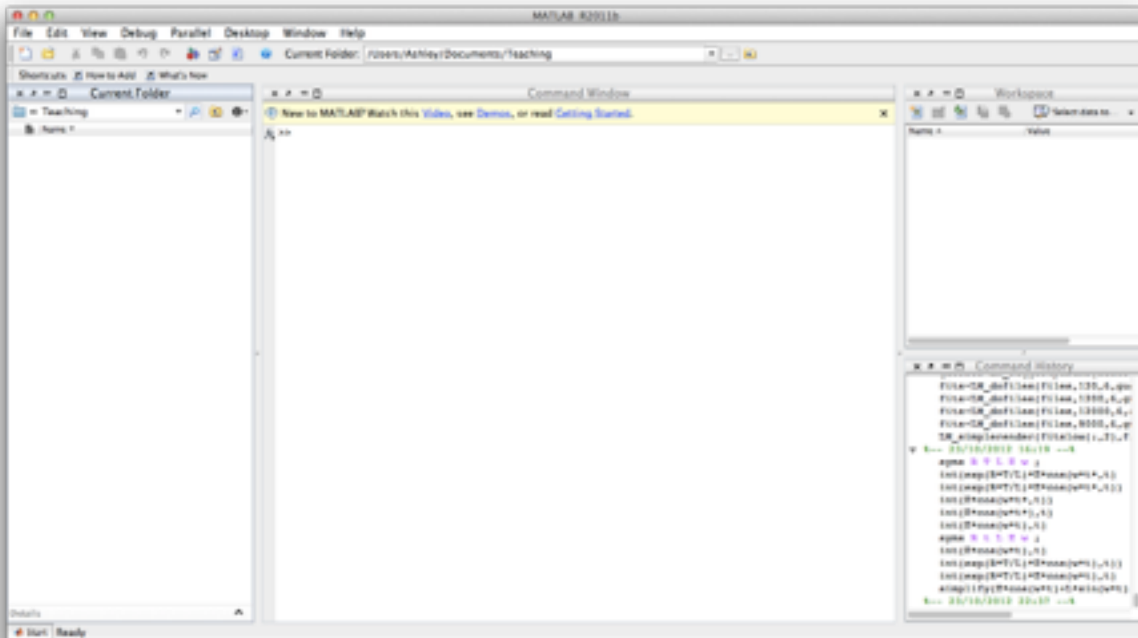
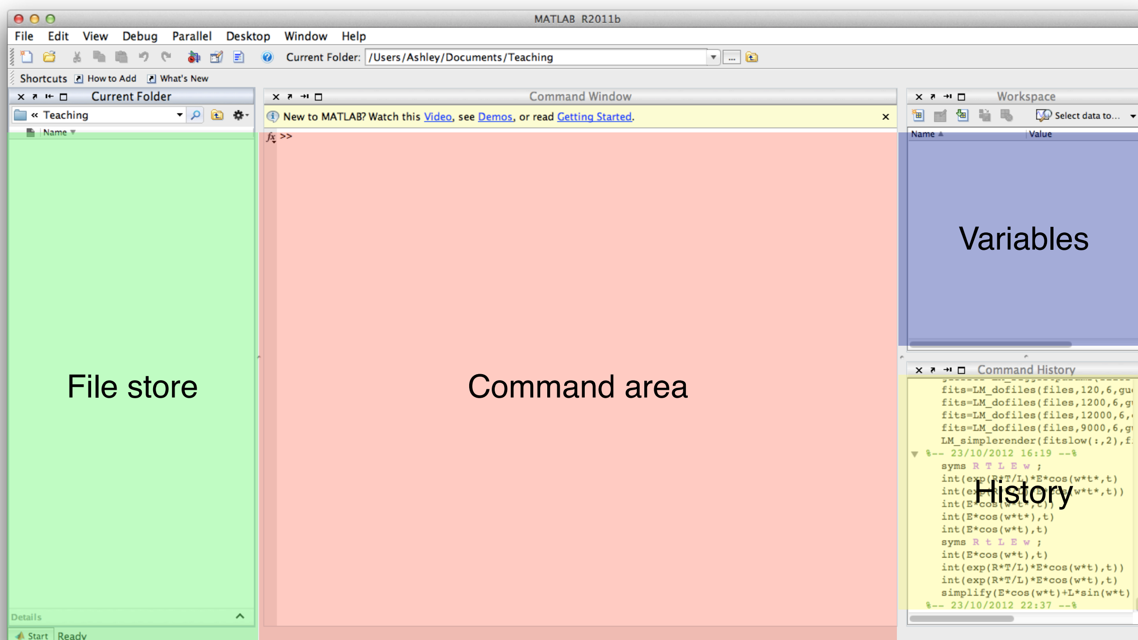
Matlab is a computer program designed to aid scientists with mathematical procedures. In this laboratory class we are going to use Matlab to solve some simple problems.





The main window of Matlab is broken up in to several section; 1.) The command area, this allows us to type commands directly in to Matlab. 2.) The history window shows us the commands we have previously used. 3.) The variables window shows us all the variables and data we have stored in the computer's memory. 4.) The file store shows us the current directory we are working on and any files that we have saved into that directory.

To allow us to get a feel for Matlab we are going to start by typing some commands directly in to the command area.

A very simple example would be to type **1+1** and press return.
The window will look like this

```
>> 1+1
ans =
    2
>>
```

Many equations, simple and complex, can be entered in to command line.
Try these
1. Cos(3.14)
2. 6/4
3. 3+3

Computers work in a very similar way to a post office, the computer stores numbers in little pigeon holes with in the computers memory. You will notice that when you press return the next line on the command window shows

```
ans =
    6
>>
```

Here the computer stores the answer to our last question in a pigeon hole called **ans**. We can make any number of pigeon holes with nearly any name to store information. Each one of these pigeon holes, is a memory address with in the computers memory.

Try this

A=3
B=2

This will create two variables **A** and **B** which will appear in the variables window. The variable A being set to 3 and B being set to 2.

We can now type

A*B   this will give us the answer 6.

A/B   this will give us 1.5

A+B   this will give us 5

A-B   this will give us 1

That's all very well if we want to deal with single numbers, but often we want to deal with many numbers, such as collected data. Matlab gets its name from being able to deal with matrices and actually it stores all the data you will ever give it in a matrix, in fact the variables **A** and **B** are just 1x1 matrices.

Type magic(3)
and you should get

```
8   1   6
3   5   7
4   9   2
```

Matlab has created a 3x3 matrix where each row column and diagonal sum to 15. Hence the name of the function "magic". You could type magic(4) and get

```
16    2    3    13
5     11   10   8
9     7    6    12
4     14   15   1
```

here each column and row sums to 34. You can see that magic(3) gives you a 3 x 3 matrix, and magic(4) gives you a 4 x 4 matrix. We can store one of these magic squares in a set of memory addresses.

A=magic(3)
now A contains the magic matrix.

```
8   1   6
3   5   7
4   9   2
```

We can address each matrix element individually.
Type A(1,1) and the answer returned will be 8, here we have addressed the 1st row and 1st column.

```
>> A(1,1)
ans =
    8
```

So now we can address each cell of the matrix with the syntax **A(y,x)** where y is the row and x is the column. If we wanted to list a complete row or column we can type **A(:,3)** to get the third column. Here the ":" tells Matlab to return either a whole column or row.

```
>> A(:,3)
ans =
    6
    7
    2
>>
```

the third row is

```
>> A(3,:)
ans =
    4    9    2
>>
```

We can transpose the matrix by appending and apostrophe to the command, try this.

```
>> A
ans =
    8    1    6
    3    5    7
    4    9    2
```

```
>> A'
ans =
    8    3    4
    1    5    9
    6    7    2
```

Break point 1

This is what we can do so far;

1.) Simple maths in the command window.
2.) We can assign numbers to simple variables
3.) We can address any cell using A(y,x)
4.) We can transpose any matrix.

There are many commands we can use to evaluate our magic matrix.
Try these

sum(A)
mean(A)

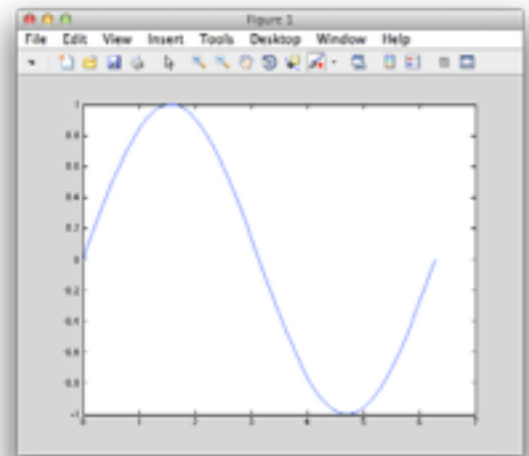
Did you get the answer you expected?
These commands always work on the columns, if we wanted the sum or mean of the rows we would have to type sum(A') and mean(A') this transposes the matrix first. For a magic matrix sum(A') is the same as sum(A) because all rows and columns add up to the same value. To sum all the numbers in a matrix we can use the command **sum(sum(A)).** This sums the sum of all the columns.

To find out the size of a matrix you can use **size(A)**.

Finally we need to know how to select sections of a matrix.
Lets make a much larger matrix using **A=magic(10)**. We can select part of the matrix using the command **A(y_1:y_2,x_1:x_2)**. This will return a section of A from y_1,x_1 to y_2,x_2 as shown in the figure to the right, to select the area in the figure you would type **A(1:5,1:5)**.

You can can also use some built in commands to define sections of a matrix, such as "end" so **A(y1:end,x1:end)**. Here the end is the last column or row you can say "end-1" if you want the last but one.

Selected area of a matrix A(1:5,1:5)



Lets look at the graphing ability of Matlab
Type in

X=linspace(0,2*pi(),100);

This command creates 100 data points from 0 to 2 pi.
The results are stored in the Variable **X**. Note that the ';' stops Matlab outputting to the screen, if you want try the above command with out the';'.

Now we can use this as our X axis,
Type in;

Y1=sin(X);
This command now fills the new variable **Y1**, with the sin of X, note the newly formed variable in the variable window.
Now we have X and Y, we can plot them using the command.

plot(X,Y1);

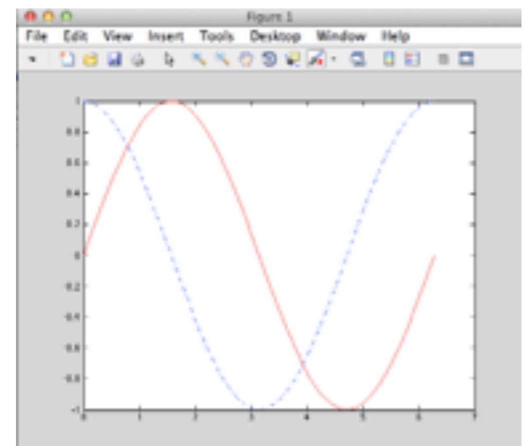
We can change the plot by adding a third option to the plot command such as **plot(X,Y1,'*')** or **plot(X,Y1,'r'),** or **plot(X,Y1,'r-.').** Right Click on the plot command to get extra information on how to format the plot command.

If we wanted to plot more than one data set at a time we can define a second data set such as Y2, using the command,
Y2=cos(X);

Now plot the two data sets by typing

plot(X,Y1,'r',X,Y2,'-.');
We can alter our display with the commands;

>> hold on;	**This hold the current display**
>> axis([0 3.14 -1 1])	**This sets the axis to [xmin xmax ymin ymax]**
>> hold off;	**This releases the current display**



At this point its useful to point out that if we wanted to plot a function that was a little more complex, such as tan(x)*cos(x), we could not just type **tan(X)*cos(X)** because mat lab would try to perform a matrix multiplication on this and would get very confused, we have to tell Matlab that it's dealing with a scalar set of numbers we do this by typing

Y3=tan(X).*cos(X); Notice the "." before the "*".

This tells Matlab to perform a scalar multiplication. Try it.

You can use Matlab to produce plots suitable for reports using some of the commands listed below

Presentation, use the following commands to turn your plot into one suitable for presentations.

```
title('Trigonometric Functions', 'fontsize', 10);
```

title adds a title to the plot.

```
xlabel('angle');
ylabel('sin(x) and cos(x)');
```

The x and Y label commands allow you to label the X and Y axis.

Finally the legend command can be used to add a legend to your data. The command

```
legend('y','z','w')
```

Adds a legend to a graph with three plots, naming them X, Y and Z.

It is possible to produce a huge range of different plots in Matlab they are highlighted on the following webpage,

http://www.mathworks.co.uk/discovery/gallery.html

Your Turn

Plot the functions;
Y = 3*X+ 2,
Y = 3*X^2+2,
Y = 3*X^3+2;

Plot the three functions with different colours, line styles and suitable symbols.

Make sure your plot has a title, a legend and the axis are labelled.

Note that X^2 should be written as X.^2 (as Matlab will again try to use X as a matrix rather than a scalar).

Once you have done this show it to a lab demonstrator.

A little break (or the cheat section). Break point 2.

Matlab is more an environment to perform maths rather than just a simple program for plotting data. In this environment there are many different plugins which can be used to help scientist, these include data analysis and experimental control plugins. One of the most useful is the symbolic maths toolkit. Here we are going to explore the application by solving some simple symbolic maths.

Type

syms A B t; This define three symbolic variable, these are variables that do not store numbers but allow Matlab to manipulate in a symbolic way, all will become clear.

Type

simplify((A^2/B)+(B^2/A))

and Matlab will give you

(A^3 + B^3)/(A*B)

Here Matlab has simplified the equation for you, Matlab can do so much more, make sure you miss off the ';' or Matlab will not give you an answer.

diff(A*sin(B*t),t)

Here Matlab has differentiated the function with respect to t, the **,t** tells Matlab what to differentiate with respect to

A*B*cos(B*t)

It should be noted that you can only use symbolic variables you have defined using the **syms** command.

Integration can be done using

int(cos(A*t)*sin(A*t),t)

Scripts and Functions.

It is a little time consuming to type commands in to the command line each time. Matlab is a fully functioning language and so for the rest of the laboratory script we will concentrate on writing actual computer code. The computer code we will write will be in two slightly different formats one is called a script and one is called a function.

The script
A Matlab script is simply a set of commands which are performed by Matlab and can access the variables currently in memory.

The Function
A Matlab function can be called from within a script or another function but it has to be supplied with all the information that it requires to perform its task, it can not access the variables in memory with out explicitly being passed them.

Our first script.
We first need to create a script to work within, use the menus to create a script.

File-> New-> scripts
Type the following and then save the script using

```
% This is my first script
```

File-> Save
You can save the script with any name you want, but it should not contain any spaces, “first” is a good name. The file should appear in the file store window. In the command window type the name of the file you have just saved (leaving off the .m). Nothing should happen because the % character in the `% This is my first script` line sets the rest of that line as a comment for humans and is ignored by Matlab.

Double click on file you have just saved in the file store window, it should open and you can edit the script. Type the following lines in to the script you have just opened.

```
% This is my first script
% Define some variable

X=linspace(0,2*pi(),1000);
```

Save the file.
In the command window type

clear and hit return

This clears the variables in memory, so that all the variables you have defined previously are now gone.

Then type the name of the script. You will see that the variable X appears in the variables window. We have just created a script that performs exactly the same function as if we had typed the command in to the command window. OK so that’s not hugely impressive lets try something a little different which can only be done by a script.

We are going to write a little computer program to work out if the diagonal of our magic matrix has the same sum as that of the columns (and therefore the rows).
The script is given below, start a new script type out the following and save it, you don't have to type out the comments. The name **magicsum** is a good name.

```
% we are going to need to store some number
% So we have two variables, temp and value
temp=0;
value=0;
%Here we defined two variables and set them to
%values of 0


%The next thing we need to do is make our magic
%matrix

X=magic(6);
%We have stored our magic matrix in the
%variable X

S=sum(X);
[width height] = size(X);

%We can learn a little about our matrix
%Firstly we use the sum command to get
%the values of the columns
%The second command is a little more interesting
%It tells us the width and height of the matrix
%but it kindly stores them in two different variables
%making life a little easier


%So the next for lines of code are the reason to use
%a script. This is a for loop and it will be dealt with
%in the lab text but basically this for loop
%performs the same function a set number of time
%and allows you to change variables with in the
%loop


for i = 1:width
    temp=X(i,i);
    value=value+temp;
end

%This loops performs the same number of loops as
%there are columns or row (they are the same) in the
%matrix and each time it does it the variable i
%increases by 1.  So that the first time the loop runs
%i has a value of 1, the second time i has a value of
%2.  The loop finishes on the loop where i has the same
%value as width.

%on each loop the variable temp is set to the value of
%the matrix cell (i,i) which is on the diagonal
%The value of temp is then added to the variable "value"
```

type **clear** and hit return in the command window and then the name of the script you have just saved. When you do that you should see the variables window become populated with all the variables you have defined in the script.

Lets have a look at them.
Type
X
This will show you the magic matrix
Type
S
This will show you the value of the sum of the columns
Type
value
This will show you the value of the sum of the diagonal
Are **S** and **value** the same? They should be.

Functions
We are going to turn the above script in to a function to see the difference between the two. The simple way to do this is to open the script and create a function and copy and past the text from the script in to the function. To do this;

1.) Double click on the script in the file store window.
2.) select **Edit->select all** from the menu.
3.) select **copy** from the **edit** menu.
4.)Select **File->New->Function** to create a new Function. You will se that it already has some text in it.
5.)Click on the line below `%   Detailed explanation goes here`
6.)And select Edit->Paste from the edit menu.
7.)I have removed the comments to make the function a little more readable it should look like this.

```
function [ output_args ] = untitled4( input_args )
%UNTITLED4 Summary of this function goes here
%   Detailed explanation goes here

temp=0;
value=0;

X=magic(6);

S=sum(X);
[width height] = size(X);

for i = 1:width
    temp=X(i,i);
    value=value+temp;
end

end
```

There are a couple of things to point out here, the first is that the function already had the line

function [output_args] = untitled4(input_args)

when we created it, it also had an **end** on the last line.

This first line defines the function and we are going to have to fill it in before we can proceed. Change this **function** line from

```
function [ output_args ] = untitled4( input_args )
```

to

```
function [ value S] = Magicfun( number )
```

What we have done here is called the function Magicfun, when we save the file in a moment we will have to call it Magicfun. We have also told Matlab that when we use this function we will give it an input called “number”. It will return two outputs “value” and “S”.

We need to make one small change to the code and that is the line;
`X=magic(6);` needs to be **`X=magic(number);`**

This simple change means that when we send the “number” input, the function will use that to create a magic square of any size we want.

I have also added the line;

```
S=S(1);
```

This is just to save space, as the variable S will be the same size as the input “number” we could end up returning a large array full of the same number (as all the columns have the same sum), so to save having to move around all that data we are just going to return the first number from the array. If you don’t understand this line just ask one of the lab demonstrators.

```
function [ value S] = Magicfun( number )
%UNTITLED4 Summary of this function goes here
%   Detailed explanation goes here

temp=0;
value=0;

X=magic(number);

S=sum(X);
S=S(1);
[width height] = size(X);

for i = 1:width
    temp=X(i,i);
    value=value+temp;
end

end
```

Ok, save the file. Remember it has to be called **Magicfun**.
lets try using this function.
type
clear
To clear the variables.
Then type
[A B] = Magicfun(5)

The function returns two variables A and B both have the number 65 in them. You can see here that we can call the returned variables anything we want and none of the variables that the function defines such as temp, S or value are saved in the variables window. This means we do not have to worry about clearing up variables or overwriting variables that we want to save.

We are now going to use a function with in a script to perform a task repeatedly. We are going to see if the column sum and the diagonal sum are the same for all magic squares.

```
start=1;
stop=100;
store = [0 0 0];


for i = start:stop;
    [A B]=Magicfun(i);
    store(i,1)=A;
    store(i,2)=B;
    store(i,3)=i;
end
plot(store(:,1),store(:,2),'.');
```

Write the script out below.
Lets look at this code, its pretty simple
```
start=1;
stop=100;
store = [0 0 0];
```
Sets ups three variables start, stop and store. Start and stop are the limits of our loop and store is a three value array, which as you may guess from its name we are going to store things in.
Next up is a for loop

```
for i = start:stop;
    [A B]=Magicfun(i);
    store(i,1)=A;
    store(i,2)=B;
    store(i,3)=i;
end
```

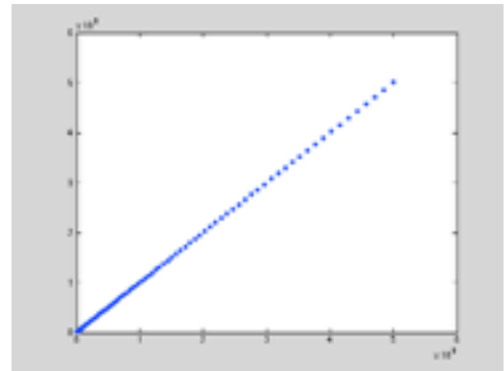
Our for loop calls **Magicfun** with the value (i) each time the loop is executed and returns the values A and B. We use the array **store** to hold our data, with column 1 holding the value of **A**, column 2 holding **B** and column three holding **i**. We can see that each time the loop is executed the row where the data is stored in incremented by one.

So for the second execution store(i,1)=A is actually store(2,1)=A. We do this for all values of i.

Finally the line

```
plot(store(:,1),store(:,2),'.');
```

Plots the values of A and B, in a pretty graph.



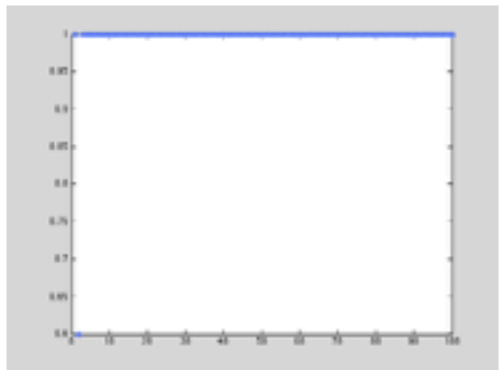
So create a new script, type out the above code, save it and run it. You should get the following which seems to indicate that (as the graph is a straight line) the diagonals and the column sums are the same. But, it is a large graph, lets re-plot the data a little.

Type this in to the command window

ratio=store(:,1)./store(:,2);

Note the full stop before the division(/) command is to force a scalar division not a matrix division and then

plot(store(:,3),ratio,'*')

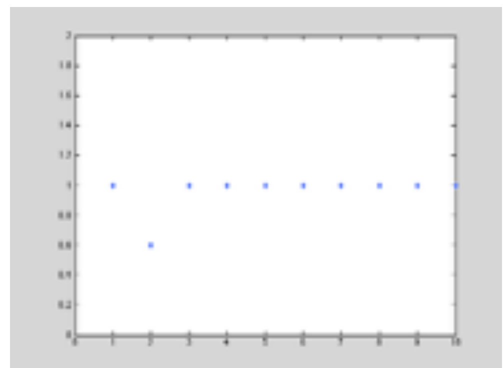


We have now plotted the ratio of A/B against the Magic square size. Here we can see that all the ratios are unity except for a single point at the beginning of the data set.

Type

hold on;
axis([0 10 0 2])
hold off;

```
>> hold on;
>> axis([0 10 0 2])
>> hold off;
```



We now have a graph that look like the lower plot. We can clearly see that for a value of two the diagonal and column sum is not the same.

We can test this by typing

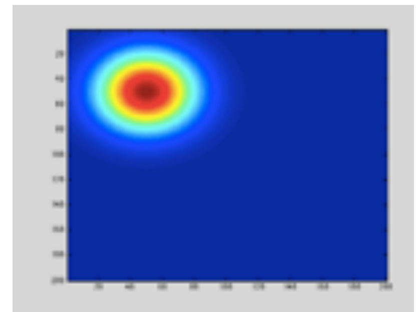
[A B] = Magicfun(2)

and we can see that this is indeed the case.

Some computing basics
In this section we are going to look at some very simple scripts to explore some basic programming. Each section will give you an example of a computing technique. Type the code out and have a play with it, you can use either a script or a function.

The For loop.
The for loop allows you to perform the same function multiple times with different variables.

A prettier example
In these examples the commands between the **for** and the **end** are carried out **k** times and the value of the variable **n** or **m** is increased by 1 each time the loop is executed. Once you have plotted you Gaussian in 2D try it in 3D using the command **"surf(Gaussian)"**. The loop does not have to go from 1 to X with simple integer steps, the examples below show that you can use any numbers you want



```
k = 200;
Amplitude = 10;      %The amplitude of the gaussian
centreX   =  50;     %The X centre
centreY   =  50;     %The Y centre
widthX    =  30;     %the width in X
widthY    =  30;     %the width in Y


Gaussian = zeros(k,k);

for m = 1:k
    for n = 1:k
        Gaussian(m,n) = Amplitude* ((exp(-1*(((centreX-m)^2))/
(widthX^2)))*(exp(-1*(((centreY-n)^2))/(widthY^2))));  %This is all on one line
    end
end

imagesc(Gaussian);
```

For loops are very powerful and don't have to be used with integers or with fixed increments. The following examples show how for loops can be used, type them out.

Steps of increments of -0.1, and display the step values:

```
for s = 1.0: -0.1: 0.0
  disp(s)
end
```

Execute statements for a defined set of index values:

```
for s = [1,5,8,17]
  disp(s)
end
```

While loops

Very similar to a for loop is a while loop. A for loop will loop an exact number of times before finishing, where as a while loop will stop at a set condition. Look at the code below. This little program calculate 1/n! for all the values of n until the value of 1/n! is less that a given value. The while statement here performs the loop to calculate the value of 1/n! and will continue to perform that task for ever or until the result satisfies the break condition **`result > 0.000000001`** that is defined at the start of the loop. If we had missed out the **n = n+1** command then this loop would have gone on indefinitely because the value of n would never have changed and so the result would always be 1 which is greater than the condition needed for the loop to finish. The while loop is a great way of searching through

```
result =1;
n=1;
while result > 0.000000001

result = 1/factorial(n);
disp(result);
disp(n);
n = n+1;
end
```

data for an event.

Conditional statements

One of the last types of programming structure we will look at is the conditional statement. A conditional statement takes the form;

If this then do this, else do that.

Look at the code below.

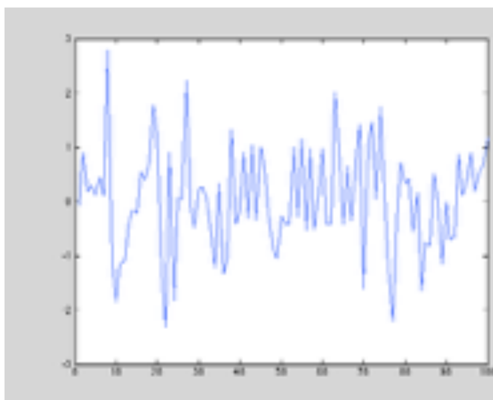
```
function [ X ] = Bubblesort( X )

[s k]= size(X);
count =1;

while count > 0

    for n =1:k-1

        if X(n+1) > X(n)
            Tempc =X(n);
            X(n)=X(n+1);
            X(n+1)=Tempc;
            count = count+k;
        else
            count = count -1;
        end

    end
end


end
```

It is a function that takes an array **X** and passes back the same array **X**. This is a very simple sort algorithm called a bubble sort. This bubble sort has three distinct features.

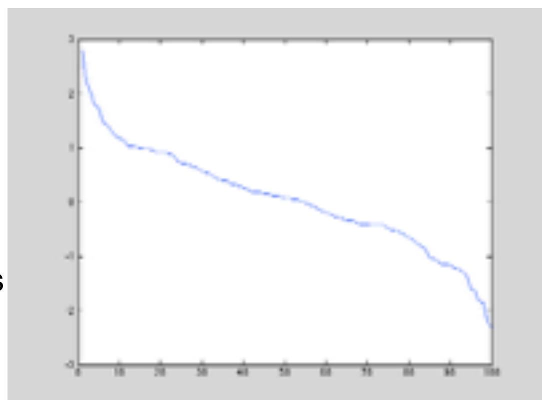
1.)The first and most boring, is the setting up of the variables **K** and **count**. **K** is the size of the array passed to the function, it is used so that we can scan through an array item by item for any sized array we are given. The s variable would be the number of columns, we never use this. However, if you want to you can change the code so that it sort a data set with more than one column, it is quite easy. The count variable here is set to 1 so that the first time the while loop runs it has a value which fits the criteria to execute the loop, we will look more at the while loop later.

2.) **The while loops:** This loop runs until the value of count is negative. You will notice that the loop any runs from 1 to the length -1 **(K-1)**. Can you see why?

3.)**The if condition:** This part of the code does all the work, it takes two consecutive array items **n** and **n+1** and compares them. If the **n+1** item has a greater value than the **n** item, then it swaps them around and the value that was in the cell (n) becomes the value in the cell (n+1) and the value in the cell (n+1) takes the value of cell (n). This means that once this has been performed a lot of times all the data in the array is ordered going from high



to low. We use the while loop because although we could run through the array k times which is the maximum number of



times we could possible need to, the while loop allows us to stop when no swaps have been made. We do this by setting the count variable high when we have to perform a swap, when we don't perform a swap, the **else** condition, we then subtract one from count. On the loop when nothing is swapped the value of count will be negative by the number of cells in the array, one single swap will mean that the value of count is positive.

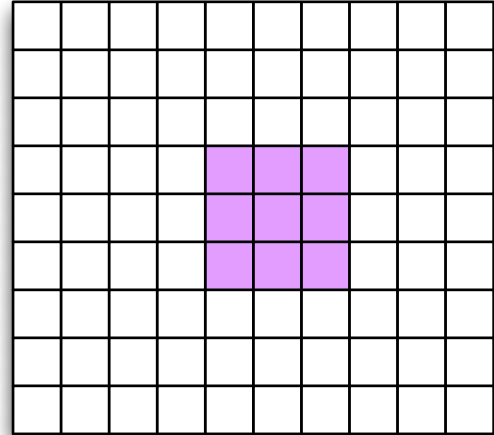
Lets have a play with this type in the code as a function and save it as Bubblesort.
The type
X=random('Normal',0,1,1,100);
This will generate a set of random numbers, if you plot them with **plot(X);** you will get, the first plot shown on the left.

Now type
X1=Bubblesort(X);
plot(X1);

You can see that the data is now sorted as shown in the right hand side plot. That's it for now, next time we will look at some more complicated programming and some fitting procedures.

Now its your turn.

1.) Write a function that takes three integers A,B and C. The function should return the 9 cells centered at the position B,C from the magic square with width and height A. Make sure your code can cope with the conditions of B or C =1 and B or C = A.
I.e The edges of the magic square.



2.) Write a script to calculate all the prime numbers between 1 and 100. It might be useful to know that the command **mod(X,Y)** returns the remainder of the division of X by Y.

3.) A particle can move either up, down, left or right once each time step. Calculate the average distance a particle moves in 1000 step.
The command "`random('bino',1,.5);`" will return the numbers 1 or 0 with an equal probability of either.
The command "X=random('unid',n)" will produce either a discrete distribution up to n.
The distance the particle moves after 1000 steps is =sqrt (X^2+Y^2). Where X is the left right distance and Y is the up down distance. Note that you will have to perform this calculation for many different particles to get a decent distribution.

4.) Newton's method: The Newton method is used to find the roots of a function. The idea is as follows: one starts with an initial guess which is reasonably close to the true root, then the function is approximated by its tangent line, and one computes the x-intercept of this tangent line. This x-intercept will typically be a better approximation to the function's root than the original guess, and the method can be iterated.

The tangent line is calculated using the derivative of the function. For f(x), the expression for a linear function representing the tangent line at the point x_n is given by

$$y(x) = xf'(x_n) + f(x_n) - x_n f'(x_n)$$

here f'(x) is the derivative of f(x).

Thus the intercept of this function (i.e. the next iteration) is calculated from

$$y(x) = 0$$ as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

When the difference between the intercept and the value of f(x) becomes small enough you have found the root of the function. Try to obtain the roots of $f(x)=x^2-3x-2$, for an accuracy of better than 10^{-12}