

Introduction to Matlab

Andy Philippides

1. Overview

Matlab is a very powerful tool for generating, manipulating, analysing and presenting data. The object of this course is to enable you to start using Matlab and to show you how and where you can find out more about it.

By the end you will be able to:

- Use the command line to call a function with various inputs and get output from it
- Generate vectors and matrices and manipulate them using colon notation and point-to-point operations
- Use 'for' loops to randomly select vectors from a matrix
- Plot 2 and 3D figures
- Experiment with a model system and see how theory translates to reality

Caveat:

Matlab is a vast open-ended programming language and there are usually several ways to do anything. I will therefore not be able to explain everything thoroughly. Instead I will show examples of the kind of thing Matlab is capable of, point out options that you could take and how you could find the same information and options yourselves.

Given this style, I will explain syntax and the use of various Matlab commands as they come up in the examples, and allow you to experiment with the examples yourselves.

The idea is to work through this sheet, together with the associated m-files that you can edit and run and hopefully re-use. The initial section is straightforward (and probably a little dull) and gives the necessary basics. The subsequent sections deal with vectors and matrices, for, while and if statements and plotting functions. They are more interactive and demonstrate useful code fragments.

I have also included some extra material that you can go through on your own. This is marked as:

ASIDE:

THE FINAL SECTION IS THIS WEEK'S PROBLEM SHEET.

It has a working program that you can experiment with by observing the effects of changing parameters. This is an example of the type of experimentation you will encounter in the problem sheets.

2. Using Matlab from the Command line

You can use Matlab like a powerful calculator which allows you to generate and analyse data quickly and easily, to perform mathematical operations on it and to visualise it.

Before we run through a couple of examples of this, it is necessary to cover some basics on variables. Also, note that doing `help function_name` gives details of how the command `function_name` works and gives any related functions. It is probably the single most useful command in Matlab. Eg to see how `help` works do:

```
>> help help
```

(where you see `>>` type the commands after `>>` at the command line and press enter/return).

2.1 Numerical Matlab Variables 1: Vectors

Numerical data are held in matrices and vectors and are assigned a variable name (anything you want as long as it doesn't start with a number). Try entering the following at the command line:

```
>> a=10
>> 1a=10
```

Entering vectors: A vector is a 1D set of numbers and is signified by **square** brackets. If there is only one number brackets aren't needed. The *elements* (ie numbers) of the vector are separated by spaces/commas eg do:

```
>> x=[1, 2, 3]
>> z=[3 2 1]
```

A useful way of making vectors (especially for plotting) is using the *colon notation*. Try:

```
>> x=2:10
```

Here the colon, `:`, stands for 'to' and we read the above as 'x is a vector going from 2 to 10'. There is also a long form of the colon notation '`a:b:c`' which means 'Start at **a** and go to **c** in increments of **b**' eg try:

```
>> l=5:5:41
>> m=16:-2.5:10
```

You can also put vectors together via eg:

```
>> n=[1 5 m]
```

Try generating a vector that goes from 1 to 5 and then join it to a vector going from 5 to 1.

Extracting data: To access elements of the vector **round** brackets are used, eg to multiply the 3rd element of `l` by 2 do:

```
>> l(3)*2
```

To access multiple elements, use a vector of indices eg to divide the last 4 elements of `l` by 5 do:

```
>> l([5 6 7 8])/5
or: >> l([5:8])/5
or: >> l([5:end])/5
```

Note the use of the colon for consecutive values and the command '`end`' used as a vector (or matrix) index to indicate the final element. Try extracting the 3rd to the 2nd to last element of `l`.

ASIDE: The above are all row vectors. A column vector such as $y = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$ is written as:

```
>> y=[1; 2; 3]
```

(as semi-colon means 'start a new row') or:

```
>> y = x'
```

since x' means x^T ie x transpose.

Example 1: Plotting a function for analysis

Suppose you want to examine the function $y = x^4 \sin(x) e^{-0.1x}$ over the range -10 to 10. First generate a vector of x values over the required range, then calculate a vector of y-values (one for each x):

```
>> x=-10:0.01:10;
>> y=x.^4.*sin(x).*exp(-0.1*x);
```

Note that \wedge means 'to the power of'. I will explain why there are full stops before mathematical operators shortly. Also note the semi-colons after the definitions of x and y. They stop x and y being printed out on screen. To see the difference try:

```
>> x=-10:0.01:10
```

TIP: using the 'up' cursor brings commands you have just typed to the current line, so pressing 'up' 3 times will bring the definition of x to the command line. Alternatively, typing the beginning of a line eg x= and pressing the 'up' cursor consecutively brings up all commands starting with x= that were typed. Now plot x against y using:

```
>> plot(x,y)
```

Use the 'magnifying glasses' on the figure window to examine parts of the plot in more detail. (**ASIDE:** Axis limits can also be set via the commands: `axis`, `ylim` and `xlim`). You can also add/take-away grid lines via:

```
>> grid
```

There are many different ways plot can be used (see `help plot` for full options), but the most useful option is setting the colour/style of the lines drawn eg to do a scatter plot do:

```
>> plot(x,y,'rx')
```

Plotting several lines on one graph: As you may have noticed from the above, the default behaviour when you call plot is to erase the previous line. Suppose you also want to plot the function $z=100\sin(10x^2)$. Use the function `hold` to keep/release current plots via

```
>> z=100*sin(10*(x.^2));
>> hold on
>> plot(x,z,'g:*')
>> hold off
```

`hold on` its own toggles the state of the figure. Alternatively, call plot with more vectors ie:

```
>> plot(x,y,x,z,'g:*')
```

Annotating figures: You can put arrows, lines and text on figures using the 3 icons to the left of the magnifying glasses on the figure window. Also use the 'insert' menu (or commands `xlabel`, `ylabel`, `title` and `legend`) to annotate axes etc (NB for those who know LaTeX, matlab interprets text strings as LaTeX so eg `\mu` generates the Greek letter μ). You can also

interactively change figures by clicking the left-most arrow icon on the figure window then clicking lines, axes labels etc to highlight objects. Highlighted objects can be deleted, or by right-clicking, their properties can be changed. Highlight then right-click one of the lines on the graphs to see what happens.

Saving and outputting figures: You can save figures from the figure window as .fig files. They can be re-opened using `openfig('fig_name')` or the open icon on the figure window. In this form, figures can be re-edited. If you want to print figures, you can copy and paste them into many applications (including word and powerpoint) using 'copy' from the edit menu on the figure (you may have to do 'paste special' into some applications. Alternatively, you can print them directly from the file menu. Finally, they can be exported in many file formats from the file menu. You can also set default preferences from the file menu.

Useful functions: if you want to plot several different things, `figure` generates another figure to plot in while `subplot` lets you plot several figures in the same window but in separate axes. If you have several figures/axes, `plot` plots in the last one generated or activated (ie by clicking on it). `close` closes the most recently activated figure and `close all` closes all figure windows.

Try plotting a few functions eg $\exp(x) \cdot \log(x)$ for x between 0 and 10.

ASIDE: If you double click a part of the figure, the property editor window opens. This allows you to alter all properties of the figure eg tick-spacing etc. You can also open the editor by typing `propedit`. The most powerful way of altering figures however, is using the commands `set` and `get`. These allow you to get and alter properties of any object using its 'handle' (an identifier) which can be set when you generate the object, or can be called via the identifiers `gcf` and `gca` which get the handles of the most recent figure and axis respectively. Thus eg:

```
>> set(gcf, 'FontSize', 16, 'TickDir', 'out', 'Box', 'off')
```

sets the font-size property of the current axis (and objects belonging to it to 16), the direction of axis ticks to outwards and deletes the box around the axis. While some property settings are useful, they are quite fiddly and I tend to get a figure roughly ready, then export in eg illustrator format for final editing.

2.2 Numerical Variables 2: Matrices

The other type of numerical variable in Matlab is a matrix such as: $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$

Matrices can be thought of as collections of row or column vectors (in fact, vectors are special cases of matrices) and are entered in a similar way eg:

```
>> M=[1 2 3;4 5 6]
>> V=[3 2 1]
```

(remember that ';' means 'start a new row'). As with vectors, matrices can be built up from other matrices, but note that matrices must be rectangular ie each row/column must have the same number of rows. Try and predict which of the following will work and what shape the resulting matrix will have:

```
>> N=[M; V]
>> N=[M V]
>> N=[M [1;2]]
```

Extracting data: Matrices are indexed as (row, column) so to get at the element in the 1st row and 2nd column do:

```
>> M(1,2)
```

In contrast, $M(2,1) = 4$. You can also use vectors as indices to pull out multiple bits of data from matrices eg if you want elements from the 2nd row and 1st and 3rd columns do:

```
>> M(2, [1 3])
```

However, the main way data is extracted from matrices is by extracting whole rows or columns. This is accomplished using a colon ':' which stands for '1:end'. Eg enter the matrix

$$s = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 6 & 9 & 12 \\ 1 & 4 & 12 & -1 \end{pmatrix}$$

via:

```
>> s=[1:4; 3:3:12; 1 4 12 -1]
```

To extract the 2nd row do:

```
>> secrow = s(2,:)
```

to extract the 3rd column do:

```
>> s(:,3)
```

The command 'end' can also be used if you don't want to extract the whole row eg

```
>> s(1,2:end)
```

which will extract the all of the 1st row of s apart from the 1st element.

Special Matrices: There are several special functions used to generate useful matrices all with the same syntax specifying the size of the matrix produced. `ones` generates a matrix of 1s, `zeros`, 0's `eye` the identity matrix and `rand` random numbers between (but not including) 0 and 1. To see how they work try:

```
>> a=ones(3)
>> b=eye(6)
>> c=zeros(3,10)
>> d=rand(size(c))
>> e=20.5*ones(4,6)
```

Now practice extracting data from them eg `plot i=1:10` against the 3rd row of d

ASIDE: note the use of the command `size(matrix_name)` which returns the dimensions of `matrix_name` as a vector

Matrix arithmetic

Matrix arithmetic is easy (it's what Matlab was designed for), but be aware it is matrix arithmetic and if dimensions do not match you will get an error. As a general rule, adding/multiplying/dividing by scalars works fine, and you can add/subtract matrices if they are the same size, but there is no division and multiplication/raising to powers will only work if the dimensions are correct.

It is more usual to use the point-by-point versions of mathematical operators which are indicated by a '.' prefixed to the operation (as in the plotting example, above). This applies the arithmetic operation to all the elements of the matrix individually. Eg if you wanted to square every element of M you would do:

```
>> M.^2
```

However, you must be sure the sizes match eg to divide every element of x by the corresponding element in y

```
>> x./y'
```

gives an error whereas

```
>> x./y
```

works as x has the same dimensions as y . Note that Matlab has all (and more) very fast matrix operations you can think of (eg covariance, eigenvectors, pseudo-inverse etc etc etc).

ASIDE Eg 2: Logical operators and matrices

Using logical operators like AND (&), OR (|), greater (> or >=) and less than (< or <=) can greatly simplify matrix analysis. For instance, suppose we want to find out how many elements in v are between 0.2 and 0.7.

```
>> v=rand(10)
>> v>0.2
```

produces a matrix of 1's and 0's according to whether the elements of v are greater or less than 0.2 (very useful for eg thresholding images). Similarly $v \leq 0.7$ produces a matrix with 1's where the elements are less than or equal to 0.7. The two commands can be combined with the logical AND operator to find elements of v which satisfy both conditions ie:

```
>> n=(v>0.2)&(v<=0.7)
>> NumOnes=sum(sum(n))
```

Here the inner 'sum' calculates how many 1's are in each column, and the outer 'sum' adds the column totals up.

2.3 3D Plotting

Matlab's 3-dimensional plotting is one of its strongest features. There are several types of plots such as `surf`, `mesh`, `pcolor` etc (see help on any of these functions to see all the options). The basic way these functions work is to plot a matrix as heights so that the value in the ij 'th position in the matrix is at position i on the x-axis and j on the y-axis eg try:

```
>> v=rand(10)
>> surf(v)
>> shading interp
>> pcolor(v)
>> colorbar
>> pcolor(double(v>0.5))
>> pcolor((v>0.5).*v)
```

In the penultimate example, the command `double` is used as the result of $v > 0.5$ is a logical array ie 1's and 0's interpreted by matlab as 'trues' and 'falses'. The command `double` makes matlab interpret them as numbers.

Matlab also has some 'easy' versions of the plotting functions such as `ezmesh`. Do:

```
>> help ezmesh
```

for a list of its properties and try a few of the examples eg:

```
>> ezmesh('x*exp(-x^2 - y^2)')
>> ezmesh('s*cos(t)', 's*sin(t)', 't')
>> ezmesh('x*y', 'circ')
```

NB You can copy and paste the commands from the help comments. Alternatively (on newer versions of matlab), you can highlight text and execute it by pressing F9

This can lead to very impressive figures in papers, but is not as easy as the 'ez' makes out. Another nice feature is that you can rotate the axes by clicking on the round arrow next to the magnifying glasses in the figure window, clicking on the plot and dragging.

Try experimenting with a few of the features. Also look in the examples of eg `plot3` and the other functions listed under the *See Also:* section of `ezmesh`

ASIDE: 2.4 Saving and loading the workspace

Variables can be saved and loaded using the 'save' and 'load' commands (see help-files for details). The simplest way to do this is via:

```
>> save MyWorkspace_2_3
```

This saves all the variables in the current workspace (ie all the ones you have been using) to a matlab file called 'MyWorkspace_2_3.mat' which can be re-loaded via:

```
>> load MyWorkspace_2_3
```

Alternatively, you can save named variables rather than all of them. To see a list of all the current variables use the command `whos` and then select some to load ie:

```
>> whos
>> save MyWorkspace_2_3 v m n x
```

You can also save/load variables as plain text (ascii) separated by spaces/tabs etc eg:

```
>> save MyVariable.dat v -ascii
```

You can thus use matlab to analyse/graph data from other programs. Note however that as it is a single variable, if you are loading a text file, it must have the same number of values in each row and it will be assigned to a matrix variable.

ASIDE: The `clear` command can be used to delete named variables, or all variables in the workspace if no variables are named. To save in another directory, change directory using the browser or the `cd` function (NB `cd` on its own tells you which directory you are in).

3. Working from M-files

3.1 Functions and m-files

The other major way of working with Matlab is by writing your own functions. These are simply collections of commands that you would have written at the command line. Functions are written in m-files (a file with a .m extension) and are named the same as the function contained within them. Eg **FunctionExample.m** will have a first line (not counting comments) of:

```
% This is a comments line. Use % to comment things out
% Function to add and take away 2 numbers a and b

function[TheSum,TheDifference] = FunctionExample(a, b)
```

Input and output: Round brackets are needed around input variables, as well as an equals sign between the 'function' statement and the function name. Square brackets are needed around the outputs (if any). Inputs and outputs are separated by commas. They do not have a specified type and so anything can be input/output, but errors will occur if the variables you enter are incompatible with the function commands. Open the file **FunctionExample.m**. You can generate/open an m-file and open the matlab editor by using the file menu at the top of the command window. If it is in a separate window, you can open it in the same window as the command prompt by doing: Dock: FunctionExample from the View menu.

Running functions: A function is run by typing its name at the command line. Note that the function's name is the name of the m-file it is in and NOT the name inside the m-file (though by convention, these are the same). You can also use the 'Run' command from the editor, but you cannot input parameters or assign output variables.

However, matlab only 'knows' about functions that are in the current directory or directories on its 'path'. Before running functions you therefore need to set the path so that it contains the directories where your m-files are. I tend to have one directory with all my matlab files in. Before you can run the functions you therefore need to:

- a) Copy **FunctionExample.m** to a directory you want to keep your m-files in
- b) From the File menu choose 'Set Path...'
- c) Click 'add folder' and from the pop up browser, select the directory with **FunctionExample.m** in
- d) Click OK, then save, then close

You can now run FunctionExample from the command line via:

```
>> [s,d] = FunctionExample(2,4)
```

When you run a function, each command in the m-file is executed in order. There is no specific command to signify the end of a function. The function is terminated either by the end of the file, by a return statement or by another 'function' statement. Functions can only 'see' and therefore manipulate variables that are defined within it, known as local variables.

As with the function definition, when calling functions with input/output from the command line (or from within other functions) you need round brackets around inputs and square brackets around multiple output variables (as above). These variables are now in the workspace and can be used. Eg do

```
>> s+d
```

If you do:

```
>> FunctionExample(2,4)
>> s = FunctionExample(2,4)
```

Only the first output will be assigned (and displayed).

Ending lines: Lines are ended with a semi-colon. If no semi-colon is entered the line is

printed out (useful for debugging): try deleting the semi-colon from the end of the "TheDifference = a-b;" line. Save the file and re-run it. You **MUST** save the file before any changes will take effect. This is a **VERY** common cause of programs mysteriously not working/updating.

Comments: Things are commented out by a % at the beginning of a line. These lines are not run. Uncomment by deleting the %. Blocks of commands can be commented/uncommented by highlighting the selection and using the commands from the Text menu.

Subfunctions: Subfunctions (or internal functions) are written below the main function and are signified by a 'function' command. They can be called by any other functions within that m-file, but not by functions in other m-files or from the command line. If you have a common routine, write it in another m-file.

Now uncomment the subfunction at the bottom of the file, comment out the line starting 'TheSum', insert the line: `TheSum = SubFunctionSum(a,b);` and save. This shows subfunctions in use.

Stopping execution: `ctrl+c` stops function execution (useful if you're stuck in an infinite loop).

ASIDE: De-bugging: The error messages that you get are usually pretty self-explanatory, so do read them. If there is an error in an m-file and you click on the error message it will take you to that point in the m-file.

There is also an in-built de-bugger with standard commands (on the Debug menu or via the icons at the top of the editor window). Debug mode is automatically entered if a breakpoint is encountered when the function is being run. Breakpoints can be set at a line via the menus or by clicking on the 'hyphen' at the left hand side of the editing window, between the line number and the edit-able space. Place a break point by the 'TheSum' line and run the program. A green arrow will appear to tell you where you are in the m-file and placing your cursor over variables shows you their value. In the command window the prompt becomes:

```
K>>
```

You can now manipulate variables local to the function you are in from the command window and see the results. Eg, using the debug commands (either from the menu, or the icons on the right hand-side of the edit window), step into the subfunction and do

```
K>> a-c
```

Then step out of it again and do:

```
K>> TheSum*a
```

One final tip when debugging: if you keep getting strange error messages it may be because you have named a variable with one of matlab's function names. To check this do:

```
>> help variable_name
```

and see if any help comments come up.

3.2 If, for and while statements

If statements are used to do a set of commands if a given specified condition is satisfied. The set of commands is started by an `if` or `else` statement and ended by an `else` or `end`. They

have the following structure:

```
if (i==1)
    a=1;
elseif (j~=3)
    a=2;
else
    a=3;
end
```

Note that you need a double == or the function `isequal` to test for equality. Inequality is specified by ~=

To run sets of commands multiple times, 'for' and 'while' loops are used. 'For' loops do the commands written between the `for` and `end` statements (the 'for' loop) once for each condition specified eg

```
sum = 0;
for i=1:10
    sum=sum+i;
end
```

does the command 'sum=sum+i;' once for i values of 1 to 10. The main difference to other programming languages is that `for` loops are indexed using vectors. Thus the line 'for i=3:6' means 'generate a vector i=[3, 4, 5, 6] and step through the `for` loop using each element of i in turn'.

`while` loops do the commands between the `while` and `end` statements (the 'while' loop) until the condition specified is satisfied eg:

```
sum = 0;
while (sum<10)
    sum=sum+1;
end
```

will keep adding 1 to sum until it reaches 10. Simple examples using `for`, `while` and `if` loops are contained in **LoopExamplesSimple.m**. Copy this across to your matlab directory and open it in the editor. Comment/uncomment the various sub-functions to see how they work. Note the use of `break` to terminate loop execution.

Now open **LoopExamplesComplex.m**. This contains various slightly more complex loop examples as follows:

Task 1: Extract the 3rd – 6th columns of a matrix and put them in a new matrix a

Run **LoopExamplesComplex.m**. This runs the 1st of the subfunctions, `LoopEg1`. This function prints out `i` and builds up the new matrix column by column.

Note that we have to initially define `a` as the empty matrix (line 15) otherwise when we arrive at the 1st instance of line 19 `a` is undefined and an error is generated. Comment out line 15 and see what happens (remember to save before re-running).

Also, as matrices have to be rectangular, this only works since as we build up the matrix, each new column has the same number of rows. Change line 15 to `a=1` to see what happens

The shape of `a` is determined by the way we specify the internal indices in the right hand side of line 19. Change line 19 to the following to see how this works:

```
a=[a ; testmatrix(:,i)]
a=[a ; testmatrix(:,i)']
```

ASIDE Task 2: Extract selected columns of a matrix and put them in a new matrix `a`

Due to the vector nature of the for loop indexing, this is easy. We simply need to change the vector `i` eg to select columns 1 4 and 7 change line 17 to

```
for i = [1 4 7]
```

Alternatively, to select all even columns do

```
for i = 1:2:size(testmatrix,2)
```

Note `size(m,2)` returns the number of columns in `m` while `size(m,1)` returns the number of rows. Similarly `length(x)` returns the length of vector `x`

The only problem with this approach is if one needs to index something within the for loop eg to generate a vector containing the *i*'th value of each column selected on the *i*'th pass through the loop. To do this one would need to do something like

```
v = [1 4 7];
for i = 1:length(v)
    i
    b(i) = testmatrix(i,v(i))
end
```

Task 3: Extract every column of a matrix but in a random order. If the whole column is 1's, exit the loop

Extracting rows in a random order is a common task in neural network training when input training vectors should be presented in a random order. It is easily achieved with the function `randperm(n)` which returns a random permutation of the integers 1 to `n`. Try:

```
>> randperm(5)
```

To see this in action, run `LoopEg2`. This function also demonstrates the use of `isequal` which compares every element of 2 matrices and `break` to exit a loop.

Task 4: Extract each column in turn. If the 6th element of a column is 0 skip that and the next two columns.

An attempt to do this in a for loop and a while loop is in `LoopEg2`. Comment out the call to `LoopEg1` and uncomment `LoopEg2`. Run `LoopExamples`, and note the difference between the 2 versions. This demonstrates the disadvantage of indexing for loops with a vector: despite the fact that `i` is re-set inside the loop, the index vector has already been 'set' and so it simply continues.

Task 5: Check the 1st element of each column of a matrix. If it is less than 0.5, put the column in a new matrix `a`

Sometimes though a task seems to need a loop, it might not be so. Matlab has lots of handy functions which avoid loops as this is beneficial in terms of speed and elegance of code. One of the most useful is `find` which returns the indices of elements which satisfy a specified criteria. `NonLoopEg` shows `find` in action. Run it to see its effects.

ASIDE: Eg 3 Plotting iteratively

Plotting lines iteratively is a good way of observing the evolution of a system/solution over time. There are 2 ways of doing this illustrated in `PlottingExamples.m`. `IterativePlotEG1` uses the function `pause` which waits until a key is pressed. Run `PlottingExamples` and see its behaviour.

Using `pause` is fine if you only have a small number of steps to do. However, suppose we want to look at the behaviour of the system in more detail eg over steps of 0.05. This would mean we would have to loop through hundreds of steps many of which – given the cyclical nature of the system – may very well be repetitive and will strain our finger pressing key. I therefore often use the function `input` (which waits for user input) instead. Together with a couple of `if` statements, this allows you to examine the initial behaviour, then see what happens at the end without looking at all the intermediate steps. Run `PlottingExamples` with `IterativePlotEG2` uncommented to see an example.

Note that the function `sin` takes radians rather than degrees (1 radian = 180°). In maths texts, it is generally assumed that angles are given in radians rather than degrees (for more details, ask).

ASIDE: Eg 4 loading and manipulating data from text files

Suppose we want to analyse reaction time data from 7 subjects that are held in text files which were output from another program. The data is held in 2 columns, each row of which holds 1 of 30 responses. The first column holds the response made (either: 0, 1, or 2, where a value of 1 indicates a correct response) and the second column holds the reaction time.

We want to analyse the data to find out for each participant the number of correct (ie '1') responses, the mean reaction time for these responses and the standard deviation, and to output this data to another text file.

The m-file performing the above is **DataFileEg.m**. In it, the data is first generated using the sub-function **GenerateData** (lines 58-96), and then the 1st 6 files are analysed. There are several new functions used within this m-file which I will briefly explain (for more detail, see the help files).

find: The most important function used here is `find`. `find` works by returning the indices of all elements of a variable that match the (logical) condition specified as a vector. For instance try:

```
>> a=5:10
>> b=find(a>7)
>> c=find(a~=8)
```

Though it might not sound it, this is extremely useful:

- Firstly, in conjunction with the `length` function (which returns the number of elements in a vector), it can be used to find out how many elements match the condition (see lines 15-19).
- Secondly, it can be used to perform an operation only on those elements that match the condition. This is done by using the vector of indices as an index into the matrix, as in lines 83-86.
- Thirdly, the vector of indices can be used to 'mark out' rows (or columns) that match the condition. Thus in lines 23-24 as we know the indices of all the 1's from the 1st column, it is a simple matter to identify and operate on the reaction times where a 1 response was recorded.

Functional form of load and save: When loading/saving data from text files it is often useful to use the functional form of `load` as this allows you to have file names that vary and to assign loaded data to a variable eg:

```
>> fname='DataFileSubject1.dat';
>> h=load(fname, '-ascii');
```

To do this, the file name, `fname`, and any parameters such as `-ascii`, must be strings. Strings are formed by putting single quotes around text. `int2str` and `num2str` (see help-

files) are useful functions used to put variables into a string. In these cases you need square brackets around the whole string eg

```
>> s=['andy is ' int2str(32) ' but looks ' num2str(68.5) \'.']
```

and lines 9 and 90.

Other functions that are used are:

`subplot(x,y,n)`: which splits the current figure into an x by y array of subaxes and makes the n'th one current for plotting in

`bar(x)`: which plots a bar chart of the values in x. This is often used in conjunction with values output by `hist`

`mean(x)`, `std(x)`: which generate the mean and std of x respectively.

Run the file to see the results. Note that because there were no '1' responses for file 7, the mean and s.d. come up as NaN which stands for 'not a number'. This can cause problems and so it is often advisable to check if any responses have been found using eg:

```
isempty(is)
```

ASIDE: If you want to load all the files in a given directory use the function `dir`. It returns details of the files in the current directory as a structure. See `help struct` for details of how to access members of a structure.

4. Getting help

Matlab has many, many built in functions or functions written by people on the web. Before writing **ANY** mathematical operation it is worth searching for it.

The first place to get help is to do: `help FunctionName`.

If there is not enough detail there, try looking at the help-files for the related functions. Alternatively, there is more detail in the on-line help pages. To do this do:

```
>> doc FunctionName
```

Or:

```
>> helpwin
```

which brings up the matlab help window and lets you search for functions etc. Alternatively, go to the matlab web-site (www.mathworks.com), which has reference material (including pdfs of their reference books). It also has details of all their toolboxes together with demos (see below). Finally doing:

```
>> type functionexample
```

displays all the code in `FunctionExample.m`

Normally, however, you do not know the name of the function you want to use. In this case use `lookfor` which searches the help comments of m-files. Eg to find all functions which have the word 'gradient' in the help comments do:

```
>> lookfor gradient
```

Or:

```
>> lookfor gradient -all
```

Do 'help lookfor' to see the difference between the 2. If this doesn't help, it might be worth looking through the demos for the various Matlab toolboxes. Typing:

```
>> demo
```

brings up the toolboxes and their demos. Toolboxes are collections of functions for a particular topic, some distributed by Matlab and some by individuals (and usually free: eg netlab, PsychToolbox etc). Looking through the demos will give you an idea of the functions that can be performed how to use them, though some are considerably better than others.

Another port of call is the matlab central file exchange located at:

```
http://www.mathworks.com/matlabcentral/fileexchange/loadCategory.do
```

which can be searched for particular functions.

Finally, a general internet search for a key word and matlab often turns up good free software.

FCS Problem Sheet 3: Central Limit Theory

Copy **CentralLimitTheoryEg.m** into your matlab directory and open it. This file demonstrates the central limit theorem. Mathematical details of the theorem can be found at: <http://mathworld.wolfram.com/CentralLimitTheorem.html>. Random variables, and probability in general, will be covered in week 9's lecture. You do not need to know all the mathematical details to run the example, however. The idea is to experiment with the model to see whether it matches theoretical predictions and how its parameters interact and affect the results. You therefore only need a general notion of the theory, as described below.

If you roll a dice many times, you would assume that each number would come up roughly the same number of times (with equal frequency). If you plotted the frequencies of each number you would expect the graph to be roughly flat. Moreover, the more times you tossed the dice the flatter you would expect the graph to look.

However, if you had 2 dice and added (or averaged) their scores, you would expect to get many more middling numbers (6, 7, 8) than extreme numbers like 2 or 12. This is simply because there are many more ways to get a score of 7 (6 ways: 1&6, 2&5, 3&4, 4&3, 5&2 or 6&1) than there are to get 12 (1 way: 6&6). This means that if you rolled both dice many times and plotted the *frequencies* of each score (ie how many times each score occurred), you would get a graph with higher frequencies in the middle, and lower ones at the edges.

If you were to use 3 or more dice, you would get even less extreme values and more middling ones since it's a rarer occurrence to get all 1's on 7 dice than it is to get all 1's on 2. The question is, however, if one were to plot the frequencies of the scores for, 2, 10 or 20 dice what would the graph look like? Is it more steeply humped in the middle for 10 dice rather than 2 dice? What happens at the extreme values?

The Central Limit Theorem begins to answer these questions. It says that if you have enough *random variables* (dice, in this case) their *mean* (average) score will approach being *normally distributed* (that is, the graph will start to look bell-shaped) with a mean equal to the mean of the random variables (the graph will be centred on 3.5 in the dice case) and a *variance* (how widely spread the bell is) which depends on the number of random variables used. The question is, therefore, how many random variables is **enough** and how **closely** does the distribution of mean scores resemble a normal distribution ...

We will investigate this by examining the frequencies of the means of different numbers of random variables. We shall therefore perform the following steps (referenced with the relevant line numbers from **CentralLimitTheoryEg.m**):

1. **Generate NumVars random variables, 'throwing' them each NumEgs times.** This is performed in line 13 using `rand`. Each row of the matrix `RandomVariables` holds `NumEgs` instances (ie the throws) of a different random variable. However, note that unlike in the dice example, where the scores are an integer between 1 and 6 (ie they are *discrete*), here each instance is a number between 0 and 1 (they are *continuous*).
2. **Work out the mean score of each throw.** The *i*'th column of `RandomVariables` holds the *i*'th instance of all the random variables, so the *i*'th mean score is the mean of the *i*'th column. We can therefore get the means using the function `mean` as in line 34. `mean(x)` works in a similar way to many matlab functions (eg `sum`, `std`, `var`, `diff` etc. See help for details): if `x` is a vector (row or column), it will return a single number, whereas if `x` is a matrix it will return the mean of each column as a vector. As `RandomVariables` may be a matrix or a vector (if `NumVars=1`), we override the default behaviour via `mean(x,1)`, which forces column means (whereas `mean(x,2)` forces row means).
3. **Plot a bar chart of frequencies of the average scores.** Now we have `NumEgs` average scores, we need the frequency of each score (ie how many times it appears). However, as the random variables are continuous (as discussed in 1), no two mean scores will have exactly the same value (so each has frequency 1). We therefore have to divide the possible range of mean scores (from 0 to 1) into a number (specified by `NumBins`) of

equal sized 'bins' eg 4 bins means: [0 - 0.25], [0.25 - 0.5], [0.5 - 0.75] and [0.75 - 1]. Each mean score is then assigned to a bin, and the number in each (its frequency) recorded. This process is performed by the function `hist`. It can be called in many ways, but the most convenient here is to pass in the mean scores and a vector specifying the centre of each bin (bin centres set in line 18). This returns the frequencies (line 37), which can be used together with `bar` (or `stairs` or other functions) to plot a bar chart (line 44).

4. **Compare a plot of the frequencies of the mean scores, with a plot of the theoretical normal distribution they should resemble.** As we want to compare the frequencies with a (normal) probability distribution we need to turn the frequencies into a probability distribution. This is done by rescaling the frequencies so that the sum of the area of the bars is 1 (line 42) and plotting them (line 44). The plot is then 'held' (line 50) and the normal distribution (bell-shaped curve) they should look like, drawn using the subfunction `DrawNormal` (lines 69-80). Note that as the variance (spread of the curve) depends on the number of random variables (`NumVars`), it has to be passed to the subfunction. Also, note how point-by-point operations are used to plot the normal curve.

Questions:

3 parameters affect the behaviour of the system: `NumVars`, the number of random variables; `NumEGs`, the number of instances of each variable and `NumBins`, the number of bins the mean scores are assigned to. The following questions investigate their effect.

1. Start by examining the effect of `NumVars`. The program has been written so that a vector `NumVars` can be entered, with a graph produced for each member of `NumVars` (remember `close` all closes all open figure windows). Start with:

```
>> CentralLimitTheoryEg([1:5 10],100,10)
```

What happens as the number of variables increases? Try `NumVars =20`. What's the lowest `NumVars` that gives a good approximation to the normal distribution? [3 marks]

2. How is the behaviour affected by changing the number of instances, `NumEGs`? If you have 500 instances do you need a smaller `NumVars` to get a good approximation to a normal distribution? What happens if you only have 10? Can you explain this behaviour? [3 marks]
3. Now examine the effect of `NumBins`. Are its effect linked to the value of `NumEGs`? What if we have more bins than there are instances (try `NumBins =200`, `NumEGs =50`)? What happens as you decrease the number of bins? Is there an 'optimal' number of bins for `NumEGs=50`? How does this change as you change `NumEGs`? [4 marks]

Postscript: While the effect of increasing `NumVars` could be deduced from theory, the effect of the other parameters is more easily seen by practical experimentation. Indeed, the interplay between these 2 factors is an important issue in *probability density estimation*, where data distributions are inferred from a sample of points (see Bishop, 95 pp. 49-59). The goal is to get as detailed a picture of the distribution as possible (ie try to maximize `NumBins`) without introducing artifacts due to the scarcity of data. Thus `NumBins` can be seen as a *smoothing parameter* (which you may come across in the Neural Networks course).

This is the style of practical experimentation I am aiming for on this course. Theoretical mathematical issues will be introduced and then tested by building and experimenting with a model. In the course of this process, design decisions will be made (normally parameters to be set) and experimented with, which can highlight practical difficulties with certain techniques. It is important, however, to try to relate observations back to the theory so one can explain **why** such-and-such 'quirky' behaviour occurred.