# ***Part 1 – Matlab***

## *What is Matlab?*

Matlab is a program that performs various numerical operations. Like a big calculator. Computer languages are generally divided into *low-level languages*, that interact with the specific hardware directly and need to be both written and compiled for the specific setting you are using. This is very powerful, because it allows you to use the resources of your machine in whatever way you choose. *High-level languages*, on the other hand, can be transferred from machine to machine (and, in some cases, from operating system to operating system), but often will need to be compiled for a specific setting. Matlab functions as a *scripting language.* Scripting languages are high-level computer languages. However, above and beyond the portable nature of most high-level languages, a system-specific interpreter interprets them online, as they run. Therefore, you will not need to compile the programs you write on Matlab. Scripting languages are relatively easy to learn. However, they do not retain the same level of flexibility as low-level languages. Moreover, because they need to be interpreted as they run, they are often slower than the equivalent program written in a compiled high-level language.

## *Starting to work with Matlab:*

You will need a computer running Matlab. Matlab runs on Macs and PCs running Windows or Unix The student version of Matlab doesn't cost that much ($95 at TSW) and can be used for everything we will be doing here, but notice that it can't do everything that you can do with the standard version of Matlab.

Start Matlab. A window will appear that's divided into a number of sub-windows. The "command window" is the one which has a little prompt '>>'. The prompt is your interface with Matlab, for now. When you type things at the prompt and press Enter, your commands are processed by Matlab.

Type at the prompt:

```
>>str1='I have no clue what I am doing'
```

You just told the computer to create a list of letters 'I have no idea what I'm doing' and to name that list of letters str1. str1 is a ***variable*** - The single quotes tell the computer that str1 is a list of letters (not numbers, more on that later). Any list of letters is called a ***string***. Now type:

```
>>who
```

This command asks your computer to give you a list of all the variables you have. At the moment the only variable you have is str1.

```
Text that looks like this is stuff
that is happening in the command
window. Either stuff you are typing
in at the prompt, or stuff that the
command window is spitting back out.
```

Text that looks like this is me telling you what's happening in Matlab for a particular command.

Text that looks like this is me giving you a general overview.

Typing the name of a variable asks your computer to tell you what is contained within that variable.

```
>>str1
```

The computer should show you what's in str1

```
str1 =

I have no clue what I am doing
```

Using disp also displays what a variable is, but only shows the contents, instead of repeating the name of the variable

```
>>disp(str1)
```

So all the computer does is display the contents of the variable as follows:

```
I have no clue what I am doing
```

```
>>str2='Is it all going to be boring?';
>>str2='Is it all going to be boring?'
```

The first time you typed this you added a semi-colon at the end. The second time you didn't. The semi-colon tells the computer whether or not you want it to display the output of each command. This will be useful later when you write scripts. When you debug your programs, sometimes you want the result of a line to be output to the command window, as a sanity check. You can control that by adding or removing semi-colons.

```
>>who
```

Now you have both str1 and str2

```
>>str1='I still have no clue what I am doing'
```

Now you are re-defining str1 by making it represent a slightly different list of letters

```
>>str1
```

See - the list of letters contained within str1 has changed

```
>>str1(3)
```

You've asked the computer to display the third letter in str1. This is called ***indexing*** or ***subscripting***. 3 is an index (or subscript) into the third character in str1. Now try:

```
>>str1(6)
```

You can see from this that the computer is counting spaces

```
>>mixstr=str1;
```

Now you've created a new variable called mixstr. You've told the computer to make mixstr the same as str1

```
>>mixstr
```

See - they are exactly the same

```
>>mixstr(3)=str1(1);
```

Now you are telling the computer to make the 3rd letter in mixstr the same as the 1st letter in str1

```
>>mixstr(1)=str1(3);
```

Now make the 1st letter in mixstr the same as the 3rd letter in str1

```
>>mixstr
```

You should now have:

```
's Itill have no clue what I am doing'
```

You can also create lists of numbers. These are called ***arrays*** or ***vectors***. Here's four different ways of creating a vector list that goes from 2 to 9 in steps of 1. A ***variable*** is a generic term that can be used to describe a ***character*** (a single letter), a ***string*** (a string of characters), a ***double*** (a single number, more on that later), a ***vector***, or a ***matrix*** (a two or more dimensional set of numbers) and other types of data structures.

```
>>array1=[2 3 4 5 6 7 8 9]
>>array1=linspace(2, 9, 8)
```
Here you are saying you want a list of 8 numbers that are evenly spaced between 2 and 9. You can imagine that this command would be useful if you had collected 8 pieces of data evenly spaced between two and 9 seconds.

```
>>array1=2:1:9
```
Here you are saying that you want a list of numbers that goes from 2 to 9 with a step-size between each number of 1. You can imagine that this command would be useful if you collected data that went from 2 seconds, to 9 seconds, and had collected data every second.

```
>>array1=2:9
```
Matlab assumes a default step-size of 1, so you can simply skip it for this particular way of creating vectors.

You can also index vectors. 2 indexes the second integer in array1
```
>>array1(2)
```

You can also index more than one number in an array or string.
```
>>array1(2:4)
```

```
>>disp(array1)
```

`disp` can also be used to display numbers

## *Matrices and Calculations*

The real power of Matlab comes from using matrix computations (Matlab actually stands for 'Matrix Lab'!). As mentioned earlier, you can have lists of numbers as well as of letters. These list of numbers can be either one-dimensional (vectors, or arrays), or n - dimensional (for n>1; matrices). Let's give this a shot:

```
>>mat1=[1 54 3; 2 1 5; 7 9 0; 0 1 0]
>>mat2=[1 54 3
  2 1 5
  7 9 0
  0 1 0]
```
Take a look at mat1 and mat2. As you can see there is more than one way of entering a matrix. `mat1` and `mat2` were entered differently, but both have 4 rows and 3 columns. When entering matrices a semi-colon is the equivalent of a new line. You can find the size of matrices using the command "size.
```
>>size(mat1)
```
For a two dimensional matrix the first value in `size` is the number of rows. The second value of `size` is the number of columns.

Now try:
```
>>vect1=[1 2 4 6 3]
>>vect2=vect1'
```
Vectors can be tall instead of long, ' (the little symbol below the double quote on your keyboard) is a ***transpose***, and allows you to swap rows and columns.
Remember there is also the command `whos,` which will tell you the size of all your variables.
Use `whos` to look at the size of mat1, mat2, vect1 and vect2.
```
>>whos
```

What does `mat1'` look like?
You can perform various calculations on matrices and arrays.

You can add a single number (also called a scalar) to a vector

```
>>vect1+3
```

You can subtract

```
>> vect2-3
```

You can add a vector onto itself

```
>>vect1+vect1
```

You can also add two vectors as long as they are the same size. You can't add vect1 and vect2 together since they are different sizes.

```
>>vect1+vect2
??? Error using ==> plus
Matrix dimensions must agree.
```

The reason you got an error is that you can't add something with 1 row and 4 columns to something else that has 4 rows and 1 columns.

```
>>vect3=[2 4 8 12 6]
>>vect1+vect3
```

These two vectors are the same size, so you can add them together.

## *Matrix multiplication and division*

When you want to add two vectors or matrices to each other you need to know that there are two sorts of multiplication and two sorts of division. The simple kind of multiplication and division is called array multiplication (also known as **scalar multiplication**). If you are multiplying/dividing a vector/matrix by a single number, you can do that as you did before:

```
>>vect1*3
>>mat1*0.5
>>vect1/2
```

However, if you are multiplying a vector/matrix by another vector matrix, you need to tell matlab what kind of multiplication to do (as you will see below, there is more than one way to multiply matrices). Scalar multiplication will proceed element by element: each element in the first vector is multiplied by the corresponding element in the second vector. You do this using the symbols `.*` and `./`

Note that the vectors must be the same shape and size:

```
>>vect1.*vect3
>>vect1./vect3
>>vect3./vect1
```

Watch out for the transpose in the next 2 examples:

```
>>vect1.*vect2'
>>vect1.*vect3'
??? Error using ==> times
Matrix dimensions must agree.
```

You will get this error a lot. What an error like this means is that the two vectors or matrices that you are trying to perform an operation on (such as multiplication) aren't the right size or shape to do what you are doing. Lots of operations are fussy about making sure the sizes of the vectors or matrices are consistent. So when you get this error the first thing you should do is check the `size` of all the variables that you are tying to manipulate, and try to work out whether they might be different sizes. Often it's the case that simply transposing one of the variables is all you need to do. This is an important thing to understand, so don't rush through the examples above. Make sure you understand what's going on.

```
>> mat1./vect1
??? Error using ==> rdivide
Matrix dimensions must agree.
```

This is not going to happen for you, regardless of how you transpose mat1 and vect1. vect1 and mat1 will never be the same size, no matter what you do.

So, with point-wise multiplication and division (which is what you will be doing most of the time) you can:

1) multiply or divide a matrix or vector by a single number (Notice that in this case either * or .* will work)

2) multiply or divide a matrix or vector by a matrix/vector that is the same size (make sure you use .* ).

It's best to get into the habit of using .* all the time, unless you are specifically using matrix multiplication (which you will only use rarely)

The second kind of multiplication and division is matrix multiplication and matrix division. In general matrix multiplication and division occurs through the following formula:

For the m-by-n matrix A and the n-by-p matrix B, the m-by-p matrix of multiplying the both will be defined by :

$$(A \cdot B)_{i,j} = \text{Sum}_{r=1\ldots n} \{A_{i,r} \cdot B_{r,i}\}$$

where i goes from 1 to m and j goes from 1 to p. Notice that you can use this in order to multiply any two matrices for which the inner dimensions agree, so a 1-by-n matrix can multiply an n-by-n matrix (how large will this product be?), but an n-by-n matrix cannot multiply a 1-by-n matrix. Conclusion – matrix multiplication is not commutative – so be careful!

This produces two general types of matrix multiplication.

*Outer product:*

```
>>B = [1; 2; 3; 4; 5]
B =
     1
     2
     3
     4
     5

>>C = [2 3 4 3 2]
C =
     2     3     4     3     2


>> whos
  Name        Size                     Bytes  Class

  B           5x1                         40  double array
  C           1x5                         40  double array
```

The first vector is tall and thin, and the second vector is short and fat. Calculating the ***outer product*** of the two vectors:

```
>>B*C

ans =
      2      3      4      3      2
      4      6      8      6      4
      6      9     12      9      6
      8     12     16     12      8
     10     15     20     15     10
```

If we transpose both B and C and reverse the multiplication order, then the number of rows in C' again matches the number of columns in B', and we can calculate another outer product, which is in fact the transpose of the previous outer product:

```
> C'*B'

ans =

      2      4      6      8     10
      3      6      9     12     15
      4      8     12     16     20
      3      6      9     12     15
      2      4      6      8     10
```

*Inner product:*

Here the first vector is short & fat, and the second vector is tall & thin.

```
>>C*B
```

With inner products you can again transpose both vectors and swap the order of the multiplication. This time you get the same answer – 42.

## *Creating Programs*

By now you should be getting a little irritated with having to type in every command one at a time. We are therefore going to create a ***program*** – a program is simply a document containing a sequence of commands.
In Matlab programs are written in documents called ***m-files***. So now we are going to put the commands you just did in a m-file.

Make sure the command window is at the front. Now go to the menu bar and choose File->New->M-file.
You'll get a blank document in a new editor window in which you can write your program.
Let's look at a program that uses some of the things we've learned so far:

```
Green is comments
Purple is text that is part of a string
Blue is commands that start and end loops
```

You'll notice in the code below that some of the lines are rather long. When you are writing code and a line is longer than your page you can break it using three dots : '. . . ' . Without those three dots Matlab will treat each part of the line as a separate command and give you an error message.

```
% Calculations.m
%
% Example program that carries out a series of
% calculations on two numbers and two matrices
%
% written by IF and GMB 4/2005
%
% modified by ASR 7/2007: added matrix operations

clear;
n1=input('choose the first number ... ');
n2=input('choose the second number ... ');

% spit responses out onto the command line
disp(['first number is ', num2str(n1)])
disp(['second number is ', num2str(n2)])
disp([num2str(n1),'+', num2str(n2), ...
 '=', num2str(n1+n2)]);
disp([num2str(n1), '-', num2str(n2), ...
'=', num2str(n1-n2)]);
disp([num2str(n1), '*', num2str(n2), ...
 '=', num2str(n1.*n2)]);
disp([num2str(n1), '/', num2str(n2), ...
 '=', num2str(n1./n2)]);
if (round(n1)==n1)
   disp(['n1 is an integer, n2 rounded = ', ...
       num2str(round(n2))]);
end
if (round(n2)==n2)
   disp(['n2 is an integer, n1 rounded = ', ...
       num2str(round(n1))]);
end;
disp([num2str(n1), ' to the power of ', ...
   num2str(n2), ' is ' ,num2str(n1.^n2)]);
disp(['the smallest number of ',num2str(n1), ...
    ' and ',num2str(n2), ' is ', num2str(min(n1, n2))]);
if n1>0 & n2>0
   disp('Both n1 and n2 are greater than zero');
end

A = [0:n1:n1*10 ; 0:n2:n2*10].*n1;
B = ([0:n2:n2*10 ; 0:n1:n1*10]./n2)';

disp('The matrix A is:');
disp(A);
disp('The matrix B is:');
disp(B);

disp ('A * B = ');
disp(A*B);
disp('B * A =');
disp(B*A);
```

```
54      disp ('A.*n1 = ')
55      disp(A.*n1);
56      disp('B./n2 = ')
57      disp(B./n1);
```

Lines 1-8: Every program should begin with a few lines of documentation. This is called a ***header***. Good headers contain the following information:
1) The name of the program
2) A description of what it does
3) Who wrote it, and when
4) If you are changing an existing program, add a comment about that. The time you did it and what you changed

You can run Calculations.m in two ways. One is to type the name of the program into the command window:
`>Calculations`
Alternatively you can go to the menu bar and choose Debug-Run (can also be invoked by pressing F5).

***Lines 11-12.*** `input` prints a string onto the command window and waits for a response. The expected response needs by default to be a number. If you try running the program and type a letter in (a *character*) rather than a number – you will get an error:

```
??? Error using ==> input
Undefined function or variable 'r'.
```

]It's possible to get input to accept characters and strings instead of numbers by explicitly telling it that the input won't be a number. Try this at the command window:
`>> yourname=input('What is your name? ', 's')`
Here the `'s'` tells `input` that the input will be a string instead of a number.

***Lines 14-23.*** num2str converts the number n1 into a string. So:
`['first number is ', num2str(n1)]`
is one long string, which is then displayed by disp.

***Line 25***. The == checks to see if round(n1) is the same number as n1. Try
`>3==3`
This will give you a 1 since it is true
`>3==3.4`
This will give you a 0 because it is untrue.
It's important to remember the difference between == and =,
The single = tells Matlab to make the variable x be 3.
`>x=3`

```
x =

     3
```

The double == asks Matlab to check whether or not x is equal to 3, and returns and answer of 1 if x *does* equal 3 and an answer of 0 otherwise.

```
>x=3.4;
>x==3
ans =

     0
```

Remember how we explained that numbers could either be integers (e.g. 3) or be doubles (3.12). On Line 22 we check to see whether n1 and n2 are round numbers by seeing if the rounded version of the number is the same as the number itself. If this is confusing try:

```
>round(3.15)
>round(3)
>round(3.14)==3
>round(3.14)==4
```

Remember that the way == works is that you get an answer of 1 if the numbers on either side of the == are the same, and a 0 otherwise.

`if` loops work in the following way: The if statement checks the condition that follows the if. For this if statement the condition is (round(n1)==n1). If the output of that condition is a 0 (the condition is false) then the statement between the `if` and the `end` (line 23-24) is **not** executed. If the output of the condition is anything but 0 (the condition is true), then the statement after the `if` is executed. Traditionally a true condition is represented by a 1, but in Matlab an `if` loop will be executed unless the condition following the `if` results in a 0.

***Line 37*** The `&` operator is used when you only want to carry out a loop only when more than one condition are both true. `&` checks to see if *both* the statement before and after the & are true. I.e. `Condition 1 & Condition 2` gives you a 1 if both conditions are true, and gives you a 0 otherwise. Try the following:

```
> 3>0 & 2>0
> 3>0 & -1>0
> -1>0 & 3>0
```

***Lines 41- 56*** Demonstrate the use of matrices and calculations done on matrices. Notice that in order for the multiplication to work, one of the matrices needs to be transposed, so that the inner dimensions of the matrices match in both multiplications (outer and inner).

Now we need to save the file. Make a folder called "PTBTutorial" somewhere. Create a subfolder called "Misc". **Don't** put these folders inside the Matlab application folder or you will lose everything if you reinstall Matlab.

Save the file as Calculations.m (the same as the header) in the "Misc" folder.

Now go back to the command window and type

```
>>help Calculations
```

You will almost certainly get an error message:

```
Calculations.m not found.
```

When the computer says that a file is "not found" that means the computer can't find an m-file that has that particular name. The reason the computer can't find the file even though you saved it in the folder 'Misc' is because the computer is only allowed to look for files in certain places.

One place that the computer always looks for files is the ***current directory*** or ***working directory***. In fact this is the first place that the computer looks. When Matlab opens it automatically links to a particular folder (the default setting is made by Matlab). If you don't tell it otherwise it will save files to that folder. You can see what folder Matlab thinks is the current directory using the ***print working directory*** command:

```
>>pwd
```

The other places that the computer can find files are in folders that are in Matlab's ***search path***. This ***path*** is a simply a list of folders that the computer is allowed to look in

whenever it is trying to find a file. Again Matlab comes with a default set of paths. You can get a list of the current folders in the path very easily:

```
>>path
```

To tell the computer where to look, you ***set the path*** …

## *Setting the path via the menu bar*

Make sure the command window is at the front, and go to:
File->Set Path in the menu bar. A pop-up window will appear
Choose "Add With Subfolders", choose the "Misc" folder, and click OK. Then choose Save and Close.

Now type

```
>> help Calculations
```

You should see the information entered in the header.

The command `help` tells the computer to display the header you wrote. That's why you always need to write headers for every m-file that describe clearly what the program does and how to use it.

An important thing to remember about setting your path is that if there are two files with the same name, and your path allows the computer to see both of them you won't get a warning, the computer will just use the first one it finds!
You can find out which file the computer is using by typing

```
>>which Calculations
```

Matlab should spit out something like the following depending on where you put Calculations.m on your computer.

```
/Users/ariel/PTBTutorial/Misc/Calculations.m
```

You only have one version of Calculations. If you had multiple versions, Matlab would print out the path for each version that was within the path. The one at the top of the list is the version that Matlab will use by default at that current moment.

But be careful – which version that is used depends on what directory is the current working directory for Matlab. If the current directory changes it is possible that the version of MixStrings that is used will also change. This can lead to some really weird ***bugs*** (a bug is any time a program doesn't do what you want it to do and you have no idea why). One really confusing thing that can happen if more than one file of the same name is in the path is that you can make changes to a file, but the changes don't affect what the computer does. What's happening is that the computer is not actually using the file that you are changing – it is using a different file of the same name somewhere else in the path.

*So you need to be careful about two things:*
1) Don't be sloppy about having multiple m-files with the same name sitting in different directories
2) Be careful about your path.
Make sure that old directories with out-of-date files in them aren't still in your path. One good technique for path management is not to simply put folders in your default path so

they are permanently in the Matlab path. Instead, when you write a program you simply add the paths you will need for that program at the beginning of the program using Matlab commands (instead of the Menu bar). This technique lets you have different paths depending on which programs you are running, instead of having one mega-path. Using this technique to manage your path minimizes the probability of having out-of-date folders in your path.

## *Changing the path within Matlab*

Play with the following commands and use them to move to your MatlabClass/Misc directory.

```
>>pwd
```

This tells you the current directory

```
>>cd ..
```

Moves up one directory

```
>>ls
```

Lists the files in that directory. You can move to any of the files listed in that directory:

```
>>cd Misc
```

Moves to folder Misc (provided you are in a folder that contained the subfolder Misc). You should be able use these commands to navigate to your Misc folder. Then you can add Misc to your path as follows:

```
>>addpath(pwd)
```

What you are doing here is telling the computer to add the folder you are currently in (`pwd`) to the path.

Remember how in my computer I used which to find that Calculationss lived in the folder /Users/PTBTutorial/Misc
Well I can now use that information at the beginning of Calculations to make sure that the folder Misc is in my path by just adding the line
addpath('/Users/ariel/PTBTutorial/Misc') at the beginning of the program

## *Help – your new bestest of friends*

One of the advantages of working with Matlab is that there exist built-in functions to do most of the numerical operations you can think of (and a whole lot of numerical operations you have never thought of). How would you know if the operation you want to do is already built-in? One way to do this is through the exstensive help module. Open the help module and type in the search bar your favorite mathematical operation. For example, try typing in 'cross-correlation'. You will get several pages of documentation, among them is a help page for the function 'xcorr', that actually computes the cross-correlation function of two vectors.
If you know the name of the built-in script you want to get help about, another way to access the documentation on this script is to type:

```
>>help xcorr
```

This types the header of the script to the command window. If the programmer writing this script has done a good job, this will provide you with enough information to run the script. Don't be shy to use help – you will probably use this feature of Matlab more often than any other feature of the program.

## *Functions*

A function is a self-contained block of code that performs a coherent task of some kind. When you send a task to a function, you give the function all the variables it needs to know, and it returns the variables that you want. The calculations within the function are hidden.

*Why Functions?*
There are three main reasons for writing functions.
1) To make your code readable – hiding pieces of code in functions makes the overall structure of your code clearer and easier to read.
2) To shorten your code – if you do the same thing repeatedly putting it in a function allows you to call the function repeatedly instead of repeating the same lines of code again and again.
3) To give you a library of routines. You will do many things again, and again in your career writing code. Writing a function to do it means that the next time you need to do that particular manipulation you don't need to rewrite that piece of code.
4) To save memory. When we create a function, we are only interested in the output of that function. All the other variables that are created within a function are only there in order to help us get to that answer. The memory taken by these variables will be freed once the function has terminated.

Whenever you write code, think about whether you will ever need that particular piece of code again – if you will, make it a function. (Even if it's just two lines – it will be easier finding a function then rummaging through old code for those precious two lines.)

Here is a very simple function.

```
1       function output=SimpleFunction(input)
2       %
3       % a very simple function
4       %
5       % written by if 4/2007
6
7       output=10*input;
8
```

The first line of your m-file defines this piece of code as a function. The terminology is that any variables you want to ***return*** from the function come ***before*** the equals sign. Variables you want to ***send*** into your function go inside the brackets ***after*** the function name.

You can send more than one variable in, and get more than one variable out.

```
1       function [output1, output2]=SimpleFunction2(input1, input2)
2       %
```

```
3       % Another very simple function
4       %
5       % written by if 4/2007
6
7       output1=input1.*input2;
        output2=input1./input2;
```

Actually you have been using functions already, many of the commands you have been using already are functions – e.g. `round` – you give the command `round` a number as input and it returns the closest integer as the output.

We will see extensive use of functions next time, when we start programming an experiment.

## *Plotting*

For completeness' sake, I add this section on plotting. One of the uses of matlab is creating visualizations of data. This can be done through a rather sophisticated system of plotting functions. We will use some of these when we turn to analysis of data from psychophysical experiments. You can get more information about plotting by using Matlab's interactive help.

Let's start by plotting a linear function:

```
>>figure(1)
>>x=0:5:100;
>>y=3*x+4;
>>plot(x, y)
```

The figure window has various properties. You can access them with the command:

```
>>get(gcf)
```

`gcf` stands for ***get current figure***. It basically provides a handle through which we can access the properties of the last figure that was referred to. It will tell you a lot about the current properties of the figure.

You can find out about the possible options for a figure using
```
>>set(gcf)
```

You can also change various figure properties using the set command
```
>>set(gcf, 'Color', [1 1 .5])
>>set(gcf, 'Position', [100  100  300 300])
>>set(gcf, 'Pointer', 'ibeam')
>>set(gcf, 'PaperOrientation' , 'landscape');
>>set(gcf, 'Name', 'My Data Figure');
```

If you have more than one figure and you want to be able to access and change properties on more than the most recent figure. In that case you need to create a handle to each figure.

```
close all
fh1=figure(1);
fh2=figure(2);
set(fh1, 'Color', [1 1 .5])
set(fh2, 'Color', [.5 1 1])
```

You can also put figure handles into an array

```
>>close all
>>clist=[1 1 .5; .5 1 1]
>>for i=1:2; fh(i)=figure(i); set(fh(i), 'Color', clist(i, :)); end
```

*Subplots*

You might want the figure to contain subplots. In Matlab each subplot is called an ***axis***. If you don't define the number of subplots then Matlab assumes you have a single axis (the whole figure).

```
>close all
>figure(1)
>subplot(2,2,1)
```

Like figures, subplots have various properties. You access them very similarly to the way that you access figure properties

```
>get(gca)
```

Here `gca` stands for ***get current axis***. It provides a handle to the last axis that Matlab created. It will tell you a lot about the current properties of the axis. Note that an axis has different properties from those of a figure. Once again you can look at the various options you have for axis properties using

Here's a table of the most useful properties of axes that you might want to change. But you can find and modify lots of other properties by using `get(gca)` and `set(gca)`

`>>help plot` is also a really good way of getting information about plotting

***** These axis properties can also be set as part of plot properties.

****** These axis properties are defined using a slightly different command structure

| ***Axis property*** | ***How to set the property*** | ***Comments*** |
|---|---|---|
| Color | `set(gca, 'Color', [1 1 .7])` | Sets the color inside the subplot. Takes as input a single color. |
| *ColorOrder | `set(gca, 'ColorOrder', gray(8))` | Defines the order of colors used in plot commands. Takes a colormap as input |
| FontName | `set(gca, 'FontName', 'Times')` | Defines the font, uses normal computer fonts, e.g. 'Helvetica' , or 'Arial' |
| FontSize | `set(gca, 'FontSize', 12)` | Defines font size |
| FontWeight | `set(gca, 'FontWeight', 'bold')` | Thickness of the font, can take the properties of [ light \| normal \| demi \| bold ] |
| *LineStyleOrder | `set(gca, 'LineStyleOrder', … {'-'; '--'; '-'})` | Defines the order of line styles used in plotting the figures. Takes a structure defining the line styles |
| LineWidth | `set(gca, 'LineWidth', 2)` | Sets the line width of the axis lines |
| ** title | `title('my subplot ')` *or* | Title |

| | | |
|---|---|---|
| | `title(gca, 'My Title')` | |
| ** xlabel / ylabel | `xlabel('my X axis') / ylabel('my Y axis')`<br>*or*<br>`xlabel(gca, 'My x label')` | Labels x and y axis |
| ** XLim / YLim<br>*or*<br>axis | `set(gca, 'XLim', [0 3])`<br>`axis([0 3 0 1])` | Sets the X or Y limits of the plot. Two ways of setting the axes |
| XTick / YTick | `set(gca, 'XTick', 0:1.5:3)` | Decides where you want the tick values, takes as input a vector of where you want the tick values |
| XTickLabel / YTickLabel | `set(gca,'XTickLabel',…`<br>`[5 35 100])`<br>`set(gca,'XTickLabel', …`<br>`{'One';'Two';'Three'})` | Decides what you want to label the ticks as instead of using the numbers in XTick. Can either take a vector, or a structure containing the strings you want to use as axis labels |

Other useful commands are:

`>>axis equal`
sets the axis so tick marks are equally spaced on x and y axes

`>>axis square`
makes the current axis square

`>>axis off`
turns off all axis labeling, tick marks etc.

Like figures, if you have more than one subplot and you want to be able to access and change properties on more than the most recent subplot, you need to create a handle to each subplot, and use that to change the properties

```
>>close all
>>fh=figure(1)
>>set(fh, 'Color', [1 .6 1])
>>sp1=subplot(1, 2, 1);
>>sp2=subplot(1, 2, 2)
>>set(sp1, 'Color', [ 1 1 .6]);
>>set(sp2, 'Color', [.6 1 1]);
```

## *Line plots*

Once again, plots have properties that you can alter. But for plots there is no handy command like gcf and gca that allow you to reference the most recent plot (it's possible, but tricky). So you should work on defining handles from the beginning if you are planning to change plot properties. The kinds of properties you can change actually depend on the kind of plot you are doing.

Here are some of the properties you may often want to change for a **line plot** (like the linear function we plotted before)

| ***Line plot property*** | ***How to set the property*** | ***Comments*** |
|---|---|---|
| Color | `set(ph, 'Color', [ 1 0 0]) or`<br>`set(ph, 'Color', 'r')` | Sets the color of the plot line. Color can either define red, green or blue guns or you can use a shorthand for some specific colors. Options include: 'b', 'g', 'r', 'c', 'm', 'y', k' {blue, green, |

| | | |
|---|---|---|
| | | red, cyan, magenta, yellow black} |
| LineStyle | `set(ph, 'LineStyle', '--')` | Sets the style of the line. Options include:<br>[ {-} \| -- \| : \| -. \| none ]<br>(solid line, dashed line, dotted line, dash-dot line, no line) |
| LineWidth | `set(ph, 'LineWidth', 1)` | The thickness of the line |
| Marker | `set(ph, 'Marker', 'v')` | What the plot marker is. Options include: [ + \| o \| * \| . \| x \| square \| diamond \| v \| ^ \| > \| < \| pentagram \| hexagram \| {none} ]<br>The v< >^ symbols represent triangles of various orientations |
| MarkerSize | `set(ph, 'MarkerSize', 5)` | The size of the markers |
| MarkerEdgeColor | `set(ph,'MarkerEdgeColor', [0 0 1])` | The color of the marker edging |
| MarkerFaceColor | `set(ph, 'MarkerFaceColor', [0 0 1])` | The color of marker fill |

## *Legend and Text*

In much the same way as we have manipulated figures, axes and plots using handles, legends and texts can also be manipulated:

`text` takes as input the x and y position of the string you want to use, and the string. These text objects can also be assigned handles and you can change their properties. To see what those properties can be, you simply use `get` or `set`.

```
>>th1 = text (10,20, 'This is a piece of text')
>>set(th1)
>>get(th1)
```

`legend` takes in a list of the plot handles for which you want to create the legend, and a cell array containing the strings that will be the legend labels.
Again, you can use `get` and `set` to look at properties of the legend. You can change things like the location or the font size or type.

```
>>lh=legend(ph1, {'x'})
>>set(lh, 'Location','West', 'FontName', 'Arial',  'FontSize',
10)
```

```
1       % plotExample.m
2       %
3       % Demonstrates the use of plotting commands and of  %
4       annotation of plots and figures
5       %
6       % ASR made it 7/2007, using code from IF
7
8       close all
9       x=0:2:100;
10      y1=2*x;
11      y2=3*x;
12      y3=3.5*x;
13
```

```
14      ph1=plot(x,y1, 'r-');
15      hold on
16      ph2=plot(x,y2,'g--');
17      ph3=plot(x,y3, 'b:');
18      title('fake data')
19      xlabel('my x data');
20      xlabel('my y data');
21      th1=text(x(13), y1(13), 'red line');
22      th2=text(x(15), y2(15), 'green line');
23      th3=text(x(17), y3(17), 'blue line');
24      set(th1,'Color', 'r')
25      set(th2,'Color', 'g')
26      set(th3,'Color', 'b')
27      lh=legend([ph1 ph2 ph3], {'x2'; 'x3'; 'x3.5'})
```

Line 15: 'hold on' tells matlab not to remove the old plots from this figure, when new plots are added. This can be reversed using 'hold off'

## *More plotting in 2D*

As I mentioned before, Matlab plotting is rather sophisticated and enables you to create rather complicated plots. We continue with some more plotting commands in 2-dimensional plots

*Errorbar plots*
This type of plot presents errorbars.

```
close all
x=0:8:100;
y=3*x+4;
err=sqrt(y/4);
ph=errorbar(x, y, err)
```

In order to achieve maximal flexibility in determining the properties of the error bars separately from the properties of the plotting of the data, plot each of these separately (using '`hold on`' when adding a new plot to the figure).

*Bar plots*

You can also make barplots.

```
>close all
>x=0:5:20;
>y=[3*x+4]
>ph=bar(x, y)
```

Here are some of the properties you may often want to change for a simple bar plot

| ***Line plot property*** | ***How to set the property*** | ***Comments*** |
|---|---|---|
| BarWidth | `set(ph, 'BarWidth', .5)` | Width of bars |
| LineWidth | `set(ph, 'LineWidth', 1.5)` | Thickness of lines surrounding bars |
| EdgeColor | `set(ph, 'EdgeColor',[1 0 0 ])` | Color of the lines surrounding bars |

FaceColor `set(ph, 'FaceColor',[0 0 1 ])` Color of inside of bars

With more complex bar plots you get a handle for each set of data. Here we have two sets of y data for every x-value

```
close all
x=0:5:20;
y=[3*x+4;2*x+2]'
ph=bar(x, y)
set(ph(1), 'FaceColor',[1 0 1 ])
set(ph(2), 'FaceColor',[0 .3 1 ])
set(gca, 'XTickLabel', {'Day 1', 'Day 5', ...
    'Day 10', 'Day 15','Day 20'})
legend([ph(1) ph(2)],  'Happy', 'Sad', 'Location', 'NorthWest')
```

## *Bar plots with error bars*

Matlab doesn't have a function to do this, so you need to use a work around. In the same way as I showed you how to have more control over you error bar plots by plotting the error bars on top of the original data plot, you can plot error bars on top of a bar graph. You may have to fiddle with the parameter width to make it look right.

```
hold on
width=get(ph(1), 'BarWidth')-.1;
eh1=errorbar(x-width, y(:, 1), y(:, 1)/4, ...
    'Marker', 'none','Color', 'k', 'LineStyle', 'none')
eh2=errorbar(x+width, y(:, 2), y(:, 2)/5, ...
    'Marker', 'none','Color', 'k', 'LineStyle', 'none')
```

## *Hist*

`Hist` creates histograms. One way to use `hist` is simply to tell it how many bins you want

```
clear all
close all
data=randn(3000, 1);
subplot(1, 3,1)
hist(data, 9);
set(gca, 'XLim', [-4.5 4.5])
```

If you want to know what the center of the bins are you can ask `hist` to return a list of the number of data points in the bin (`n1`) and the centers of the bins (`x1`) instead of plotting the data. If you want to both know the centers of the bins and plot the histogram then you will need to call `hist` twice.

```
[n1, x1]=hist(data, 9)
```

You can also give `hist` a vector of the bin centers. Here we are simply returning the number of data points that fell within each bin. We already know where the bin centers are, because we defined them. We can then plot the histogram using `bar`.

```
bins=-4:1:4
subplot(1, 3,2)
n1=hist(data, bins);
bar(bins, n1)
set(gca, 'XLim', [-4.5 4.5]);
```

`histc` takes in a vector that contains the upper and lower limits of each bin. It returns the number of data points that fell within each bin. We can plot this again, using bar

```
subplot(1, 3,3)
n2=histc(data, bins);
bar(bins,n2,'histc')
set(gca, 'XLim', [-4.5 4.5])
```

Note that the number of bins is 9. That means we have 9 bins in the second subplot with the bin centers of -4, -3, -2 … 2, 3, 4. In the third subplot we only have 8 bins, The first bin contains values from -4->-3, the second bin contains values from -3->-2, the third contains values from -2->-1 and so on …

## *Plotting in 3d*

*Mesh*

Let's say you have an experiment in which the data is 3 dimensional. Suppose you were looking at how age affects your ability to perform a task as a function of the time of day. So you have two independent variables, the time of day and the age of the subject, and one dependent variable, which is performance. Your data might look like this

| | | **Time of day** | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | **9** | **11** | **1** | **3** | **5** | **7** | **9** |
| **Age of subject** | **17** | 90 | 90 | 80 | 80 | 85 | 80 | 80 |
| | **17.5** | 90 | 80 | 80 | 80 | 90 | 90 | 95 |
| | **18** | 90 | 90 | 90 | 80 | 90 | 85 | 90 |
| | **18.5** | 80 | 90 | 75 | 70 | 80 | 90 | 85 |
| | **19** | 80 | 80 | 80 | 70 | 80 | 100 | 90 |
| | **19** | 85 | 80 | 90 | 75 | 90 | 85 | 85 |
| | **40** | 90 | 85 | 80 | 70 | 80 | 67 | 40 |
| | **45** | 100 | 80 | 80 | 70 | 80 | 58 | 30 |
| | **47** | 80 | 75 | 70 | 65 | 75 | 60 | 36 |
| | **50** | 80 | 80 | 70 | 80 | 70 | 60 | 40 |
| | **60** | 90 | 90 | 90 | 80 | 50 | 40 | 20 |
| | **65** | 90 | 90 | 90 | 70 | 80 | 40 | 30 |
| | **67** | 85 | 90 | 87 | 70 | 80 | 35 | 22 |
| | **68** | 80 | 85 | 80 | 75 | 65 | 45 | 22 |
| | **69** | 95 | 83 | 85 | 85 | 75 | 38 | 25 |

This gives us the following data in Matlab

```
>>age=[17 17.5 18 18.5 19 19 40 45 47 50  …
 60 65 67 68 69 70];

 tod=[9    11     1     3     5     7     9];
>>tod(3:end)=tod(3:end)+12;
```

Note that that last line of code transforms the time variable into the 24 hour clock!

```
>>perf=[90    90    80    80    85    80    80
    90    80    80    80    90    90    95
    90    90    90    80    90    85    90
    80    90    75    70    80    90    85
```

```
    80    80    80    70    80   100    90
    85    80    90    75    90    85    85
    90    85    80    70    80    67    40
   100    80    80    70    80    58    30
    80    75    70    65    75    60    36
    80    80    70    80    70    60    40
    90    90    90    80    50    40    20
    90    90    90    70    80    40    30
    85    90    87    70    80    35    22
    80    85    80    75    65    45    22
    95    83    85    85    75    38    25
    90    80    90    80    80    36    18];
```

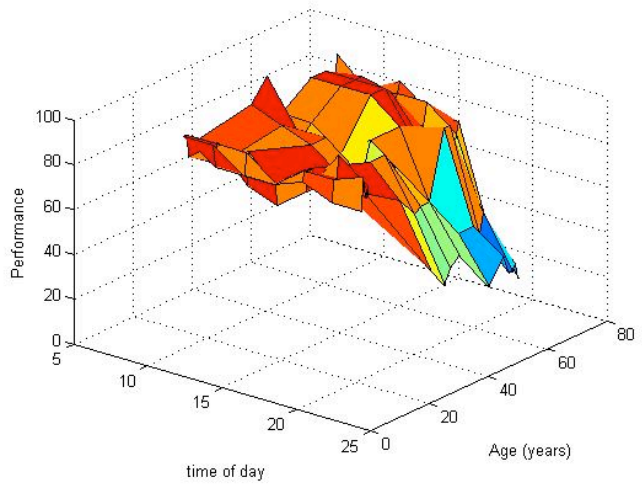
```
  >whos
Name        Size                     Bytes  Class

  age         1x16                     128  double array
  ans         1x16                     128  double array
  perf       16x7                      896  double array
  tod         1x7                       56  double array
```

We are going to use mesh(x,y,Z). The first two vectors must have length(x) = n and length(y) = m where [m,n] = size(Z).

```
>> sh=surf(tod, age, perf)
>> xlabel('time of day')
>> ylabel('Age (years)')
>> zlabel('Performance')
```

You can rotate the view of this three-D graph using the mouse if you hit the little rotation icon on the top of the menu bar. You can now play with various aspects of the figure




Try

```
>> shading flat
>> shading interp
>> colormap(gray)
```

A useful command is `view`. If you rotate the graph to a good view, you might want to save the values that represent that view in a matrix `v` (a 4x4 matrix) so you don't have to manipulate it by hand the next time.

```
>>v=view
```

You can set the view to the desired view which you previously saved by using

```
>view(v)
```

Try saving a view, then rotate the graph manually, then use view to restore the graph to the view that you saved/

Or you can provide the graph with the azimuth an elevation you desire (the azimuth and elevation are basically simpler forms of the 4x4 matrix describing the view contained in `v`)

```
>view(60, 50)
```

There are also three default views

```
>view(1)
>view(2)
>view(3)
```

Also try

```
> mesh(tod, age, perf)
```

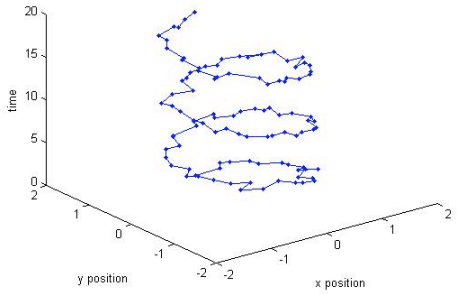
Surf and mesh can deal with missing data values as long as they are set to be `NaN`. It will simply interpolate between the missing values.

```
> perf(3,1)=NaN; > perf(5,3)=NaN; perf(7,2)=NaN;
> sh=surf(tod, age, perf)
```

## *Plot3*

This is often useful when you have one independent and two dependent variables. One example would be if you were tracking the x and y position of a rat over time.

```
> t = linspace(0,6*pi,101);
> x = sin(t)+.1*randn(size(t));
> y = cos(t)+.1*randn(size(t));
> plot3(x, y, t, '.-');
> xlabel('x position');
> ylabel('y position');
> zlabel('time')
```




This rat is a little confused.

Let's suppose you were interested in seeing whether the rat reached a certain location (the food or the swim platform?) within a certain time window. First let's plot the position of this target on the time graph.

```
1       %plotExample2.m
2       %
3       %Example of plotting 3 dimensional data and locating a
4       certain area in the
```

```
%plot
%
%Plots the trajectory of a random process through space-time
(space 2d,
%time 1d).
%
%Written by IF (?) and commented by ASR 07/2007
%

close all
t= linspace(0,6*pi,101);
x= sin(t)+.1*randn(size(t));
y = cos(t)+.1*randn(size(t));
plot3(x, y, t, '.-');
xlabel('x position');
ylabel('y position');
zlabel('time');

twin=[10 17];
xwin=[ 1 2];
ywin=[-.5 1.5];

%line commands make a box in red around the time of interest
and location
%of interest:
lh=line([xwin(1) xwin(2)],[ywin(1) ywin(1)], [twin(xwin(1))
twin(xwin(1))]);
set(lh, 'Color', 'r')
lh=line([xwin(1) xwin(2)],[ ywin(1) ywin(1)], [twin(2)
twin(2)]);
set(lh, 'Color', 'r')
lh=line([xwin(1) xwin(2)],[ywin(2) ywin(2)], [twin(xwin(1))
twin(xwin(1))]);
set(lh, 'Color', 'r')
lh=line([xwin(1) xwin(2)],[ ywin(2) ywin(2)], [twin(2)
twin(2)]);
set(lh, 'Color', 'r')

lh=line([xwin(1) xwin(1)],[ywin(1) ywin(2)], [twin(xwin(1))
twin(xwin(1))]);
set(lh, 'Color', 'r')
lh=line([xwin(1) xwin(1)],[ ywin(1) ywin(2)], [twin(2)
twin(2)]);
set(lh, 'Color', 'r')
lh=line([xwin(2) xwin(2)],[ywin(1) ywin(2)], [twin(xwin(1))
twin(xwin(1))]);
set(lh, 'Color', 'r')
lh=line([xwin(2) xwin(2)],[ ywin(1) ywin(2)], [twin(2)
twin(2)]);
set(lh, 'Color', 'r')


lh=line([xwin(1) xwin(1)],[ywin(1) ywin(1)], [twin(xwin(1))
twin(xwin(2))]);
set(lh, 'Color', 'r')
lh=line([xwin(1) xwin(1)],[ ywin(2) ywin(2)], [twin(1)
twin(2)]);
set(lh, 'Color', 'r')
```

```
62      lh=line([xwin(2) xwin(2)],[ywin(1) ywin(1)], [twin(xwin(1))
63      twin(xwin(2))]);
64      set(lh, 'Color', 'r')
65      lh=line([xwin(2) xwin(2)],[ ywin(2) ywin(2)], [twin(1)
66      twin(2)]);
        set(lh, 'Color', 'r')
```

This is clearly a little “wordy”. Here’s a way to shorten it up.

```
1       %plotExample3.m
2       %
3       %Same example as plotExample2.m, but shorter and less
4       "wordy".
5       %
6       %Made by IF (?) and commented by ASR 07/2007
7
8
9       close all
10      t= linspace(0,6*pi,101);
11      x= sin(t)+.1*randn(size(t));
12      y = cos(t)+.1*randn(size(t));
13      plot3(x, y, t, '.-');
14      xlabel('x position');
15      ylabel('y position');
16      zlabel('time');
17
18      twin=[10 17];
19      xwin=[0.7 1.7];
20      ywin=[-.5 1.5];
21
22      %'mat' represents the facets of the area of interest
23      mat=[1 2   1 1   1 1; ...
24           1 2   1 1   2 2; ...
25           1 2   2 2   1 1 ; ...
26           1 2   2 2   2  2; ...
27           1 1   1 2   1 1; ...
28           1 1   1 2   2 2; ...
29           2 2   1 2   1 1 ; ...
30           2 2   1 2   2  2; ...
31           1 1   1 1   1 2; ...
32           1 1   2 2   1 2; ...
33           2 2   1 1   1 2 ; ...
34           2 2   2 2   1  2];
35
36       %The lines are drawn with a 'for' loop:
37
38       for i=1:size(mat, 1)
39             lh=line([xwin(mat(i, 1)) xwin(mat(i, 2))], ...
40                 [ywin(mat(i, 3)) ywin(mat(i, 4))], ...
41                 [twin(mat(i, 5)) twin(mat(i, 6))]);
42           set(lh, 'Color', 'r');
43       end
44
45      ind=find(t>=twin(1) & t<twin(2) & ...
46          x>=xwin(1) & x<xwin(2) & ...
```

```
47          y>=ywin(1) & y<ywin(2));
48
49      hold on
50      plot3(x(ind), y(ind), t(ind), 'g.-');
```

***Lines 45-50.*** We've added some line finding out whether the rat made it into the box at the specified time, and we've re-plotted those points as green