Image Processing with MATLAB

This tutorial discusses how to use MATLAB for image processing. Some familiarity with MATLAB is assumed (you should know how to use matrices and write an M-file).

[References : MATLAB Basics | Image Processing Toolbox Guide]

Contents

Image representation

A digital image differs from a photo in that the values are all discrete. Usually they take integer values. A digital image can be considered as a large two dimensional array of discrete cells, each of which has a brightness associated with it. These dots are called pixels.

Aspects of Image Processing

Image Enhancement: Processing an image so that the result is more suitable for a particular application. (sharpening or deblurring an out of focus image, lighlighting edges, improving image contrast, or brightening an image, removing noise)

Image Restoration: This may be considered as reversing the damage done to an image by a known cause. (removing of blur caused by linear motion, removal of optical distortions)

Image Segmentation: This involves subdividing an image into constituent parts, or isolating certain aspects of an image. (finding lines, circles, or particular shapes in an image, in an aerial photograph, identifying cars, trees, buildings, or roads.

Image representation

There are five types of images in MATLAB.

1. **Grayscale.** A grayscale image *M* pixels tall and *N* pixels wide is represented as a matrix of double datatype of size *M*×*N*. Element values (e.g., MyImage(m,n)) denote the pixel grayscale intensities in [0,1] with 0=black and 1=white.

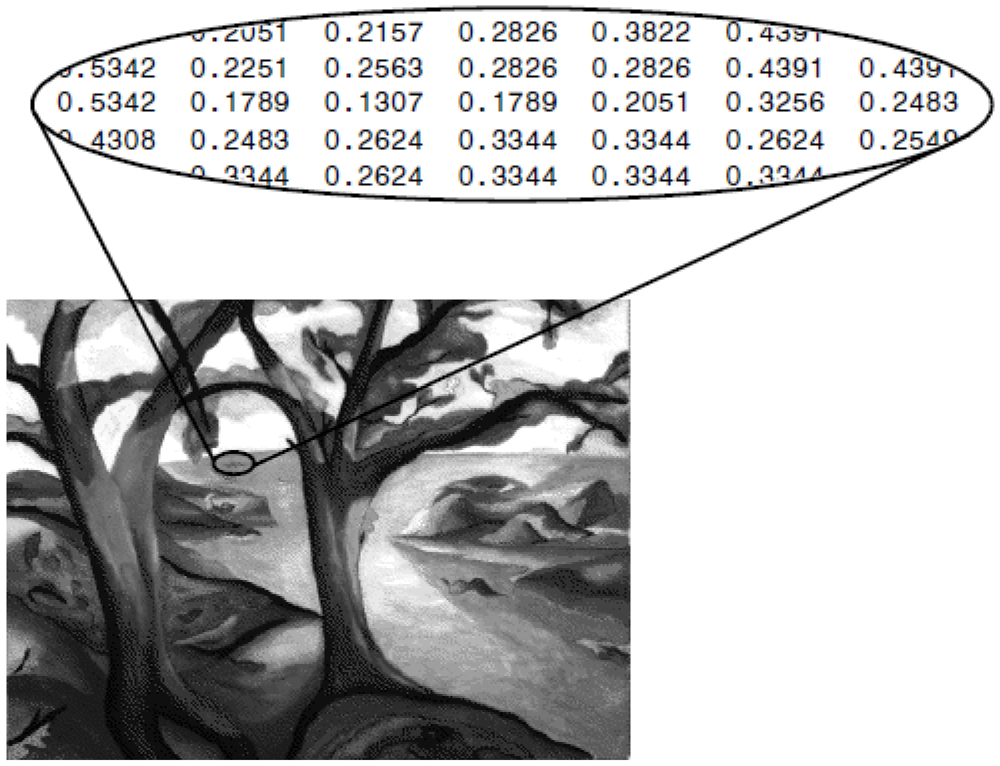


Fig: Grayscale image of class Double:

2. **Truecolor RGB.** A truecolor red-green-blue (RGB) image is represented as a three-dimensional*M*×*N*×3 double matrix. Each pixel has red, green, blue components along the third dimension with values in [0,1], for example, the color components of pixel (m,n) are MyImage(m,n,1) = red, MyImage(m,n,2) = green, MyImage(m,n,3) = blue.

If each of these components has a range 0–255, this gives a total of **2563** different possible colors. Such an image is a "stack" of **three matrices**; representing the **red**, **green** and **blue** values for each pixel. This means that for every pixel there correspond 3 values.

True color image may be of class Int or Double.

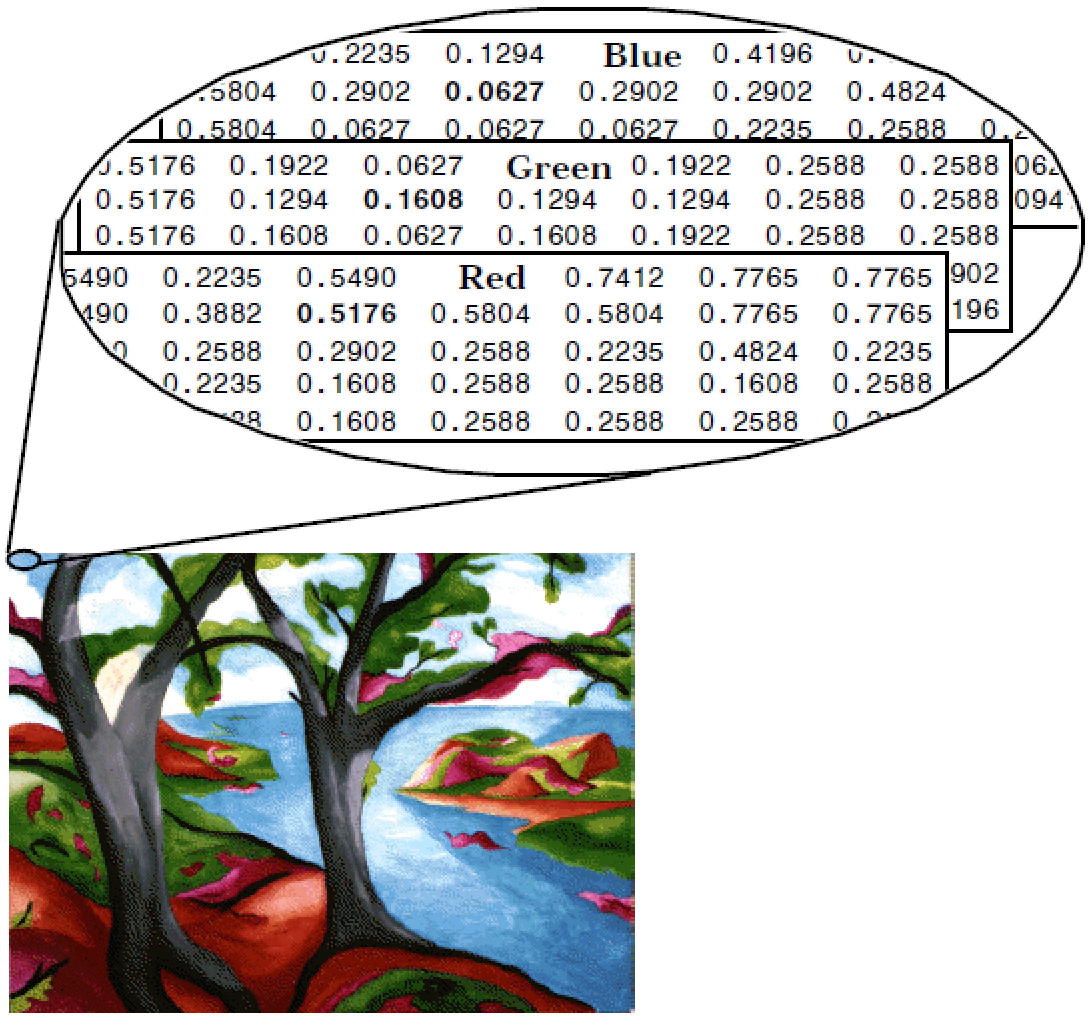
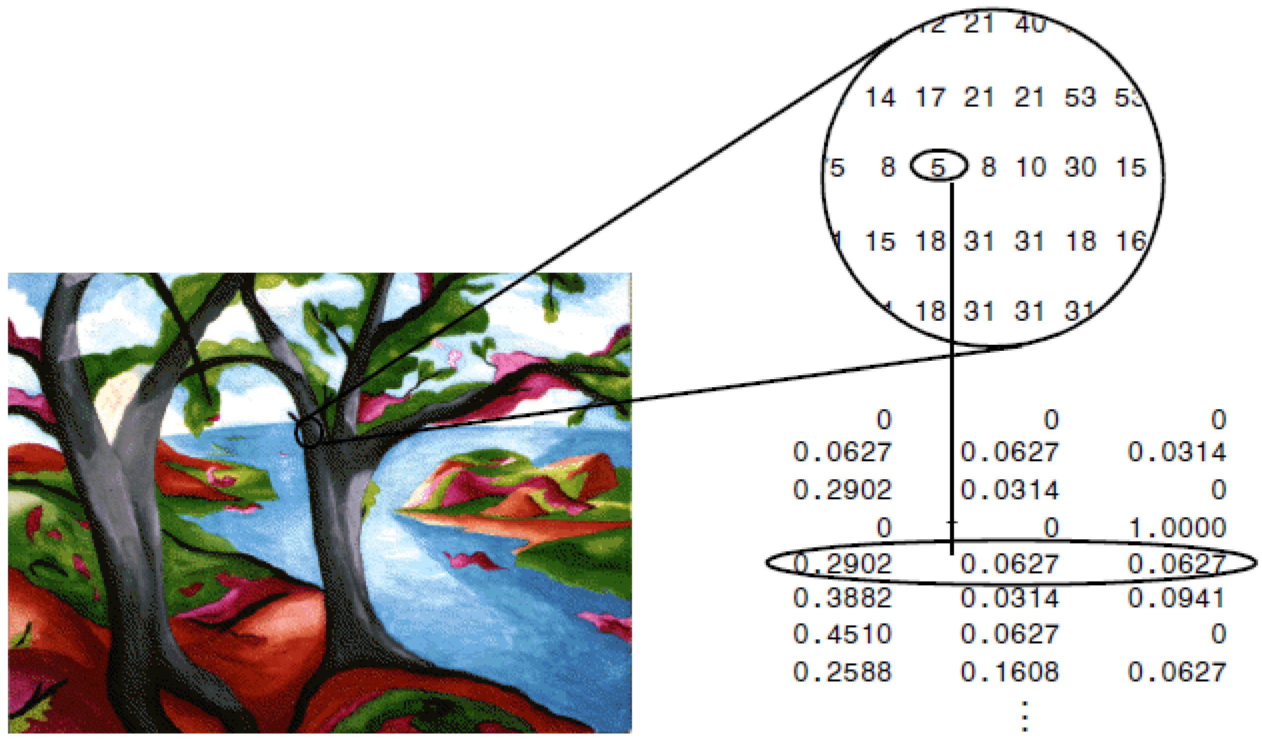


Fig: RGB image of class double

3. **Indexed.** Indexed (paletted) images are represented with an index matrix of size M×N and a colormap matrix of size K×3. The image has in total K different colors. The colormap holds all colors used in the image and the index matrix represents the pixels by referring to colors in the colormap. For example, if the 22nd color is magenta MyColormap(22,:) = [1,0,1], then MyImage(m,n) = 22 is a magenta-colored pixel.



0	0	0
0.0627	0.0627	0.0314
0.2902	0.0314	0
0	0	1.0000
0.2902	0.0627	0.0627
0.3882	0.0314	0.0941
0.4510	0.0627	0
0.2588	0.1608	0.0627
	⋮	

Fig: Indexed Image

4. **Binary**. A binary image is represented by an M×N logical matrix where pixel values are 1 (true) or 0 (false).

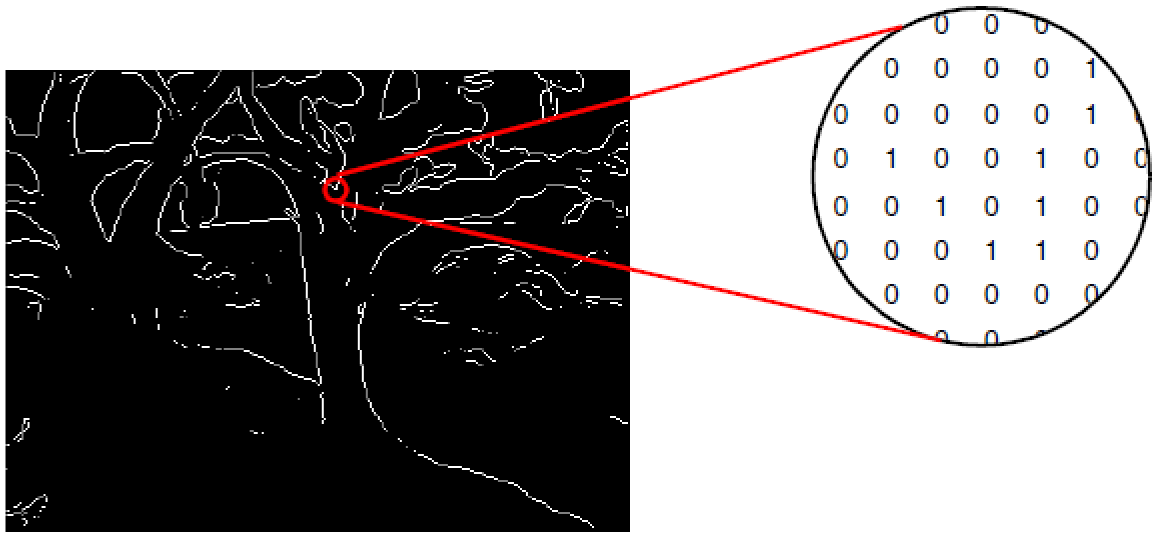


Fig: Binary Image

Grayscale is usually the preferred format for image processing. In cases requiring color, an RGB color image can be decomposed and handled as three separate grayscale images. Indexed images must be converted to grayscale or RGB for most operations.

General Commands

Operation	Matlab command
Read an image. (Within the parenthesis you type the name of the image file you wish to read. Put the file name within single quotes ' '.)	`imread();`
Write an image to a file. (As the first argument within the parenthesis you type the name of the image you have worked with. As a second argument within the parenthesis you type the name of the file and format that you want to write the image to. Put the file name within single quotes ' '.)	`imwrite();`
Creates a figure on the screen	`Figure;`
displays the matrix g as an image.	`imshow(g);`

turns on the pixel values in our figure.	`pixval on;`
the command returns the value of the pixel (i,j)	`impixel(i,j);`
Information about the image	`iminfo;`
Zoom in (using the left and right mouse button).	`zoom on;`
Turn off the zoom function.	`zoom off;`

Below are some common manipulations and conversions

Operation	**Matlab Command**

Display a grayscale or binary image	`image(MyGray*255);`
	`axis image`
	`colormap(gray(256));`

Display an RGB image (error if any element outside of [0,1])	`image(MyRGB);`
	`axis image`

Display an RGB image (clips elements to [0,1])	`image(min(max(MyRGB,0),1));`
	`axis image`

Display an indexed image	`image(MyIndexed);`
	`axis image`
	`colormap(MyColormap);`

Separate the channels of an RGB image	`MyRed = MyRGB(:,:,1);`
	`MyGreen = MyRGB(:,:,2);`
	`MyBlue = MyRGB(:,:,3);`

Put the channels back together	`MyRGB = cat(3,MyRed,MyGreen,MyBlue);`
Convert grayscale to RGB	`MyRGB = cat(3,MyGray,MyGray,MyGray);`
Convert RGB to grayscale using simple average	`MyGray = mean(MyRGB,3);`
Convert RGB to grayscale using NTSC weighting [Image Toolbox]	`MyGray = rgb2gray(MyRGB);`
Convert RGB to grayscale using NTSC weighting	`MyGray = 0.299*MyRGB(:,:,1) + 0.587*MyRGB(:,:,2) + 0.114*MyRGB(:,:,3);`
Convert indexed image to RGB	`MyRGB =`

[Image Toolbox]	`ind2rgb(MyIndexed,MyColormap);`
Convert indexed image to RGB	`MyRGB = reshape(cat(3,MyColormap(MyIndexed,1),MyColormap(MyIndexed,2),MyColormap(MyIndexed,3)),size(MyIndexed,1),size(MyIndexed,2),3);`
Convert an RGB image to indexed using K colors [Image Toolbox]	`[MyIndexed,MyColormap] = rgb2ind(MyRGB,K);`
Convert binary to grayscale	`MyGray = double(MyBinary);`
Convert grayscale to binary	`MyBinary = (MyGray > 0.5);`

Make sure to use semi-colon ; after these commands, to prevent t output being unnecessarily displayed in the command window

Loading and saving variables

Operation	**Matlab command**
Save the variable X .	`save X;`
Load the variable X .	`load X;`

How to display an image in Matlab

Here are a couple of basic Matlab commands (do not require any tool box) for displaying an image.

Displaying an image given on matrix form

Operation	**Matlab command**
Display an image represented as the matrix X.	`imagesc(X)`
Adjust the brightness. s is a parameter such that -1<s<0 gives a darker image, 0<s<1 gives a brighter image.	`brighten(s)`

Change the colors to gray.	`colormap(gray)`

If you are using Matlab with an Image processing tool box installed, it is recommended that you use the command imshow to display an image.

Displaying an image given on matrix form (image processing tool box)

Operation	**Matlab command**
Display an image represented as the matrix X.	`imshow(X)`
Zoom in (using the left and right mouse button).	`zoom on`
Turn off the zoom function.	`zoom off`

Spatial Resolution

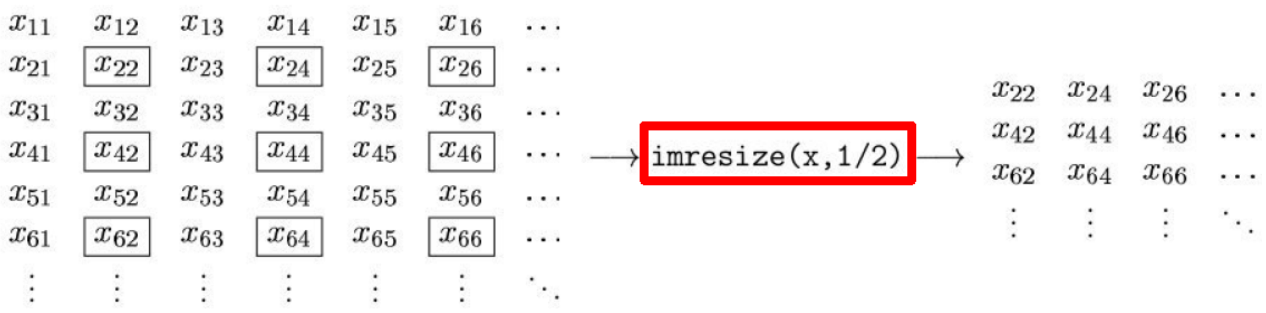
Spatial resolution is the density of pixels over the image: the greater the spatial resolution, the more pixels are used to display the image.

Halve the size of the image: It does this by taking out every other row and every other column, thus leaving only those matrix elements whose row and column indices are even.

Double the size of the image: all the pixels are repeated to produce an image with the same size as the original, but with half the resolution in each direction.

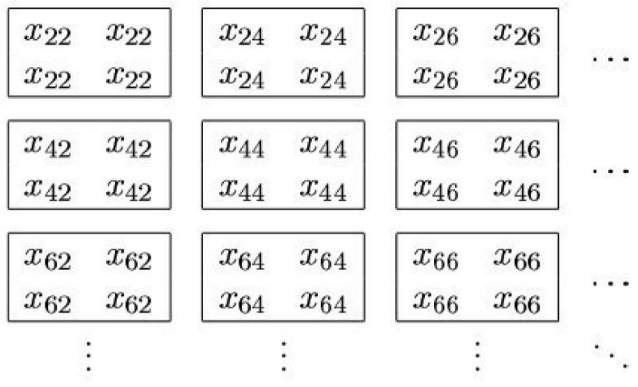
Interpolation

```
Imresize(x,1/2);
```



Extrapolation

Imresize(x,2);



Histograms

Given a grayscale image, its histogram consists of the histogram of its gray levels; that is, a graph indicating the number of times each gray level occurs in the image. We can infer a great deal about the appearance of an image from its histogram.

In a **dark** image, the gray levels would be clustered at the lower end

In a **uniformly bright** image, the gray levels would be clustered at the upper end.

In a **well contrasted** image, the gray levels would be well spread out over much of the range.

Problem: Given a poorly contrasted image, we would like to enhance its contrast, by spreading out its histogram. There are two ways of doing this. Histogram stretching and histogram equalization.

Histogram Stretching

Poorly contrasted image of range [a,b]. We can stretch the gray levels in the center of the range out by applying a piecewise linear function. This function has the effect of stretching the gray levels [a,b] to gray levels [c,d], where a<c and d>b according to the equation:

$$j = \frac{(c-d)}{(b-a)} \cdot (i-a) + c$$

imadjust(I,[a,b],[c,d]);

Pixel values less than c are all converted to c, and pixel values greater than d are all converted to d.

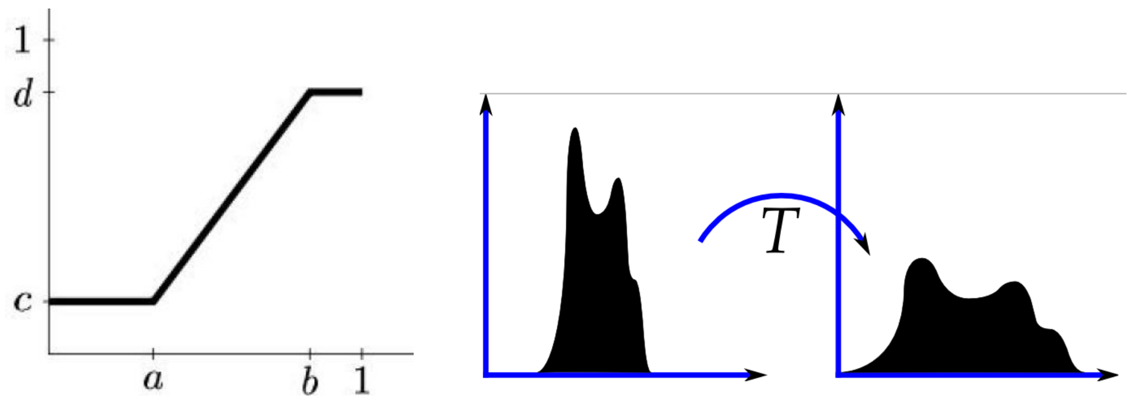


Fig: Example of Histogram Stretching

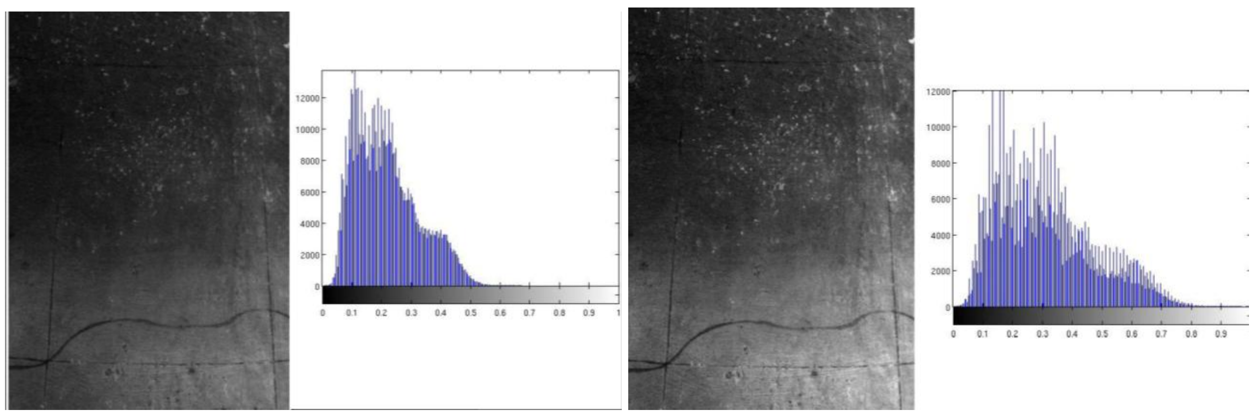


Fig: Another example of Histogram Stretching

Histogram Equalization

The trouble with the previous method of histogram stretching is that they require user input. Histogram equalization, is an entirely automatic procedure. Suppose an image has L different gray levels $0,1,2,\ldots,1\text{-}L$ and that gray level i occurs n_i times in the image. Suppose also that the total number of pixels in the image is n so that $n_0+n_1+n_2+\ldots n_L=n$. To transform the gray levels to obtain a better contrasted image, we multiply each gray level i by

$$\left(\frac{n_0+n_1+\cdots+n_i}{n}\right)(L-1)$$

rounded to the nearest integer. A roughly equal number of pixels is mapped to each of the L levels, so that the histogram of the output image is approximately flat.

Basic operations

Below are some basic operations on a grayscale image (u):

Operations	**Matlab Command**
Statistics	
Compute the maximum value	`uMax = max(u(:));`
Compute the minimum value	`uMin = min(u(:));`
Power	`uPower = sum(u(:).^2);`
Average	`uAvg = mean(u(:));`
Variance	`uVar = var(u(:));`
Median	`uMed = median(u(:));`
Plot histogram	`hist(u(:),linspace(0,1,256));`
`Basic Manipulations`	
Clip elements to [0,1]	`uClip = min(max(u,0),1);`
Pad image with one-pixel margin	`uPad = u([1,1:end,end],[1,1:end,end]);`
Pad image with k-pixel margin [Image Toolbox]	`uPad = padarray(u,[k,k],'replicate');`
Crop Image	`uCrop = u(RowStart:RowEnd,ColStart:ColEnd);`
Flip in the up/down direction	`uFlip = flipud(u);`
Flip left/right	`uFlip = fliplr(u)`
Interpolate image [Image Toolbox]	`uResize = imresize(u,ScaleFactor);`
Rotate by k*90 degrees with integer k	`uRot = rot90(u,k);`
Rotate by Angle degrees [Image Toolbox]	`uRot = imrotate(u,Angle);`
Stretch contrast to [0,1]	`uc = (u - min(u(:))/(max(u(:)) - min(u(:)));`

Quantize to K graylevels {0,1/K,2/K,...,1}	`uq = round(u*(K-1))/(K-1);`
`Simulating noise`	
Add white Gaussian noise of standard deviation sigma	`uNoisy = u + randn(size(u))*sigma;`
Salt and pepper noise	`uNoisy = u; uNoisy(rand(size(u)) < p) = round(rand(size(u)));`
`Debugging`	
Check if any elements are infinite or NaN	`any(~isfinite(u(:)))`
Count how many elements satisfy some condition	`nnz(u > 0.5)`

(Note: For any array, the syntax u(:) means "unroll u into a column vector." For example, if u = [1,5;0,2], then u(:) is [1;0;5;2].)

Converting The Image Type of Images

For certain operations, it is helpful to convert an image to a different image type. (Within the parenthesis you type the name of the image you wish to convert.)

Operation	**Matlab command**
Convert between intensity/indexed/RGB format to binary format.	**`dither()`**
Convert between intensity format to indexed format.	**`gray2ind()`**
Convert between indexed format to intensity format.	**`ind2gray()`**
Convert between indexed format to RGB format.	**`ind2rgb()`**
Convert a regular matrix to intensity format by scaling.	**`mat2gray()`**
Convert between RGB format to intensity format.	**`rgb2gray()`**
Convert between RGB format to indexed format.	**`rgb2ind()`**

Linear Filtering

Filtering is a technique for modifying or enhancing an image. For example, you can filter an image to emphasize certain features or remove other features. Filtering is a *neighborhood operation,* in which the value of any given pixel in the output image is determined by applying some algorithm to the values of the pixels in the neighborhood of the corresponding input pixel. A pixel's neighborhood is some set of pixels, defined by their locations relative to that pixel.

Convolution

In MATLAB, linear filtering of images is implemented through two-dimensional *convolution*. In convolution, the value of an output pixel is computed by multiplying elements of two matrices and summing the results.

One of these matrices represents the image itself, while the other matrix is the filter. For example, a filter might be:

k = [-1 1 -1; 1 4 1; -1 1 -1]

This filter representation is known as a *convolution kernel*. The MATLAB function conv2 implements image filtering by applying your convolution kernel to an image matrix. conv2 takes as arguments an input image and a filter, and returns an output image. For example, in this call, k is the convolution kernel, A is the input image, and B is the output image.

B = conv2(A,k);

Pixel Values and Statistics

The Image Processing Toolbox provides several functions that return information about the data values that make up an image. These functions return information about image data in various forms, including:

- The data values for selected pixels (pixval, impixel)
- The data values along a path in an image (improfile)
- A contour plot of the image data (imcontour)
- A histogram of the image data (imhist)
- Summary statistics for the image data (mean2, std2, corr2)
- Feature measurements for image regions (imfeature)
- This example illustrates the erosion operation described above.

Thresholding

Single thresholding:

A grayscale image is turned into a binary
image by first choosing a gray level T in the original image, and

then turning every pixel black or white according to whether its
gray value is greater than or less than T.
A pixel becomes white if its gray level is > T
A pixel becomes black if its gray level is <= T

Double thresholding:

Here we choose two values T1 and T2 and
apply a thresholding operation as:
A pixel becomes white if its gray level between T1 and T2
A pixel becomes black if its gray level is otherwise

Color Detection

Operation	**Command**
Read standard MATLAB demo image	`rgbImage = imread('onion.png');`

Display the original image	`subplot(3, 4, 1);`
	`imshow(rgbImage);`
	`title('Original RGB Image');`

Maximize figure	`set(gcf, 'Position', get(0, 'ScreenSize'));`

Split the original image into color bands	`redBand = rgbImage(:,:, 1);`
	`greenBand = rgbImage(:,:, 2);`
	`blueBand = rgbImage(:,:, 3);`

Display them	`subplot(3, 4, 2);`
	`imshow(redBand);`
	`title('Red band');`
	`subplot(3, 4, 3);`
	`imshow(greenBand);`
	`title('Green band');`
	`subplot(3, 4, 4);`
	`imshow(blueBand);`
	`title('Blue Band');`

Threshold each color band	`redthreshold = 68;`
	`greenThreshold = 70;`
	`blueThreshold = 72;`
	`redMask = (redBand > redthreshold);`
	`greenMask = (greenBand < greenThreshold);`
	`blueMask = (blueBand < blueThreshold);`

Display the masksa	`subplot(3, 4, 6);`
	`imshow(redMask, []);`
	`title('Red Mask');`
	`subplot(3, 4, 7);`
	`imshow(greenMask, []);`

	`title('Green Mask');`
	`subplot(3, 4, 8);`
	`imshow(blueMask, []);`
	`title('Blue Mask');`

Combine the masks to find where all 3 are "true."	`redObjectsMask = uint8(redMask & greenMask & blueMask);`
	`subplot(3, 4, 9);`
	`imshow(redObjectsMask, []);`
	`title('Red Objects Mask');`
	`maskedrgbImage = uint8(zeros(size(redObjectsMask)));`
	`maskedrgbImage(:,:,1) = rgbImage(:,:,1) .* redObjectsMask;`
	`maskedrgbImage(:,:,2) = rgbImage(:,:,2) .* redObjectsMask;`
	`maskedrgbImage(:,:,3) = rgbImage(:,:,3) .* redObjectsMask;`
	`subplot(3, 4, 10);`
	`imshow(maskedrgbImage);`
	`title('Masked Original Image');`

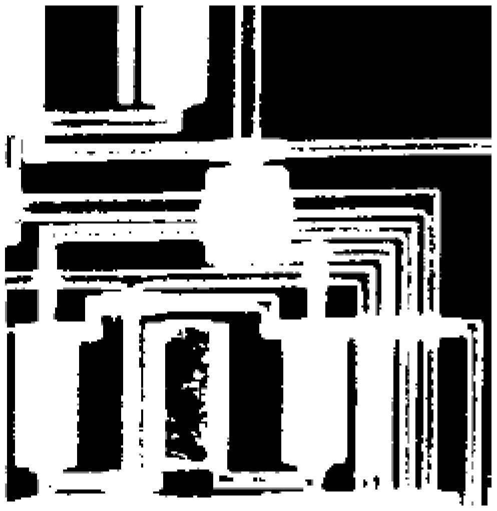
Noise Reduction

The main morphological operations are *dilation* and *erosion.* Dilation and erosion are related operations, although they produce very different results. Dilation adds pixels to the boundaries of objects (i.e., changes them from off to on), while erosion removes pixels on object boundaries (changes them from on to off).

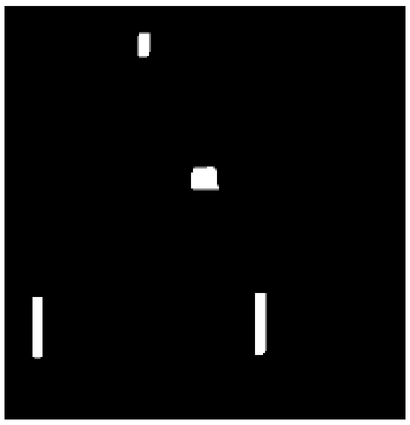
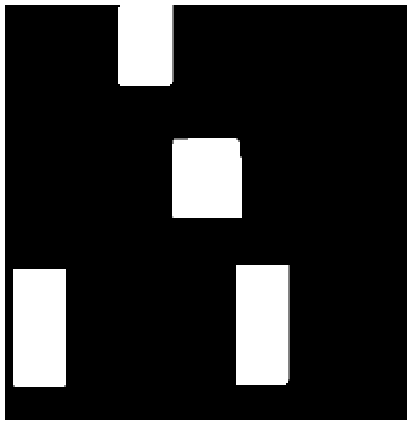
The neighborhood for a dilation or erosion operation can be of arbitrary shape and size. The neighborhood is represented by a structuring element, which is a matrix consisting of only 0's and 1's. The center pixel in the structuring element represents the pixel of interest, while the elements in the matrix that are on (i.e., = 1) define the neighborhood.

```
>> BW1 = imread('circbw.tif');
>> SE = eye(5);
>> BW3 = erode(BW1,SE);
>> BW2 = dilate(BW1,SE);
>> imshow(BW1)
>> figure, imshow(BW2)
>> figure, imshow(BW3)
```

Erode:



Dilate:

Webcam Capture [Image Acquisition Toolbox required.]

Firstly, Matlab needs to find out what are the webcam devices that are connected to your computer.

Firstly, a `imaqhwinfo` gives information about the existing adaptors for your webcam device. You can get more information on each adapter, by using `imaqhwinfo('winvideo')` where *winvideo* is one of the adaptors. In this, (if you have a device connected) you shall get a Device IDs attached to your webcam device. Further information pertaining to the device can be obtained by `imaqhwinfo('winvideo',1)` where 1 is the Device ID you saw earlier.

This gives you much needed information regarding the capture device. The resolution (800×600, 1024×768, 1600×1200, etc.), format (RGB, YUV, etc.) which needs to be selected when creating a video object.

Armed with all this imaqhwinfo (**im**age **acq**uisition **h**ard**w**are **info**rmation) you are ready to create your own video object.

```
vidobj = videoinput('winvideo',1,'RGB_1024x768');
```

The most important command now would be to start your video object `start(vidobj)`. It is at this point, or during the creation of video object, that the light (if any) on your webcam would start glowing indicating capture.

You can obtain snapshots of capture by using the `frame = getsnapshot(vidobj);` or view the continuous stream of frames by saying `preview(vidobj);`.

A safe closure (unlocking of the video handles) of the video object is extremely important so that it can be started again easily. A `stop(vidobj)` followed by `delete(vidobj)` is the best way to follow.

All the options can be seen by `imaqhelp(videoinput)`.

Get Image from camera
get snapshot can be put inside a while loop to process frames of the video

```
>> vid = videoinput('winvideo', 1);
>> set(vid, 'ReturnedColorSpace', 'RGB');
>> % Begin loop
>> img = getsnapshot(vid);
>> imshow(img)
>> % Process the image
>> % End loop
```

Serial Communication

We need to create a serial port object. Serial port object is just a name given to that serial port so that we can use it in later commands.

```
>> s = serial ('COM1');
     Serial Port Object : Serial-COM1
          Communication Settings
               Port: COM1
               BaudRate: 9600
               Terminator: 'LF'
          Communication State
               Status: closed
               RecordStatus: o_
          Read/Write State
               TransferStatus: idle
               BytesAvailable: 0
               ValuesReceived: 0
               ValuesSent: 0
```

Setting up serial port object

This line of command only constructs the object. It does not check/setup/initialize the communication. This command will still work even if the serial port is not connected to any device. Many objects can be created for a serial port but only one can be connected at a time. This shows all the property of the constructed serial portobject.

In MATLAB, s is a structure which has all the above properties. We can access/modify them using dot(.) operator. Note that the Status is closed. It implies that the serial port is not connected.

Baud Rate is the rate of transfer of the data in bits/sec.

We can change the BaudRate using the set method as follows-

```
>> set(s, 'BaudRate', 9600);
>> s.BaudRate = 9600;
```

You can also setup different BaudRate while making the serial port object as follows-

```
>> s = serial('COM1','BaudRate', 9600);
```

You can verify the change using get method-

```
>> get(s, 'BaudRate')
ans = 9600
```

The following will also show the similar result-

```
>> s.BaudRate
ans = 9600
```

Setup the connection

Before you actually write the data to the serial port, you must connect to device. This is like a JAVA lock. Only one entity can acquire the lock at a time. Use fopen to acquire the lock and setup the connection.

```
>> fopen(s)
```

Notice the status property of the serial port object-

```
>> s.Status
ans = open
```

If the lock is already acquired, fopen will give an error. To avoid error, first check the Status property of the serial port object. If it is closed then try to setup the connection.

Writing to Serial Port in MATLAB

MATLAB can write any kind of data to the serial port binary, string, int, oat etc. with speci_ed precision. We use *fwrite* or *fprintf* to write data.
Transfer an int/float array-

```
>> fwrite(s, vector array, 'precision');
```

The precision speci_es the datatype of the vector array. It can be 'int8', 'int16', 'oat32', 'oat64', 'uint8', 'char' etc.

String-

```
>> fwrite(s, 'string');
```

You can use *fprintf* for strings as well-

```
>> fprintf(s, 'string');
```

You can specify the format in *fprintf* for a string.

Reading from Serial Ports in Arduino

You can follow these steps to read data in Arduino-

```
     void setup(){
          Serial.begin(9600);
     }
     Void loop(){
          if(Serial.available()>0){
               byte b = Serial.read();
               //Process the Data
          }
     }
```

You can choose the kind of data you are expecting, otherwise byte dataype can be used.

Mouse Operations

To extract the coordinates of a point in a figure, you may select the figure and use ginput.

```
>> figure(10);
```

% 10 can be replaced with any index of the figure you want to activate.

```
[x,y] = ginput(4);
```

% This will return 4 point coordinates of mouse clicks in the last figure

Alternate method:

Define the `WindowButtonDownFcn` of your figure callback using the set command and a `@callbackfunction` tag.
Like so:

```
>>
function mytestfunction()
f=figure;
set(f,'WindowButtonDownFcn',@mytestcallback)

function mytestcallback(hObject,~)
pos=get(hObject,'CurrentPoint');
disp(['You clicked X:',num2str(pos(1)),', Y:',num2str(pos(2))]);
```

You can also pass extra variables to callback functions using cell notation:

```
set(f,'WindowsButtonDownFcn',{@mytestcallback,mydata})
```

If you're working with uicontrol objects, then it's:

```
set(myuicontrolhandle,'Callback',@mytestcallback)
```

References

1. **Tutorials, review materials**
    a. **MATLAB tutorial** (via David Kriegman and Serge Belongie)
    b. **More MATLAB tutorials:** basic operations, programming, working with images (via Martial Hebert at CMU)
    c. **Linear algebra review** (via David Kriegman)
    d. **Random variables review** (via David Kriegman)
2. MATLAB Reference
    a. **MATLAB guide from Mathworks**
    b. **MATLAB image processing toolbox**

3. Introduction with Image Processing on Matlab
http://visl.technion.ac.il/labs/anat/An%20Introduction%20To%20Digital%20Image%20Processing%20With%20Matlab.pdf
4. MATLAB Image Processing - ETHZ
http://www.igp.ethz.ch/photogrammetry/education/lehrveranstaltungen/photogrammetry/matlab_imageprocessing.pdf
5. Tutorials for Image Processing – imageprocessingplace.com
Tutorials for IP
http://www.imageprocessingplace.com/root_files_V3/tutorials.htm
6. PPTs for revision
http://www.imageprocessingplace.com/DIP-3E/dip3e_classroom_presentations_downloads.htm
7. Video Capture in MATLAB
http://makarandtapaswi.wordpress.com/2009/07/09/webcam-capture-in-matlab/
8. Serial Communication in MATLAB
http://home.iitb.ac.in/~rahul./ITSP/serial_comm_matlab.pdf
http://www.mathworks.in/products/instrument/supported/serial.html
9. While matlab is easy to learn and use, mex and opencv support highly optimized functions. Here is something which might help in the future to use them together.
http://www.cs.stonybrook.edu/~kyamagu/mexopencv/
http://www.cs.stonybrook.edu/~kyamagu/mexopencv/matlab/
10. OpenCV Documentation
http://opencv.itseez.com/index.html