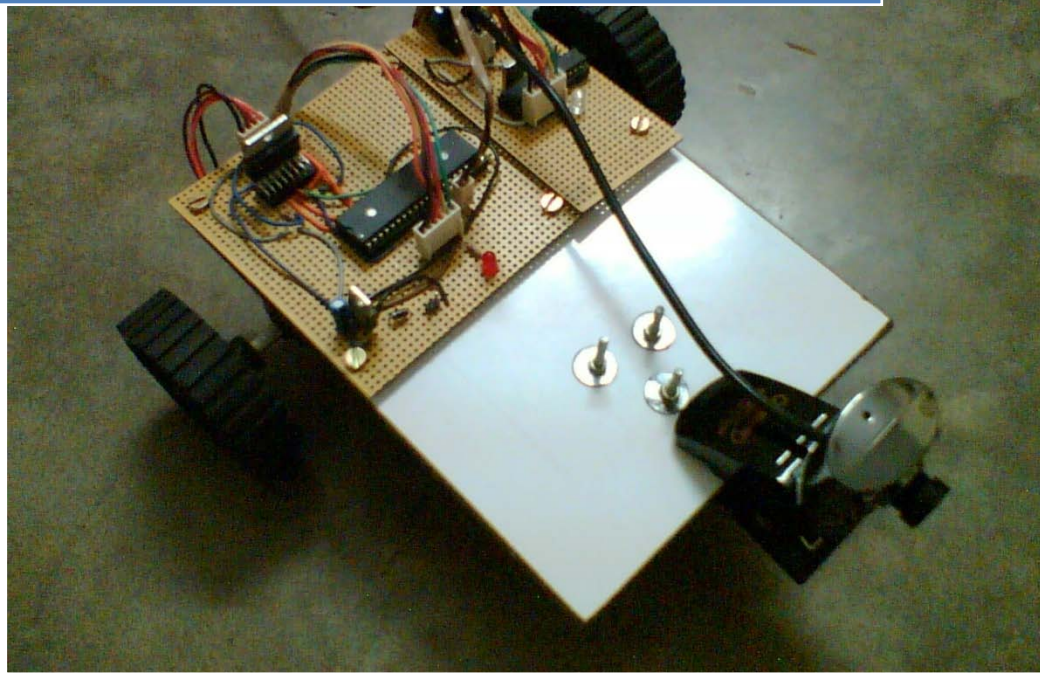


2009

Elementary Introduction to Image Processing Based Robots

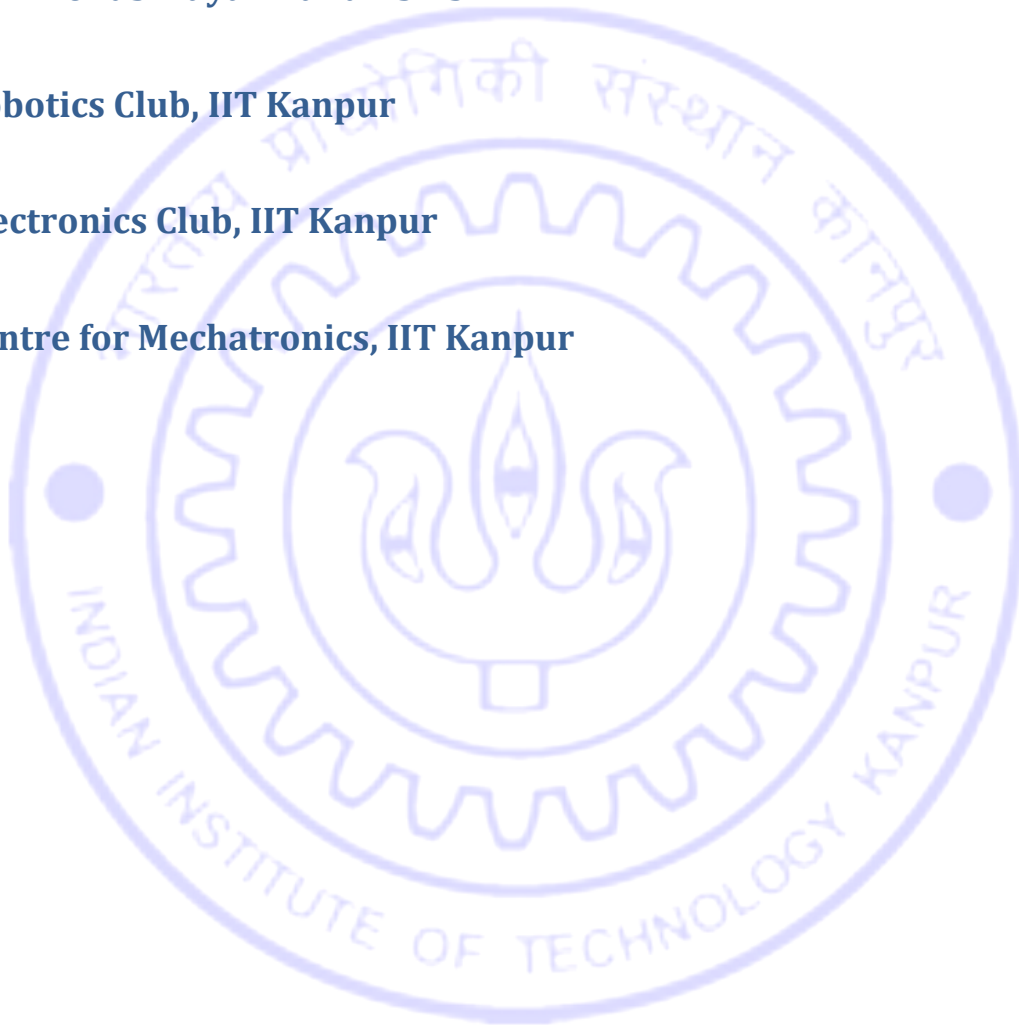


Ankur Agrawal

IIT Kanpur

Acknowledgement

- **My Senior Sourabh Sankule**
- **My Friends Mayank and Ashish**
- **Robotics Club, IIT Kanpur**
- **Electronics Club, IIT Kanpur**
- **Centre for Mechatronics, IIT Kanpur**



Contents

Introduction	4
MATLAB	4
What does MATLAB stand for?	4
Getting acquainted with MATLAB environment	4
General functions/commands	7
Trigonometric functions	8
The colon operator (:)	8
Relational operators	9
Frequently used functions and commands	9
Taking up Images	11
Important terms and types of Images	11
Representation of an Image in MATLAB	13
Reading and displaying Images	13
Making M-files and functions	14
M-file	14
Functions	15
Removing Noise	18
Getting the properties of different regions	21
bwboundaries	21
Label Matrix (L)	23
regionprops	24
Working in Real Time	24
Getting Hardware information	24
Previewing video	26
Capturing Images	27
Interfacing via PC Ports	28
Parallel Port	29
Serial Port	31
Some suggestions	33

Introduction

Here is a small tutorial that suffices you with the basic concepts required to put up eyes on your robot. Trust me, if you are into robotics, you are going to enjoy the next few pages. After all, making “a robot that can see” would really be cool! So let’s get started!

A vision based robot has an image acquisition device like a webcam as its eyes. Then we need a processor that can make sense out of those captured images and actuators like dc motors for navigation. One key point to note is that Image Processing has huge computational requirements, and it is not possible to run an image processing code directly on a small microcontroller. Hence, for our purpose, the simplest approach would be to run the code on a computer, which has a webcam connected to it to take the images, and the robot is controlled by the computer via serial or parallel port. The code is written in a software that provides the tools for acquiring images, analyzing the content in the images and deriving conclusions. MATLAB is one of the many such software available which provide the platform for performing these tasks.

MATLAB

What does MATLAB stand for?

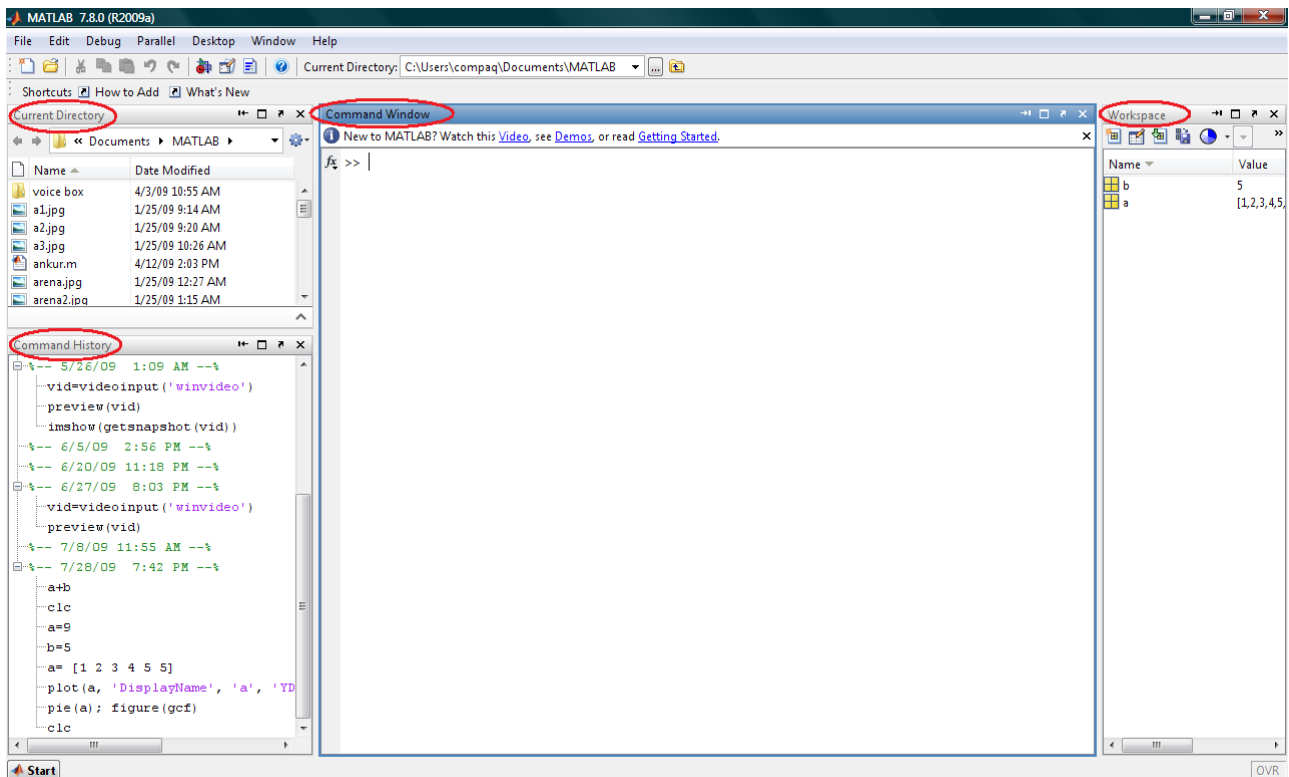
MATLAB stands for MATrix LABoratory. Hence, as the name suggests, here you play around with matrices. Hence, an image (or any other data like sound, etc.) can be converted to a matrix and then various operations can be performed on it to get the desired results and values.

Image processing is quite a vast field to deal with. We can identify colors, intensity, edges, texture or pattern in an image. In this tutorial, we would be restricting ourselves to detecting colours (using RGB values) only.

Getting acquainted with MATLAB environment

For those who have just finished installing MATLAB on their system and can’t figure out from where to start, no need to worry! This tutorial will first make you well acquainted with its very basics and then move further.

So, a typical MATLAB 2009 window looks like:

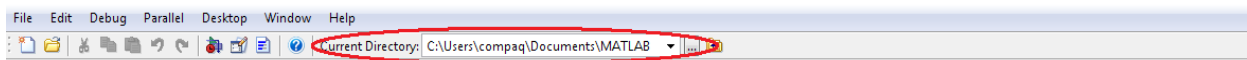


There are 4 main windows:

Command window: This is the main window where you write the commands, as well as see the outputs. In other words, here is your interaction with the software.

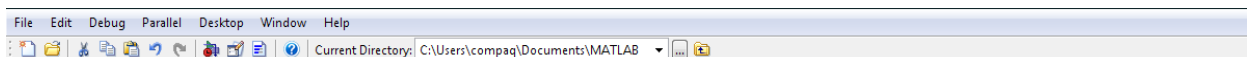
Command History: As the name suggests, it shows the list of the commands recently used in chronological order. Hence, you can double click on a command to execute it again.


Current directory: It is the default directory (folder) for saving your files. All the files which you make (like m-files, as discussed later) are saved here and can be accessed from here directly. The location of the current directory is shown in the toolbar at the top. You can change it by changing the address here.

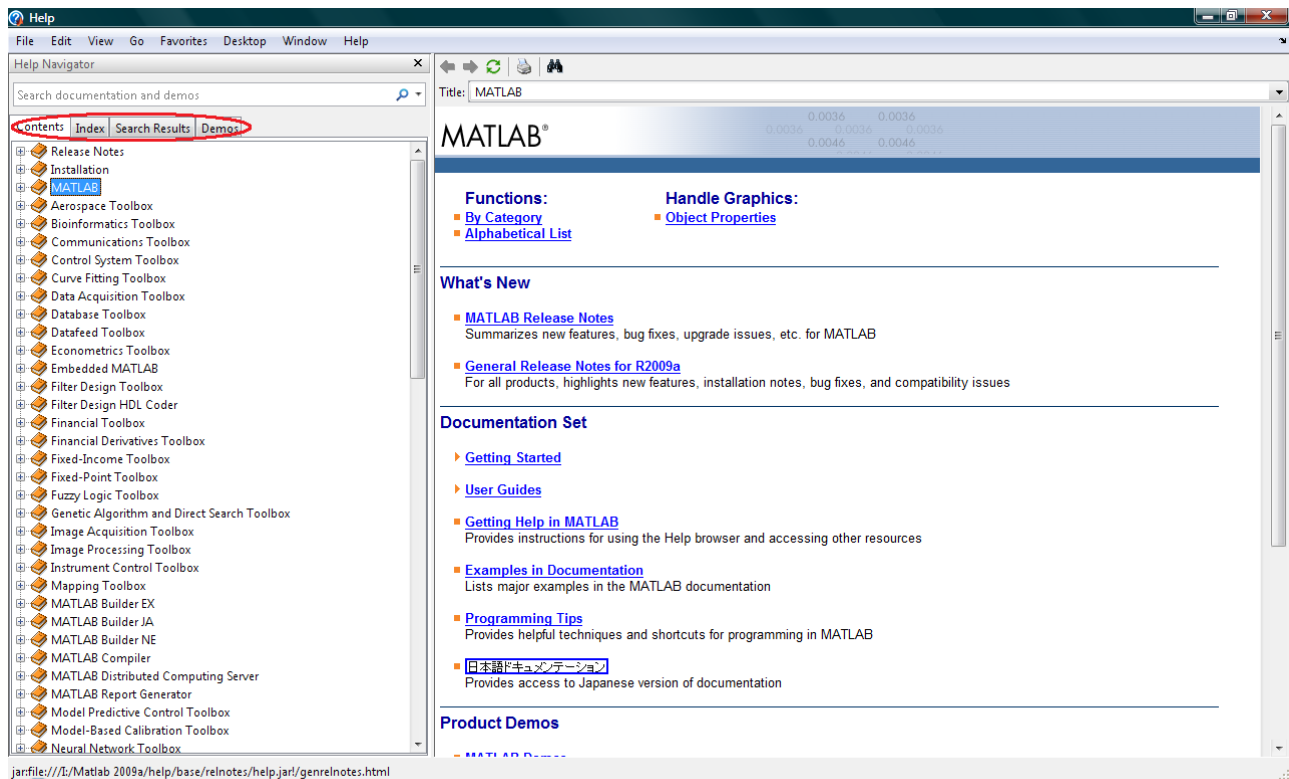


Workspace: It displays the list of the variables defined by you in the current session of MATLAB.

The Menu bar and Toolbar:

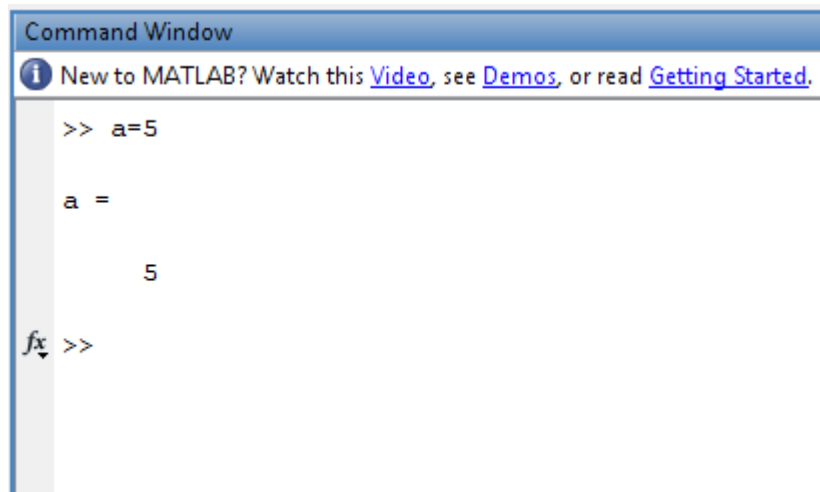


The toolbar has buttons for common operations like cut, copy, paste, undo, redo. The most important button here is the HELP button . It opens the MATLAB help window which has looks somewhat like this:



You can get the details of any MATLAB command/function here and many demos of some commonly used applications. Locate the four tabs: Contents, Index, Search Results and Demos on the left. One of the best ways to learn by yourself is to look for demos of your interest and type the command/ function which you encounter there as the search term. You will then get the complete details of the function like its use, syntax, as well as few examples on how to use it. You can also have a look at some of the related functions at the end of page under the heading “See Also”. The demos related to Image Processing can be found under Image Processing Toolbox and Image Acquisition Toolbox.

Now once we are done with knowing the essential features of MATLAB, let’s start typing something in the command window, say: `a=5` and press enter.



Yes... as you can see that MATLAB creates a variable with name 'a', stores value 5 in it and displays it in the command window itself. Hence you can see how user-friendly MATLAB is.

Variables are stored in the form of Matrices in MATLAB. Hence, 'a' is a 1X1 matrix in the above example. Similarly, you can make one dimensional, two dimensional, etc. matrices as follows:

```
>> a=[1 3 5 7 9]
a =
     1     3     5     7     9

>> b=[1 2 3;4 5 6;7 8 9]
b =
     1     2     3
     4     5     6
     7     8     9
```

To avoid the display of the variable, we use semi-colon (;) at the end of instruction.

```
>> b=[1 2 3;4 5 6;7 8 9];
```

The indices of matrices in MATLAB start from 1 (unlike C/C++ and Java where they start from 0).

We refer to a particular element of the matrix by giving its indices in parenthesis ().

```
>> b(2,1)
ans =
     4
```

Now with the variables in hand, you can perform various mathematical operations on them directly.

```
>> a=[1 2 3];
>> b=[6 7 8];
>> a+b
ans =
     7     9    11
```

ans is the default variable of MATLAB. You can also store the result in another variable as

```
>> c=a+b
c =
     7     9    11
```

General functions/commands

clc: To clear the command window, giving you a 'clear screen'.

clear: To remove all variables from the workspace. This frees up system memory.

Trigonometric functions

For angle in radians-

```
>> sin(1)
ans =
    0.8415
```

For angle in degrees-

```
>> sind(30)
ans =
    0.5000
```

Inverse trigonometric-

```
>> asin(1)
ans =
    1.5708
```

```
>> asind(.5)
ans =
    30.0000
```

Similarly we have cos(), cosd(), acos(), tan(),...etc.

The colon operator (:)

The colon is one of the most useful operators in MATLAB. It can create vectors, subscript arrays, and specify for iterations. In a very crude language, we can say that the colon (:) means “throughout the range”.

$j:k$ is the same as $[j, j+1, \dots, k]$

$j:i:k$ is the same as $[j, j+i, j+2i, \dots, k]$

$A(:,j)$ is the j th column of A

$A(:,j:k)$ is $A(:,j), A(:,j+1), \dots, A(:,k)$

$A(:)$ is all the elements of A , regarded as a single column.

Example,

```
>> a= [1 2 3; 4 5 6; 7 8 9]
a =
     1     2     3
     4     5     6
     7     8     9
```



```
>> a(:,2:3)
ans =

     2     3
     5     6
     8     9
```

Relational operators

Operator	Description
==	Equal to
~=	Not equal to
<	Less than
<=	Less than or equal to
>	More than
>=	More than or equal to

Frequently used functions and commands

if, else: Execute statements if condition is true, false respectively.

Syntax:

```
if condition1
statement
elseif condition 2
statement
else
statement
end
```

Example,

```
a=10;
>> if a<10
b=a/2;
else
b=a*2;
end
>> b
b =
    20
```

***Note:** As a block is contained in braces {} in C/C++/Java, a block is terminated by the 'end' statement in MATLAB.

for: To create a loop, i.e., execute a block of code specified number of times.

Syntax:

```
for variable = initval:endval
    statement
    ...
    statement
end
```

Example,

```
>>c=[1 2 3 4 5]; b=0;
>>for i=1:5
b=b+c(i);
end
>> b
b =
    15
```

while: Again to create loop, that executes till a specified condition is true.

Syntax:

```
while condition
    statements
end
```

Example,

```
>> c=2009; i=1;
while c>1
b(i)=mod(c,10);
c=c/10; i=i+1;
end
>> b
b =
    9.0000    0.9000    0.0900    2.0090
```

zeros(): Create array/matrix of all zeros.

B = zeros(n) returns an n-by-n matrix of zeros.

B = zeros(m,n) returns an m-by-n matrix of zeros.

Example,

```
>> z=zeros(2,4)
z =
     0     0     0     0
     0     0     0     0
```

Similarly we have ones() function for all values 1.

size(): Returns matrix dimensions.

Example, for the above matrix z,

```
>> size(z)
ans =
      2      4
```

length(): Returns the length of a vector. For an array, it returns the size of the longest dimension.

Example,

```
>>x = ones(1,8);
>>n = length(x)
n =
      8
```

dot(): Returns dot product of two vectors.

Example,

```
C = dot(A,B)
```

sqrt(): Returns square root of each element of an array

min(): Returns smallest elements in an array.

Syntax:

```
C = min(A)
```

If A is a matrix, min(A) treats the columns of A as vectors, returning a row vector containing the minimum element from each column.

Similarly we have **max()** function.

sort(): Sorts array elements in ascending or descending order

Syntax:

```
B=sort(A,mode)
```

where value of mode can be

```
'ascend' : Ascending order (default)
'descend' : Descending order
```

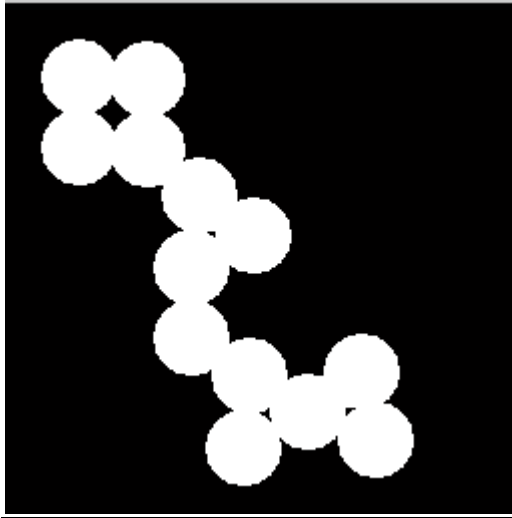
plot(): Creates a 2-D line plot.

Taking up Images

Important terms and types of Images

Pixel: Pixels are the building blocks of an image. In other words, a pixel is the smallest possible image that can be depicted on your screen.

Binary Image: An image that consists of only black and white pixels.



A Binary Image



A Grayscale Image

Grayscale Image: It contains intensity values ranging from a minimum (depicting absolute black) to a maximum (depicting absolute white) and in between varying shades of gray. Typically, this range is between 0 and 255.

***Note:** In daily language what we refer to as black-and-white (as in old photos) are actually grayscale. Hence avoid confusion here in technical terms.

Color Image: We all have seen this! Such an image is composed of the three primary colors, Red, Green and Blue, hence also called an RGB image.



A Color Image

RGB value: All colors which we see around us can be made by adding red, blue and green components in varying proportions. Hence, any color of the world can uniquely be described by its

RGB value, which stands for Red, Blue and Green values. This triplet has each value ranging from 0 to 255, with 0 obviously meaning no component of that particular color and 255 meaning full component. For example, pure red color has RGB value [255 0 0], pure white has [255 255 255],

pure black has [0 0 0] and  has RGB value [55 162 170].

Representation of an Image in MATLAB

An image in MATLAB is stored as a 2D matrix (of size $m \times n$) where each element of the matrix represents the intensity of light/color of that particular pixel. Hence, for a binary image, the value of each element of the matrix is either 0 or 1 and for a grayscale image each value lies between 0 and 255. A color image is stored as an $m \times n \times 3$ matrix where each element is the RGB value of that particular pixel (hence it's a 3D matrix). You can consider it as three 2D matrices for red, green and blue intensities.

Reading and displaying Images

imread(): To read an image and store in a matrix.

Syntax:

`IM=imread('filename')`

where IM is a matrix. If the file is in the current directory (as described above), then you only need to write the filename, else you need to write the complete path. Filename should be with extension (.jpg, .bmp,..). There are some default images of MATLAB like 'peppers.png', 'cameraman.tif', etc. You can try reading them as

```
>>im=imread('peppers.png');
```

It is always advised to use a semi-colon (;) at the end of the statement of reading an image, otherwise... you can try yourself what happens!

imshow(): Displays the image.

Syntax:

`imshow('filename')`

or

`imshow(im)`

Example,

```
>>imshow('cameraman.tif');
```

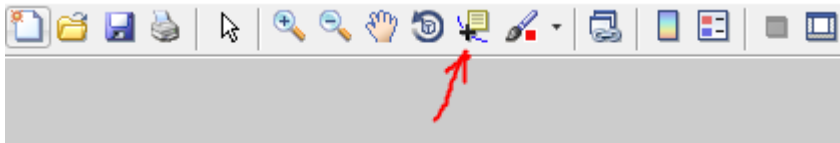
OK, now let's make our own image, try this:

```
>>a(1,1)=0;
>>for i=1:200;
    for j=1:200
        a(i+1,j+1)=1-a(i,j);
    end
end
```

```
>>imshow(a);
```

You try out making many different types of images like this just to make yourself comfortable with the commands learnt till now.

Data cursor: To see the values of the colors in the figure window, go to Tools>Data Cursor (or select from the toolbar), and click over any point in the image. You can see the RGB values of the pixel at location (X,Y).



A better option of data cursor is the function **imtool()**. Type the following

```
>>imtool('peppers.png');
```

and see the pixel info on lower left corner as you move mouse pointer over different pixels.

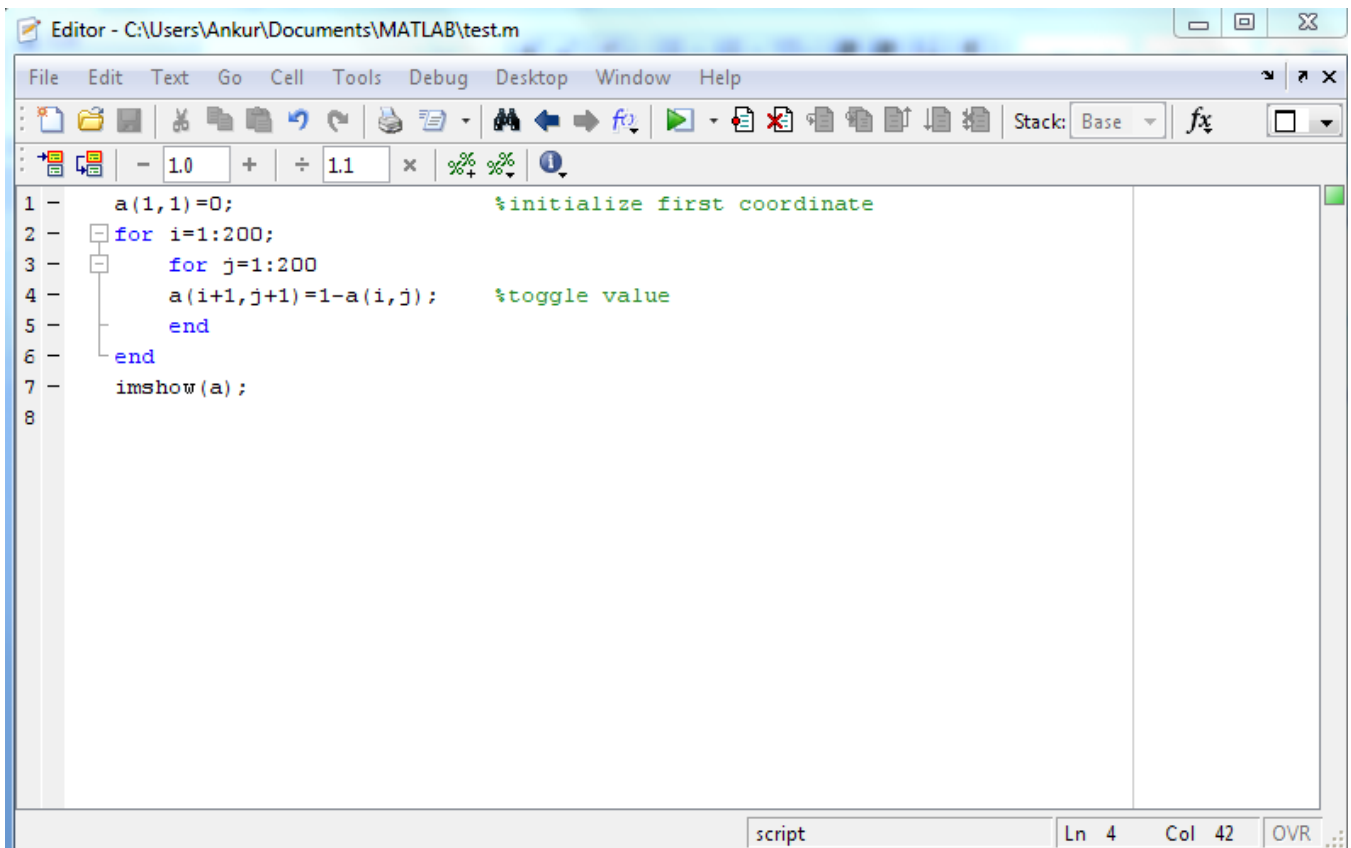
Making M-files and functions

M-file

It is a provision in MATLAB where you can execute multiple commands using a single statement. Here the group of commands is stored as a MATLAB file (extension .m).

Go to File->New->Blank M-file

MATLAB editor window opens where you can write the statements which you want to execute and save the file.



Here we have saved the m-file by the name “test.m”. Now as you type

```
>>test
```

in MATLAB command window, all the above commands will execute.

Comments: As we have comments in C/C++/ Java using double slash (//), in MATLAB we use symbol % to write comments, i.e., statements that are not considered for execution. You can see comments in green in the snapshot above.

Functions

Functions, as some of you might know, are written to organize the code efficiently and make debugging easier. The set of statements within a function can be executed as and when required by just calling it, thereby avoiding repetitions. The data which is needed within the function can be passed as arguments and then the required values can be returned. You can return any no. of values and they can be matrices also.

A function is saved as an m-file with the same name as the name of the function.

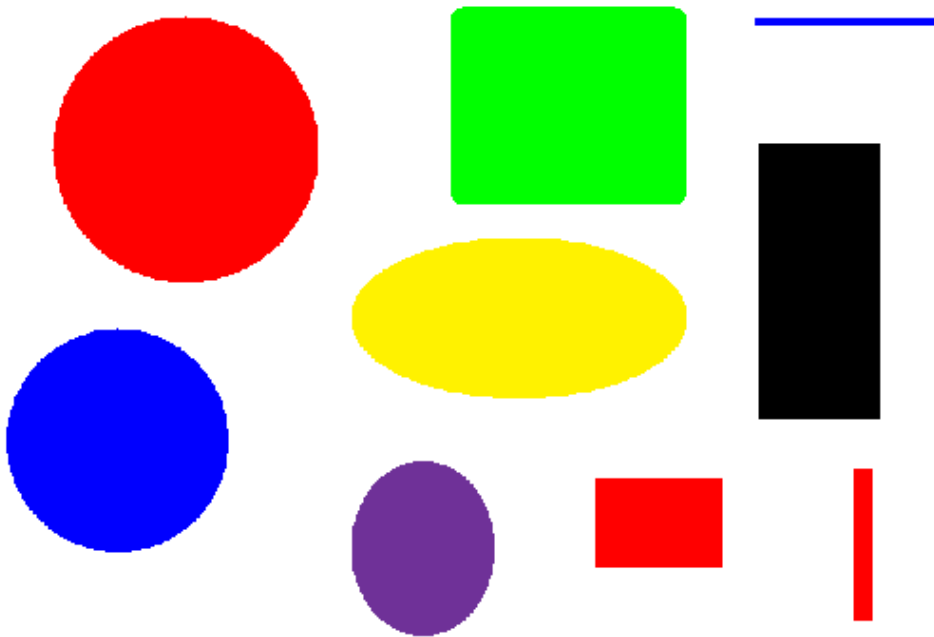
For Example, the following function takes the input as a colored image and returns a binary image where the green pixels have been replaced as white, rest are black and also returns the total no. of green pixels. (This task is required frequently whose use is described later)

```

function [bingreen, num]=green(im)
[m,n,t]=size(im);
bingreen=zeros(m,n); num=0;
for i=1:m
    for j=1:n
        if(im(i,j,1)==0 && im(i,j,2)==255 && im(i,j,3)==0)
%Red and Blue components must be zero and Green must be full
            bingreen(i,j)=1;
            num=num+1;
        end
    end
end
end

```

Now suppose the input image is 'shapes.bmp' (not a default image of MATLAB, hence you can save it in the working directory to perform the following operations or make yourself in paint)

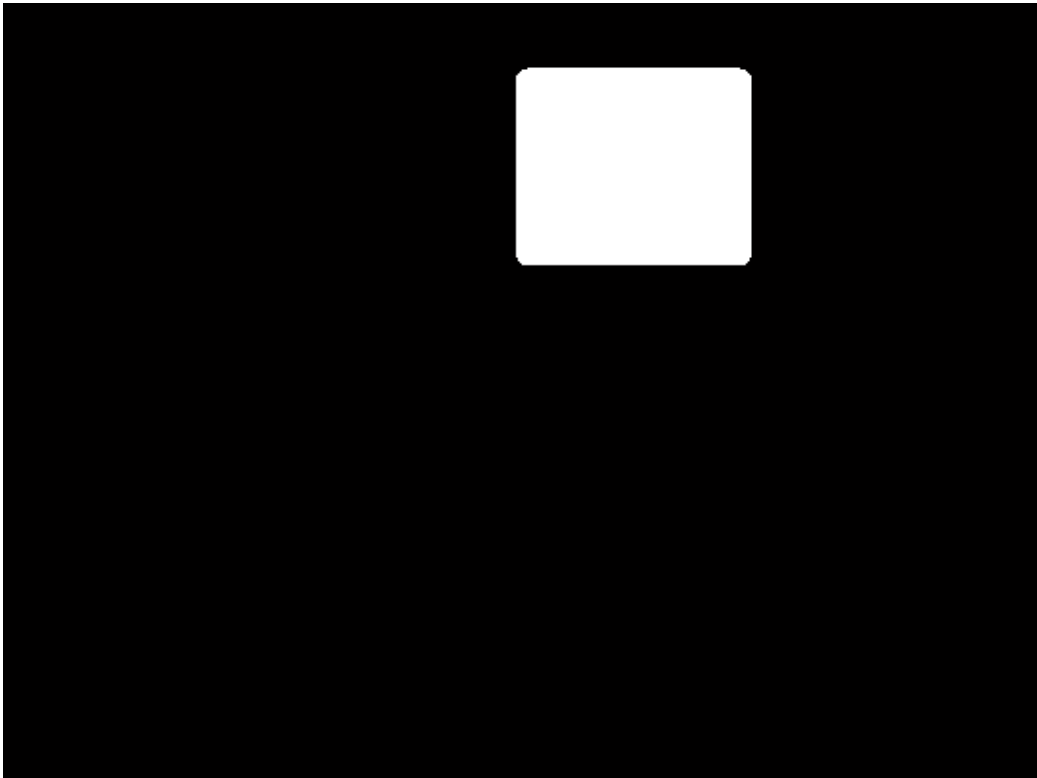


So, you first read the image and then call the function by typing in the command window

```

>> I=imread('shapes.bmp');
>> [img, n]=green(I);
>> n
n =
    28753
>> imshow(img);

```

As you can see, this is a binary image with white pixels at those coordinates which were green in input image.

Now this was an ideal image where each color was perfect. But in practice, you don't have images with perfect colors. So, we give a range of RGB values to extract our required region of image.

Suppose the image at hand is the following (again it's not a default image) and we need to extract the red region.



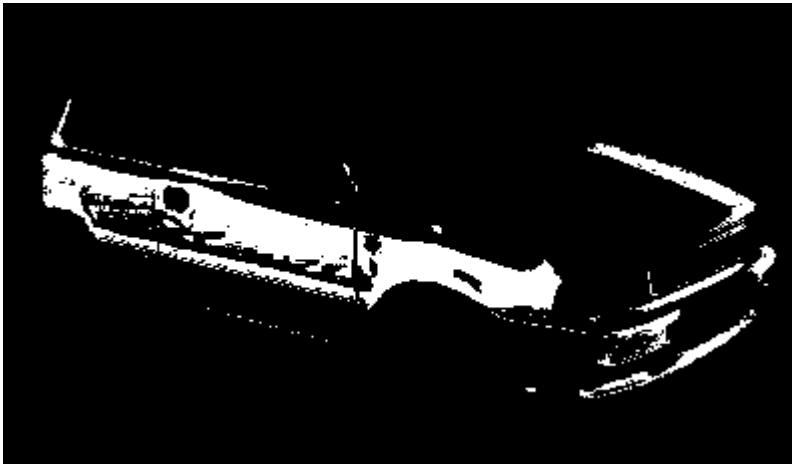
So, we use the same function with the main conditional statement changed as

```

function bw=red(im)
[m,n,t]=size(im);
bw=zeros(m,n);
for i=1:m
    for j=1:n
        if(im(i,j,1)>150 && im(i,j,2)<50 && im(i,j,3)<50)
            %Specifying range for red
            bw(i,j)=1;
        end
    end
end
end

```

You can define more specific range as per your requirement. Now when we see the output image, it looks like



*Tip: To get an idea of what range of RGB values you must define, use `imtool()` function (or data cursor) with the original file, move the cursor over the region which you need to extract and note down the RGB values. Expand/ Contract the range as per your requirement by trying several times. This is actually how the **calibration** is done.

Removing Noise

As you can see, the binary image obtained above has quite a lot of noise. You need to smoothen the edges, remove the tiny dots scattered here and there so that at last you have some countable number of objects to work upon. There are several functions available in MATLAB to remove noise in an Image. Some of them are ('bw' is the above binary image):

imclose(): Performs morphological closing operation. It fills in the gaps between two objects and smoothen the edges. The degree and type of smoothening and joining depend on the structuring element, i.e., the basic shape which is used to perform the operation. The structuring element can be a disk, a diamond, a line, etc. To create a structuring element, use **strel()** function.

For example,

```
>> se=strel('disk',5); % a disk of radius 5
```

```
>>bw=imclose(bw,se);           %Perform the closing operation on  
                                original image itself  
>>imshow(bw);
```



imfill(): Remove holes from the image
Holes are background pixels surrounded by foreground (image) pixels.

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



Now you need to remove the spatters. For that you can use the function `imopen()`.

imopen(): Morphologically open the image.

```
>>se=strel('disk',2);  
>>bw=imopen(bw,se);  
>>imshow(bw);
```

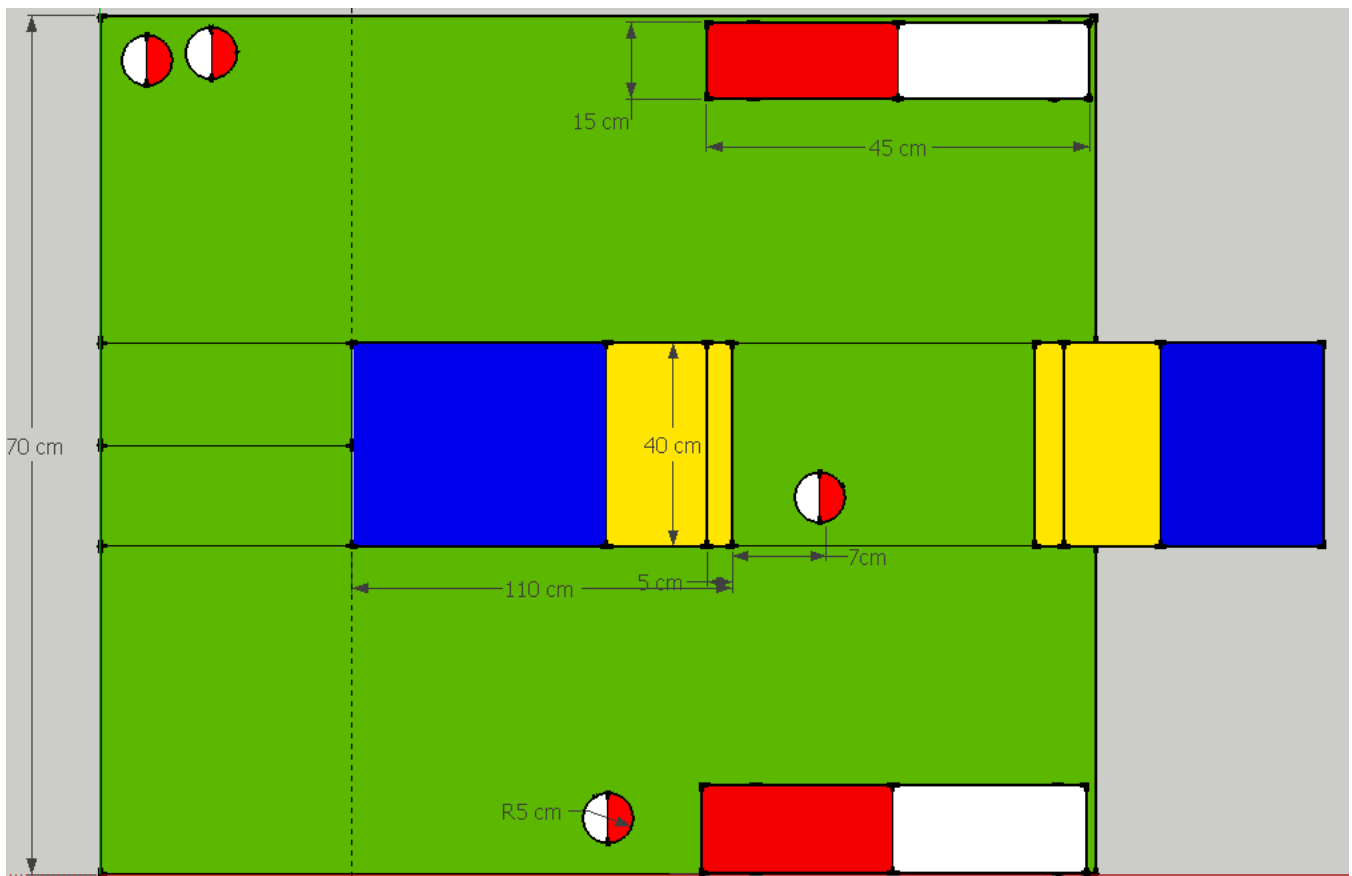


***Note:**

- The purpose of these functions might not be clear at first. Hence, it is strongly recommended that you explore these functions yourself by trying out on different images and have a look at these functions in MATLAB Help section also.
- The sequence of performing these operations is very significant, as the subsequent operation is performed on the converted image and not the original image. The sequence I have adopted is NOT necessary, and you must try yourself which sequence suits your requirement.
- The size and shape of structuring element is also to be determined experimentally, so that the number of objects you get in the binary image is same as what you require. This final image obtained in this example is still not workable, and you can try yourself to get a consolidated image.
- There are many other functions available for noise removal like `imerode()`, `imdilate()`, etc. Please refer to MATLAB Help section for details.
- An object is defined as a connected region in a binary image.

The use of making a binary file:

In most of problem statements of robotics based on image processing, we are required to find the centroid, area, and no. of objects of a particular color. MATLAB has in-built functions for these tasks which operate on binary images only. Hence we create binary images corresponding to different colors. For example, in the problem statement “Brooklyn Builder” of Techkriti’10, the image from the top (overhead camera) looks somewhat like (again it’s an ideal image; the real one will be far different and full of noise)



Now we first take a sample image of arena, note down the RGB values (using `imtool()`) and then extract the regions of different color by making binary images. Then we can find the centre of different objects using the function `bwboundaries()` and `regionprops()`. The piers and decks can be differentiated on the basis of their area.

Getting the properties of different regions

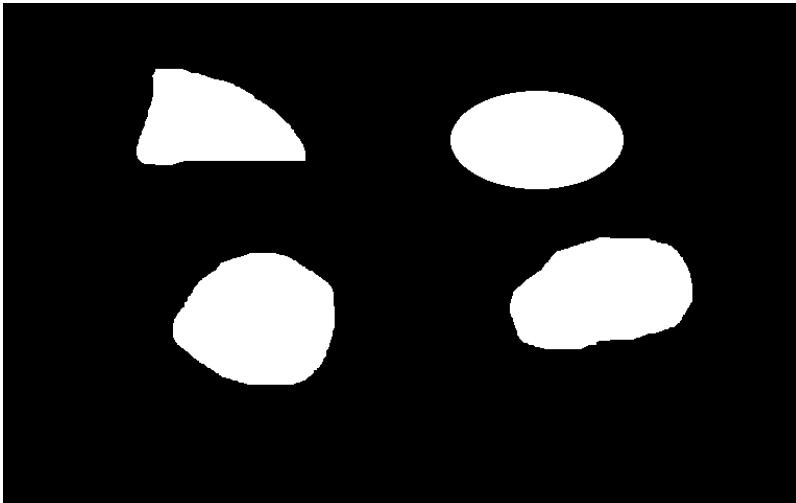
`bwboundaries`

It traces the exterior boundaries of objects in a binary image.

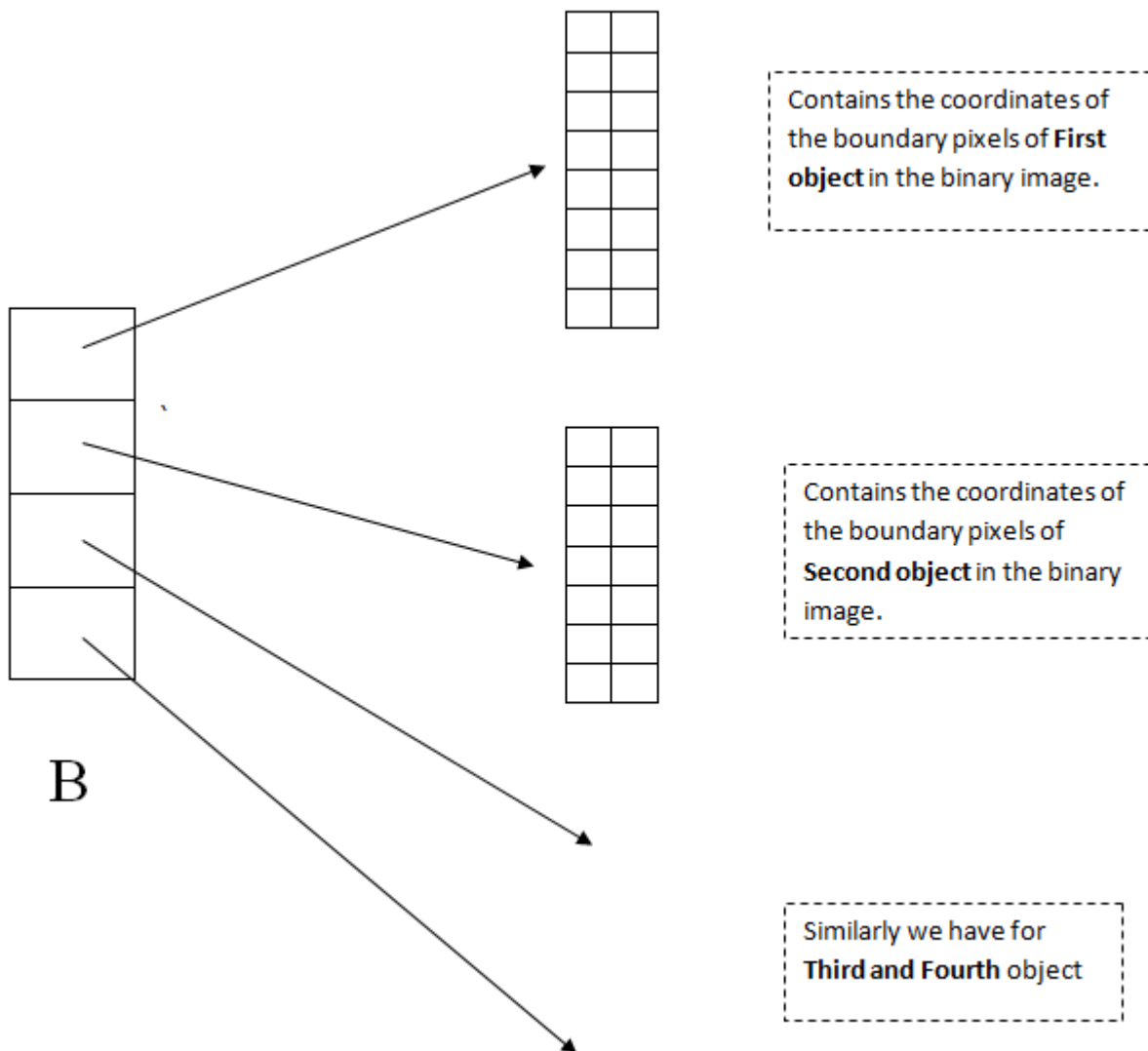
`B=bwboundaries(bw)`

`bwboundaries` returns `B`, a `P`-by-1 cell array, where `P` is the number of objects and holes. (A cell array is one where each element of the array is a matrix itself). Each cell in the array `B` contains a `Q`-by-2 matrix. Each row in the matrix contains the coordinates (row and column indices) of a boundary pixel. `Q` is the number of boundary pixels (perimeter) for the corresponding region. In other words, `B` contains the coordinates of pixels constituting the perimeter of each object and hole.

For example, suppose there are four objects in a binary image.



Then the matrix B can be understood as shown in the diagram below



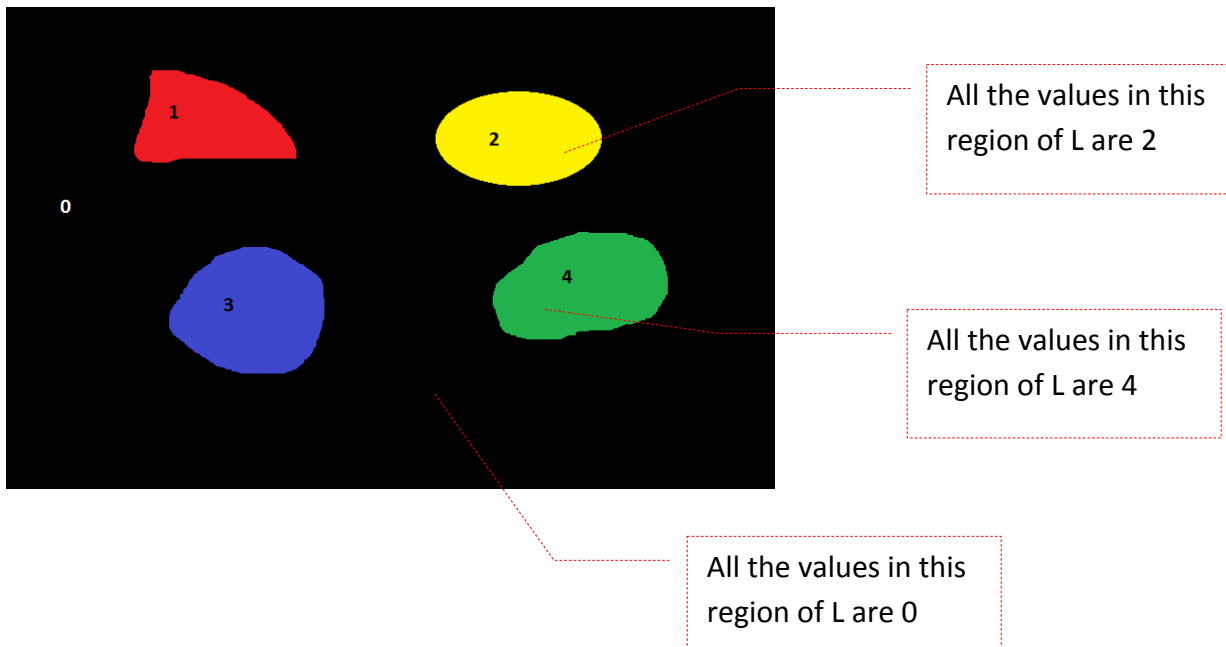
*Note:

- In the coordinate matrix, the **first coordinate is the row number (y - coordinate)** and **second coordinate is the column number (x-coordinate)** of the boundary pixel. Take this always into consideration as it can create a lot of confusion.
- `bwboundaries(bw)` will trace the hole boundaries too. So, a better option is to use `bwboundaries(bw,'noholes')`

Label Matrix (L)

A label matrix(L) corresponding to binary image is a 2D matrix of the same size as that of the image. Each object in the binary image is numbered 1,2,3,... and all the pixels of L corresponding to the objects in binary image have value respectively 1,2,3,... The background pixels are 0 by default. In other words, the region in L corresponding to first object in the image is marked 1, corresponding to second object is marked 2 and so on.

The label matrix corresponding to the above binary image would look somewhat like this (colors are just representative, the matrix will consist of values 0,1,2,3 and 4 only)



The label matrix(L) is can either be obtained from the function `bwboundaries()` as

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

or from function `bwlabel()` as

```
L=bwlabel(bw);
```

Now why is this label matrix needed? It is required to use the following function

regionprops

It is used to measure properties of image regions.

Once we have an image consisting of all the objects labelled, we must now know the centroid, area, etc. of each.

Syntax:

`STATS =regionprops(L,properties)`

where STATS is a structure array with length equal to the number of labelled objects in L. The fields of the structure array denote different properties for each region, as specified by *properties*.

properties are be a comma-separated list of strings. See the help section of MATLAB for all the properties available. We will use Centroid and Area.

Example,

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

The properties in the array stats are obtained by dot(.) operator.

```
>>a=stats(3).Area;           %to get area of 3rd object
>>c=stats(2).Centroid;       %to get centroid of 2nd object
```

*Note: `length(stats) = length(B) = max(L(:)) = no. of objects in bw`

Working in Real Time

Getting Hardware information

Till now we have been working on images already saved on our computer. But in actual practice, we need to work in real time, i.e., we need to take images continuously from the current environment using a webcam and then process them. Hence, the Image Acquisition toolbox of MATLAB provides support in this regard.

To start with working in real time, you must have a functional USB webcam connected to your PC and its driver installed. MATLAB has built-in **adaptors** for accessing these devices. An adaptor is a software that MATLAB uses to communicate with an image acquisition device. You can check if the support is available for your camera in MATLAB by typing the following:

```
>> imaqhwinfo           % stands for image acquisition hardware info
>> cam=imaqhwinfo;
>> cam.InstalledAdaptors
```

I tried this on my computer and I got the following information


```
>> imaqhwinfo

ans =

    InstalledAdaptors: {'coreco'  'winvideo'}
    MATLABVersion: '7.8 (R2009a)'
    ToolboxName: 'Image Acquisition Toolbox'
    ToolboxVersion: '3.3 (R2009a)'

>> cam=imaqhwinfo;
>> cam.InstalledAdaptors

ans =

    'coreco'    'winvideo'
```

To get more information about the device, type

```
>> dev_info = imaqhwinfo('winvideo',1)
```

```
>> dev_info = imaqhwinfo('winvideo',1)

dev_info =

    DefaultFormat: 'YUY2_160x120'
    DeviceFileSupported: 0
    DeviceName: 'Webcam-101'
    DeviceID: 1
    ObjectConstructor: 'videoinput('winvideo', 1)'
    SupportedFormats: {1x5 cell}
```

Most probably you would have 'winvideo' installed and use that. If it is not available, use whichever adapter is shown by 'imaqhwinfo'.

If you are using a laptop, you may also have a webcam in it. So note down the DeviceName shown as above. If it is not the USB webcam, then probably DeviceID = 2 should work. Hence, type

```
>> dev_info = imaqhwinfo('winvideo',2)
```

***Note:** From now onwards, I will refer Adapter by 'winvideo' and DeviceID by '1'. You must check yourself what is available on your system and change the commands described further accordingly.

Note down the supported formats by your camera as

```
>> dev_info = imaqhwinfo('winvideo',1);
```

```
>>dev_info.SupportedFormats
```

```
>> dev_info.SupportedFormats

ans =

    Columns 1 through 4

    'YUY2_160x120'    'YUY2_176x144'    'YUY2_320x240'    'YUY2_352x288'

    Column 5

    'YUY2_640x480'
```

As you can see, there are 5 supported formats in my camera. The numbers (160x120, 176x144....) denote the size of the image to be captured by the camera.

Previewing video

You can preview the video captured by the camera by defining an object (say by the name 'vid') and associate it with the device.

```
>>vid=videoinput('winvideo',1, 'YUY2_160x120')
or
>>vid=videoinput('winvideo',1, 'RGB24_160x120')
                                % depends on availability
```

It should give information somewhat like this

```
>> vid=videoinput('winvideo',1,'YUY2_160x120')

Summary of Video Input Object Using 'Webcam-101'.

Acquisition Source(s):  input1 is available.

Acquisition Parameters:  'input1' is the current selected source.
                        10 frames per trigger using the selected source.
                        'YUY2_160x120' video data to be logged upon START.
                        Grabbing first of every 1 frame(s).
                        Log data to 'memory' on trigger.

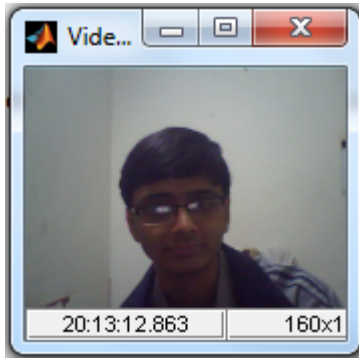
Trigger Parameters:  1 'immediate' trigger(s) on START.

Status:  Waiting for START.
        0 frames acquired since starting.
        0 frames available for GETDATA.
```

'vid' is has been declared as the video input object, and now the camera can be referenced by this name.

To see a preview of the video through your camera, use preview command

```
>> preview(vid)
```



You can check out the preview with different formats. You will see that the size of preview window changes. Use a format that suits your size requirement. A larger size gives greater clarity and is easier to work with, but consumes more memory and therefore is slow. But in a smaller image, it is difficult to differentiate between two objects.

Capturing Images

Now you have a video stream available and you need to capture still images from it. For that, use getsnapshot() command.

```
>> im=getsnapshot(vid);           % where vid is video input object
```

Here 'im' is the 3D matrix and you can see the image by the usual imshow() command

```
>> imshow(im);
```

If you get an unexpected image (with shade of violet/green/pink and low clarity), there is nothing to worry. You must be using a format starting with 'YUY2_...' which means that your image is in YCbCr format and not RGB format. Therefore, you must convert it in RGB format by using

```
>> im=ycbcr2rgb(im);
>> imshow(im);
```

If a format somewhat like 'RGB24_160x120' (anything starting with RGB24_....) is used, then you directly get the image in RGB format.

If you want to store an image captured, so that you can view it later (like .jpg, .png, .bmp), you can use imwrite()

```
>> imwrite(im, 'myimage.jpg');
```

The file *myimage.jpg* is saved in the current working directory.

***Note:** It takes time for the camera to be active and sufficient light to enter it after giving preview() command. Hence, there should be a time gap of 2-3 seconds between preview() and getsnapshot(). If you are typing in command window, then you can maintain this time gap manually. But generally you will use these commands in an m-file, hence, the delay must be given using pause() command. Also in a vision based robot, the images have to be taken continuously, so use an infinite loop.

```
vid=videoinput('winvideo',1, 'YUY2_160x120');
preview(vid);
pause(3);
while(1)
    img=getsnapshot(vid);

    % Do all image processing and analysis here

end
```

Once you have the image matrix 'im', you can perform all the operations like extracting region of particular colors, finding their centroid and area, etc. by the methods described previously.

SIGH...! Now sufficient Image Processing has been learnt. Great! What next?

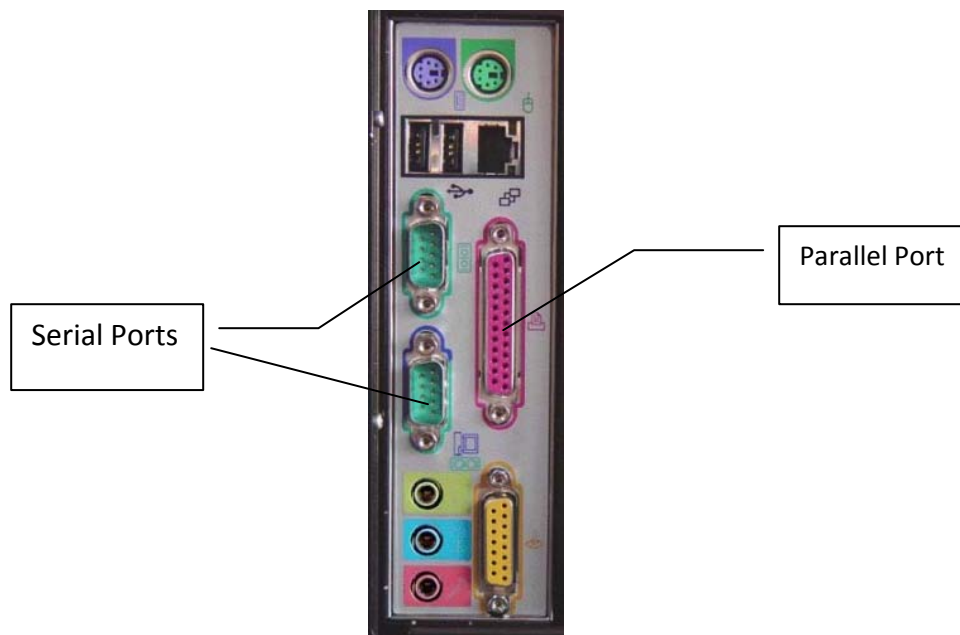
Something apart from image processing must be known to make and run the robot. How to interface your computer with your robot via serial/parallel port? How can you send instructions from MATLAB to your robot and control it?

The control of the robot is a dynamic process. You continuously take images using webcam, process them, find the required information like position of an object, orientation of robot (if there is an overhead camera) and finally the task to be performed (move robot left / right / forward / backward / pick object, etc.). According to the task at hand, some data is output on the serial/parallel port.

Interfacing via PC Ports

The Instrument Control toolbox of MATLAB provides support to access serial port (also called as COM port) and parallel port (also called as printer port or LPT port) of a PC.

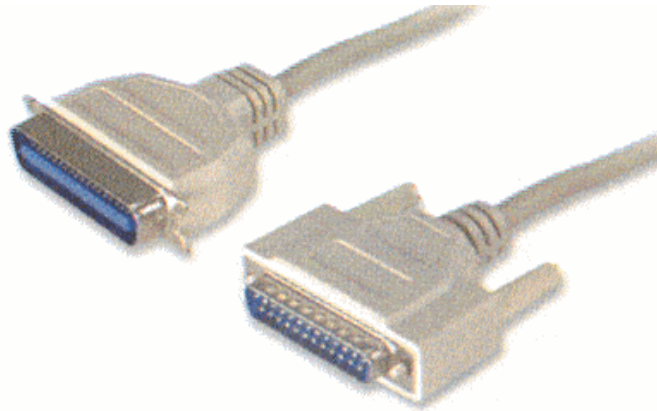
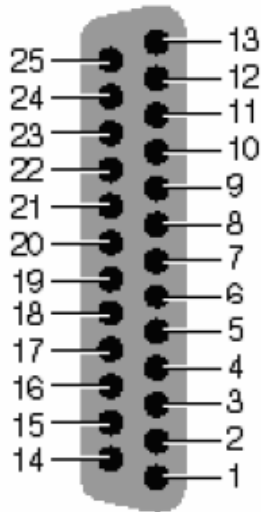
If you are using a desktop PC or an old laptop, you may have both, parallel and serial ports. However in newer laptops, none of them may be available and you will have to use USB to Serial Converter cable. Please note that USB to Parallel converter is not recognised as a virtual parallel port.



USB to Serial Converter

Parallel Port

Parallel port has 25 pins as shown. Parallel port cables, commonly referred as printer port cables are easily available. These cables are handy to connect port pins with your circuit. Pins 2-9 are bi-directional data pins (pin 9 gives the most significant bit (MSB)), pins 10-13 and 15 are output pins (status pins), pins 1,14,16,17 are input pins (control pins), while pins 18-25 are Ground pins.



Parallel port cables

courtesy: www.cknow.com

MATLAB has an adaptor to access the parallel port (similar to adaptor for image acquisition). First you must check from the Device Manager of your system (My Computer->System Properties->Device Manager->Ports) that what is the name given to the parallel port of your computer. Let's say it is 'LPT1'.

To access the parallel port in MATLAB, define an object; let's say by the name 'parport'

```
>> parport = digitalio('parallel','LPT1');
```

You may obtain the port address using,

```
>> get(parport,'PortAddress')
>> daqhwinfo('parallel');           % To get data acquisition
                                     hardware information
```

You have to define the pins 2-9 as output pins, by using **addline()** function

```
>> addline(parport,0:7,'out')
```

Now put the data which you want to output (depends on motions of robot required) to the parallel port into a matrix (See function **logical()** in MATLAB Help); e.g.

```
>> dataout = logical([1 1 0 1 0 1 0 1]);
```

Now to output these values, use the **putvalue()** function

```
>> putvalue(parport,dataout);
```

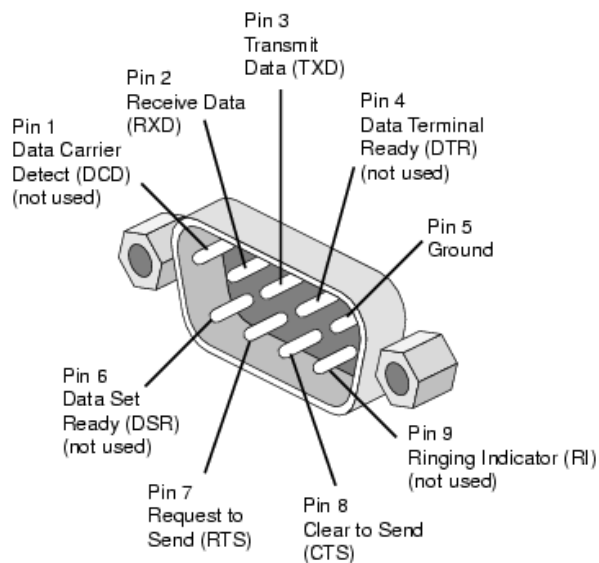
Alternatively, you can write the decimal equivalent of the binary data and output it.

```
>> data = 213;
>> putvalue(parport,data);
```

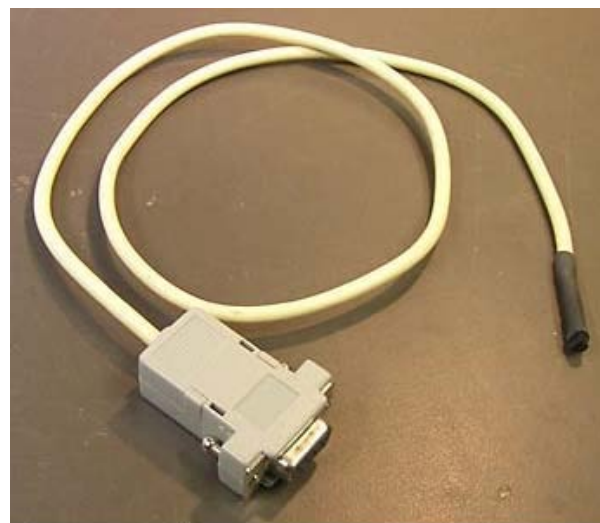
You can connect the pins of the parallel port to the driver IC for the left and right motors of your robot, and control the motion of the vehicle. You will need an H-bridge for driving the motor in both clockwise and anti-clockwise directions (like L293 or L298).

Serial Port

A serial port has 9 pins as shown. If you have to transmit one byte of data, the serial port will transmit 8 bits as one bit at a time. The advantage is that a serial port needs only one wire to transmit the 8 bits. Pin 3 is the Transmit (TX) pin, pin 2 is the Receive (RX) pin and pin 5 is Ground pin. Other pins are used for controlling data communication in case of a modem, hence not required for our purpose. In fact, even pin 2 is also not required if we don't need to receive any data from our robot.



Pins in a Serial Port



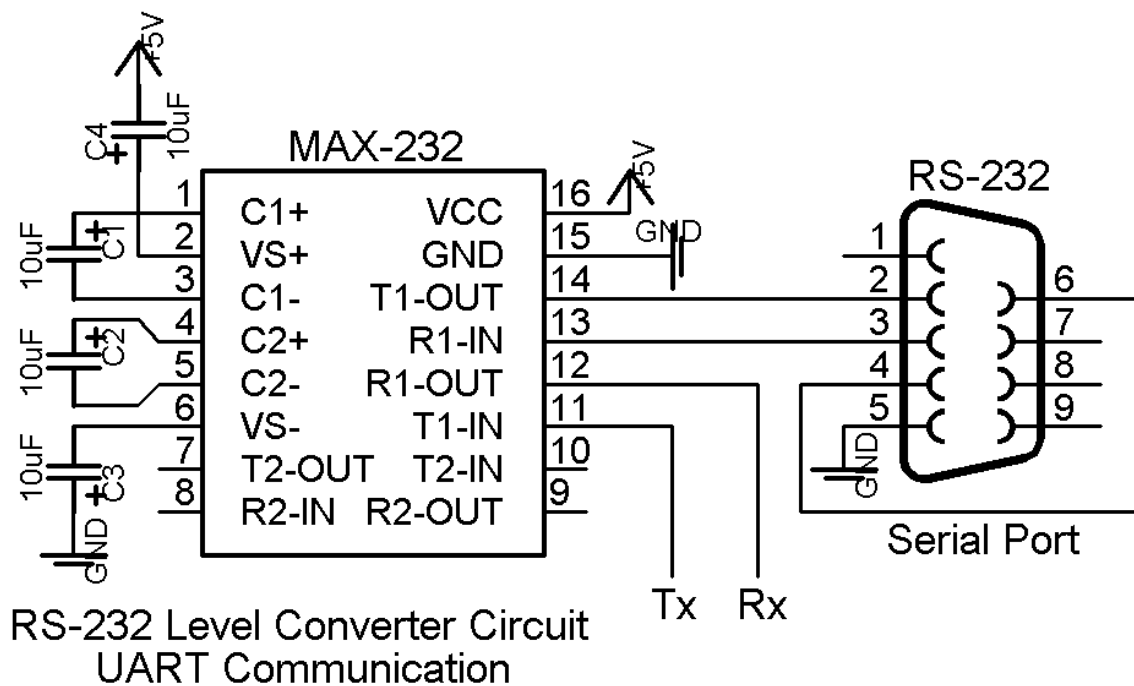
courtesy: martybugs.net

A Serial Port Cable

At the receiver's side (robot), a microcontroller with UART (Universal Asynchronous Receiver Transmitter) is required aboard the robot, to interpret the serial data. Most of the microcontrollers like AVR ATMEGA 16, Atmel/Philips 8051 or PIC microcontrollers have a UART. The UART needs to be initialized to receive the serial data from PC.

Something about serial communication

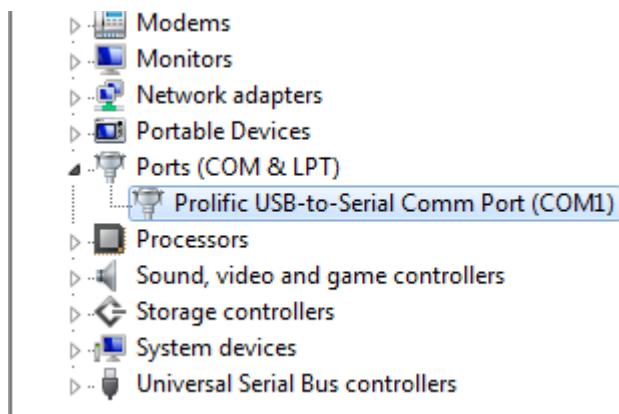
The standard used for serial communication is **RS-232** (Recommended Standard 232). The RS-232 standard defines the voltage levels that correspond to logical one and logical zero levels. Valid signals are plus or minus 3 to 15 volts. Now we know that this is not the voltage level at which our microcontroller works. Hence, we need a device which can convert this voltage level to that of CMOS, i.e., logic 1 = +5V and logic 0 = 0V. This task is carried out by an IC MAX 232, which is always used with four 10uF capacitors. The circuit is as shown:



***Note:** The Tx and Rx shown in above figure (pins 11 and 12 of MAX232) are the Tx and Rx of microcontroller. For example, PD0 and PD1 in Atmega 16.

For controlling the robot via serial port, the microcontroller is connected to the motor driver ICs which control the right and left motors. After processing the image, and deciding the motion of the robot, some predefined codewords (let's say 'A'=Left, 'B'=Right, 'C'=Forward, 'D'=Backward, 'E'=Pick object, etc.) are sent to microcontroller through the serial port.

First you must check from the Device Manager of your system (My Computer->System Properties->Device Manager->Ports) that what is the name given to the serial port of your computer (or the virtual serial port of USB to Serial Converter). Let's say it is 'COM1'.



Now we can access the serial port from MATLAB by defining a serial port object, let's say by the name 'ser' as

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

This defines the Baud rate as 9600 bps. You can use any Baud rate, but **make sure** that the same Baud rate has been configured in the onboard receiver microcontroller also. 9600 bps is the default baud rate.

Now we open the serial port and send data through it.

```
>>fopen(ser);  
>>fwrite(ser, 'A');           % Move robot left  
>>fwrite(ser, 'E');           % Pick object
```

As the microcontroller interprets the serial data from the PC, it suitably controls the motors through its output pins which are connected to the motor driver or H-bridge (like L293 or L298).

Some suggestions

As mentioned in the start of this tutorial, the best way to learn by yourself is to go through the demos provided by Matlab. Some of the demos which I suggest are:

In Image Processing Toolbox:

- Identifying Round Objects
- Measuring Angle of Intersection
- Measuring the Radius of a Roll of Tape

In Instrument Control Toolbox:

- Serial Port Tutorials

So that's all I have in store! This should be enough to get to the first taste of image processing in robotics. I hope this introduction excites you towards venturing further into this field.

GOOD LUCK!

If you have any suggestions that you would like to share with me, feel free to contact me at:

ankur.ag12@gmail.com

You can also visit my homepage

www.ankuragrawal.co.nr

References

Following resources were referred while compiling this tutorial:

- The documentation and demos provided in the Help section of MATALB
- <http://www.thinklabs.in/resources/?p=47>
- <http://techfest.org/competitions/ibots/freekick/resources/FreeKick%28Tutorial%29.pdf>
- <http://www.sourabh.sankule.com/tutorial/avr/uart.html>