Assignment 03

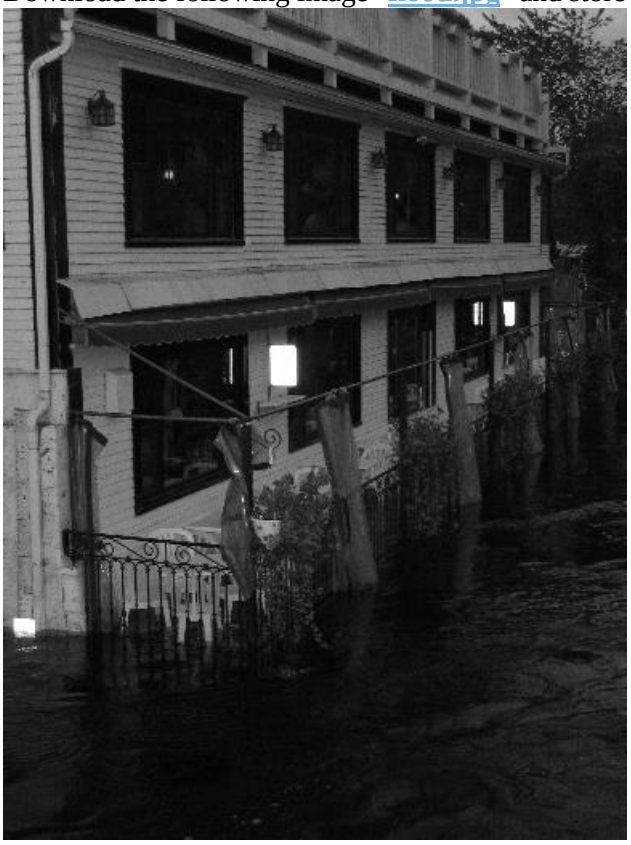
Course: CSD331- Digital Image Processing **Due Date: November 8, 2019**

Histograms Processing and Edge Detection

Note: There are two question in this assignment. Question 1 is about Histogram processing and Question 2 is related to edge detection. You are suggested to go through the literature provided here before writing code.

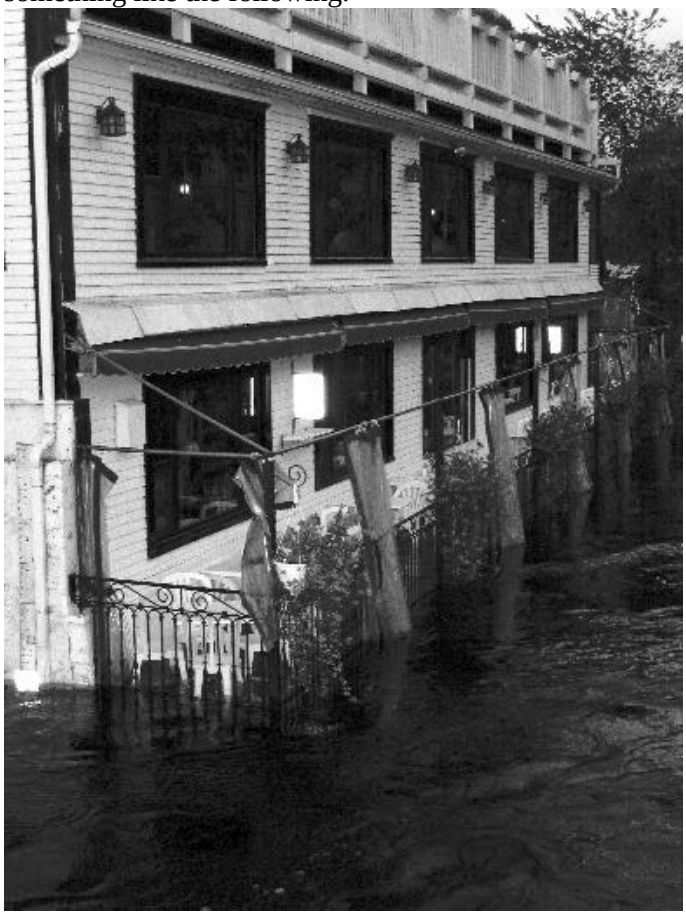
Question No. 1 Histogram Processing **10 marks**

1. Download the following image "**flood.jpg**" and store it in MATLAB's "Current Directory".



2. Load the image data.
3. Display the histogram of the image.
4. Use histogram equalization to create a new image with more contrast.
5. Display the histogram of the image created in step 4.

6. Use histogram matching (using M-Files from above notes) to produce an image that looks something like the following:



7. Display the histogram of the image created in step 6.

Tips

- manualhist might be helpful, but it is not necessary. You only have to use the twomodegauss function to produce the histogram to match. It takes arguments in the same order as manualhist.

Deliverables:

- An M-File called histograms.m with a script that displays the following (as described above)
  - Original image
  - Histogram of original image
  - Image produced from histogram equalization
  - Histogram of image produced from histogram equalization
  - Image produced from histogram matching
  - Histogram of image produced from histogram matching

Question No. 2 (Edge Maps) **10 marks**

1. Download the following image "**building.jpg**" and store it in MATLAB's "Current Directory".

2. Identify which of the following is the result of zerocrossing or Canny detector

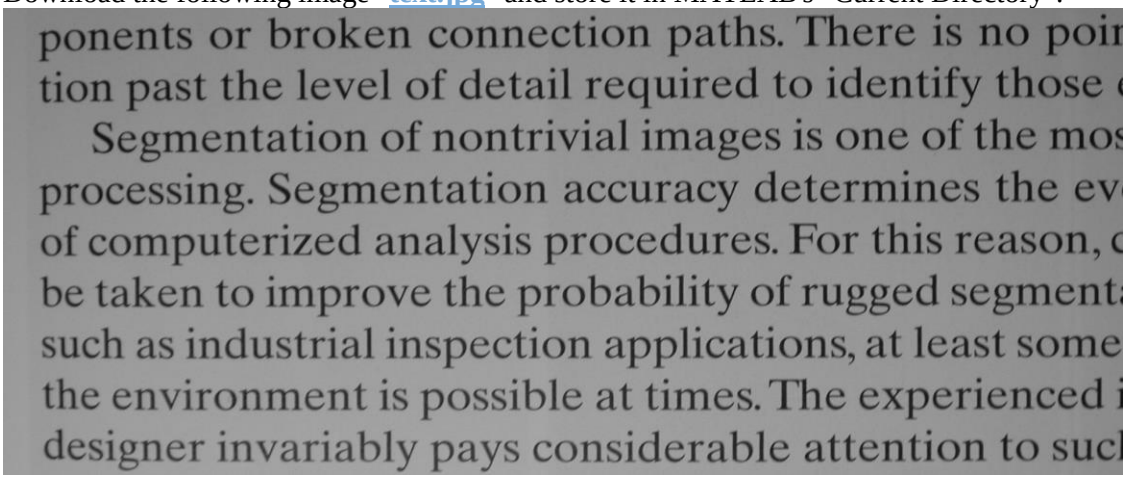
Image 1

Image 2

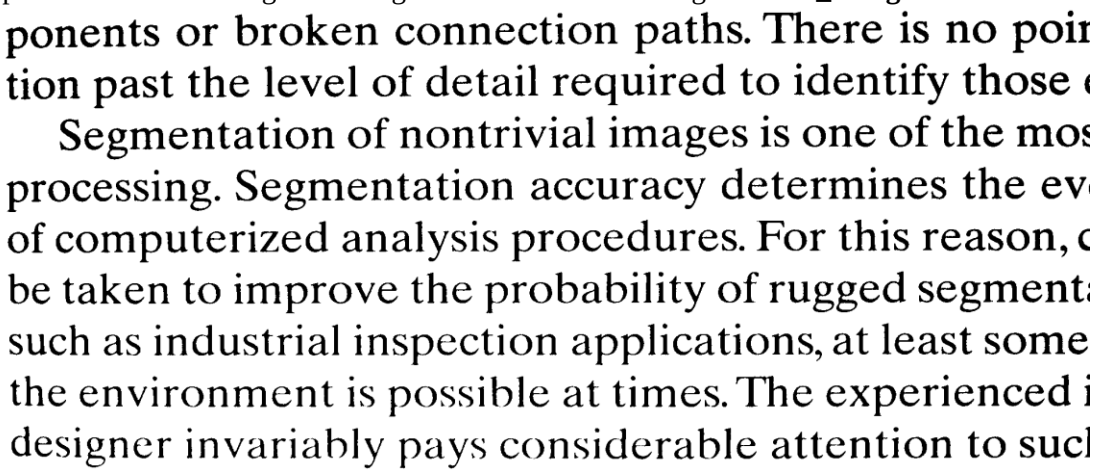
3. Reproduce the results for input image **building.jpg**. I encourage you to try different T and Sigma to get the best result.

Question No. 3 (Automatical Thresholding)

1. Download the following image "text.jpg" and store it in MATLAB's "Current Directory".



2. Reproduce the results to get the image looks like the following **enhance_text.gif**.



Deliverables:

- Part 1
  - A script called edge_maps.m that:
    - Identifies, with code comments, which edge detection method was used for the two images.
    - Produces zero crossing and canny edge maps for **building.jpg**.
- Part 2
  - A script called my_threshold.m that:
    - finds and prints the optimal T for text.jpg
    - uses T to threshold text.jpg and displays the results.

1. Image Processing Toolbox (Basic Concepts)

"***The Image Processing Toolbox is a collection of functions that extend the capability of the MATLAB numeric computing environment. The toolbox supports a wide range of image processing operations.***" (from the online **Image Processing Toolbox, User's Guide** - "Getting Started" - "What is Image Processing Toolbox?"

- The Image Processing Toolbox provides a set of tools, which allow you to view and manipulate images
- A few classic things that the Image Processing Toolbox allows you to do are:
    - histogram equalization (imhist)
    - filtering (imfilter)
    - fast Fourier transform (fft2)
    - converting color images to grayscale (rgb2gray)
    - edge detection (edge)
- To see demos of what you can do with the image processing toolbox, you can type: iptdemos
- The discussion that follows is necessarily limited. Please refer to help or doc pages for more details on the functions discussed.

1.1 Reading and Displaying Images

- There are a series of photos that come as part of the image processing toolkit. To get the list of images and credits, you can type:
  help imdemos
  or
  doc imdemos

- If you want to view any of these photos, you can use the imshow, which opens a separate window displaying the image. For instance:
  imshow('football.jpg');
  imshow('coins.png');
  imshow('autumn.tif');
  imshow('board.tif');

- If we want to modify or enhance one of these pictures, we should store it in a temporary array. We can do this using imread. For instance, to create an array I that holds the information about an image called pout.tif, we use the following syntax:
  I=imread('pout.tif');
  to then view the image in I:
  imshow(I);

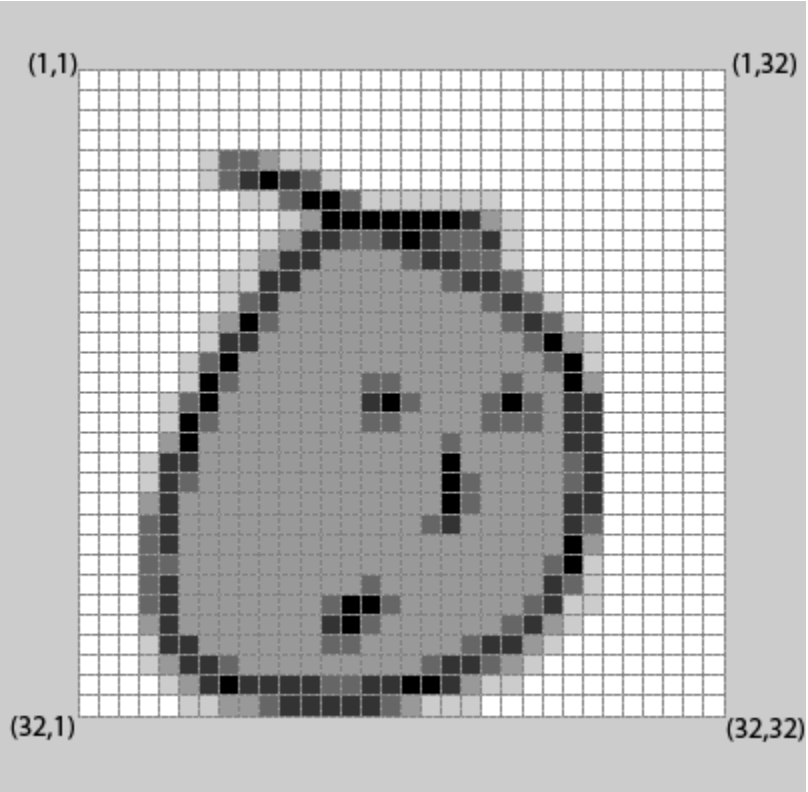
- The following table is meant to summarize the commands you can use for reading or displaying images

Command	Description
I=imread('filename');	Reads a picture stored in 'filename' and stores it in I
[I,map]=imread('filename.gif');	Reads an indexed picture stores it in I with colormap map.
imshow(I)	Displays the image stored in I in a separate window
imshow(I, map)	Displays the indexed image stored in I in a separate window using colormap map.

imtool(I)	Displays the image. Allows you to look at pixel values in a region, etc. Displaying indexed images is similar to imshow.

1.2 What is stored in an image?

Don't panic. When we get to images, we are still working with matrices that can be accessed through indices. The upper left-hand corner of the image is indexed by (1,1). The general format of indexing is (row, col) or (y,x). Be careful here because you are used to thinking in Cartesian (x, y) coordinates. The following diagram shows the indices to all four corners of a small image:

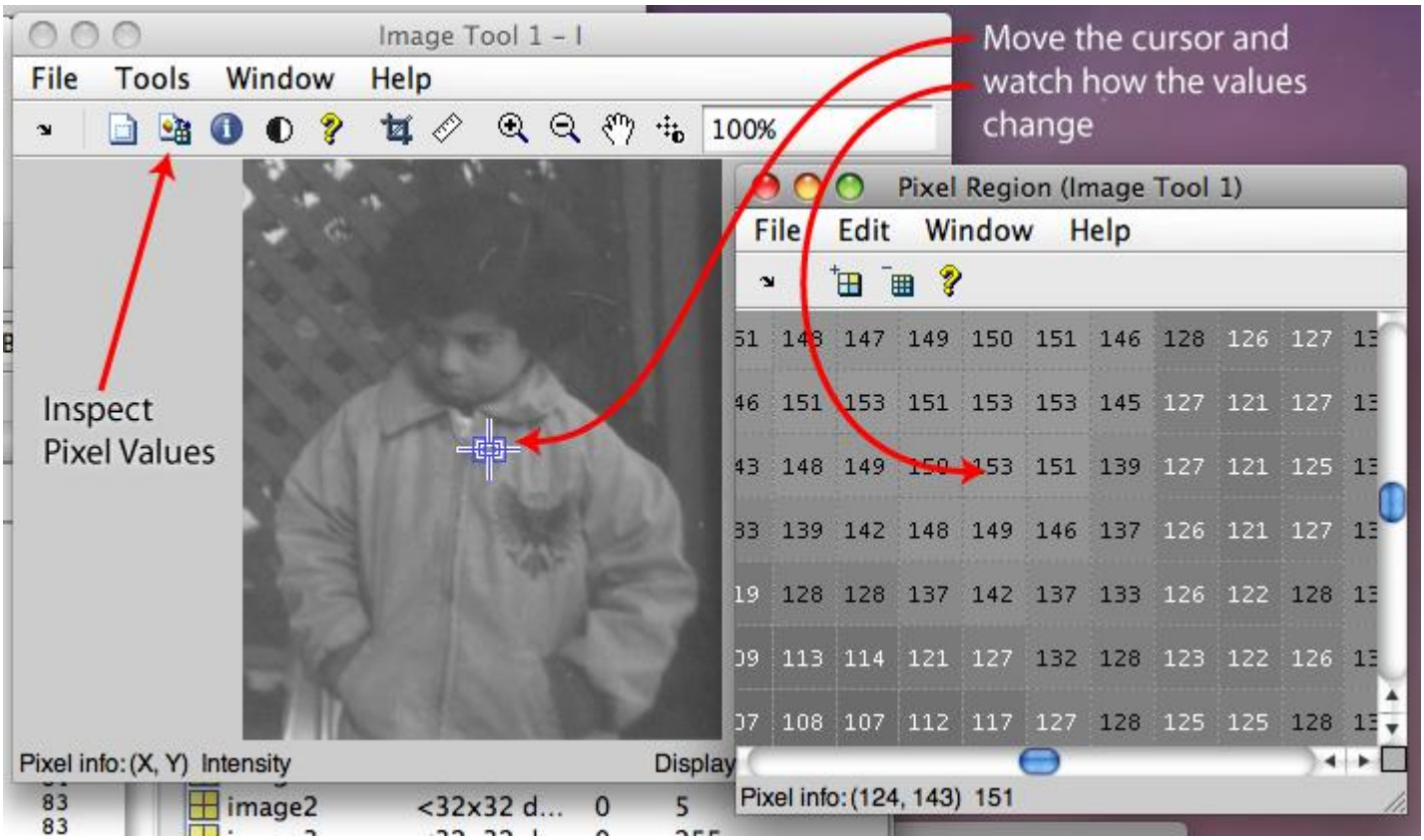


All the basic matrix operations that you have learned still apply. What you might be asking yourself is what is stored in an image matrix? Let's use a grayscale image ("pout.tif") as an example. This image is stored in a 291 x 240 matrix. That means there are 291 rows and 240 columns. Each element corresponds to a single pixel value.

In the case of "pout.tif", the values are between 0 and 255. Zero is full black and 255 is white. Values between zero and 255 are shades of gray.

- Let's assume that you have already read "pout.tif" into I.
- If you type imtool(I), you will start up the image tool.

- You can get a close up view the pixels of a particular region using the "Inspect pixel values" option from the toolbar.
- You can resize the view area by dragging the corners of a box on the main image. If the box is small enough, the close up view will show pixel values.
- Click the middle of the box and drag the cursor around on the image to see that brighter areas are closer to 255 and darker areas are closer to 0.



Note: The "Pixel info" at the bottom displays the value at (column, row) or (x,y). This is different from the MATLAB subscripting syntax, which is in (row, column) or (y,x) format.

1.3 Displaying Image Information

The following table summarizes some ways to get information about an image. These are not specific to the Image Processing Toolbox.

Command	Description
whos	To get information about size, type, and bytes, of all variables.
whos I	For information about an image stored in I.
size(I)	To get the size of the image stored in I.

class(I)	To get type of data stored in I

Color Formats

There can be different types of data stored in the image array (or matrix). The following table, based on the table on page 2-3 of the ***Image Processing Toolbox User's Guide***, summarizes the different image types and values.

Image Type	Intensity Values/Ranges
Binary	• logical (boolean) array of 0's and 1's • 0 is black and 1 is white

Indexed (pseudocolor)	• values in image are indices into a colormap
	• range of values where p is the length of the colormap:
	  ○ single or double arrays [1, p]
	  ○ logical, uint8, or uint16 [0, p-1]

Grayscale (intensity)	• Values have ranges as follows:
	  ○ single or double arrays [0, 1]
	  ○ uint8 [0, 255]
	  ○ uint16 [0, 65 535]
	  ○ int16 [-32 768, 32 767]

Truecolor (RGB)	• Values have the following range:
	  ○ single or double arrays [0, 1]
	  ○ uint8 [0, 255]
	  ○ uint16 [0, 65 535]

Truecolor

Until now you have worked with grayscale images and you have sees that that there are different image types. Most of them work similarly. For each pixel there's a single value, from a range of values appropriate to a data type, stored in a two dimensional matrix. To represent color we have to add another dimension. For instance, if you read image "autumn.tif" into I2 and view the details of I2 using:

```
I2=imread('autumn.tif');
whos I2
```

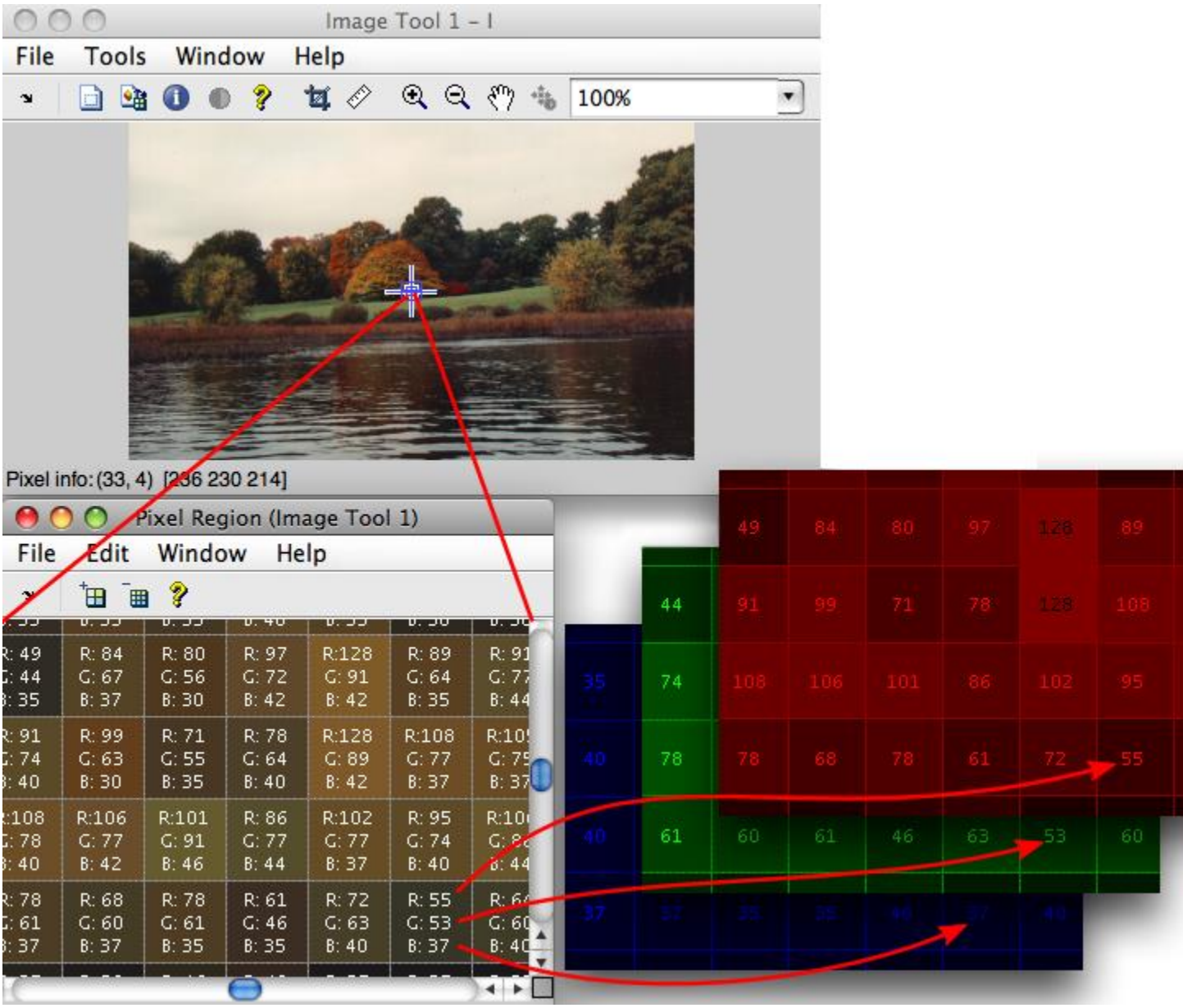
The image size is: 206x345x3

The extra dimension is the color component. For each pixel, we now have a representation of the intensity of red, green, and blue. In autumn.tif, 255 represents the highest intensity, and 0 represents the lowest intensity. For instance, to make a pure red, red would be 255, green would be 0, and blue would be 0.

At pixel (93,180) in autumn.tif,

- the red intensity is represented by I2(93,180,1)
- the green intensity is represented by I2(93,180,2)
- the blue intensity is represented by I2(93,180,3)

The following diagram represents the three color components at pixel values. Often, they are represented as channels as is shown in the diagram below. Usually, there is a red channel, a green channel and a blue channel.



If you only wanted to view the red channel, which of the following statements would you write?

a. imshow(I2(1, :, :));
b. imshow(I2(:, :, 1));
c. imshow(I2(:, 1, :));

What does the image look like?
What happens if you try another color channel?

1.4 Converting Between Different Formats

The following table lists commands provided by the Image Processing Toolbox that convert between the different formats given above.

Image format conversion (Within the parenthesis you type the name of the image you wish to convert.)	
MATLAB command:	**Operation:**
gray2ind()	Convert from intensity format to indexed format.
ind2gray()	Convert from indexed format to intensity format.
ind2rgb()	Convert from indexed format to RGB format.
mat2gray()	Convert a regular matrix to intensity format by scaling.
rgb2gray()	Convert from RGB format to intensity format.
rgb2ind()	Convert from RGB format to indexed format.
im2bw()	Convert an intensity/indexed/RGB image to a thresholded black and white binary image.
dither()	Convert from intensity/indexed/RGB format to dithered indexed format with custom colormap. Defaults to black and white if no colormap is provided,

The command mat2gray is useful if you have a matrix representing an image but the values representing the gray scale range between, let's say, 0 and 1000. The command mat2gray automatically re-scales all entries so that they fall between 0 and 255 (if you use the uint8 class) or 0 and 1 (if you use the double class).

As you read through the lab notes, you will see examples of these functions, often to change an indexed or RGB image to grayscale to simplify processing.

Here are some examples of how to use some conversion commands:

```
%Start with a truecolor image
img=imread('onion.png');
imshow('onion.png')
```

```
%Convert to binary/Black and White
img_bin=im2bw(img);
imshow(img_bin)
```

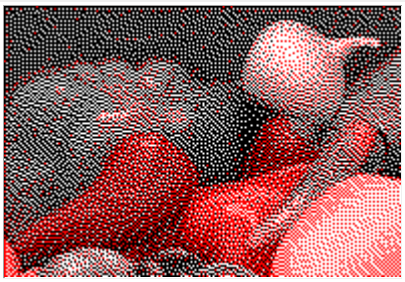
```
%Convert to binary/Black and White
%with custom threshold (default is 0.5)
img_bin2=im2bw(img, 0.3);
imshow(img_bin2)
```

```
%Convert to grayscale
img_gray=rgb2gray(img);
imshow(img_gray)
```

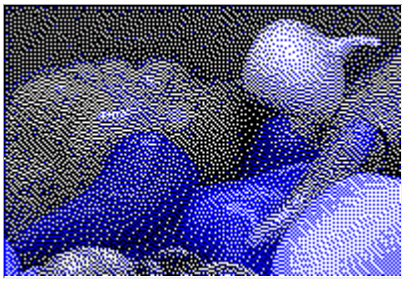
```
%reduce to indexed color
[img_indexed, map]=rgb2ind(img,8);
imshow(img_indexed, map)
```

```
%reduce to indexed color
%without dithering
[img_nodither, map2]=rgb2ind(img,8, 'nodither');
imshow(img_nodither, map2)
```

```
%reduce to indexed color
%using a custom colormap
map_custom=[
  1 1 1;  %white
  1 0 0;  %red
  0 0 0]; %black
img_custommap=dither(img, map_custom);
imshow(img_custommap, map_custom)
```



```
%Of course you can swap colormaps...
map_custom2=[
  1 1 1;  %white
  0 0 1;  %blue
  0 0 0]; %black
imshow(img_custommap, map_custom2)
```



1.5 Saving Images

To save an image, use the imwrite function. The parameters you send to the function depend on the file format and image type.

Saving a Binary, Grayscale or Truecolor Image

When saving a binary, grayscale or truecolor image, all you need to specify are the matrix and filename. For example:

```
imwrite(img_bin, 'onion_bw.png')
```

In this example, the image would be stored as a 1-bit image in png format. The format is automatically guessed from the file extension.

Saving an Indexed Image

When you save an indexed image, it is important to provide the colormap like this:

```
imwrite(img_indexed, map, 'onion.gif');
```

Notes

- If you save an indexed image to a format that doesn't support indexed images, it will be converted to a compatible format automatically. With JPEG file types, this may lead to an undesirable result.
- If you forget the colormap, the image will still be saved—the indexing numbers will be interpreted as shades of gray and may look chaotic. If your picture had a small colormap, the result may look black.

Options

There are various options that you can send to imwrite to control how different file types save. For example you can change the compression level of a JPEG like this:

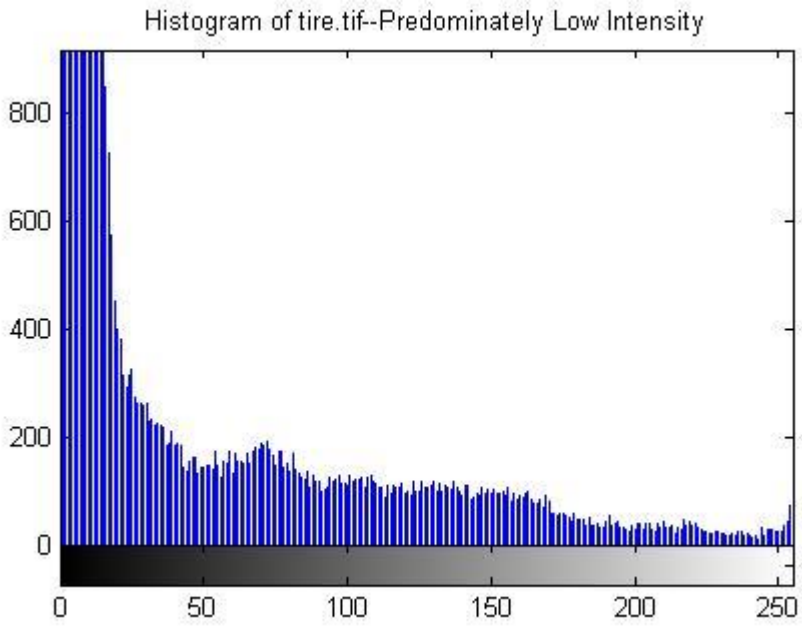
```
imwrite(img, 'onion.jpg', 'Quality', 25);
```

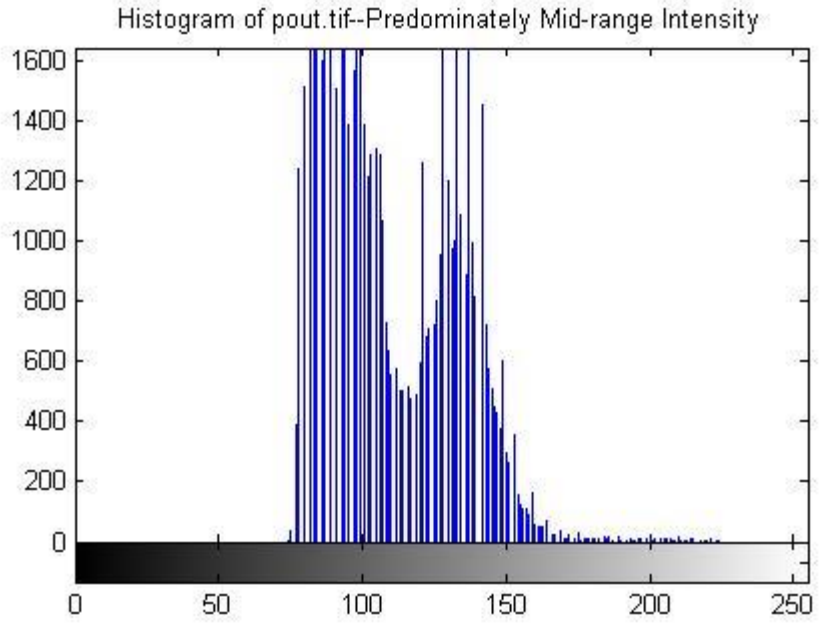
---

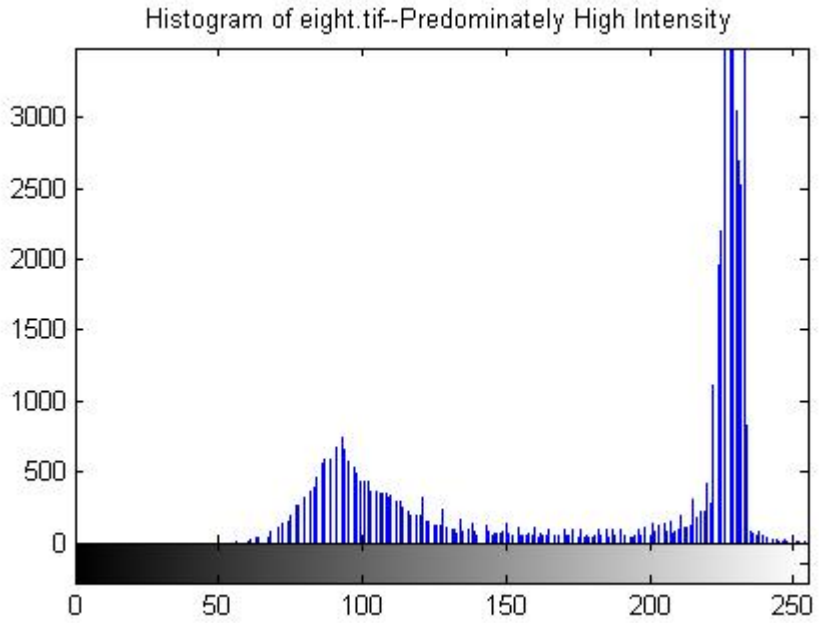
2. Histograms processing

Histograms are a way of visualizing the predominant intensities of an image. As a definition, image histograms are a count of the number of pixels that are at a certain intensity. When represented as a plot, the x-axis is the intensity value, and the y-axis is the number of pixels with that intensity value. The following are examples of histograms with predominately low, mid and high range intensities.

Which image would you expect to be darkest? lightest?







Notice that the x-axis is the intensity value from 0 to 255 (these images are uint8). The y-axis varies depending on the number of the pixels in the image and how their intensities are distributed.

MATLAB easily displays image histograms using the function imhist(I). The above plots were created with the following commands:

```
tire=imread('tire.tif');
figure,imhist(tire);
title('Histogram of tire.tif--Predominately Low Intensity');
```

```
pout=imread('pout.tif');
figure,imhist(pout);
title('Histogram of pout.tif--Predominately Mid-range Intensity');
eight=imread('eight.tif');
figure, imhist(eight);
title('Histogram of eight.tif--Predominately High Intensity');
```

2.1 Histogram Equalization

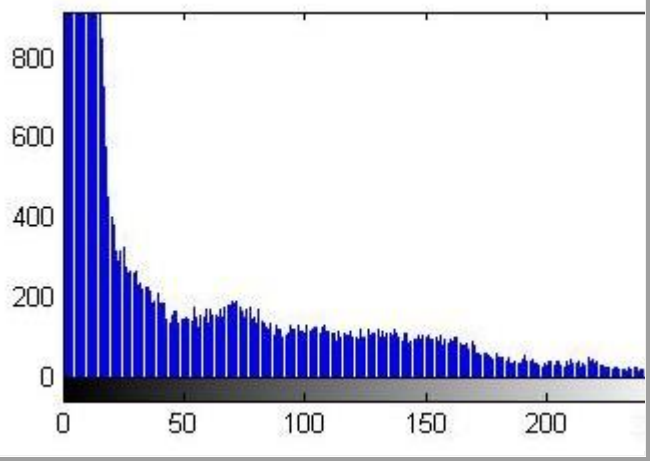
See Chapter 5.5 in your textbook.

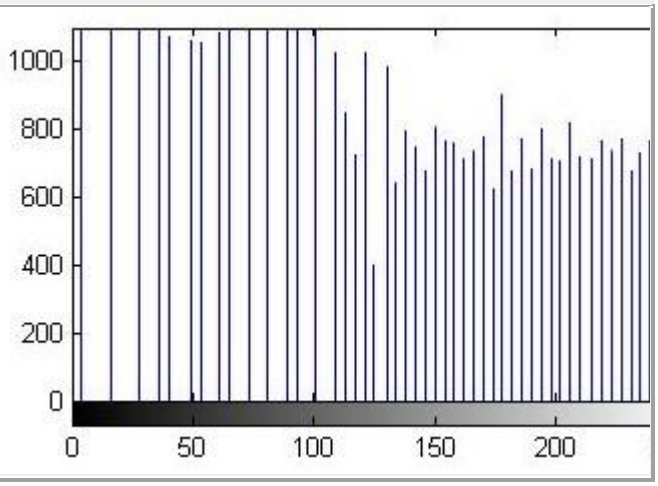
The idea behind Histogram Equalization is that we try to evenly distribute the occurrence of pixel intensities so that the entire range of intensities is used more fully. We are trying to give each pixel intensity equal opportunity; thus, ***equalization***. Especially for images with a wide range of values with detail clustered around a few intensities, histograms will improve the contrast in the image.

In MATLAB, the function to perform Histogram Equalization is histeq(I).

Following are some examples of Histogram Equalization and its results on the images that have predominately low and mid range intensities:

2.1.1 Low Range Intensities

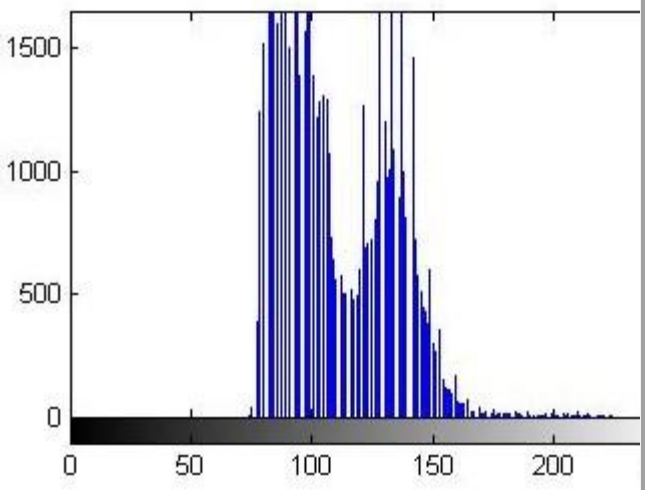
	Image	Histogram
Original		

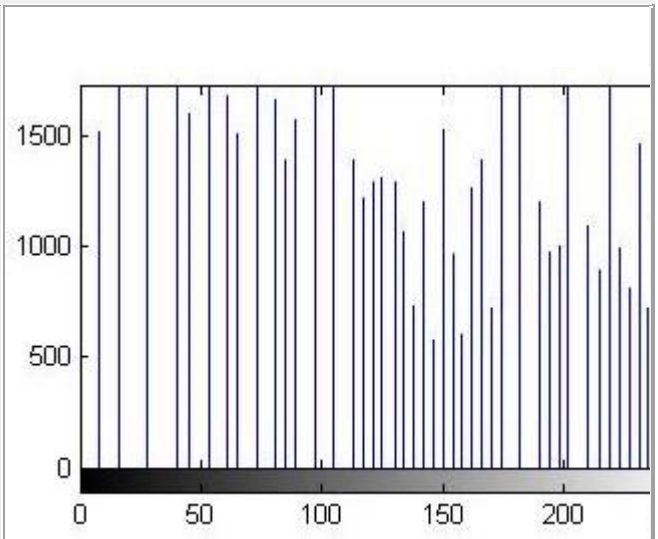
With Histogram Equalization		

The above diagrams were created using the following calls:

```
tire=imread('tire.tif');
imshow(tire);
figure, imhist(tire);
tire_eq=histeq(tire);
figure, imshow(tire_eq);
figure, imhist(tire_eq);
```

2.1.2 Mid Range Intensities

	Image	Histogram
Original		

With Histogram Equalization		

The above diagrams were created using the following calls:

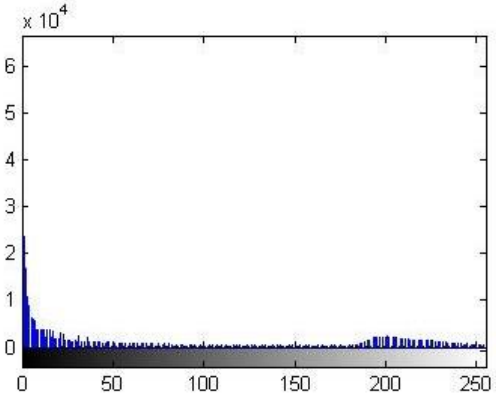
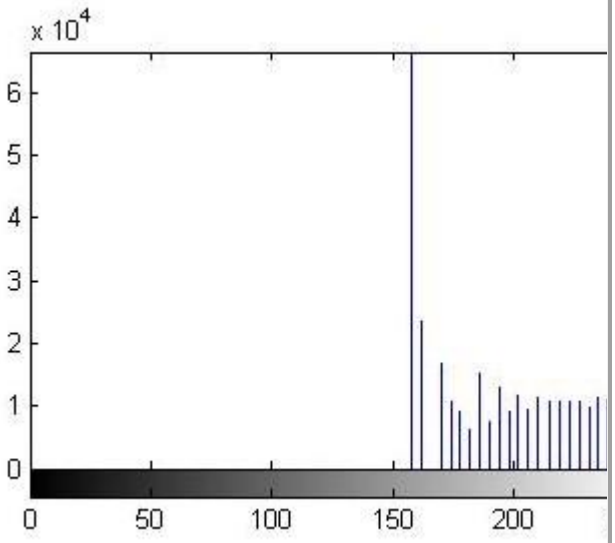
```
pout=imread('pout.tif');
imshow(pout);
pout=rgb2gray(pout);
figure, imhist(pout)
pout_eq=histeq(pout);
figure, imshow(pout_eq);
figure, imhist(pout_eq);
```

Note: imshow(image,[low, high]) also performs a histogram stretch: all values less than or equal to low will be shown as black;all values greater than or equal to high will be shown as white; the rest will be stretched from 0 to 255.

imshow(pout, [75, 160])	

2.2 Histogram Matching

Sometimes, histogram equalization does not produce the contrast or results that we expect. Consider the following example, taken from ***Digital Image Processing, Using MATLAB***. The **original image** is courtesy of NASA.

	Image	Histogram
Original		
With Histogram Equalization		

The above images were created using these steps:

1. Download Fig0310(a)(MoonPhobos).tiff
2. Find it in the Current Directory window.
3. Double click on Fig0310(a)(MoonPhobos).tiff—it will now appear in your Workspace window with the name Fig03100x28a0x290x28MoonPhobos0x29.
4. Use the following code in the Command Window.
    - ***Hint:*** press tab to autocomplete long names.

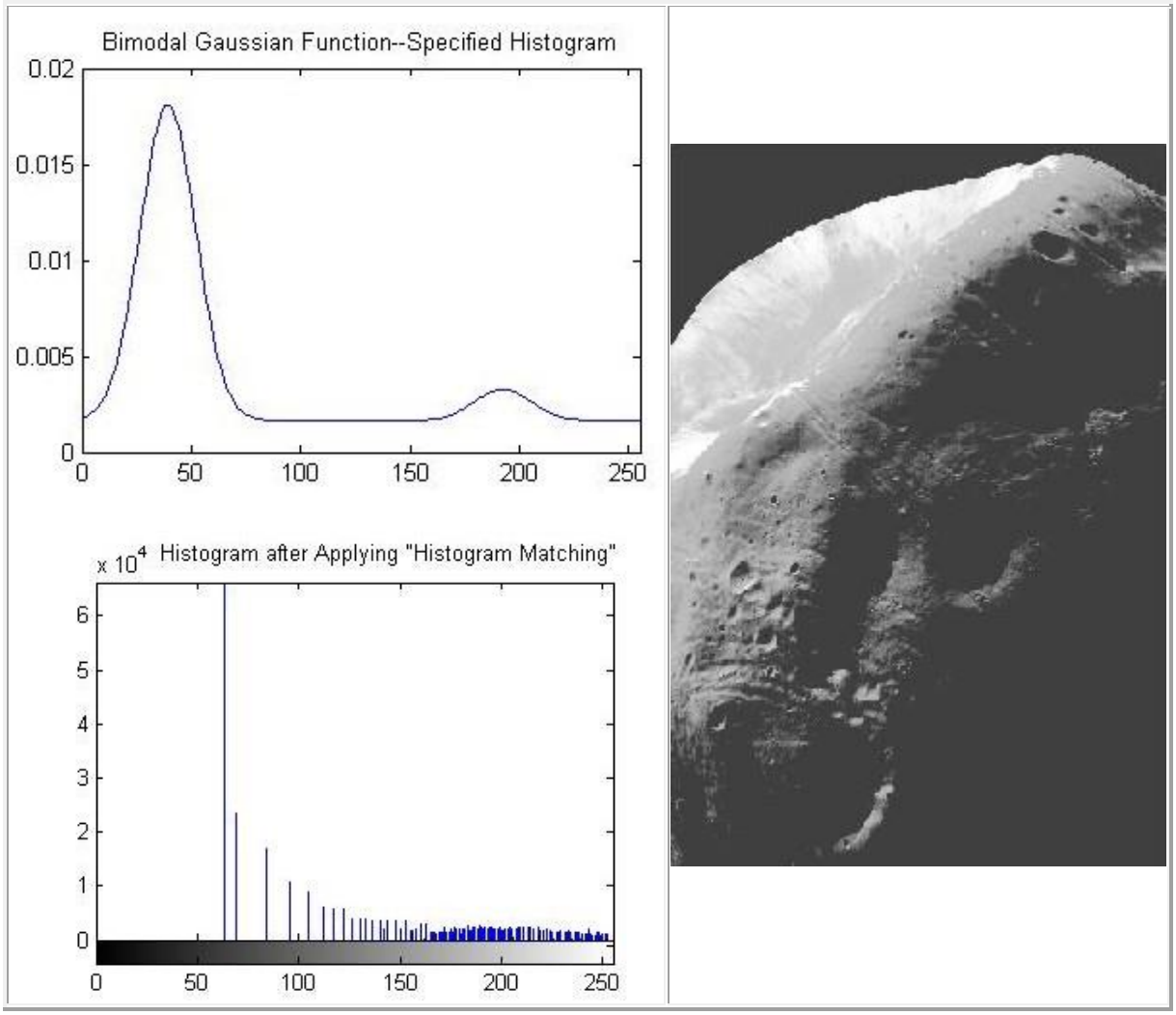
```
moon=Fig03100x28a0x290x28MoonPhobos0x29;
imshow(moon);
figure, imhist(moon);
moon_eq=histeq(moon);
figure, imshow(moon_eq);
figure, imhist(moon_eq);
```

You will notice that there are very few low intensity values in the histogram equalized image above. To fix this, we will use histogram matching (or histogram specification).

To provide some background, histogram equalization tries to create an equal probability of each intensity occurring. This equal probability is represented by a horizontal line. If you wanted to emulate the results above, you could pass a horizontal line as a second argument to histeq. The following code can be used to yield results similar to the above histogram equalization

```
p(1:256)=1
g=histeq(moon,p);
figure, imshow(g);
figure, imhist(g);
```

With histogram matching, we provide a multimodal (multi-peaked) Gaussian function as an argument to histeq. In contrast to the horizontal line of histogram equalization, the Gaussian function specifies that certain intensity ranges will have more probability of occurring than others. To speak intuitively, by passing a multimodal Gaussian function as a second argument to histeq, the modified image histogram will have peaks that approximately match the peaks of the Gaussian function. Note this approximate match in the the moon example below:



In order to create the above results, two M-functions were used

- **manualhist** provides a user interface to type in different arguments and see the resulting histogram (created by calling the twomodegauss function)
- **twomodegauss** generates a bimodal Gaussian-like function using seven arguments

With the above two functions in the "Current Directory", the following calls were made to produce the results in the above table:

```
p=manualhist;
%the following arguments were typed at the prompt
% 0.15 0.05 0.75 0.05 1 0.1 0.002  - input is looped don't forget to end with x
g=histeq(moon, p);
figure, imshow(g);
figure, imhist(g);
```

3. The Return of the M-Files

At the end of this section, we are going to take a look at a function, written in MATLAB, that is able to apply all of the intensity transformations mentioned above. Before doing that, the following provides some background information about M-Files and functions.

3.1 Scripts

Earlier, we briefly looked at M-Files. Specifically, we saw an m-file script. Remember:

- The extension is .m
- If we call our file myScript.m, to run it in MATLAB:
  - we need to have our "Current Directory" set to the location of myScript.m
  - we can invoke it by typing its name without the extension:

```
myScript
```

- The contents of the script are regular MATLAB code

Scripts do not take input arguments or return anything as output. However, anything that is created when the script is run remains in the workspace.

3.2 Functions

By contrast, functions can accept input arguments and return output. Functions are distinguished from other M-files by the keyword function in the first line of code. Let's take a look at a sample function acosd. The code below can be seen in MATLAB using the command:
type acosd

```
function y = acosd(x)
%ACOSD  Inverse cosine, result in degrees.
%   ACOSD(X) is the inverse cosine, expressed in degrees,
%   of the elements of X.
%
%   Class support for input X:
%      float: double, single
%
%   See also COSD, ACOS.

%   Copyright 1984-2004 The MathWorks, Inc.
%   $Revision: 1.1.6.3 $  $Date: 2004/06/25 18:51:38 $

if ~isreal(x)
   error('MATLAB:acosd:ComplexInput', 'Argument should be real.');
end

y = 180/pi*acos(x);
```

A couple of things to note about this code are the following:

- the keyword function is used
- the function's name is acosd. Therefore, the name of the M-File will be acosd.m
- the return, "y", is indicated in the function statement as y = acosd(x)
- Any arguments to the function are specified in parenthesis after the function name. In acosd, "x" is the only input argument.
- Anything after the % is ignored as a comment
- The first block of comments up to the first blank line or executable line are displayed when you type
  help acosd
  (all functions are like this)

3.3 Function Inputs and Outputs

MATLAB functions are very flexible compared to those of many other languages. They can have multiple and varying numbers of inputs—which is pretty normal. They can also have **multiple and varying numbers of *outputs***—which may be new to you. To help you manage this complexity, MATLAB has some special functions. The following table summarizes these functions:

Function	Description
nargin	returns the number of arguments input into the function
nargout	returns the number of arguments output from a function
nargchk(low, high, number)	returns "Not enough input parameters" if number is less than low and returns "Too many input parameters" if number is greater than high. If the value of number is between (or equal to) low and high, then an empty matrix is returned. Often, nargchk is used in conjunction with error. The error function stops execution and prints the message if number is either too high or too low. A sample usage of nargchk inside a function is: error(nargchk(2, 3, nargin)); %This checks if the input arguments (nargin) are between 2 %and 3 (inclusive). If they aren't, then execution stops and %an error message (with the string returned from nargchk) is %printed
varargin	accepts a variable number of arguments
varargout	returns a variable number of arguments

3.3.1 varargin

To write a function with a variable number of arguments, you can write:
function m=testFunction(varargin)
This means that you could have zero to as many arguments as you want for testFunction. The following code is meant to test this.

```
function m=testFunction(varargin)
%TESTFUNCTION tests variable number of arguments
%   The following are valid calls to testFunction
%   a=testFunction
%   b=testFunction(val)
%   c=testFunction(val1,val2)
```

```
%   d=testFunction(val1,val2,...moreval)


number=nargin;

switch number
   case 0
      m=0;
   case 1
      m=1;
   case 2
      m=2;
   otherwise
      m='more than two';
end
```

Notice that there are no return statements, instead, m (the return variable) is assigned a value. For instance, if you have zero arguments, then m will be set to 0; if you have one argument, then m will be set to 1; if you have three or more arguments, then m will be assigned a message of: 'more than two'.

If you want to access any one of the arguments sent as a variable argument, you can use the notation varargin{#}. For instance, varargin{1} will return the first argument. The following function constructs m as an array that contains all of the input arguments.

```
function m=testFunction2(varargin)
%TESTFUNCTION2 displays all of arguments sent to the function
%   The following are valid calls to testFunction2
%   a=testFunction
%   b=testFunction(val)
%   c=testFunction(val1,val2)
%   d=testFunction(val1,val2,moreval)


number=nargin;
% m=[];  %this goes with the commented line below


if number>0
   for i=1:number
     m(i)=varargin{i};  %add one more argument to m
     % m=[m varargin{i}];  %this way works as well
   end

else
   m='no arguments';
end
```

This function uses if/else and for statements. Notice that end is used to end these statements.
The for makes use of the colon operator; in each loop, i will have an incremented value from 1 to the number of arguments.

You can use varargin with other arguments, but varargin must come last. For example, if you require one argument and then a variable number of arguments, you would write this:
function m=testFunction3(x, varargin)
In this case, you require at least one argument and it will be stored in x.

3.3.2 Varargout

The idea behind varargout is not far from varargin. However, first we will describe something that you might not be familiar with in other languages—MATLAB can return multiple arguments. To have a function with multiple returns, you can write:
function [m, n] = testFunction4(x)
This indicates that there are two returns to be stored in m and n.

You can also specify a variable number of arguments to be returned. The following code returns a variable number of arguments depending on the number of arguments specified when the function was called:

```
function [varargout] = testFunction5(x)
%TESTFUNCTION5 tests how variable arguments work
%  testFunction5(x) does not return anything
%  m=testFunction5(x) returns one argument (1+x)
%  [m,n]=testFunction5(x) returns two arguments (1+x and 2+x)
%  [m,n,o]=testFunction5(x) returns three arguments (1+x, 2+x, and 3+x)
%  so on.

number=nargout;

if number>0
   for i=1:number
      varargout{i}=i+x;
   end
end
```

For instance, if you make a call like:
[m,n,o]=testFunction5(1)
The following will be returned:

```
m =
    2

n =
    3

o =

    4
```

Edge Detection

1. Edge Detection

In the field of Image Processing, the extraction of geometric features from images is very common problem. Over the years, several different approaches have been devised to extract these features.

These different approaches can be characterized and classified in several different ways.

Some of the techniques involve **global examination** of the image, while others only involve **local examination** of each pixel in the image.

1.1 What is Edge Detection?

Typically, the first step in the process is to perform some form of edge detection on the image, and to generate a new **Binary edge** image that provides the necessary segmentation of the original image.

Edge detection algorithms operate on the premise that each pixel in a grayscale digital image has a **first derivative**, with regard to the change in intensity at that point, if a significant change occurs at a given pixel in the image, then a white pixel is placed in the binary image, otherwise, a black pixel is placed there instead.

In general, the gradient is calculated for each pixel that gives the degree of change at that point in the image. The question basically amounts to how much change in the intensity should be required in order to constitute an edge feature in the binary image.

A threshold value, **T**, is often used to classify edge points.

Some edge finding techniques calculate the **second derivative** to more accurately find points that correspond to a local maximum or minimum in the first derivative. This technique is often referred to as a **Zero Crossing** because local maxima and minima are the places where the second derivative equal zero, and its left and right neighbors are non-zero with opposite signs.

1.2 Edge Detection With the edge Function in MATLAB

The Image Processing Toolbox's edge function provides several derivative estimators based on the criteria just you learned above.

For some of these estimators, it is possible to specify whether the edge detector is sensitive to horizontal or vertical edges or to both. The general syntax for the edge function is:

```
[g, t]= edge(f, 'method', parameters ... , options ... );
```

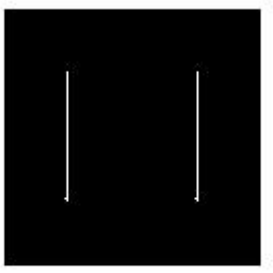
where f is the input image. The most popularly used approaches are listed in the following table. Additional approaces and control parameters are discussed in MATLAB's documentation. Your lab instructor will give you a tour of some of the details of the three approaches below in the lab lecture.

In the output, g is a logical array with 1's at the locations where edge points were detected in f and 0's elsewhere. Parameter t is optional, it gives the threshold used by edge to determine which gradient values are strong enough to be called edge points.

Edge Detector	Basic Properties
Sobel	Finds edges using the Sobel approximation to the derivatives.
Canny	Finds edges by looking for local maxima of the gradient of f(x,y). The gradient is calculated using the derivative of a Gaussian filter. The method uses two thresholds to detect strong and weak edges, and includes the weak edges in the output only if they are connnected to strong edges. Therefore, this method is more likely to detect true weak edges.
Zero Crossing	Finds edges by looking for zero crossings after filtering f(x,y) with a user defined filter. This method is often combined with the Laplacian of a Gaussian filter, but any filter that approximates the second derivative of the image's data will do. If you don't provide your own filter, Zero Crossing uses the equivalent of this filter: H = FSPECIAL('log', 13, 2); See also the log section of help FSPECIAL.

Now let us try those different methods and parameters to see what is the difference between them:

```
T=100;
f=zeros(128,128);
f(32:96,32:96)=255;
[g1, t1]=edge(f, 'sobel', 'vertical');
imshow(g1);
t1
```



```
sigma=1;
f=zeros(128,128);
f(32:96,32:96)=255;
[g3, t3]=edge(f, 'canny', [0.04 0.10], sigma);
```

```
figure,imshow(g3);
t3
```