreground, FMM can be a good method to try.

*Bison at Yellowstone National Park courtesy of the National Park Service.*

For FMM, you need the seed mask and a grayscale image that approximately marks the shapes of the objects you want to identify. There are several ways to do this.

Create a grayscale intensity difference array where each element is the pixel's distance from a specified threshold, here 0.

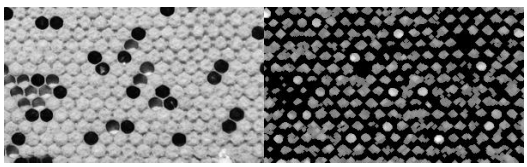
```
wts = graydiffweight(gs,0);
imshow(log(wts),[])
```

Create a weight array that represents each pixel's distance from the average intensity in the region of the initial segmentation.

```
wts = graydiffweight(gs,mask);
imshow(log(wts),[])
```

Create a weight array where each weight is inversely proportional to the gradient values at that pixel location.

```
wts = gradientweight(gs,"RolloffFactor",3.8);
imshowpair(gs,wts,"montage");
```



```
>> BW = imsegfmm(wts,mask,threshold)
```

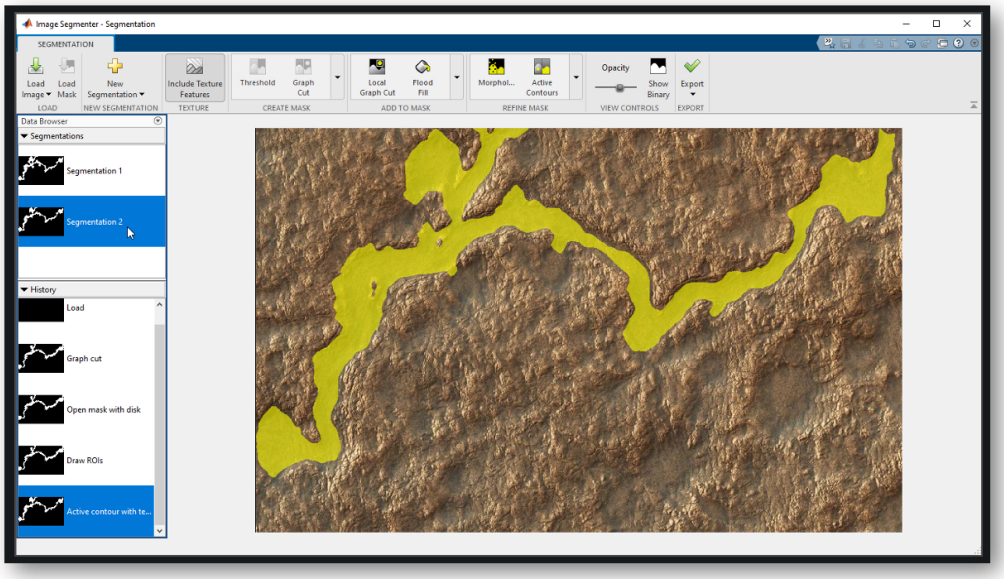
| Outputs | |
|---|---|
| `BW` | The output mask |

| Inputs | |
|---|---|
| `wts` | A grayscale weight array where images in the foreground are bright |
| `mask` | An initial seed that marks the locations of foreground objects |
| `threshold` | Threshold level used to obtain the binary image between [0, 1]. Lower values typically result in larger foreground regions. |

### Image Segmenter App

You can use the Image Segmenter App for creating masks and improving them with morphological operations, iterative growing methods, and other interactive options. You can experiment with different techniques and toggle between segmentation results.

Find it in the Apps tab under "Image Processing and Computer Vision" or type `imageSegmenter` at the command prompt.




*Depressions and Channels on the Floor of Lyot Crater*
*Image credit: NASA/JPL-Caltech/University of Arizona.*

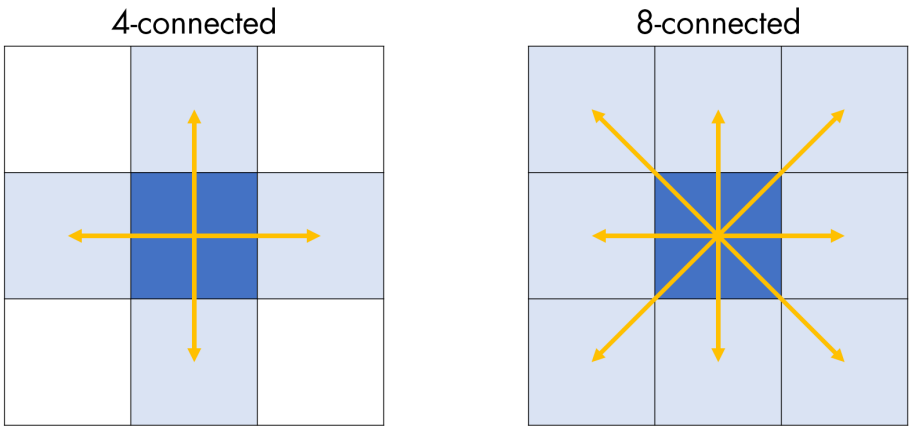
---

# 7. Finding and Analyzing Objects

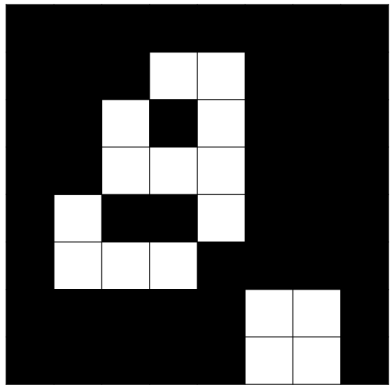
## Summary: Finding and Analyzing Objects

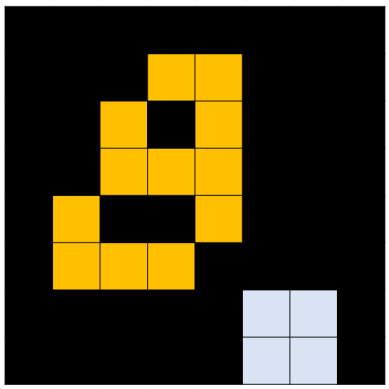
### What Are Connected Components?

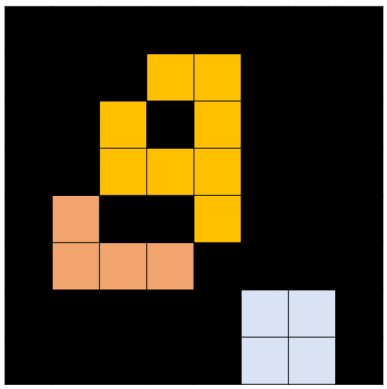
Connected components are separate regions of foreground, a collection of pixels that are all touching. There are two types of connectivity.




This image has 3 connected components using 4-connectivity and 2 if you are using 8-connectivity.

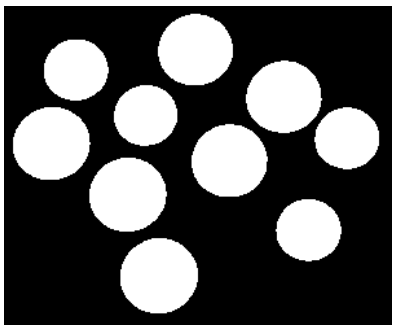






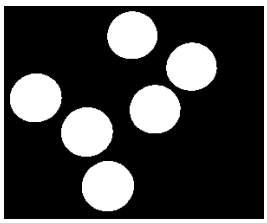
### Working with Connected Components

There are several functions you can use to identify and filter connected components. Consider this binarization of coins on a table.



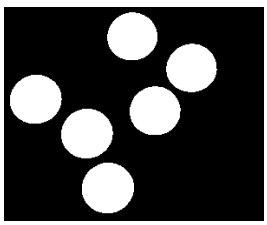
Filter all components of a certain area (e.g. number of pixels)

```
nickel = bwpropfilt(coinsMask,"Area",[2000 3000]);
```



You can also filter the n largest (or smallest) connected components based on a given property

```
nickel = bwpropfilt(coinsMask,"Area",6,"largest");
```



Find connected components in a binary mask

```
coinsCC = bwconncomp(coinsBW)
```

```
     nickelCC = struct with fields
         Connectivity: 8
            ImageSize: [246 300]
           NumObjects: 10
          PixelIdList: {[2651x1 double] ...}
```

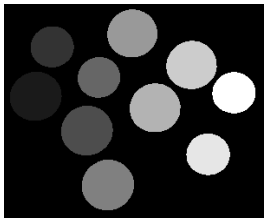
You can access information about the connected components using dot notation

```
numCoins = coinsCC.NumObjects
```

```
          10
```

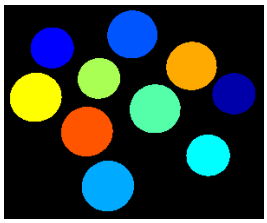
Create a label matrix that assigns a different number to pixels in each connected component.

```
coinsLab = labelmatrix(coinsCC);
imshow(coinsLab,[])
```



Convert a label matrix to an RGB image.

```
coinsRGB = label2rgb(coinsLab,"jet","k","shuffle");
imshow(coinsRGB)
```

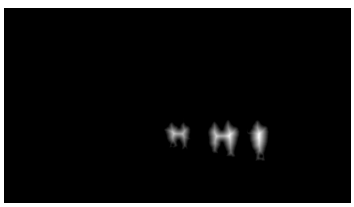


### Separating Overlapping Objects with Watershed

You can apply the watershed algorithm to separate overlapping connected components.

Create a grayscale image that marks the separate objects. Here the distance transform roughly identifies the bison.

```
d = bwdist(~BW);
imshow(d,[])
```

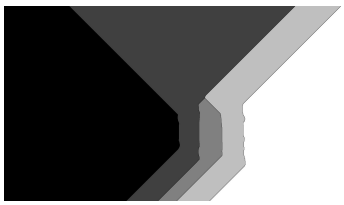


If needed, take the complement of the grayscale image so that the background is bright.

```
d = imcomplement(d);
imshow(d,[])
```

Even out the gray values to remove small minima. This will prevent oversegmentation by the watershed algorithm. Then apply watershed to the grayscale image.

```
dHmin = imhmin(d,5);
bisonSep = watershed(dHmin);
imshow(bisonSep,[])
```



Set the background to black to visualize the bison.

```
bisonSep(~bisonBW) = 0;
imshow(bisonSep,[])
```

Overlay the connected components on the original image.

```
bisonOverlay = labeloverlay(bison,bisonSep);
imshow(bisonOverlay)
```

### Measuring Shape Properties

You can use the `regionprops` function to calculate measurements about connected components from a label matrix or a binary mask. See the documentation for a list of measurements.

```
>> stats = regionprops("table",L,props)
```

**Outputs**

| | |
|---|---|
| `stats` | A table or structure with the requested statistics |

**Inputs**

| | |
|---|---|
| `"table"` | The format of the output. You can also specify `"struct"`. |
| L | A label image or binary mask |
| `props` | A string array specifying the properties you want to extract |

---

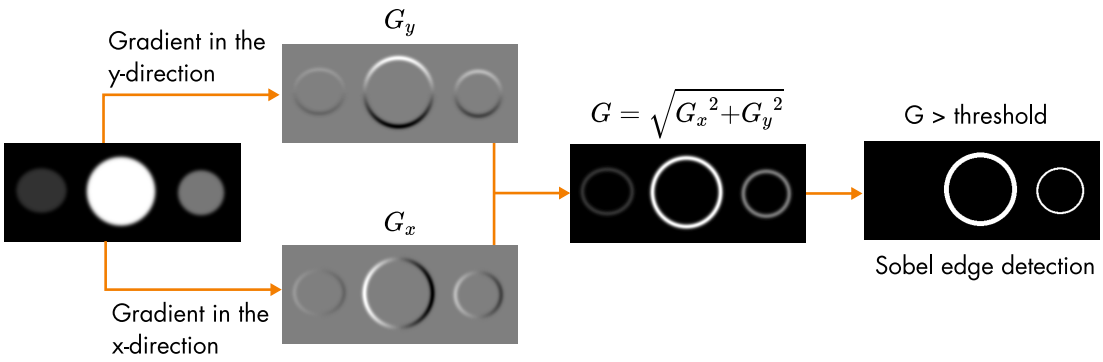
## 8. Detecting Edges and Shapes

### Summary: Detecting Edges and Shapes

## Detecting edges in grayscale images

You can detect object edges in grayscale images using `edge`.

| | |
|---|---|
| `edge` | *Finds edges in a grayscale image* |




*The edge function uses the `"Sobel"` method by default. This method detects edges by thresholding the gradient magnitude.*

If the default edge detection is not sensitive enough, try using the `"Canny"` method, which applies two thresholds. A lower threshold is applied to pixels connected to strong edges in order to keep more of the edge.

## Detecting edges in binary images

For binary images, use `bwboundaries` to detect edges.

| | |
|---|---|
| `bwboundaries` | *Finds edges in a binary image* |

The function `bwboundaries` is typically used as part of a larger workflow.

Create and filter a binary mask.
Note: `createMask` was generated by the Color Thresholder app.

```
I = imread("oranges.jpg")
BW = createMask(I);
BWadj = imopen(BW,strel("disk",5));
BWadj = imclose(BWadj,strel("disk",5));
imshow(BWadj)
```

Find edges in the binary image and visualize them.

```
B = bwboundaries(BWadj,"noholes")
imshow(I);
hold on
visboundaries(B)
hold off
```

## Detecting circles

You can detect circles using `imfindcircles`.

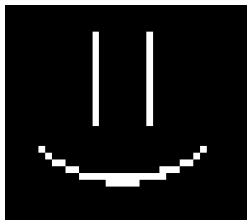
| | |
|---|---|
| `imfindcircles` | *Finds circles in an image using the circular Hough transform* |

### Detecting straight lines

You can detect straight lines using the Hough transform workflow.
The Hough workflow requires the use of three `hough-` functions.

Load or create a binary mask.

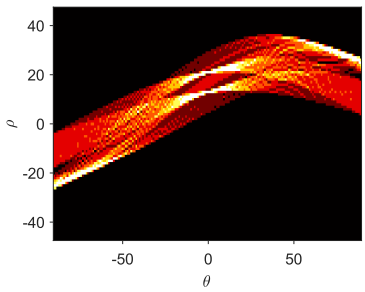
```
BW = imread("smiley.png");
```



Compute the Hough transform matrix

`hough`

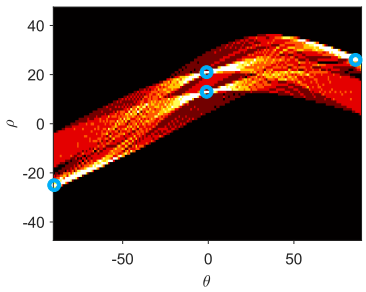
```
[H,theta,rho] = hough(BW);
```




Find the peaks in the Hough transform

`houghpeaks`

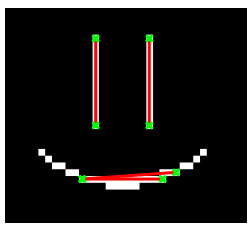
```
peaks = houghpeaks(H);
```




Find straight lines in the binary image.

`houghlines`

```
lines = houghlines(BW,theta,rho,peaks);
```



---

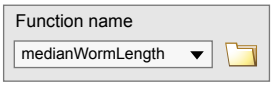
# 9. Batch Processing

## Summary: Batch Processing

### Batch processing workflows

MATLAB has two primary ways to batch process images: the Image Batch Processor app (interactive) and image datastores (programmatic).

---

| Image Batch Processor app | Programmatic workflow using datastores |
|---|---|
| Select folder and load images<br>  | Create an image datastore<br>`ds = imageDatastore("imageFolder")`<br>and read images<br>`I = read(ds);` |
| Create and select a compatible processing function<br>  | Create a function that processes an image<br>`med = medianWormLength(I)` |
| Process some or all of the images<br>  | Loop through the datastore and process images<br>`while hasdata(ds)`<br>`    ...`<br>`end` |

## The Image Batch Processor App

### Processing function

The processing function must be a MATLAB function with one input and one output, where:

1. the input is a single image represented as a MATLAB variable (not a filename), and
2. the output is a structure whose fields contain the results of the processing.

```
function result = processFcn(img)
    result.minValue = min(img,[],"all");
    result.maxValue = max(img,[],"all");
end
```

*Example of a correctly formatted processing function*

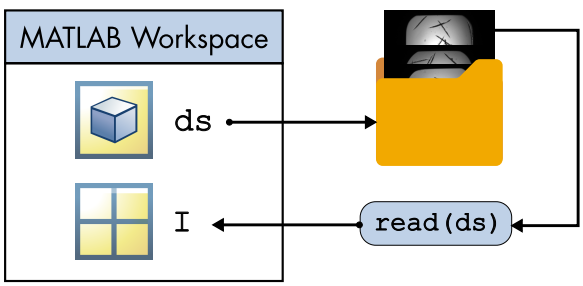
### Export options

1. **Export results**: Save selected results in a workspace variable. By default, the app returns the results in a table.
2. **Generate function**: Create a MATLAB function file that can be called later to batch process images in a folder.

## Image datastores

A datastore stores the location and other basic metadata for image files, without actually loading them into memory. When you need an image from a datastore, you can load it with a call to `read`.

```
ds = imageDatastore("dataFolder")
```




If you wish to include subdirectories when creating the datastore and create labels based on the folders in which the data resides, use the `"IncludeSubfolders"` and `"LabelSource"` options.

```
ds = imageDatastore("data","IncludeSubfolders",true,"LabelSource","foldernames")
```

Additional functions used in the image datastore workflow are summarized in the table below.

| Function or keyword | Example usage | What it does |
|---|---|---|

| Function or keyword | Example usage | What it does |
|---|---|---|
| read | `I = read(ds)` | Reads the next image in `ds` |
| readimage | `I = readimage(ds,k)` | Reads the $k^{th}$ image in `ds` |
| hasdata | `hasdata(ds)` | Determines if any images remain to be read |
| countEachLabel | `T = countEachLabel(ds)` | Counts how many images have each label in `ds` |
| while | `while <condition>`<br>`    …`<br>`end` | Executes body code until `<condition>` returns `false` |

---

# 10. Aligning Images with Image Registration

## Summary: Aligning Images with Image Registration

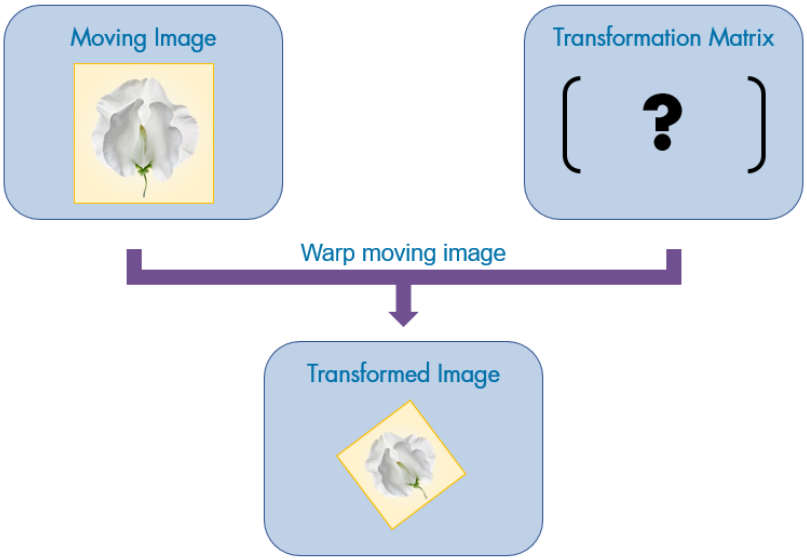
### Defining a Known Geometric Transformation

*This flower has been rotated approximately $45^{\circ}$, scaled by 0.5 and translated to the upper right part of the frame.*

```
I2 = imrotate(I,angle);      % rotate an image (angle in degrees counterclockwise)
I2 = imresize(I,scale);      % scale an image
I2 = imtranslate(I,[x y]);   % translate an image
```

### Estimating a Geometric Transformation

When you don't know the exact rotation, scale, or translation, you can perform phase correlation with `imregcorr` to estimate the transformation matrix.




```
tForm = imregcorr(moving,fixed,type)
spRef = imref2d(szFixed);
transformed = imwarp(moving,tForm,"OutputView",spRef);
```

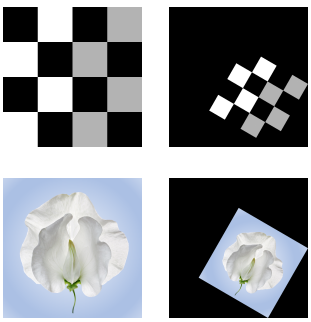
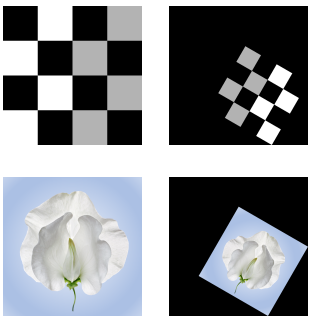
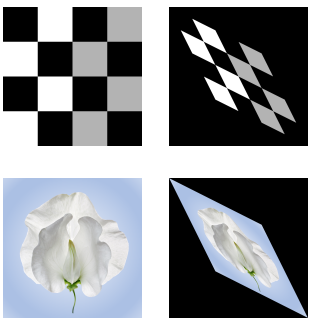
| Transformation Type | Description | Example |
|---|---|---|
| `"translation"` | Translation |  |
| `"rigid"` | Translation and rotation |  |
| `"similarity"` (default) | Translation, rotation, and scaling<br><br>`imregcorr` does not detect scale differences less than 1/4 or greater than 4. |  |

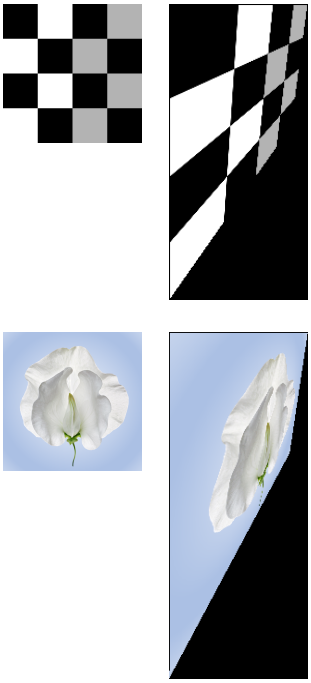
### Mapping Control Points

| | |
|---|---|
| Open the Control Point Selection Tool and select the required number of control points. | `cpselect(movIm,fixedIm)` |

| | |
|---|---|
| Use cross-correlation to fine-tune control points. | `movPtsCor = cpcorr(movPts,fixPts,movIm,fixIm);` |
| Fit a geometric transformation of a specified type. | `tForm = fitgeotrans(movingPts,fixedPts,type)` |
| Create a spatial referencing object. | `spRef = imref2d(size(fixIm))` |
| Warp the image. | `movTransf = imwarp(movIm,tForm,"OutputView",spRef);` |

Depending on the complexity of the transformation, some transformation types require more control points for estimation.

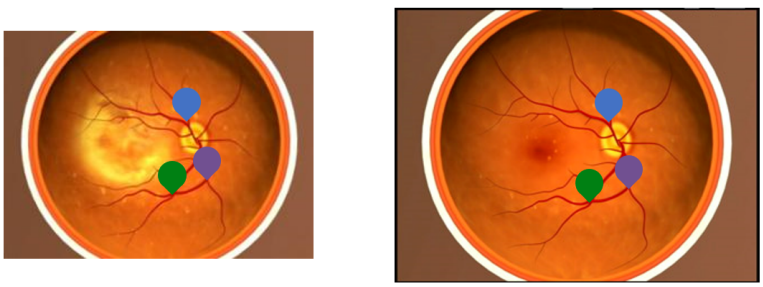
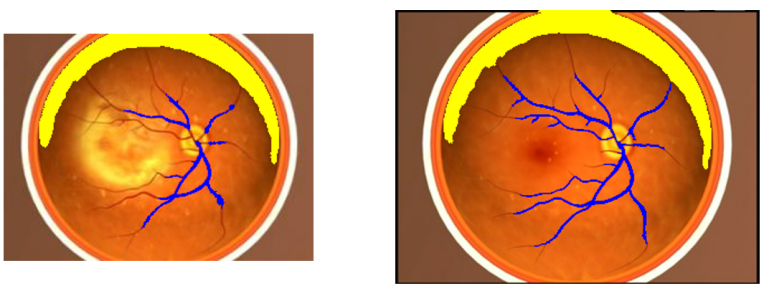
| Transformation Type | Description | Minimum Number of Control Points | Example |
|---|---|---|---|
| `"nonreflective similarity"` | Translation, rotation, scaling<br><br>Use this transformation when shapes in the moving image keep their form. Straight lines remain straight, parallel lines remain parallel, and angles between lines are preserved. | 2 |  |
| `"similarity"` | The same as `"nonreflective similarity"`, but also includes reflection. | 3 |  |
| `"affine"` | Shearing<br><br>Unlike the previous transformations, shapes are distorted. Straight lines remain straight, parallel lines remain parallel, but angles are not preserved (rectangles become parallelograms). | 3 |  |

| Transformation Type | Description | Minimum Number of Control Points | Example |
|---|---|---|---|
| `"projective"` | Projective Transformation<br><br>Use this transformation when the scene appears tilted. Straight lines remain straight, but parallel lines converge toward a vanishing point. | 4 |  |

There are other nonlinear geometric transformations you can estimate using control points. You can find a list of the supported transformation types in the documentation, along with descriptions and the number of control points to required to fit a transformation.

## Matching Image Features

Three types of features are corners, regions, and blobs.

| Feature | Example |
|---|---|
| **Corners**<br>In Computer Vision Toolbox, you can detect corner features using FAST, Harris, and Brisk techniques. |  |
| **Regions**<br>Regions are calculated using the MSER algorithm. |  |

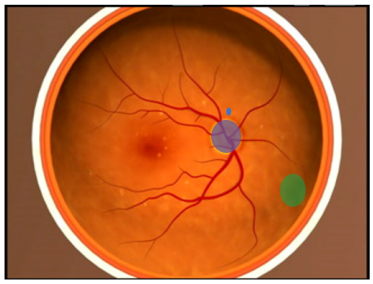
| Feature | Example |
| --- | --- |
| **Blobs**<br>Blobs are circular or elliptical regions identified with the SURF algorithm. |   |

Image Courtesy: National Eye Institute, National Institutes of Health NEI/NIH)

Run the `registrationEstimator` command to test different techniques and parameters using the Registration Estimator App.

## 11. Conclusion