Tutorial 6

Useful Image Processing Techniques (MATLAB) for Your TurtleBot3 Practical/Project

Images in MATLAB

The basic data structure in MATLAB is the *array*, an ordered set of real or complex elements. This object is naturally suited to the representation of *images*, real-valued ordered sets of color or intensity data.

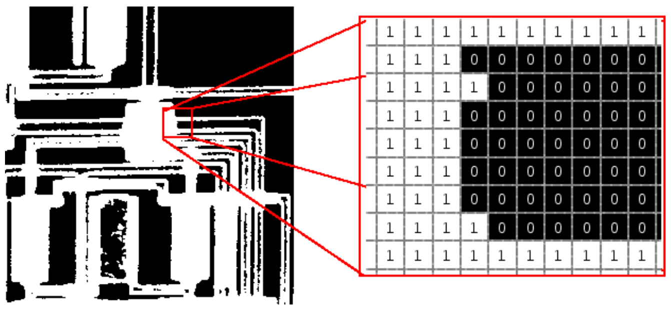
MATLAB stores most images as two-dimensional matrices, in which each element of the matrix corresponds to a single discrete *pixel* in the displayed image. (Pixel is derived from *picture element* and usually denotes a single dot on a computer display.) For example, an image composed of 200 rows and 300 columns of different colored dots would be stored in MATLAB as a 200-by-300 matrix.

Some images, such as truecolor images, represent images using a three-dimensional array. In truecolor images, the first plane in the third dimension represents the red pixel intensities, the second plane represents the green pixel intensities, and the third plane represents the blue pixel intensities. This convention makes working with images in MATLAB similar to working with any other type of numeric data, and makes the full power of MATLAB available for image processing applications.

Commonly Used Image Types

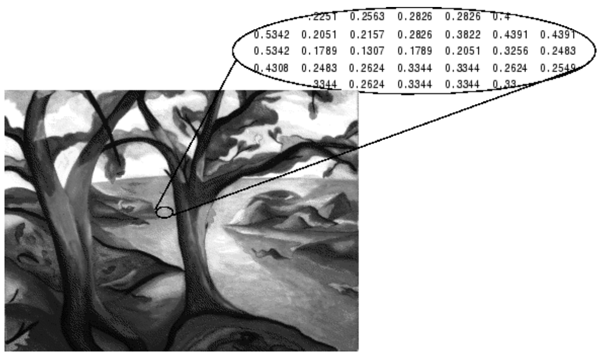
Binary Images

Image data are stored as an *m*-by-*n* logical array. Array values of 0 and 1 are interpreted as black and white, respectively.



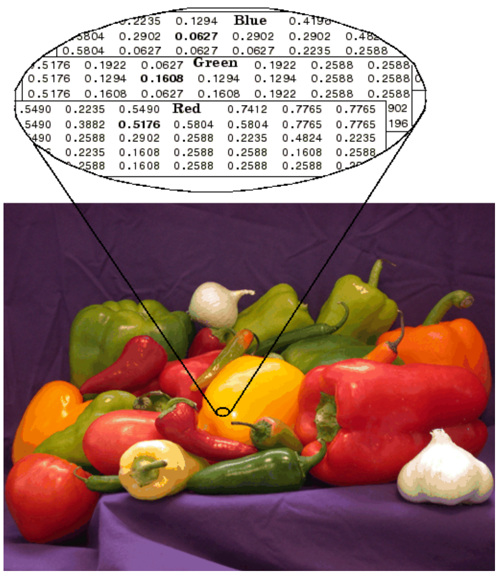
Grayscale Images

A grayscale image (also called gray-scale, gray scale, or gray-level) is a data matrix whose values represent intensities within some range.



Truecolor Images (RGB)

A truecolor image is an image in which each pixel is specified by three values — one each for the red, blue, and green components of the pixel's color. MATLAB store truecolor images as an *m*-by-*n*-by-3 data array that defines red, green, and blue color components for each individual pixel. The color of each pixel is determined by the combination of the red, green, and blue intensities stored in each color plane at the pixel's location.



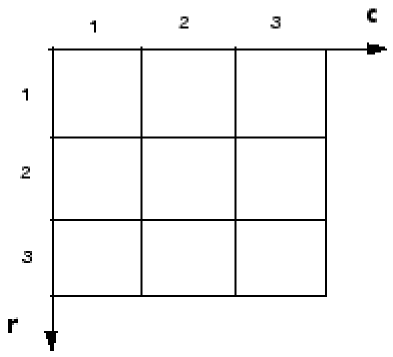
To determine the color of the pixel at (2,3), you would look at the RGB triplet stored in (2,3,1:3). Suppose (2,3,1) contains the value `0.5176`, (2,3,2) contains `0.1608`, and (2,3,3) contains `0.0627`. The color for the pixel at (2,3) is

```
0.5176 0.1608 0.0627
```

Image Coordinate Systems

MATLAB stores most images as two-dimensional arrays (i.e., matrices), in which each element of the matrix corresponds to a single *pixel* in the displayed image.

Often, the most convenient method for expressing locations in an image is to use pixel indices. The image is treated as a grid of discrete elements, ordered from top to bottom, and left to right, as illustrated by the following figure.



For pixel indices, the row increases downward, while the column increases to the right. Pixel indices are integer values, and range from 1 to the length of the row or column.

There is a one-to-one correspondence between pixel indices and subscripts for the first two matrix dimensions in MATLAB. For example, the data for the pixel in the fifth row, second column is stored in the matrix element (5,2). You use normal MATLAB matrix subscripting to access values of individual pixels. For example, the MATLAB code

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

returns the value of the pixel at row 2, column 15 of the image I.

Instructions for the exercises: In this tutorial, the code blocks are provided in sequential order. Every code block is explained. When you go through this tutorial, add code blocks sequentially to your MATLAB script and run it. Add the code blocks one by one. Try to understand the first code block and see its output (workspace values and figures). Then add the second code block and understand it, and so on. If you run the entire script at once you will not understand the individual steps.

Exercise 1

Basic Image Import, Processing, and Export

Open a new MATLAB script and save it as *Tute_6_1.m*. Include the following commands in your script and run them. *lowlight_1.jpg* file is given for you.

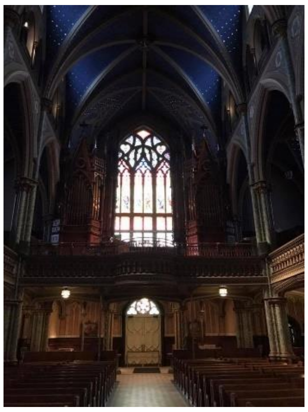
Step 1: Read and Display an Image

Read an image into the workspace, using the *imread* command.

```
I = imread('lowlight_1.jpg');
```

Display the image, using the *imshow* function.

```
imshow(I)
```



Step 2: Check How the Image Appears in the Workspace

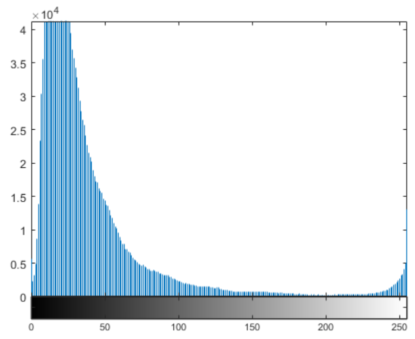
Check how the *imread* function stores the image data in the workspace, using the *whos* command. You can also check the variable in the Workspace Browser.

```
whos I
```

Step 3: Improve Image Contrast

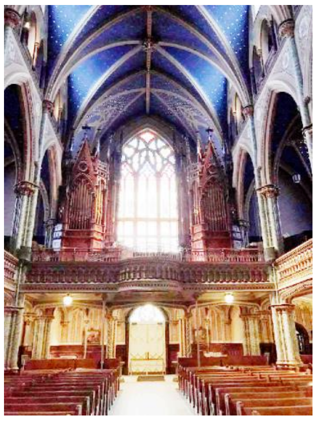
View the distribution of image pixel intensities. The image *lowlight_1.jpg* is a low contrast image. To see the distribution of intensities in the image, create a histogram by calling the *imhist* function. (Precede the call to *imhist* with the figure command so that the histogram does not overwrite the display of the image `I` in the current figure window.) Notice how the histogram indicates that the intensity range of the image is rather narrow. The range does not cover the potential range of [0, 255], and is missing the high values that would result in good contrast.

```
figure
imhist(I)
```



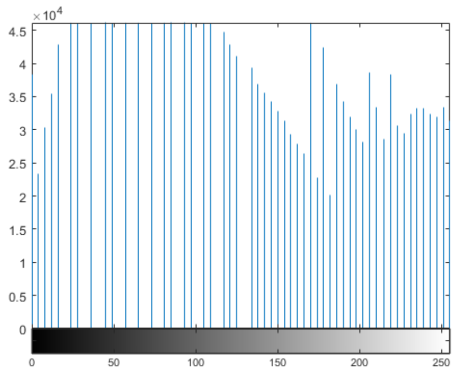
Improve the contrast in an image, using the *histeq* function. Histogram equalization spreads the intensity values over the full range of the image. Display the image.

```
I2 = histeq(I);
figure
imshow(I2)
```



Call the *imhist* function again to create a histogram of the equalized image `I2`. If you compare the two histograms, you can see that the histogram of `I2` is more spread out over the entire range than the histogram of `I`.

```
figure
imhist(I2)
```



Step 4: Write the Adjusted Image to a Disk File

Write the newly adjusted image `I2` to a disk file, using the *imwrite* function. This example includes the filename extension *'.png'* in the file name, so the *imwrite* function writes the image to a file in Portable Network Graphics (PNG) format, but you can specify other formats.

```
imwrite (I2, 'lowlight_1.png');
```

Exercise 2

Convert Between Image Types, Resizing, Cropping

The Image Processing toolbox includes many functions that you can use to convert an image from one type to another.

Create a new MATLAB script and save it as *Tute_6_2.m*.

Step 1: Convert a truecolor image to a grayscale image.

```
RGB = imread('peppers.png');
imshow(RGB)
```

Step 2: Convert the RGB image to a grayscale image and display it.

```
I = rgb2gray(RGB);
figure
imshow(I)
```

Resize an Image with imresize Function

Step 3: Specify the Magnification Value

Resize the image `I`, using the *imresize* function and assign the resized image to `J`. Specify the magnifying factor as 0.75. Google and find out how to do this.

```
J = %%%write your command here%%%
```

Display the original image next to the reduced version.

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

Step 4: Specify the Size of the Output Image

Resize the image again, this time specifying the desired size of the output image, rather than a magnification value. Pass *imresize* a vector [200 250] that contains the number of rows and columns in the output image. If the specified size does not produce the same aspect ratio as the input image, the output image will be distorted.

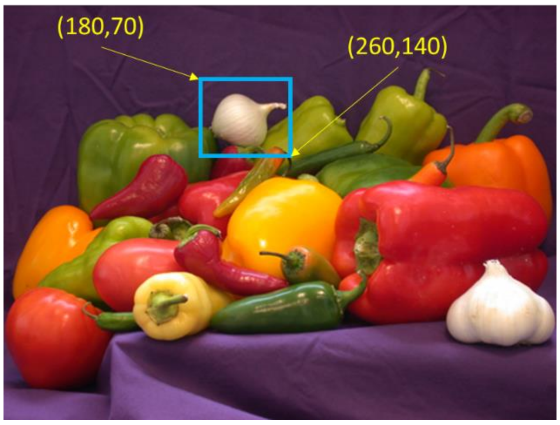
```
K = %%%write your command here%%%
figure, imshow(K)
```

Crop an Image

To extract a rectangular portion of an image, use the *imcrop* function. Using *imcrop*, you can specify the crop region interactively using the mouse or programmatically by specifying the size and position of the crop region.

Call *imcrop* specifying the image to crop, `I`, and the crop rectangle. *imcrop* returns the cropped image in `J`.

The image coordinates are given in pixels. Write your code to crop the onion image. Assign the cropped image to `L`.



```
L = %%%write your command here%%%
figure, imshow(L)
```

Exercise 3

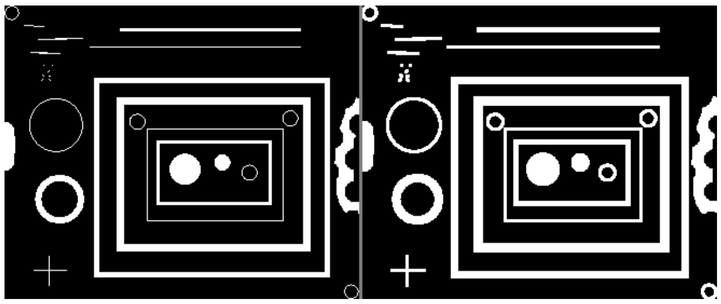
Morphological Operations

Morphology is a broad set of image processing operations that process images based on shapes. In a morphological operation, each pixel in the image is adjusted based on the value of other pixels in its neighborhood. By choosing the size and shape of the neighborhood, you can construct a morphological operation that is sensitive to specific shapes in the input image.

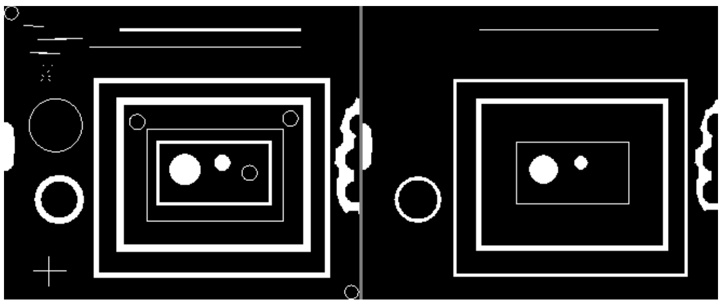
Morphological Dilation and Erosion

The most basic morphological operations are dilation and erosion. Dilation adds pixels to the boundaries of objects in an image, while erosion removes pixels on object boundaries.

Morphological dilation makes objects more visible and fills in small holes in objects.



Morphological erosion removes islands and small objects so that only substantive objects remain.



Use Morphological Opening to Extract Large Image Features

You can use morphological opening to remove small objects from an image while preserving the shape and size of larger objects in the image.

In this example, you use morphological opening on an image of a circuitboard to remove all the circuit lines from the image. The output image contains only the rectangular shapes of the microchips.

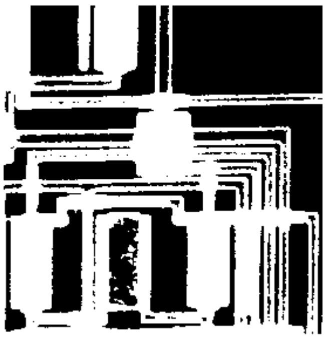
Create a new MATLAB script and save it as *Tute_6_3.m.*

Open an Image In One Step

You can use the *imopen* function to perform erosion and dilation in one step.

Step 1: Read the image into the workspace, and display it.

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



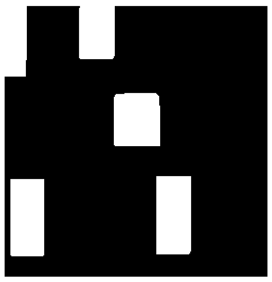
Step 2: Create a structuring element.

The structuring element should be large enough to remove the lines when you erode the image, but not large enough to remove the rectangles. It should consist of all 1s, so it removes everything but large contiguous patches of foreground pixels.

```
 SE = strel('rectangle',[40 30]);
```

Step 3: Open the image.

```
BW2 = imopen(BW1, SE);
figure, imshow(BW2);
```

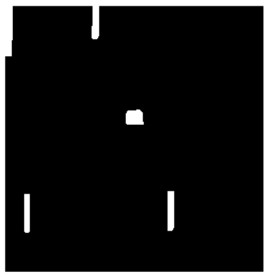


Open an Image By Performing Erosion Then Dilation

Step 4: You can also perform erosion and dilation sequentially.

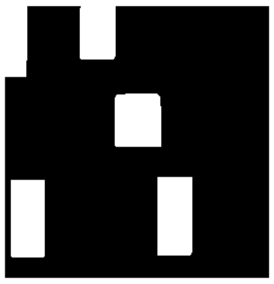
Erode the image with the structuring element. This removes all the lines, but also shrinks the rectangles.

```
BW3 = imerode(BW1,SE);
figure, imshow(BW3)
```



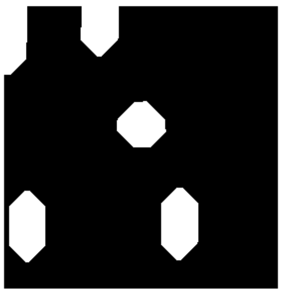
To restore the rectangles to their original sizes, dilate the eroded image using the same structuring element, SE.

```
BW4 = imdilate(BW3,SE);
figure,imshow(BW4)
```



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

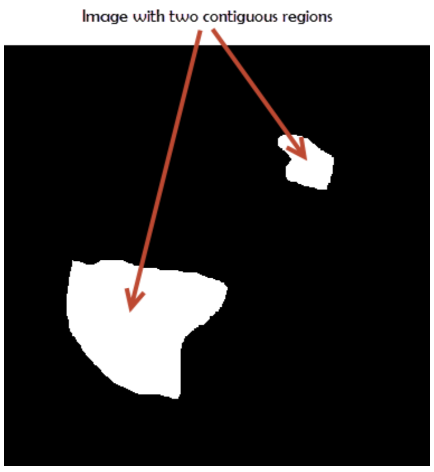
```
SE = strel('diamond',15);
BW5 = imdilate(BW3,SE);
figure,imshow(BW5)
```



Exercise 4

Image Region Properties

Image regions, also called *objects*, *connected components*, or *blobs*, can be contiguous or discontiguous. The following figure shows a binary image with two contiguous regions.



A region in an image can have properties, such as an area, center of mass, orientation, and bounding box. To calculate these properties for regions (and many more) in an image, you can use the *regionprops* function.

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 measured region properties in a table.

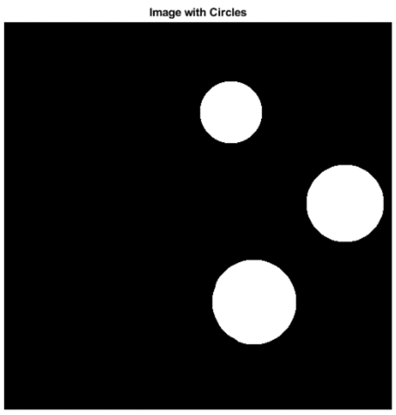
Create a new MATLAB script and save it as *Tute_6_4.m*.

Step 1: Read an image into workspace.

```
a = imread('circlesBrightDark.png');

figure, imshow(a)
```

Step 2: Turn the input image into a binary image.

```
bw = a < 100;
figure, imshow(bw)
title('Image with Circles')
```



Step 3: Calculate properties of regions in the image and return the data in a table.

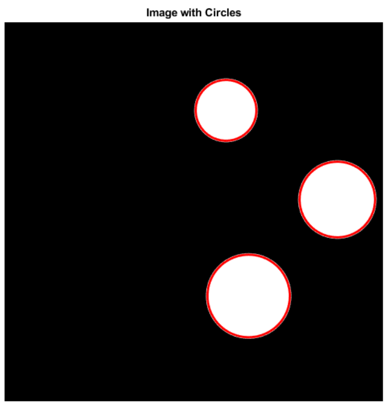
```
stats = regionprops('table',bw,'Centroid',...
    'MajorAxisLength','MinorAxisLength')
```

Step 4: 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
```



Exercise 5

Identifying Round Objects

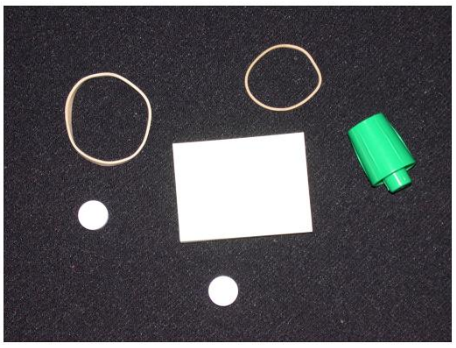
Create a new MATLAB script and save it as *Tute_6_5.m*.

This example shows how to classify objects based on their roundness using *bwboundaries*, a boundary tracing routine.

Step 1: Read Image

Read in pills_etc.png.

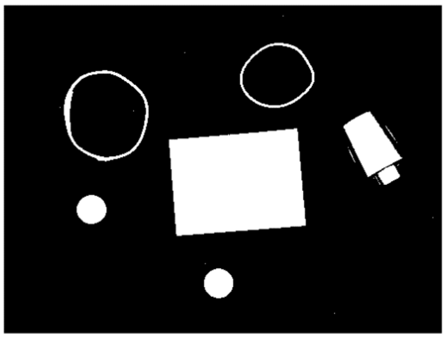
```
RGB = imread('pillsetc.png');
Figure, imshow(RGB)
```



Step 2: Threshold the Image

Convert the image to black and white in order to prepare for boundary tracing using *bwboundaries*.

```
I = rgb2gray(RGB);
bw = imbinarize(I);
figure, imshow(bw)
```

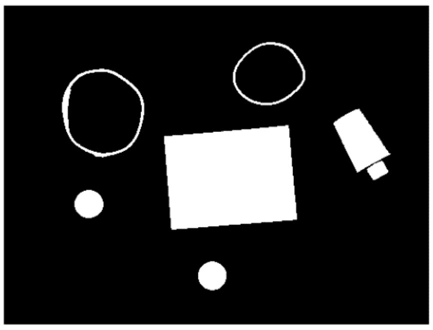


Step 3: Remove the Noise

Using morphology functions, remove pixels which do not belong to the objects of interest.

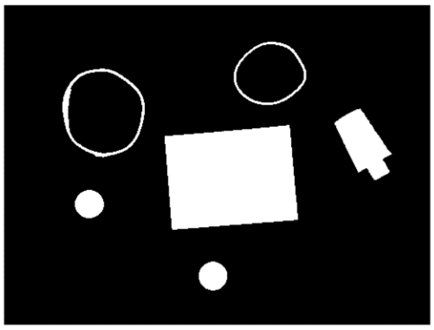
Remove all object containing fewer than 30 pixels.

```
bw = bwareaopen(bw,30);
figure, imshow(bw)
```



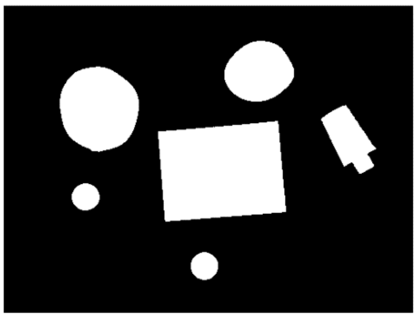
Fill a gap in the pen's cap.

```
se = strel('disk',2);
bw = imclose(bw,se);
figure, imshow(bw)
```



Fill any holes, so that *regionprops* can be used to estimate the area enclosed by each of the boundaries

```
bw = imfill(bw,'holes');
figure, imshow(bw)
```



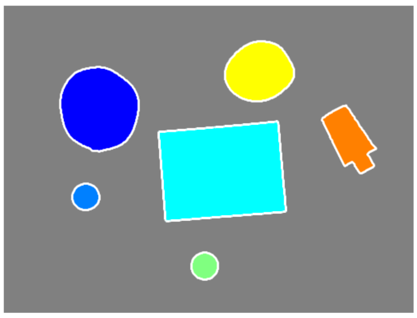
Step 4: Find the Boundaries

Concentrate only on the exterior boundaries. Option '*noholes*' will accelerate the processing by preventing *bwboundaries* from searching for inner contours.

```
[B,L] = bwboundaries(bw,'noholes');
```

Display the label matrix and draw each boundary.

```
figure, imshow(label2rgb(L,@jet,[.5 .5 .5]))
hold on

for k = 1:length(B)
  boundary = B{k};
  plot(boundary(:,2),boundary(:,1),'w','LineWidth',2)
end
```



Step 5: Determine which Objects are Round

Estimate each object's area and perimeter. Use these results to form a simple metric indicating the roundness of an object:

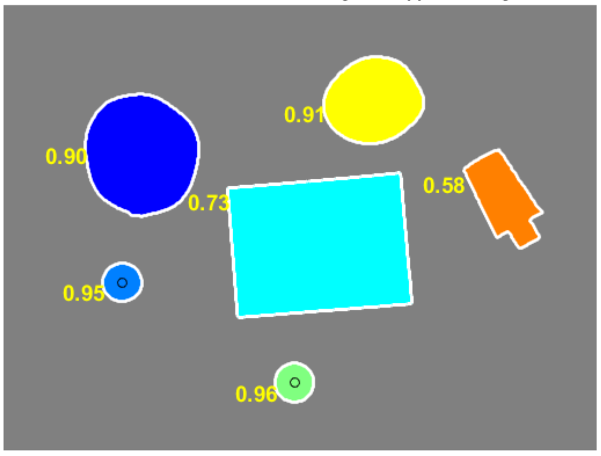
$$metric = \frac{4\pi \times area}{perimeter^2}$$

This metric is equal to 1 only for a circle and it is less than one for any other shape. The discrimination process can be controlled by setting an appropriate threshold. In this example use a threshold of 0.94 so that only the pills will be classified as round.

Use *regionprops* to obtain estimates of the area for all of the objects. Notice that the label matrix returned by *bwboundaries* can be reused by *regionprops*.

```
stats = regionprops(L,'Area','Centroid');

threshold = 0.94;

% loop over the boundaries
for k = 1:length(B)

  % obtain (X,Y) boundary coordinates corresponding to label 'k'
  boundary = B{k};

  % compute a simple estimate of the object's perimeter
  delta_sq = diff(boundary).^2;
  perimeter = sum(sqrt(sum(delta_sq,2)));

  % obtain the area calculation corresponding to label 'k'
  area = stats(k).Area;

  % compute the roundness metric
  metric = 4*pi*area/perimeter^2;

  % display the results
  metric_string = sprintf('%2.2f',metric);

  % mark objects above the threshold with a black circle
  if metric > threshold
    centroid = stats(k).Centroid;
    plot(centroid(1),centroid(2),'ko');
  end

  text(boundary(1,2)-35,boundary(1,1)+13,metric_string,'Color','y',...
       'FontSize',14,'FontWeight','bold')

end

title(['Metrics closer to 1 indicate that ',...
       'the object is approximately round'])
```

Metrics closer to 1 indicate that the object is approximately round



Reference: Mathworks online tutorials. Relevant web pages are linked to the blue color titles. (03 May, 2019).