Note that it is sometimes useful to add comments to your commands. You can do this with "%":

```
>> data=[3 5 9 6] %here is my comment

data =

     3     5     9     6
```

At any time you can see the variables in the workspace and their properties including dimensions etc:

```
>> whos
  Name      Size             Bytes  Class     Attributes

  data         1x4                32  double
```

You can save this array to a delimited text file named e.g. "data.dat"

```
>> save -ascii data.dat data
```

You can also get information about a specific variable:

```
>> length(a)

ans =

     4
```

You can concatenate these elements to other arrays:

```
>> data=[[1 5 3 8]' data']

data =

     1     3
     5     5
     3     9
     8     6
```

Note that these are all logical statements, not algebraic. They can therefore be self-referential.

Since this is a 2-dimensional array (4 rows and 2 columns), we can see the length is

```
>> length(data)

ans =

     4
```

but the number of elements is

```
>> numel(data)

ans =

     8
```

And the size reports the length of the rows and columns:

```
>> size(data)

ans =
```

```
     4     2
```

(BTW, there exists documentation on all the functions available in MATLAB; for instance
http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/refbook.pdf,
which is just for functions starting with A-E and takes up >1500 pages. The other two volumes for the rest of the alphabet are refbook2.pdf and refbook3.pdf. Rather than burden you with sorting through them, we will focus on the functions of greatest utility.)

You can address individual elements of this array (row index first, column index second):

```
>> data(1,1)

ans =

     1

>> data(4,1)

ans =

     8
```

You can also use this address syntax to assign values:

```
>> data(2,1)=18

data =

     1     3
    18     5
     3     9
     8     6
```

The colon ":" is extremely useful in Matlab. You can use it to define subsets of elements of arrays:

```
>> data(1:2,1:2)

ans =

     1     3
     5     5
```

Or, use it to create arrays of evenly spaced numbers. For unit spacing:

```
>> 1:4

ans =

     1     2     3     4
```

Or, for other spacing values e.g.

```
>> 0:5:15

ans =

     0     5    10    15
```

Note that you can also use linspace to generate equally-spaced numbers. For example, you could produce the same output as above by finding the 4 equally spaced numbers between 0 and 15 (inclusive):

```
>> linspace(0,15,4)

ans =

     0     5    10    15
```

To define simple arrays, you can use these two self-explanatory commands:

```
>> ff=zeros(2,1)

ff =

     0
     0

>> ff=ones(2,1)

ff =

     1
     1
```

To create an array of random values equally distributed between 0 and 1:

```
>> ff=rand(2,1)

ff =

    0.8147
    0.9058
```

Because these are random numbers, you will get different results everytime you execute the command. [Note that `rand`, by itself, is equivalent to `rand(1,1)`]

If you want to erase all variables from the workspace use "`clear`",

```
>> clear
```

Be aware that one of the most useful MATLAB commands is "`help`", e.g.

```
>> help mean
 MEAN   Average or mean value.
    For vectors, MEAN(X) is the mean value of the elements in X. For
    matrices, MEAN(X) is a row vector containing the mean value of
    each column.  For N-Dimensional arrays, MEAN(X) is the mean value of
    the elements along the first non-singleton dimension of X.

    MEAN(X,DIM) takes the mean along the dimension DIM of X.

    Example: If X = [0 1 2
                     3 4 5]

    then mean(X,1) is [1.5 2.5 3.5] and mean(X,2) is [1
                                                       4]

    Class support for input X:
       float: double, single

    See also median, std, min, max, var, cov, mode.

    Overloaded functions or methods (ones with the same name in other
directories)
       help timeseries/mean.m
```

```
    Reference page in Help browser
       doc mean
```

You can not only find out how to use a function, but also discover other related functions that might be helpful as well in the "`See also`" section above.

ARITHMETIC

Matrix operations are overloaded onto the scalar operations, for example: a row vector times a column vector is a scalar

```
>> size(rand(1,2)*rand(2,1))

ans =

     1     1
```

But a column vector times a row vector is a matrix

```
>> size(rand(2,1)*rand(1,2))

ans =

     2     2
```

However, it is often useful to operate on an array element-by-element. For instance, if you want to square every element, put a "." before the operator:

```
>> (1:3).^2

ans =

     1     4     9
```

Note that two row vectors cannot be multiplied together so this gives an error without the "."!

PLOTTING

For instance,

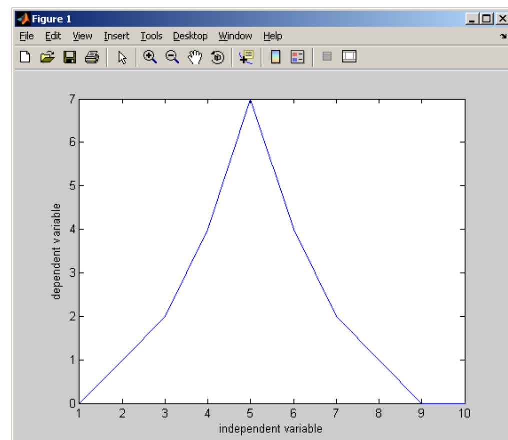
```
>> x=1:10; y=[0 1 2 4 7 4 2 1 0 0];
>> plot(x,y)
```

And label the axes:

```
>> xlabel('independent variable')
>> ylabel('dependent variable')
```

We can even print this to an image file with one of:

```
>> print('-dbmp','fig1.bmp')
>> print('-dpdf','fig1.pdf')
>> print('-djpeg','fig1.jpg')
```



etc., where the first argument specifies the file type and the second defines the filename of the resulting image.

SCRIPT AND FUNCTION FILES

You can eliminate a lot of typing at the command prompt by saving commands to a script – essentially a text file that contains the commands, but is always named with a ".m" extension. For example, consider a text file "script1.m" which contains the lines

```
clear;
N=1e2;
A=rand(1,N);
M=mean(A);
disp(M)
```

[Notice that it is very often a good idea to start every script with "clear" to avoid variable name ambiguities and errors. Also, notice that I used the exponential notation "1e2" to represent "100" in the above and that "`disp()`" displays the value of a variable.]

Now, just type the name of the script (without the ".m") at the command prompt to run it. :

```
>> script1
    0.4990
```

Create the script file above and test it in your command window. (Make sure the .m file is in your current directory!)

If we want to easily interactively see the effect of e.g. larger and larger numbers of random variables without directly editing the value of `N` in our script file, we can make another .m file which is a ***function*** that takes `N` as an argument on the command line. The following is the contents of a file func1.m:

```
function mymean=func1(N)

A=rand(1,N); %generate an array of random variables between 0 and 1
mymean=mean(A); %calculate the mean of the array
```

(Note that the function definition on the first line is the same as the filename without the ".m", and the value it returns is similarly defined on the same line before the "`=`").

Then we can set the value of `N` on the command prompt,

```
>> func1(1e6)

ans =

    0.5003
```

CONTROL STRUCTURES

There are several elements of simple programming that we will use:

Conditionals are also called “if statements”. For instance, the following function prints something conditional on the input:

```
function out=if1(in)

if in==1
    disp(['input is ' num2str(in) '!'])
    out=4;
elseif in==2
    disp(['INPUT IS ' num2str(in) '!'])
    out=9;
else
    disp(['input is not 1. It is ' num2str(in)])
    out=0;
end
```

Note that I made use of “`num2str()`” which does just what its name implies: convert a number to a string. Then, it is concatenated with string constants to create the displayed text. Another useful function like this is “`int2str()`” which only works for integers but will suppress printing out a number to 4 decimal places or more.

We can run it with arbitrary input:

```
>> b=if1(1)
input is 1!

b =

     4

>> b=if1(10)
input is not 1. It is 10

b =

     0
```

Loops are extremely important for programming MATLAB to perform repetition. For instance, if we want MATLAB to do something 10 times,

```
for ii=1:10
 rr(ii)=ii^2;
end

>> rr

rr =

     1     4     9    16    25    36    49    64    81   100
```

The loop sets “ii” equal to successive values of the array 1:10 every time it executes until it has looped through all the values. I use “ii” here so as to not confuse this iteration counter with “i”, which has special meaning in MATLAB (i.e. it is the imaginary number).

Note that this is a particularly BAD way to produce an array of squares because MATLAB is optimized to operate on whole arrays, not by looping through an array’s elements
Q: How can you construct an array of squares with a **one-line command**?

There are other types of loops. For instance,

```
ii=1;
while rand<0.9
 ii=ii+1;
end
disp(ii)
```

The loop will execute the code before the “end” statement and increment “ii” by 1 until the random number generated by “rand” is equal to or more than 0.9. Once it satisfies this constraint, it will display the number of loop iterations.
Q: what is the mean value of ii after this code executes a large number of times? Write a function and script which calls it to show this. Does this make sense? Compare to the analytical result!

Note that it is possible to write (faulty) code for a loop which will never end if the “while” statement is always satisfied. If your code gets into an infinite loop, you can stop it by pressing control-c.

MATRICES

Rather than tell MATLAB what each element of a matrix is individually, it is often useful when constructing matrices to use the diag command:

```
>> diag(ones(2,1),1)

ans =

     0     1     0
     0     0     1
     0     0     0
```

Note the second argument tells MATLAB which diagonal to use, relative to the main diagonal. Positive values go above the main diagonal, and negative integers go below, e.g.

```
>> diag(ones(2,1),-1)

ans =

     0     0     0
     1     0     0
     0     1     0
```

If you don’t specify a second argument, the array will go on the main diagonal itself.

The eig function returns eigenvalues and eigenvectors:

```
>> [v,d]=eig(rand(2))

v =

   -0.9952   -0.9199
    0.0982   -0.3922


d =

    0.5720         0
         0    1.0178
```

Note that the eigenvectors are the columns of `v` and eigenvalues are the diagonal elements of matrix `d`.

Some other useful commands: `eye(N)` makes an identity of rank N