Introducing Matlab® & Simulink®

PART1: INTRODUCTION

Since its introduction more than a decade ago, Matlab and its add-on development tools (called toolboxes) have become one of the most widely used engineering, scientific and mathematical software tools in the world. Its initial appeal was the provision of performing complex matrix algebra in a user-friendly manner. However, with continual expansion of its capability, Matlab is now employed to aid designers in the mathematics of signal processing, control, mechanical, aerospace, electrical/electronic engineering, and economics disciplines, backed up by what has become a professional suite of plotting/graphical commands for displaying data. Furthermore, as we shall see, a specialised simulation toolbox is available, 'Simulink', that enables hierarchical time-based dynamic analysis of systems to be readily carried out.

This short work package cannot explore all the functions and facilities available to you in the Matlab kernal and toolboxes. However, by showing you how to use a few of the most useful commands, and indicating what other commands are available, you should be in a position to explore the full power of Matlab for yourselves (and attack your impending assignment).

It should be noted that this work-package assumes that you have a copy of Matlab available to you. You should use Matlab interactively with this document to obtain maximum benefit in minimum time. Matlab (and Simulink) are available on the University network. It is expected that this work package should take 1-2 hours to complete.

Anything that is written in *italic* is either a command that you can type into Matlab at the 'chevron' (>>) command prompt, or the response that Matlab provides after you have typed the command---you'll get the idea quick enough.

Entering the Matlab Environment

Matlab is available on the University network; you can enter the Matlab environment by double-clicking on the Matlab icon or by selecting *******"START→APPLICATION→ACADEMIC→MATHS&STATS→MATLAB"*****.
Note: subject to change

If you cannot find the correct icon, please see one of the support staff.

Having entered the Matlab environment, at the top of the new window that has opened you should see something like:

```
To get started, type one of these: helpwin, helpdesk, or demo.
For product information, type tour or visit www.mathworks.com.

»
```

(the exact wording depends on which version of Matlab you are using, but the general outline should be similar)

The chevron » is Matlab's standard command prompt. Any command that appears on the same line as the '»' is for you to type in.

Using Matlab as a Calculator

One of the simplest uses of Matlab is as a basic calculator. Let's try it out; type *a*=0.5 at the chevron prompt (then press ENTER/CR of course), as follows

```
» a=0.5

a =
   0.5000
»
```

Notice that Matlab echoes the result of your command to show you that it has been accepted ok. Let's input another variable, *b=0.666666666666*:

```
» b=0.666666666666

b =
   0.6667
»
```

Notice that Matlab displays a 'rounded-up' version of any entered variable (typically to four decimal places). However, this is for display purposes **only**, and Matlab actually stores more than 13 significant figures for calculation purposes. If you want Matlab to display numbers in a different format, you can use the *format* command. For help, type *help format* at the command prompt. This also demonstrates how you can get information on **any** Matlab command.

```
» help format

FORMAT Set output format.
   All computations in MATLAB are done in double precision.
   FORMAT may be used to switch between different output
   display formats as follows:
     FORMAT         Default. Same as SHORT.
     FORMAT SHORT   Scaled fixed point format with 5 digits.
     FORMAT LONG    Scaled fixed point format with 15 digits.
     FORMAT SHORT E Floating point format with 5 digits.
…………..
```

Let's try displaying numbers to more decimal places, enter *format long*:

```
» format long
```

We can now have a look at the value of *b* again, just by typing the variable name at the command prompt:

```
» b

b =
   0.66666666666600
```

Matlab will now display all numbers to 14 decimal places. See what *a* looks like:

```
» a

a =
   0.50000000000000
```

Clearly, 14 decimal places can be a bit excessive, so we will change back to the standard *short* format. Type:

```
» format short
```

OK let's do some calculation. Let's multiply *a* and b and store the result in a variable called c. Type:

```
c=a*b

c =
   0.3333
```

In the same manner, you can employ all the normal arithmetic operations " +, -, *, /" with parenthesis ( ) where necessary. (You can get help on the full range of arithmetic operations by typing *help ?* at the command prompt)

OK what about something a little more exciting---- FUNCTIONS. Matlab supports all the usual functions that you would find on a calculator. Let's try calculating sin(c) and call the result d --(just for fun).

```
» d=sin(c)

d =
   0.3272
```

Note that Matlab uses radians by default.
You can also use multiple operations on a command line:

```
» d=d*360/(2*pi)

d =
   18.7469
```

Notice that Matlab already knows what *pi* is---it is a reserved name for π.

Whilst we are looking at variables, it is often instructive to call variables by a name that is more descriptive i.e. *a*,b,c,d, doesn't tell you much about the calculation you have performed. In Matlab, you can assign variable names as you like, although you cannot use spaces or arithmetic operations in variable name definitions. For instance, type:

```
» d_result=d

d_result =
   18.7469
```

{Note: Matlab variable name assignments are case-sensitive i.e. 'dummy' is a different variable to 'Dummy' }

After a long session of calculating, it is often the case that you cannot remember all the variables that you have declared and defined. You can have a look at them by typing *who* at the command prompt:

```
» who
Your variables are:

a          c          d_result
b          d
```

As with many software packages, you can go back and execute previous commands using the ↑ key on the keyboard.

Well, we can calculate the sin() of a variable and multiply and divide---whoopee ! Other common functions that are available in Matlab, and used in a similar way to the *sin* function. These include:, *exp, sinh, asin, cos, cosh, acos, tan, tanh atan, sec...., cot...., csc* etc. Use the *help* command if you are not sure about a function you require. The square-root of a number can be obtained using *sqrt* command. A minor anomaly with Matlab is that the *log* command actually returns the natural logarithm (ln); to obtain a logarithm to the base 10 you have to use the *log10* command.

Due to the clever people who wrote the kernel for Matlab, complex numbers are also catered for in the same manner as real numbers. For example, you can define *j* as the square-root of *–1*, although

Matlab naturally holds imaginary numbers with an 'i' prefix due to historical reasons. To define *j* as √(*-1*), type:

```
» j=sqrt(-1)

j =
        0 + 1.0000i
```

So complex arithmetic can also be employed. Try declaring/defining the multiplication of the two following complex numbers:

```
» a=23+j*92

a =
  23.0000 +92.0000i

» b=0.7+j*40

b =
  0.7000 +40.0000i

» c=a*b

c =
-3.6639e+03+ 9.8440e+02i
```

If you want to separate the real and imaginary parts of a number, you can use the *real* and *imag* commands:

```
» real(c)

ans =
 -3.6639e+03

» imag(c)

ans =
  984.4000
```

OK with a bit of practice then, so you should be able to use Matlab as a basic calculator. Let's have a look at something a little more useful; vectors and matrices.

Vectors and Matrices

The best way to look at how to define a matrix is by example (be careful about where commas and semicolons are used). Type:

```
» first_matrix=[1,2,3;4,5,6;7,8,9]

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

From this example it should be clear that commas , separate elements of a row and semicolons ; separate the rows. In fact, you can actually miss out the commas to separate elements of a row and just use spaces:

```
» first_matrix=[1 2 3;4 5 6;7 8 9]

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

Row and column vectors can be defined similarly:

```
» row_vector=[10 11 12]

row_vector =
   10   11   12

» column_vector=[13;14;15]

column_vector =
   13
   14
   15
```

Once you have inputted your vectors and matrices, you can perform simple matrix arithmetic eg. multiplication:

```
» result_matrix=first_matrix*column_vector

result_matrix =
   86
   212
   338
```

Note: when performing matrix operations, you have to make sure that all your matrices and vectors are compatibly dimensioned—Matlab will throw up an error otherwise:

```
» result_matrix=first_matrix*row_vector

??? Error using ==> *
Inner matrix dimensions must agree.
```

A common requirement when working with matrices is the transpose operation. Matlab uses the 'tilda' or inverted comma on your keyboard to signify this operation:

```
» mat_transpose=first_matrix'

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

Elements of a matrix or vector can be referred to, or extracted for other operations, by using its elemental position. For example, to obtain the middle value of mat_transpose i.e. the value of the element at the intersection of the second row and the second column use:

```
» element=mat_transpose(2,2)

element =
   5
```

Of course you can also perform addition and subtraction of matrices, however, the great advantage of Matlab will start becoming clear when we consider more complicated operations on matrices. For example, finding the inverse *inv* of a matrix, or the determinant (*det*) of a matrix, or finding the eigenvalues and eigenvectors. Let's define a simple matrix and see how some of these operations are performed. Define/declare the matrix *A*:

```
» A=[1,0,0;10,2,0; 20, 3, 5]

A =
    1   0   0
   10   2   0
   20   3   5
```

Let's find the inverse of A:

```
» inv(A)

ans =
    1.0000        0        0
   -5.0000   0.5000   0.0000
   -1.0000  -0.3000   0.2000
```

Notice that if you do not assign a variable name to the result of a calculation, Matlab holds the result in the variable *ans*.

We can also find the determinant *det* of *A*:

```
» det(A)

ans =
   10
```

and the eigenvalues *eig* of A:

```
» eig(A)

ans =
    5
    2
    1
```

Many in-built Matlab functions return more than one result, for example, the *eig* command can return the eigenvectors of a matrix along with the eigenvalues. We can store both the results by using two result variables:

```
» [eigenvectors, eigenvalues]=eig(A)

eigenvectors =
        0        0   0.0966
        0   0.7071  -0.9656
   1.0000  -0.7071   0.2414


eigenvalues =
    5   0   0
    0   2   0
    0   0   1

»
```

The result of the *eig* command now provides two matrices, one with the eigenvalues on the diagonals, and one of the associated eigenvectors. This demonstrates the general form of all Matlab commands viz.

[output1, output2, output3, ..., outputN]=function_name(input1,input2,....,input N)

To find out how many inputs and outputs a function might need, use the *help* command eg.

```
» help eig

 EIG    Eigenvalues and eigenvectors.
        E=EIG(X) is a vector containing the eigenvalues of a square
        matrix X.

        [V,D] = EIG(X) produces a diagonal matrix D of
        eigenvalues and a full matrix V whose columns are the
        corresponding eigenvectors so that X*V = V*D.
 .........
```

Other commands for obtaining information about the structure of matrices include *rank*, *svd* (singular value decomposition), *pinv* (pseudo inverse), *null* (null-space),... among many others. Note: you can perform many (nested) mathematical operations on a single command line, rather like modern calculators.

As we shall see, it is often the case that we want to construct a vector whose elements are incremental values. For example, we may want a vector whose elements contain time steps of 0.1 seconds between 0 seconds and 100 seconds. We could form the vector in the normal manner i.e. by typing t=[0 0.1 0.2 0.3 0.4 … 99.7 99.8 99.9 100] however, this may take us a while. Fortunately, Matlab provides us with a way of quickly obtaining such a vector. The structure of the command to do this is:

Vector_name=startvalue:increment:finalvalue

For example, in our case Type:

```
» t=0:0.1:100

t =
 Columns 1 through 7
      0    0.1000    0.2000    0.3000    0.4000
0.5000    0.6000

 Columns 8 through 14
  0.7000    0.8000    0.9000    1.0000    1.1000
1.2000    1.3000

etc
```

You will have noticed that the variable *t* that we have constructed is quite large and takes up a lot of space on the Matlab screen. This brings us onto a neat little aside. If you are not concerned with Matlab printing out the variable after you have declared it, or the result of a function, you just terminate the command with a semicolon ; . For example:

```
» t=0:0.1:100;
```

Now, although Matlab has not printed *t*, it has still stored the whole vector in memory. We can see this by using the *size* command to look at the dimensions of the vector *t*:

```
» size(t)

ans =
      1      1001
```

indicating that the vector *t* has 1 row and 1001 columns, as expected.

Matlab can also perform block operations on matrices and vectors. For example, let's begin by defining a vector *x* of numbers between 0 and 2π at increments of 0.5:

```
» x=0:0.5:2*pi

x =

 Columns 1 through 7
     0    0.5000    1.0000    1.5000    2.0000
2.5000    3.0000

 Columns 8 through 13
  3.5000    4.0000    4.5000    5.0000    5.5000
6.0000
```

Let's assume that the elements of *x* are actually angles (specified in radians), and we want to find the cosine of all the elements. We simply type:

```
» result=cos(x)

result =

 Columns 1 through 7
  1.0000   0.8776   0.5403   0.0707  -0.4161  -
0.8011  -0.9900

 Columns 8 through 13
 -0.9365  -0.6536  -0.2108    0.2837    0.7087
0.9602
```

and we have the cosine of all the elements of *x* by using only a single command. The same techniques also apply to matrices.
This brings us nicely onto plotting graphs in Matlab.

PLOTTING DATA

Over recent years, Matlab has been augmented with a whole suite of professional plotting tools for both two dimensional and three dimensional representation of data. In this short introductory text, we only have room to give you a brief overview of the most used commands.

You will probably find that 90% of the time, you will use the *plot* command, and its derivatives, so let's give it a go:

```
» plot([0 1 2 3 4 5],[16 27 4 -3 -2 9])
```

This command will have opened up an additional figure on your screen. The plot command plots each element of the first vector (x-axis) against the corresponding element of the second vector (y-axis). In this case the points (0,16) (1,27) (2,4) (3,-3) (4,-2) (5,9) are plotted. Matlab automatically scales the axes (note: you can override the Matlab default scaling using the *axis* command --- use *help axis* for more details). CLEARLY, YOU NEED THE SAME NUMBER OF ELEMENTS IN BOTH VECTORS FOR THE PLOT COMMAND—MATLAB PROVIDES AN ERROR IF THIS IS NOT THE CASE.

On the most recent versions of Matlab, there will be icons above the figure that has been opened for zooming into your graph, or putting arrows on it, or writing text on the graph. Try some of these out for yourself if possible.

You can also use variables or operations in the *plot* command (note we have already defined *x* as a vector of values):

```
» plot(x,cos(x))
```

It should be clear from the figure that is generated that Matlab connects the points on the graph by straight lines. This is often inconvenient. If you do not want the points to be connected, you can just plot the points with a symbol of your choice, for example, an asterisk '*':

```
» plot(x,cos(x),'*')
```

You MAY also want to plot more than one curve on a figure. You can do this in two ways:

i) use the *hold* command. This holds the axis scaling and the previous graph. The next plot command will plot directly over the last figure:
Try:

```
» hold
Current plot held
» plot(x,sin(x))
```

If you want start to again and plot another figure, you must release the hold command by typing *hold* again.

```
» hold
Current plot released
```

ii) Employing multiple declarations in the plot command:

```
» plot(x,cos(x),x,sin(x))
```

The two curves have now been drawn with a single command.

To professionalise your figures a bit you can type a title using, *title*, a label on the x-axis using, *xlabel*, a label on the y-axis using, *ylabel*, and you can place a grid on the plot, *grid*. Try typing in the following lines to annotate your figure.

```
» title('sin(x) and cos(x)')
» xlabel('time')
» ylabel('amplitude')
»grid
```

We have only touched on the power of the *plot* command. To see more advanced features use the help command *help plot*. Three dimensional plots can also be generated, look for *help* on *plot3, surf* and *mesh* commands for instance.

SAVING/LOADING THE WORKSPACE

After a long session of using Matlab, you may not want to loose all the data and variables you have generated. You can save the Matlab working environment (your variables) using the *save* command:

```
»save mydata
```

All your data will now be stored in a file called *mydata.mat* .

After a rest, you may want to restore your data and variables from the file. You can do this using the *load* command:

```
»load mydata
```

All the variables you have stored will now be restored into the Matlab workspace. You can find out if they are there by using the *who* command.

OK, WE'VE GOT THROUGH A LOT HERE. YOU MAY WANT TO TAKE A BREAK NOW BEFORE WE MOVE ONTO SOME MORE ADVANCED MATERIAL.

THE BEST WAY TO USE THIS INTRODUCTION IS TO QUICKLY REVIEW PART 1 NEXT TIME, AND THEN MOVE ONTO PART 2.

PART2: POLYNOMIALS AND TRANSFER FUNCTIONS

As well as defining vectors for use with linear algebra, Matlab also allows the definition of vectors for the representation of polynomials. The elements of the resulting vector are the coefficients of decreasing power. What does this mean ? Well, consider the polynomial $2x^2+3x+4$; this is represented in Matlab by defining the vector:

```
» polynom=[2 3 4]

polynom =
    2    3    4
```

That is, we define a polynomial in the same way as a normal vector; it's the functions we apply to the vector that interpret it as a polynomial. For instance, if we want to find the roots of $2x^2+3x+4=0$, we can use the *roots* command:

```
» calc_roots=roots(polynom)

calc_roots =
  -0.7500 + 1.1990i
  -0.7500 - 1.1990i
```

Conversely, if we have the roots of a polynomial, we can obtain the polynomial vector that provides those roots using the *poly* command:

```
» poly(calc_roots)

ans =
    1.0000    1.5000    2.0000
```

thus providing $1x^2+1.5x+2=0$ which is actually the same as the previous equation (both sides divided by 2). Therefore, *poly* and *roots* are complimentary operations.

This representation of polynomials, and the simple operations that can be carried out on them can be very powerful. Consider the Laplace transfer function of a system:

$$\frac{s^2+2s+2}{s^5+19.5s^4+164.26s^3+733.64s^2+1636.8s+1393.92}$$

Although we could use the quadratic rule to obtain the roots of the numerator (called the zeros) of the transfer function, there is no simple way for us to obtain the roots of the denominator (called the poles). However, we can readily obtain this information by defining the numerator and denominator as vectors of polynomial coefficients in the way we have seen previously:

```
» num=[1 2 2]

num =
    1    2    2

» den=[1 19.5 164.26 733.64 1636.8  1393.92]

den =
  1.0e+003 *

   0.0010    0.0195    0.1643    0.7336    1.6368
  1.3939
```

We can now obtain the poles and zeros of the transfer function using the *roots* command as before:

```
» zeros=roots(num)

zeros =
  -1.0000 + 1.0000i
  -1.0000 - 1.0000i

» poles=roots(den)

poles =
  -4.0000 + 4.0000i
  -4.0000 - 4.0000i
  -6.0000
  -3.3000
  -2.2000
```

ANALYSIS OF DYNAMICAL SYSTEMS

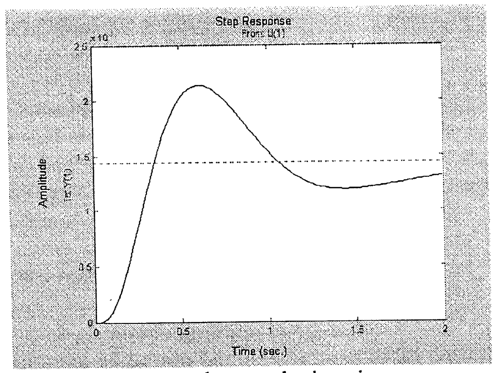
The ability to manipulate polynomials (and hence transfer functions) means that Matlab can readily perform dynamic systems analysis, for example, the generation of step responses using the *step* command, or *bode* plots using the *bode* command.

{The following section of commands can only be used if the appropriate (control and signal processing) toolboxes are available. If you have trouble with any of the commands, ask one of the supervisory staff to make sure the Matlab command PATH is set to access the control system toolbox and the signal processing toolbox. Failing this, just read through the section to get a 'feel' for the underlying concepts and move onto Part 3: Simulink.}

Let's look at the step response of the previous system:

```
» step(num,den)
```

A figure should have opened up on your screen showing the step response of the system .

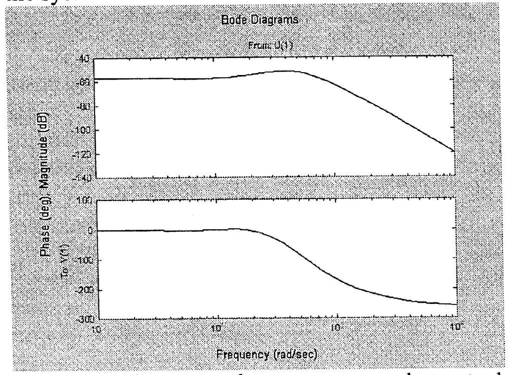


[Another command to obtain time response information is *impulse* to obtain a plot of the impulse response.]

Let's now plot the frequency response of the previous transfer function using our polynomial definition of the numerator and denominator of the transfer function:

```
» bode(num,den)
»
```

The *bode* command will have opened up a figure on your screen showing the frequency response of the system.



It's often the case that you want the actual magnitude and phase data from your transfer function at particular frequencies, rather than just a plot. The bode command allows this by using output arguments for the function; type:

```
» [mag,phase,w]=bode(num,den);
```

Notice that a semicolon has been placed after the command to stop Matlab printing out all of the data. However, Matlab has still stored the frequency response data in memory—try using the *who* command to see the variables that have been created, and perhaps the *size* or *whos* commands to see the dimensions of the resulting vectors of data.

Before we can plot a bode diagram of the data manually, we need to put the magnitude data into dB's. This should be relatively straightforward based on what we have learned so far:

```
» magdB=20*log10(mag);
```

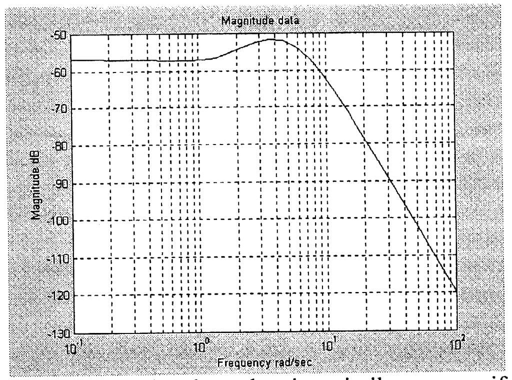
If you want to look at the magnitude and phase data you just type the name of the variable. Since a bode plot actually uses a logarithmic scale for the frequency axis, the normal *plot* command isn't appropriate for plotting bode data. We can, however, use another command to provide a logarithmic scaling of the x-axis i.e. *semilogx*:

```
» semilogx(w,magdB)
```

And you may want to use the *title*, *xlabel*, *ylabel* and *grid* commands to make your plot look more professional:

```
» title('Magnitude data')
» xlabel('Frequency rad/sec')
» ylabel('Magnitude dB')
» grid
```

The resulting plot will look something like:



You can plot the phase data in a similar manner if you want.

The *bode* command is very useful and has a number of variations, type *help bode* to get a complete description. Other commands for plotting dynamic system data include *nyquist*, *ngrid* & *nichols* and *rlocus* (root-locus—you'll come across this in the Feedback Systems Design in 2nd/3rd year modules). Use the *help* command to see what these offer.

{Although you may not have encountered discrete-time systems i.e. using z-transform transfer functions, Matlab provides analogous commands for these types of discrete systems viz. *dstep*, *dbode* etc. The numerator and denominator of a z transfer function is defined in exactly the same way as before; it's Matlab's discrete-time functions that interpret the polynomial vectors as a digital transfer function rather than a Laplace transfer function—more on this in later years}

SIGNAL PROCESSING COMMANDS

Matlab now comes with a toolbox full of useful signal processing commands viz. *xcorr* (cross correlation), *conv* (convolution), *DFT* (discrete Fourier transform), *FFT* (fast Fourier transform), *hanning* (Hanning window function), *hamming*

(Hamming window function) and many, many others. You will find these types of commands useful throughout the remainder of your course.

For clarity, we shall find the convolution of two signals *x=[0.1 0.2 –0.7 –0.3 1.2]* and *y=[-0.4 0.3 1.2 –0.3 2]* :

```
» x=[0.1 0.2 -0.7 -0.3 1.2]

x =

   0.1000   0.2000  -0.7000  -0.3000   1.2000

» y=[-0.4 0.3 1.2 -0.3 2]

y =

  -0.4000   0.3000   1.2000  -0.3000   2.0000

» result=conv(x,y)

result =

  Columns 1 through 7

  -0.0400  -0.0500   0.4600   0.1200  -1.2700
0.6100   0.1300

  Columns 8 through 9

  -0.9600   2.4000
```

You should now be in a position to find commands that will help you with a particular problem, and be able to use the commands competently, although, of course, a little practice will still be required.

m-files

When using your calculators, you will have encountered situations where calculations have to be carried out repetitively, or where many different calculation steps have to be performed before the desired result is achieved. These are some of the reasons why programmable calculators became popular before the advent of the cheap PC. Matlab has a simple programming language that has all the constructs (for-next, if-else etc.) that you would normally find in a high level language. The source files that you write are called m-files, and Matlab automatically compiles them for you. In effect, m-files allow you to supplement the standard Matlab functions that we have previously been using, by designing your own. Let's do an example to show the structure of an m-file.

In the Matlab command window, go to the FILE menu and select NEW→M-FILE. This will open up a simple editor where we are going to construct a simple m-file to plot the step responses of a second order system for different values of damping ratio (note: in general you can use any editor you like).

The example is therefore to plot the step response of the Laplace transfer characteristic:

$$\frac{\omega_n^{\;2}}{s^2 + 2\zeta\omega_n s + \omega_n^{\;2}}$$

OK. the first thing you need to do is assign a name to the m-file function. We shall use the name *dampplot.* See the example file shown below—type this in for yourselves using the editor.

```
function dampplot

%clear graph
clg

%plot step response of second order system with
damping ratio=1.
wn=1;
num=[wn*wn];
den=[1, 2*1*wn, wn*wn];
step(num,den);
hold

%plot step responses of second order system with
other damping ratios
%on same graph

for i=0.2:0.1:0.9
        wn=1;
        zeta=i;
        num=[wn*wn];
        den=[1, 2*zeta*wn, wn*wn];

        step(num,den);
end
grid
title('step responses')
```

Save the file as *dampplot.m*

[Make sure you save the file in a directory that Matlab has access to. If in doubt, ask a member of support staff.]

The function can now be executed by typing the function name at the Matlab command prompt.

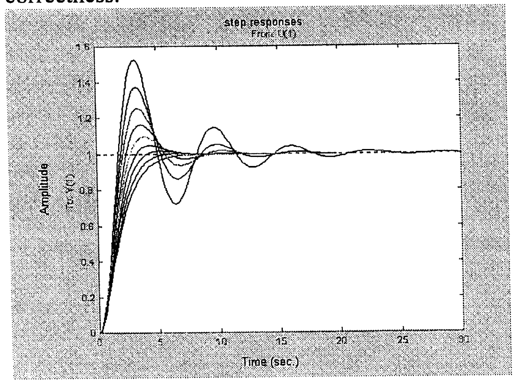
Description of function

The first line of the m-file shows the function name. Comments can be included in the program by using a percentage sign at the beginning of a line. Since we are going to plot a graph, we initially clear any graphs already plotted using the *clg* command—note: this command is now becoming obsolete but should still work. The next part of the program uses Matlab's *step* command to produce the step response of a second order system with damping ratio of unity. Notice that we have defined the numerator and denominator of the 2nd order transfer function using variables. This is fine so long as the variables have been assigned numbers before they are used. The *step* command automatically plots the step response, as we have previously seen. We now use the *hold* command so we can plot other step responses on the same graph. The next step is to construct a *for-next* loop to iteratively obtain the step response of the transfer function for different values of damping ratio. The *step* command in the for-next loop repeatedly plots the step response on the same graph. The last couple of lines of the program just adds a *title* to the graph and puts on a *grid*.

You can execute the function by typing its name.

```
» dampplot
Current plot held
```

You should get a graphical output something like what is shown below (colours and axes may vary). If not, check your m-file program for correctness.



In this way, any Matlab commands you can use on the command line (») can also be used in an m-file.

As with any Matlab function, you can pass scalars, matrices, or vectors to one of <u>your</u> m-files, and, likewise, receive scalars, matrices or vectors from <u>your</u> m-file function. Type in the m-file *steplot.m* as shown below.

```
function
[amplitude1,t1,amplitude2,t2]=steplot(zeta1,zeta2)

%calculate step response of second order system
with damping ratio=zeta1.
wn=1;
num=[wn*wn];
den=[1 2*zeta1*wn wn*wn];
[amplitude1,x,t1]=step(num,den);

%calculate step response of second order system
with damping ratio=zeta2.
wn=1;
num=[wn*wn];
den=[1 2*zeta2*wn wn*wn];
[amplitude2,x,t2]=step(num,den);
```

You now pass two parameters to the function, *zeta1* and *zeta2* (in this case the parameters are scalars; in general the parameters could be vectors or matrices) from the Matlab command line, and receive the step responses of the 2nd order systems with the two damping ratios that you provided. To execute the m-file type:

```
» [amplitude1,t1,amplitude2,t2]=steplot(0.1,1);
```

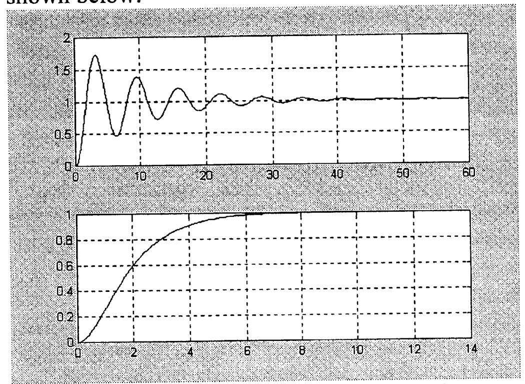
Notice that since we used a semicolon at the end of the command line, Matlab does not return all the data. Try leaving the semicolon off and see how Matlab responds. Because we have returned *amplitude1,t1,amplitude2,t2* from the m-file, the data is now in Matlab's memory. Use the *who* or *whos* command to see the variables that Matlab is storing. We now have access to the data from the Matlab command line. Let's plot the results from the function manually. We could use the plot command, as before:

```
» plot(t1,amplitude1)
» plot(t2,amplitude2)
```

although these commands mean that we only plot one result at a time. If we want to plot both results separately on the same figure we can use the *subplot* command. This effectively enables you to plot multiple <u>separate</u> plots on a single screen shot. The format of the subplot command is *subplot(number of rows of plots,number of columns of plots, plotnumber)*. For our example we have only two results to plot, and we will plot one on top of the other i.e. 2 rows and 1 column. Since there are only 2 plots the variable *plotnumber* will take on the values of 1 and 2. At the Matlab command line type the lines:

```
» subplot(2,1,1)
» plot(t1,amplitude1)
» grid
» subplot(2,1,2)
» plot(t2,amplitude2)
» grid
```

The figure resulting from these commands is shown below:



As with all other Matlab commands, *subplot* can also be used within an m-file.

Miscellaneous Commands

Other commands that are extremely useful but which we have not yet covered include:

i) *ch dir* ,this means change directory, and is used in the same way as the cd command in DOS. It enables you to make the Matlab workspace refer automatically to one of your own directories.

ii) *save filename* ,this command saves the Matlab workspace and all the variables you have declared. It enables you save your session of work so you can carry on from where you left off the next time you enter Matlab.

iii) *load filename,* loads back data that was *save*d in a previous Matlab session.

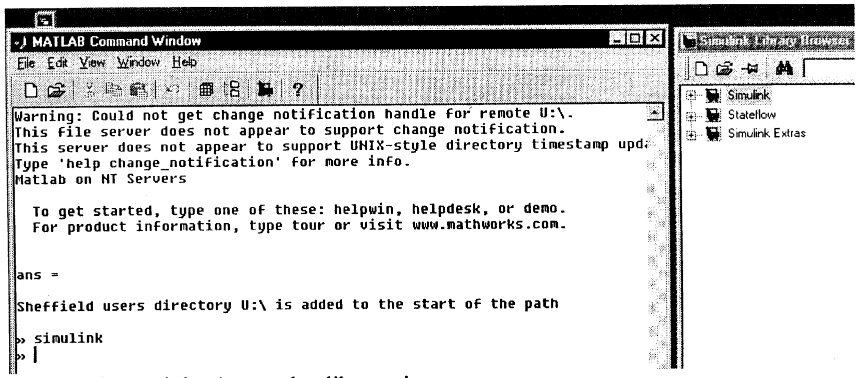
OK, AGAIN WE'VE GOT THROUGH A LOT. YOU MAY WANT TO TAKE A BREAK NOW BEFORE WE MOVE ONTO SOME MORE ADVANCED MATERIAL.

PART3: SIMULINK

Upto now we have used command line operations in Matlab to perform dynamic analysis of systems (via the definition of a transfer function). There is, however, a very simple, but powerful graphical simulation toolbox available for Matlab that enables you to very simply perform dynamic simulations of (possibly very complex) systems. You get access to this toolbox by typing *simulink* at the Matlab command line.

```
» simulink
```

In response to the command, a new window will open. If you are using a relatively new version of Matlab/Simulink, you will have obtained a Simulink library window something like the one



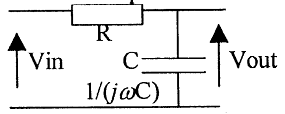
```
Warning: Could not get change notification handle for remote U:\.
This file server does not appear to support change notification.
This server does not appear to support UNIX-style directory timestamp upd
Type 'help change_notification' for more info.
Matlab on NT Servers

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, type tour or visit www.mathworks.com.


ans =

Sheffield users directory U:\ is added to the start of the path

» simulink
» |
```

shown above-right (note: the library icons may vary slightly).

This 'front-end' provides access to the main libraries of dynamic components in *Simulink*. Go to the FILE menu on the *Simulink* window and open a NEW file/model—Alternatively, click on the NEW-FILE icon.

ALL systems that can be described analytically or numerically can be simulated using *simulink.* These include electrical/electronic, mechanical, hydraulic, aerospace, economic etc etc systems. Here, we are going to construct a simple dynamic model of a simple RC low pass filter, and simulate the step response. Because of the generality of *simulink*, we describe systems mathematically rather that with resistors and capacitors etc.

Consider then the low pass filter shown below.



You will know from network theory that this system can be described in the $j\omega$ domain by:

$$\frac{V_{out}}{V_{in}}(j\omega) = \frac{1}{j\omega CR + 1}$$

For convenient input into *simulink*, we replace $j\omega$ with 's' to give the dynamic equation describing the system as :

$$\frac{V_{out}}{V_{in}}(j\omega) = \frac{1}{sCR + 1}$$

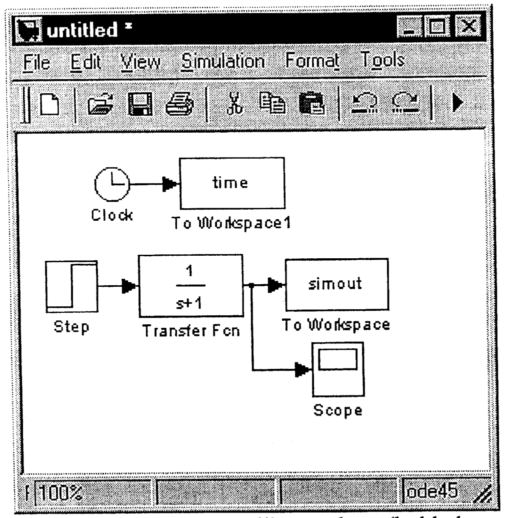
This is now in a standard form that we can use in simulink. For our particular case, we shall assume R=1kΩ and C=1μF giving a time-constant of τ=CR=1×10^{-3}seconds, and

$$\frac{V_{out}}{V_{in}}(j\omega) = \frac{1}{10^{-3}s + 1}.$$

We are going to construct a model of this system using standard *simulink* library blocks and simulate the time domain dynamics.

To see the library of blocks that Simulink possesses, click on the '+' sign next to the Simulink icon.

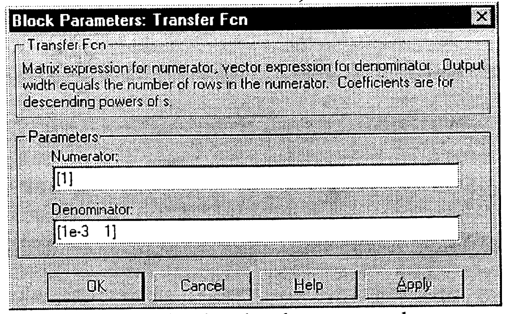
The blocks in the Simulink window are actually libraries of parts to make up a system. Open the *Math* library. This will open up the library of maths components that Simulink provides. You can also open up other library blocks if you wish, although the number of open windows may start to become difficult to handle so you may want to close them again. We are going to construct the system shown in the figure below.



From the '*Continuous*' library drag (hold down left mouse button and drag) a *Transfer Fcn* block into your new file.
The '*Clock'* and '*Step'* blocks are sources (inputs) and are help in the *simulink/Sources* library. The '*Scope*' and '*To workspace'* blocks are sinks (outputs), and are held in the *simulink/Sinks* library

Once you have the blocks in your new file, move them around with your left mouse button until they are in the formation shown above. You can now connect up the components by holding down the left mouse button at the outputs (or inputs) of the blocks and dragging a line between the output of one component and the input of the next.

Now we have connected up all the components of our system, we need to customise the block values and variables to what we want. We do this by double-clicking on the relevant component to open up a parameter window. Try double-clicking on the *Transfer Fcn* component to open the parameter window. The NUM and DEN parameters are the numerator and denominator of the 'transfer function' of our system that we want to simulate. We input the data in descending powers of 's' as follows (note: the highest power of 's' in the numerator is $s^0=1$):


Block Parameters: Transfer Fcn

Transfer Fcn

Matrix expression for numerator, vector expression for denominator. Output width equals the number of rows in the numerator. Coefficients are for descending powers of s.

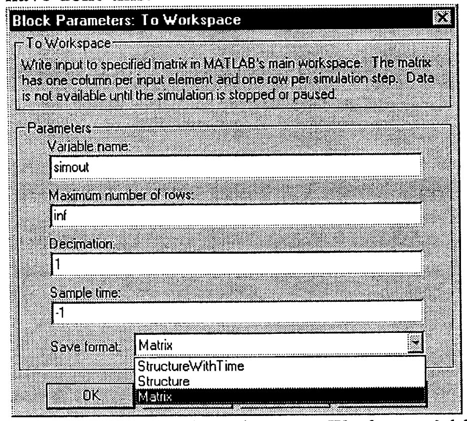
Parameters

Numerator:
[1]

Denominator:
[1e-3 1]

Press OK once the data has been entered

We can do the same for the other elements of the model. Open up the *To Workspace* block. The only thing you have to do here is change the 'Save format' option to Array (or Matrix) instead of Structure (see below). {Note: you can also change the variable name from *simout* to *something_else* if you wish.} Click OK when you have done this.


Block Parameters: To Workspace

To Workspace

Write input to specified matrix in MATLAB's main workspace. The matrix has one column per input element and one row per simulation step. Data is not available until the simulation is stopped or paused.

Parameters

Variable name:
simout

Maximum number of rows:
inf

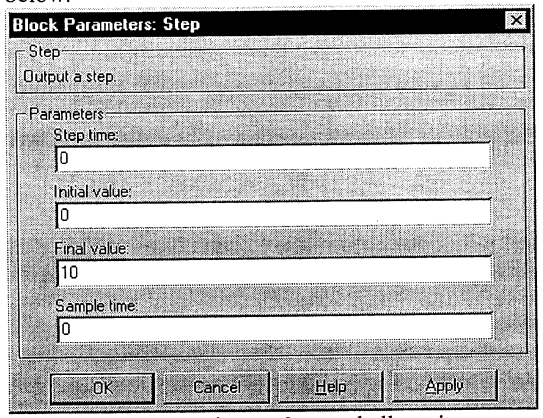
Decimation:
1

Sample time:
-1

Save format: Matrix

Do the same for the other '*To Workspace*' block and give it the variable name *time*. The purpose of the 'To Workspace' block is to save the data into the Matlab environment (for post processing). The purpose of the Scope block is to have a look at the data while the simulation is running—you can open it up by double clicking on it.

The *Step Fcn* component can also be customised (double-click on it). Set the parameters as shown below.


Block Parameters: Step

Step

Output a step.

Parameters

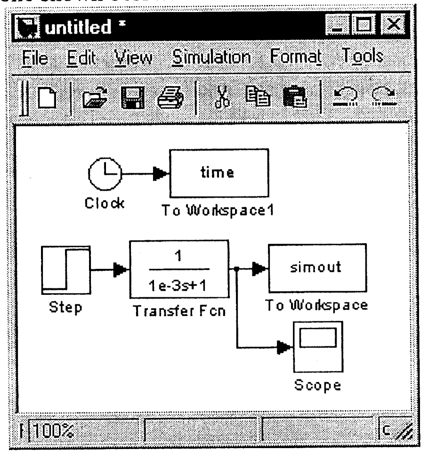
Step time:
0

Initial value:
0

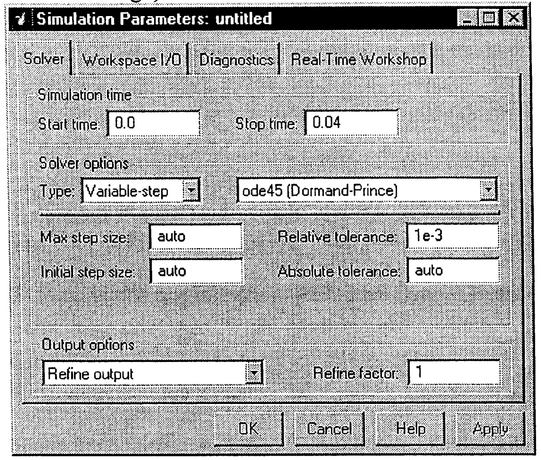
Final value:
10

Sample time:
0

This means that at time t=0, we shall excite our system with and input of 10 (Volts in our case).
Our system is now complete; it should look like the one shown below:



We now need to specify some simulation parameters. Appropriate selection of simulation parameters often takes a little practice, however, to get us 'up and running' open-up the *Simulation* menu on your model window toolbar, and enter the *Parameters* section. Type in the *Stop Time* Parameter as 0.4, as shown below.
{If you are using a different version of Simulink, leave any other variables in the window at their default settings}



Simulation Parameters: untitled

Solver | Workspace I/O | Diagnostics | Real-Time Workshop

Simulation time
Start time: 0.0   Stop time: 0.04

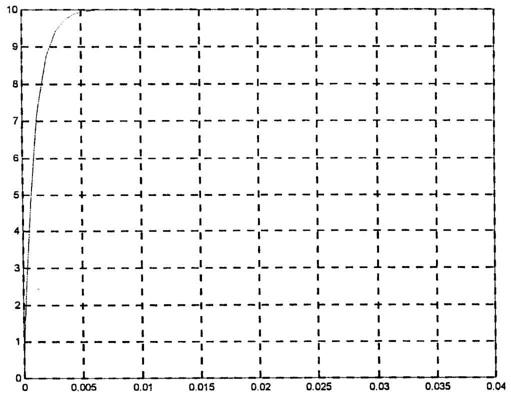
Solver options
Type: Variable-step   ode45 (Dormand-Prince)
Max step size: auto   Relative tolerance: 1e-3
Initial step size: auto   Absolute tolerance: auto

Output options
Refine output   Refine factor: 1

Press OK when the data has been entered.

We are now ready to simulate. Again go into the *Simulation* menu in your model window and select *Start*. You won't see anything on the screen, but Simulink will have simulated your system in a fraction of a second. The output of the system, (*simout* if you have not changed your *'To Workspace'* block), will now be a variable in Matlab's workspace and will contain a vector whose elements represent the output of your system (try typing *who* in the Matlab window to see the variable). You can now plot or manipulate the data as you would any other variable in Matlab. For example, let's plot the output of the system:

```
» plot(time,simout)
» grid
```



You can use *xlabel* and *ylabel* commands to label the axes from the Matlab environment:

```
» xlabel('seconds')
» ylabel('Vout (Volts)')
```

In the most recent versions of Matlab, a sophisticated graphical plotting window is provided to make adding captions and titles to figures very easy. In older versions you can use *text*, or *gtext* (use the *help* command for more information).

OK, so we've simulated a simple system (don't forget to save your simulation model if you want to use it again). Much more complex systems (digital, non-linear) are built up and simulated in the same way just by adding the appropriate dynamic blocks from the libraries. Have a look in the libraries and see what components are there.

We've now addressed the most basic features of Matlab and Simulink and you should have a basic feel of how to use the package. To get the best out of Matlab/Simulink, you have to implement a few dynamic systems of your own; have a look what is available in the other library blocks we have not used yet. Also, you will find that investing in a manual or a book showing you some of the more advanced features of the package will be well worth the money. For completeness, some of the functions available in Matlab are shown at the back of this document. You'll have to use the *help* command to get more information on them.